Chi-Hung Chi
Maarten van Steen
Craig Wills (Eds.)

LNCS 3293

Web Content Caching and Distribution

9th International Workshop, WCW 2004
Beijing, China, October 2004
Proceedings

Springer

Chi-Hung Chi Maarten van Steen
Craig Wills (Eds.)

Web Content Caching and Distribution

9th International Workshop, WCW 2004
Beijing, China, October 18-20, 2004
Proceedings

Springer

eBook ISBN: 3-540-30471-1
Print ISBN: 3-540-23516-7

Preface

Since the start of the International Workshop on Web Caching and Content Distribution (WCW) in 1996, it has served as the premiere meeting for researchers and practitioners to exchange results and visions on all aspects of content caching, distribution, and delivery. Building on the success of the previous WCW meetings, WCW 2004 extended its scope and covered interesting research and deployment areas relating to content services as they move through the Internet.

This year, WCW was held in Beijing, China. Although it was the first time that WCW was held in Asia, we received more than 50 high quality papers from five continents. Fifteen papers were accepted as regular papers and 6 papers as synopses to appear in the proceedings. The topics covered included architectural issues, routing and placement, caching in both traditional content delivery networks as well as in peer-to-peer systems, systems management and deployment, and performance evaluation.

We would like to take this opportunity to thank all those who submitted papers to WCW 2004 for their valued contribution to the workshop. This event would not have been possible without the broad and personal support and the invaluable suggestions and contributions of the members of the program committee and the steering committee.

August 2004

Chi-Hung Chi
Maarten van Steen
Craig Wills

General Chair
Chi-Hung Chi
Kwok-Yan Lam

Steering Committee
Azer Bestavros
Pei Cao
Jeff Chase
Brian Davison
Fred Douglis
Michael Rabinovich
Duane Wessels

Program Committee
Maarten van Steen (chair)
Craig Wills (vice-chair)
Gustavo Alonso
Chin-Chen Chang
Chi-Hung Chi
Michele Colajanni
Mike Dahlin
Magnus Karlsson
Ihor Kuz
Kwok-Yan Lam
Dan Li
Pablo Rodriguez
Oliver Spatscheck
Geoff Voelker
Limin Wang
Tao Wu
Zheng Zhang

Table of Contents

DotSlash: A Self-Configuring and Scalable Rescue System for Handling Web Hotspots Effectively*

Weibin Zhao and Henning Schulzrinne

Columbia University, New York NY 10027, USA
{zwb,hgs}@cs.columbia.edu

Abstract. DotSlash allows different web sites to form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site. As a rescue system, DotSlash intervenes when a web site becomes heavily loaded, and is phased out once the workload returns to normal. It aims to complement the existing web server infrastructure to handle short-term load spikes effectively. DotSlash is self-configuring, scalable, cost-effective, easy to use, and transparent to clients. It targets small web sites, although large web sites can also benefit from it. We have implemented a prototype of DotSlash on top of Apache. Experiments show that using DotSlash a web server can increase the request rate it supported and the data rate it delivered to clients by an order of magnitude, even if only HTTP redirect is used. Parts of this work may be applicable to other services such as Grid computational services.

1 Introduction

As more web sites experience a request load that can no longer be handled by a single server, using multiple servers to serve a single site becomes a widespread approach. Traditionally, a distributed web server system has used a fixed number of dedicated servers based on capacity planning, which works well if the request load is relatively consistent and matches the planned capacity. However, web requests could be very bursty. A well-identified problem web hotspots (also known as flash crowds or the Slashdot effect [2]) may trigger a large load increase but only last for a short time [14,24]. For such situations, overprovisioning a web site is not only uneconomical but also difficult since the peak load is hard to predict [16].

To handle web hotspots effectively, we advocate dynamic allocation of server capacity from a server pool distributed globally because the access link of a local network could become a bottleneck. As an example of global server pools, content delivery networks (CDNs) [27] have been used by large web sites, but small web sites often cannot afford the cost particularly since they may need these services very rarely. We seek a more cost-effective mechanism. As different web sites (e.g., different types or in different locations) are less likely to experience their peak request loads at the same time, they could form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site [10]. Based on this observation, we designed *DotSlash* which allows a web site to build an adaptive distributed web server

* This work was supported in part by the National Science Foundation (ANI-0117738).

system on the fly to expand its capacity by utilizing spare capacity at other sites. Using DotSlash, a web site not only has a fixed set of *origin servers,* but also has a changing set of *rescue servers* drafted from other sites. A web server allocates and releases rescue servers based on its load conditions. The rescue process is completely self-managing and transparent to clients.

DotSlash does not aim to support a request load that is persistently higher than a web site's planned capacity, but rather to complement the existing web server infrastructure to handle short-term load spikes effectively. We envision a spectrum of mechanisms for web sites to handle load spikes. Infrastructure-based approaches should handle the request load sufficiently in most cases (e.g., 99.9% of time), but they might be too expensive for short-term enormous load spikes and insufficient for unexpected load increases. For these cases, DotSlash intervenes so that a web site can support its request load in more cases (e.g., 99.999% of time). In parallel, a web site can use service degradation [1] such as turning off dynamic content and serving a trimmed version of static content under heavily-loaded conditions. As the last resort, a web site can use admission control [31] to reject a fraction of requests and only admit preferred clients.

DotSlash has the following advantages. First, it is self-configuring in that service discovery [13] is used to allow servers of different web sites to learn about each other dynamically, rescue actions are triggered automatically based on load conditions, and a rescue server can serve the content of its origin servers on the fly without the need of any advance configuration. Second, it is scalable because a web server can expand its capacity as needed by using more rescue servers. Third, it is very cost-effective since it utilizes spare capacity in a web server community to benefit any participating server, and it is built on top of the existing web server infrastructure, without incurring any additional hardware cost. Fourth, it is easy to use because standard DNS mechanisms and HTTP redirect are used to offload client requests from an origin server to its rescue servers, without the need of changing operating system or DNS server software. An add-on module to the web server software is sufficient to support all needed functions. Fifth, it is transparent to clients since it only uses server-side mechanisms. Client browsers remain unchanged, and client bookmarks continue to work. Finally, an origin server has full control of its own rescue procedure, such as how to choose rescue servers and when to offload client requests to rescue servers.

DotSlash targets small web sites, although large web site can also benefit from it. We focus on load migration for static web pages in this paper, and plan to investigate load migration for dynamic content in the next stage of this project. Parts of this work may be applicable to other services such as Grid computational services [12]. The remainder of this paper is organized as follows. We discuss related work in Section 2, give an overview of DotSlash in Section 3, present DotSlash design, implementation and evaluation in Section 4, 5 and 6, respectively, and conclude in Section 7.

2 Related Work

Caching [29] provides many benefits for web content retrieval, such as reducing bandwidth consumption and client-perceived latency. Caching may appear at several different places, such as client-side proxy caching, intermediate network caching, and server-side reverse caching, many of which are not controlled by origin web servers. DotSlash uses caching at rescue servers to relieve the load spike at an origin server, where caching is set up on demand and fully controlled by the origin server.

CDN [27] services deliver part or all of the content for a web site to improve the performance of content delivery. As an infrastructure-based approach, CDN services are good for reinforcing a web site in a long run, but less efficient for handling short-term load spikes. Also, using CDN services needs advance configurations such as contracting with a CDN provider and changing the URIs of offloading objects (e.g., Akamaized [3]). As an alternative mechanism to CDN services, DotSlash offers cost-effective and automated rescue services for better handling short-term load spikes.

Distributed web server systems are a widespread approach to support high request loads and reduce client-perceived delays. These systems often use replicated web servers (e.g., ScalaServer [5] and GeoWeb [9]), with a focus on load balancing and serving a client request from the closest server. In contrast, DotSlash allows an origin server to build a distributed system of heterogeneous rescue servers on demand so as to relieve the heavily-loaded origin server. DC-Apache [17] supports collaborations among heterogeneous web servers. However, it relies on static configuration to form collaborating server groups, which limits its scalability and adaptivity to changing environments. Also, DC-Apache incurs a cost for each request by generating all hyperlinks dynamically. DotSlash addresses these issues by forming collaborating server groups dynamically, and using simpler and widely applicable mechanisms to offload client requests. Backslash [25] suggests using peer-to-peer (P2P) overlay networks to build distributed web server systems and using distributed hash table to locate resources.

The Internet Engineering Task Force (IETF) has developed a model for content internetworking (CDI) [11,23]. The DotSlash architecture appears to be a special case of the CDI architecture, where each web server itself is a content network. However, the CDI framework does not address the issue of using dynamic server allocation and dynamic rate adjustment based on feedback to handle short-term load spikes, which is the main focus of DotSlash.

Client-side mechanisms allow clients to help each other so as to alleviate server-side congestion and reduce client-perceived delays. An origin web server can mediate client cooperation by redirecting a client to another client that has recently downloaded the URI, e.g., Pseudoserving [15] and CoopNet [20]. Clients can also form P2P overlay networks and use search mechanisms to locate resources, e.g., PROOFS [26] and BitTorrent [7]. Client-side P2P overlay networks have advantages in sharing large and popular files, which can reduce request loads at origin web servers. In general, client-side mechanisms scale well as the number of clients increases, but they are not transparent to clients, which are likely to prevent widespread deployment.

Grid technologies allow "coordinated resource sharing and problem solving in dynamic, multi-institutional organizations" [12], with a focus on large-scale computational problems and complex applications. The sharing in Grid is broader than simply file exchange; it can involve direct access to computers, software, data, and other resources. In contrast, DotSlash employs inter-web-site collaborations to handle web hotspots effectively, with an emphasis on overload control at web servers and disseminating popular files to a large number of clients.

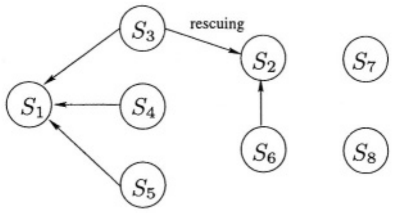


Fig. 1. An example for DotSlash rescue relationships

3 DotSlash Overview

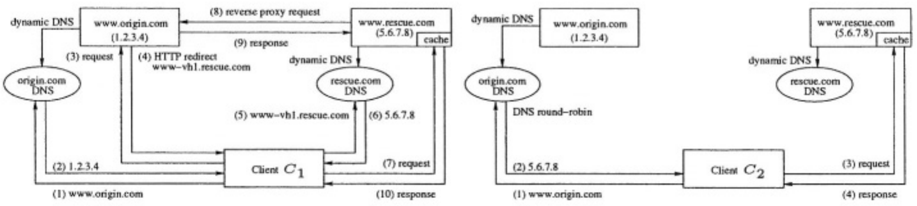
DotSlash uses a mutual-aid rescue model. A web server joins a mutual-aid community by registering itself with a DotSlash service registry, and contributing its spare capacity to the community. In case of being heavily loaded, a participating server discovers and uses spare capacities at other servers in its community via DotSlash rescue services. In our current prototype, DotSlash is intended for a cooperative environment, and thus no payment is involved in obtaining rescue services.

In DotSlash, a web server is in one of the following states at any time: *SOS state* if it gets rescue services from others, *rescue state* if it provides rescue services to others, and *normal state* otherwise. These three states are mutually exclusive: a server is not allowed to get a rescue service as well as to provide a rescue service at the same time. Using this rule can avoid complex rescue scenarios (e.g., a rescue loop where S_1 requests a rescue service from S_2, S_2 requests a rescue service from S_3, and S_3 requests a rescue service from S_1), and keep DotSlash simple and robust without compromising scalability. Throughout this paper, we use the notation origin server and rescue server in the following way. When two servers set up a rescue relationship, the one that benefits from the rescue service is the origin server, and the one that provides the rescue service is the rescue server. Fig. 1 shows an example of rescue relationships for eight web servers, where an arrow from S_y to S_x denotes that S_y provides a rescue service to S_x. In this figure, S_1 and S_2 are origin servers, S_3, S_4, S_5 and S_6 are rescue servers, and S_7 and S_8 have not involved themselves with rescue services.

3.1 Rescue Examples

In DotSlash, an origin server uses HTTP redirect and DNS round robin to offload client requests to its rescue servers, and a rescue server serves as a reverse caching proxy for its origin servers. There are four rescue cases: (1) HTTP redirect (at the origin server) and cache miss (at the rescue server), (2) HTTP redirect and cache hit, (3) DNS round robin and cache miss, and (4) DNS round robin and cache hit. We show examples for case 1 and 4 next; case 2 and 3 can be derived similarly.

In Fig. 2, the origin server is *www.origin.com* with IP address *1.2.3.4* (referred to as S_o), and the rescue server is *www. rescue. com* with IP address *5.6.7.8* (referred to as S_r). S_r has assigned an alias *www-vh1.rescue.com* to S_o, and S_o has added S_r's IP address



(a) For HTTP redirect and cache miss (b) For DNS round robin and cache hit

Fig. 2. Rescue examples

to its round robin local DNS. Fig. 2(a) gives an example for case 1, where client C_1 follows a ten-step procedure to retrieve *http://www.origin.com/index.html:*

1. C_1 resolves S_o's domain name *www.origin.com;*
2. C_1 gets S_o's IP address *1.2.3.4;*
3. C_1 makes an HTTP request to S_o using *http://www.origin.com/index.html;*
4. C_1 gets an HTTP redirect from S_o as *http://www-vh1.rescue.com/index.html;*
5. C_1 resolves S_r's alias *www-vh1.rescue.com;*
6. C_1 gets S_r's IP address *5.6.7.8;*
7. C_1 makes an HTTP request to S_r using *http://www-vh1.rescue.com/index.html;*
8. S_r makes a reverse proxy request to S_o using *http://www.origin.com/index.html* because of a cache miss for *http://www-vh1.rescue.com/index.html;*
9. S_o sends the requested file to S_r;
10. S_r caches the requested file, and returns the file to C_1.

Fig. 2(b) gives an example for case 4, where client C_2 follows a four-step procedure to retrieve *http://www.origin.com/index.html:*

1. C_2 resolves S_o's domain name *www.origin.com;*
2. C_2 gets S_r's IP address *5.6.7.8* due to DNS round robin at S_o's local DNS;
3. C_2 makes an HTTP request to S_r using *http://www.origin.com/index.html;*
4. C_2 gets the requested file from S_r because of a cache hit.

4 DotSlash Design

The main focus of DotSlash is to allow a web site to build an adaptive distributed web server system in a fully automated way. DotSlash consists of dynamic virtual hosting, request redirection, workload monitoring, rescue control, and service discovery.

4.1 Dynamic Virtual Hosting

Dynamic virtual hosting allows a rescue server to serve the content of its origin servers on the fly. Existing virtual hosting (e.g., Apache [4]) needs advance configurations: registering virtual host names in DNS, creating *DocumentRoot* directories, and adding directives to the configuration file to map virtual host names to *DocumentRoot* directories. DotSlash handles all these configurations dynamically.

A rescue server generates needed virtual host names dynamically by adding a sequence number component to its configured name, e.g., *host-vh<seqnum>.domain* for *host.domain,* where *<seqnum>* is monotonically increasing. Virtual host names are registered using *A* records via dynamic DNS updates [28]. We have set up a domain *dot-slash.net* that accepts virtual host name registrations. For example, *www.rescue.com* can obtain a unique host name *foo* in *dot-slash.net,* and register its virtual host names as *foo-vh<seqnum>.dot-slash.net.* A rescue server assigns a unique virtual host name to each of its origin servers, which is used in the HTTP redirects issued from the corresponding origin server.

As a rescue server, *www.rescue.com* may receive requests using three different kinds of *Host* header fields: its configured name *www.rescue.com,* an assigned virtual host name such as *www-vh1.rescue.com,* or an origin server name such as *www.origin.com.* Its own content is requested in the first case, whereas the content of its origin servers is requested in the last two cases. Moreover, the second case is due to HTTP redirects, and the third case is due to DNS round robin. A rescue server maintains a table to map assigned virtual host names to its origin servers. To map the *Host* header field of a request, a rescue server checks both the virtual host name and the origin server name in each mapping entry; if either one matches, the origin server name is returned. Due to client-side caching, web clients may continue to request an origin server's content from its old rescue servers. To handle this situation properly, a rescue server does not remove a mapping entry immediately after the rescue service has been terminated, but rather keeps the mapping entry for a configured time such as 24 hours, and redirects such a request back to the corresponding origin server via an HTTP redirect.

A rescue server works as a reverse caching proxy for its origin servers. For example, when *www.rescue.com* has a cache miss for *http://www-vh1.rescue.com/index.html,* it maps *www-vh1.rescue.com* to *www.origin.com,* and issues a reverse proxy request for *http://www.origin.com/index.html.* Using reverse caching proxy offers a few advantages. First, as files are replicated on demand, the origin server incurs low cost since it does not need to maintain states for replicated files and can avoid transferring files that are not requested at the rescue server. Second, as proxy and caching are functions supported by most web server software, it is simple to use reverse proxying to get needed files, and use the same caching mechanisms to cache proxied files and local files.

4.2 Request Redirection

Request redirection [8,6,30] allows an origin server to offload client requests to its rescue servers, which involves two aspects: the mechanisms to offload client requests and the policies to choose a rescue server among multiple choices. A client request can be redirected by the origin server's authoritative DNS, the origin server itself, or a redirector at transport layer (content-blind) or application layer (content-aware). Redirection policies can be based on load at rescue servers, locality of requested files at rescue servers, and proximity between the client and rescue servers.

DotSlash uses two mechanisms for request redirections: DNS round robin at the first level for crude load distribution, and HTTP redirect at the second level for fine-grained load balancing. DNS round robin can reduce the request arrival rate at the origin server, and HTTP redirect can increase the service rate of the origin server because an HTTP redirect is much cheaper to serve than the original content. Both mechanisms can increase the origin server's throughput for request handling.

We investigated three options for constructing redirect URIs: IP address, virtual directory, and virtual host name. Using the rescue server's IP address can save the client's DNS lookup time for the rescue server's name, but the rescue server is unable to tell whether a request is for itself or for one of its origin servers. Using a virtual directory such as */dotslash-vh, http://www.origin.com/index.html* can be redirected as *http://www.rescue.com/dotslash-vh/www.origin.com/index.html.* The problem is that it does not work for embedded relative URIs. DotSlash uses virtual host names, which allows proper virtual hosting at the rescue server, and works for embedded relative URIs.

In terms of redirection policies, DotSlash uses standard DNS round robin without modifying the DNS server software, and uses weighted round robin (WRR) for HTTP redirects, where the weight is the allowed redirect data rate assigned by each rescue server. Due to factors such as caching and embedded relative URIs, the redirect data rate seen by the origin server may be different from that served by the rescue server. For simplicity, an origin server only controls the data rate of redirected files, not including embedded objects such as images, and relies on a rate feedback from the rescue server to adjust its redirect data rate (see Section 4.4 for details).

Redirection needs to be avoided for communications between two collaborating servers and for requests of getting server status information. On one hand, a request sender (a web client or a web server) needs to bypass DNS round robin by using the server's IP address directly in the following cases: when a server initiates a rescue connection to another server, when a rescue server makes a reverse proxy request to its origin server, and when a client retrieves a server's status information. On the other hand, a request receiver (i.e., a web server) needs to avoid performing an HTTP redirect if the request is from a rescue server, or if the request is for the server's status information.

4.3 Workload Monitoring

Workload monitoring allows a web server to react quickly to load changes. Major DotSlash parameters are summarized in Table 1. We measure the utilization of each resource at a web server separately. According to a recent study [20], network bandwidth is the most constrained resource for most web sites during hotspots. We focus on monitoring network utilization ρ_n in this paper. We use two configurable parameters, lower threshold ρ_n^l and upper threshold ρ_n^u, to define three regions for ρ_n: lightly loaded region $[0, \rho_n^l)$, desired load region $[\rho_n^l, \rho_n^u]$, and heavily loaded region $(\rho_n^u, 100\%]$. Furthermore, we define a reference utilization $\hat{\rho_n}$ as $(\rho_n^l + \rho_n^u)/2$.

In DotSlash, we monitor outbound HTTP traffic within a web server, without relying on an external module to monitor traffic on the link. We assume there is no significant other traffic besides HTTP at a web server, and assume a web server has a symmetric link or its inbound bandwidth is greater than its outbound bandwidth, which is true, for example, for a web server behind DSL. Since a web server's outbound data rate

Table 1. Major DotSlash parameters, where type C is for configurable parameters, type O is for measured outputs, type I is for control inputs, and type D is for derived parameters

Parameter	Description	Type
ρ_n^l and ρ_n^u	lower and upper threshold for network utilization, default 50% and 75%	C
λ_d^m	maximum data rate (kB/s) for outbound HTTP traffic	C
τ	control interval, default 1 second	C
α	used in exponentially weighted moving average filter, default 0.5	C
λ_d	real data rate (kB/s) of outbound HTTP traffic	O
λ_{rd}	real redirect data rate (kB/s)	O
λ_{rd}^a	allowed redirect data rate (kB/s)	I
P_r	redirect probability	I
ρ_n	network utilization, $\rho_n = \lambda_d/\lambda_d^m$	D
$\hat{\rho_n}$	reference network utilization, $\hat{\rho_n} = (\rho_n^u + \rho_n^l)/2$	D
$\hat{\lambda_d}$	reference data rate (kB/s), $\hat{\lambda_d} = \hat{\rho_n}\lambda_d^m$	D
β	adjustment factor for control inputs, $\beta = \rho_n/\hat{\rho_n}$	D

is normally greater than its inbound data rate, it should be sufficient to only monitor outbound HTTP traffic.

Due to header overhead (such as TCP and IP headers) and retransmissions, the HTTP traffic rate monitored by DotSlash is less than the real traffic rate on the link. Since the header overhead is relatively constant and other overheads are usually small, to simplify calculation, we use a configurable parameter λ_d^m to denote the maximum data rate for outbound HTTP traffic, where $\lambda_d^m = BU$, B is the network bandwidth, and U is the percentage of bandwidth that is usable for HTTP traffic. We perform a special accounting for HTTP redirects because they may account for a large percentage of HTTP responses and their header overhead is large compared to their small sizes. For an HTTP redirect response of n bytes, its accounting size $A_r = (n + O)U$ bytes, where O is the header overhead. A web server sends five TCP packets for each HTTP redirect: one for establishing the TCP connection, one for acknowledging the HTTP request, one for sending the HTTP response, and two for terminating the TCP connection. The first TCP header (SYN ACK) is 40 bytes, and the rest four TCP headers are 32 bytes each. Thus, $O = (40 + 32 * 4) + 20 * 5 + (14 + 4) * 5 = 358$ bytes, which includes the TCP and IP headers, and the Ethernet headers and trailers.

4.4 Rescue Control

Rescue control allows a web server to tune its resource utilization by using rescue actions that are triggered automatically based on load conditions. To control ρ_n within the desired load region $[\rho_n^l, \rho_n^u]$, overload control actions are triggered if $\rho_n > \rho_n^u$, and under-load control actions are triggered if $\rho_n < \rho_n^l$. To control the utilization of multiple resources, overload control actions are triggered if *any* resource is heavily loaded, and under-load control actions are triggered if *all* resources are lightly loaded.

Origin servers and rescue servers use different control parameters. An origin server controls the redirect probability P_r by increasing P_r if $\rho_n > \rho_n^u$ and decreasing P_r if

$\rho_n < \rho^l_n$, whereas a rescue server controls the allowed redirect data rate λ^a_{rd} for each of its origin servers by decreasing λ^a_{rd} if $\rho_n > \rho^u_n$ and increasing λ^a_{rd} if $\rho_n < \rho^l_n$. An origin server should ensure the real redirect data rate $\lambda_{rd} \leq \lambda^a_{rd}$, but a rescue server may experience $\lambda_{rd} > \lambda^a_{rd}$.

We use the following control strategies. A configurable parameter τ denotes the control interval, which is the smallest time unit for performing workload monitoring and rescue control. Other time intervals are specified as a multiple of the control interval. To handle stochastics, we apply an exponentially weighted moving average filter to ρ_n, P_r and λ^a_{rd}. Using ρ_n as an example, $\overline{\rho_n(k)} = \alpha\overline{\rho_n(k-1)} + (1-\alpha)\rho_n(k)$, where $\rho_n(k)$ is the current raw measurement, $\overline{\rho_n(k)}$ is the filtered value of $\rho_n(k)$, $\overline{\rho_n(k-1)}$ is the previous filtered value, and α is a configurable parameter with a default value 0.5. If multiple rescue server candidates are available, the one with the largest rescue capacity should be used first. This policy can help an origin server to keep the number of its rescue servers as small as possible. Minimizing the number of rescue servers can reduce their cache misses, and thus reduce the data transfers at the origin server.

The DotSlash rescue protocol (DSRP) is an application-level request-response protocol using single-line pure text messages. A request has a command string (starting with a letter) followed by optional parameters, whereas a response has a response code (three digits) followed by the response string and optional parameters. DSRP defines three requests: *SOS* for initiating a rescue relationship, *RATE* for adjusting a redirect data rate, and *SHUTDOWN* for terminating a rescue relationship. An *SOS* request is always sent by an origin server, and a *RATE* request is always sent by a rescue server, but a *SHUTDOWN* request may be sent by an origin server or a rescue server. To initiate a rescue relationship, an origin server sends an *SOS* request to a chosen rescue server candidate. The request has the following parameters: the origin server's fully qualified domain name, its IP address, and its port number for web requests. When a web server receives an *SOS* request, it can accept the request by sending a "200 OK" response or reject the request by sending a "403 Reject" response. A "200 OK" response has the following parameters: a unique alias of the rescue server assigned to the origin server, the rescue server's IP address, the rescue server's port number for web requests, and the allowed redirect data rate that the origin server can offload to the rescue server.

Fig. 3 summarizes DotSlash rescue actions and state transitions. We describe rescue actions in each state next. The normal state has two rescue actions: initial allocation and initial rescue. For the first case, if a web server is heavily loaded (i.e., $\rho_n > \rho^u_n$), then it needs to allocate its first rescue server, set P_r to 0.5, and switch to the SOS state. For the second case, if a web server receives a rescue request and it is lightly loaded (i.e., $\rho_n < \rho^l_n$), then it can accept the rescue request, set λ^a_{rd} to $(\hat{\rho_n} - \rho_n)\lambda^m_d$ or a smaller value determined by a rate allocation policy, and switch to the rescue state.

The SOS state has four rescue actions: increase P_r, additional allocation, decrease P_r, and release. For the first case, if an origin server is heavily loaded and it has unused redirect capacity (i.e., $\lambda_{rd} < \lambda^a_{rd}$), then it needs to increase P_r until P_r reaches 1. For the second case, if an origin server is heavily loaded and it has run out of redirect capacity (i.e., λ_{rd} equals λ^a_{rd}), then it needs to allocate an additional rescue server so as to increase its redirect capacity. For the third case, if an origin server is lightly loaded and it still redirects requests to rescue servers (i.e., $P_r > 0$), then it needs to decrease

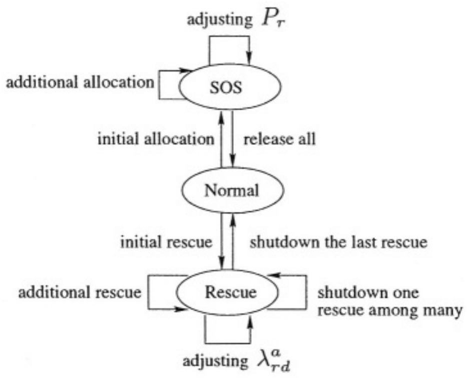


Fig. 3. DotSlash rescue actions and state transitions

P_r until P_r reaches 0. For the last case, if an origin server has been lightly loaded and has not redirected requests to rescue servers (i.e., P_r is 0) for a configured number of consecutive control intervals, then it needs to release all rescue servers. Fig. 4 gives the algorithm for adjusting P_r at an origin server, which increases P_r if $\rho_n > \rho^u_n$, and decreases P_r if $\rho_n < \rho^l_n$. The adjustment is controlled by parameter $\beta = \rho_n/\hat{\rho_n}$, where $\beta > 1$ for increase since $\rho_n > \rho^u_n > \hat{\rho_n}$, and $\beta < 1$ for decrease since $\rho_n < \rho^l_n < \hat{\rho_n}$. Further, the adjustment is smoothed by using an exponentially weighted moving average filter with $\alpha = 0.5$. To avoid infinite convergence, an increase from above 0.99 is set to 1, and a decrease from below 0.1 is set to 0. To react quickly to load spikes, an increase from below 0.5 is set to 0.5.

The rescue state has five rescue actions: decrease λ^a_{rd}, heavy-load shutdown, increase λ^a_{rd}, additional rescue, and idle shutdown. For the first case, if a rescue server is heavily loaded and its $\lambda^a_{rd} > 0$, then it needs to decrease λ^a_{rd} until λ^a_{rd} reaches 0. For the second case, if a rescue server is heavily loaded and its λ^a_{rd} is 0, then it needs to shutdown the rescue relationship. When a rescue server has shutdown all rescue relationships, it switches to the normal state. For the third case, when a rescue server is lightly loaded and $\lambda^a_{rd} < \hat{\lambda}_d$, then it can increase λ^a_{rd}. Note that a rescue server should not increase λ^a_{rd} if λ_{rd} is far below λ^a_{rd}. For the fourth case, if a rescue server is lightly loaded, and it receives a new rescue request, then it can accept the rescue request, and assign a λ^a_{rd} to the new origin server. By doing so, the rescue server will have multiple origin servers, and a separate λ^a_{rd} is assigned to each origin server. For the last case, if a rescue server has an origin server whose λ_{rd} has been 0 for a configured number of consecutive control intervals, then the rescue server should shutdown the rescue relationship so as to release rescue resources in case of the origin server failure or network separation. Fig. 5 gives the algorithm for adjusting λ^a_{rd} at a rescue server, which decreases λ^a_{rd} if $\rho_n > \rho^u_n$, and increases λ^a_{rd} if $\rho_n < \rho^l_n$. This algorithm is very similar to the algorithm shown in Fig. 4. However, these two algorithms make adjustments in opposite directions because their adjusting factors are β and $1/\beta$, respectively. We keep the adjusting factor for λ^a_{rd} within the range of [0.5, 2] to avoid over-reacting adjustments. Also, we keep the filtered value of λ^a_{rd} as an integer.

```
// Compute $\beta$
$\beta = \rho_n/\hat{\rho_n}$;
```

```
// Increase $P_r$ if $\rho_n > \rho_n^u$
if ($P_r < 1$) {
  if ($P_r < 0.5$) {
    $P_r = 0.5$;
  } else if ($P_r > 0.99$) {
    $P_r = 1$;
  } else {
    $t = \min(\beta P_r, 1)$;
    $P_r = \alpha P_r + (1 - \alpha)t$;
  }
}
```

```
// Decrease $P_r$ if $\rho_n < \rho_n^l$
if ($P_r > 0$) {
  if ($P_r < 0.1$) {
    $P_r = 0$;
  } else {
    $t = \beta P_r$;
    $P_r = \alpha P_r + (1 - \alpha)t$;
  }
}
```

Fig. 4. Algorithm for adjusting P_r at an origin server

```
// Compute $\beta$
$\beta = \rho_n/\hat{\rho_n}$;
if ($\beta < 0.5$) {
  $\beta = 0.5$;
} else if ($\beta > 2$) {
  $\beta = 2$;
}
```

```
// Decrease $\lambda_{rd}^a$ if $\rho_n > \rho_n^u$
if ($\lambda_{rd}^a > 0$) {
  $t = \lambda_{rd}^a/\beta$;
  $\lambda_{rd}^a = (\text{int})(\alpha\lambda_{rd}^a + (1 - \alpha)t)$;
}
```

```
// Increase $\lambda_{rd}^a$ if $\rho_n < \rho_n^l$
if ($\lambda_{rd}^a < \hat{\lambda_d}$) {
  $t = \min(\lambda_{rd}^a/\beta, \hat{\lambda_d})$;
  $\lambda_{rd}^a = (\text{int})(\alpha\lambda_{rd}^a + (1 - \alpha)t)$;
}
```

Fig. 5. Algorithm for adjusting λ_{rd}^a at a rescue server

4.5 Service Discovery

Service discovery allows servers of different web sites to learn about each other dynamically and collaborate without any administrator intervention. DotSlash uses the Service Location Protocol (SLP) [13] since it is an IETF proposed standard for service discovery in IP networks, and it is flexible, lightweight and powerful. Based on the SLP mesh enhancement (mSLP [33]), DotSlash uses multiple well-known service registries that maintain a fully-meshed peer relationship. A web server can use any service registry to register its information and to search information about other web servers. Service registrations received by one registry will be propagated to other registries as soon as possible, and anti-entropy [32] is used to ensure consistency among all service registries. Only a small number of such service registries are needed for reliability and scalability. All of them serve the scope "DotSlash" (reserved for DotSlash rescue services) so that they will not affect local service discovery.

The template for DotSlash rescue services has the following attributes: the domain name for the web server, its IP address which is used to bypass DNS round robin, its port number for web requests, its port number for DotSlash rescue services, and the current allowed redirect data rate λ_{rd}^a computed as $\max((\hat{\rho_n} - \rho_n)\lambda_d^m, 0)$. A web server performs service registrations and searches periodically with a configurable interval τ_r and τ_s, respectively. To get ready for load spikes, a web server maintains a list of rescue server candidates. A DotSlash service search request uses preference filters [34] that allow the registry to sort the search result based on λ_{rd}^a and to only return the desired number of matching entries, which is useful if many entries match a search request.

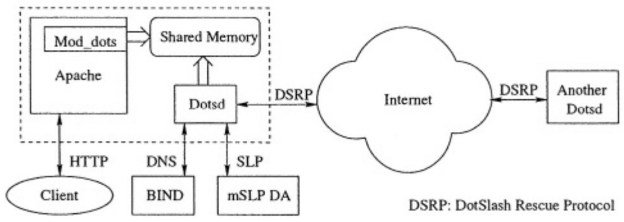


Fig. 6. DotSlash software architecture

5 Implementation

We use Apache [4] as our base system since it is open source and is the most popular web server [19]. Fig. 6 shows the DotSlash software architecture. DotSlash is implemented as two parts: *Mod_dots* and *Dotsd.* Mod_dots is an Apache module that supports DotSlash functions related to client request processing, including accounting for each response, HTTP redirect, and dynamic virtual hosting. Dotsd is a daemon that accomplishes other DotSlash functions, including service discovery, dynamic DNS updates, and rescue control and management. For convenience, Dotsd is started within the Apache server, and is shutdown when the Apache server is shutdown. Dotsd and Mod_dots share control data structures via shared memory. DNS servers and DotSlash service registries are DotSlash components external to the Apache server. We use BIND as DNS servers, and use mSLP Directory Agents (DAs) as DotSlash service registries. A web server interacts with other web servers via its Dotsd using DSRP carried by TCP.

The control data in shared memory are divided into two parts: a workload meter for the web server itself, and a peer table for collaborating web servers. The peer table maintains accounting information of redirected traffic for peers. Traffic accounting is performed in two time scales: the current control interval and the server's lifetime (from the server's starting time to now). The former accounting is used to trigger rescue actions, and the corresponding counters are reset to zero at the end of the current control interval. The latter accounting allows computing various average traffic rates by sampling the corresponding counters at desired time intervals.

Dotsd is implemented using pthread. It has two main threads: a control thread and a a DSRP server. The control thread runs at the end of each control interval for processing tasks that need to be done periodically such as computing the current workload level, triggering rescue actions if needed, and checking whether it needs to perform service discovery. The DSRP server accepts connections from other Dotsds and creates a new thread for processing each accepted connection. Dotsd also includes three clients: a DNS client for dynamic DNS updates, an SLP Service Agent for service registrations, and an SLP User Agent for service searches.

Mod_dots handles traffic accounting, performs HTTP redirects for an origin server, supports dynamic virtual hosting for a rescue server, and implements a content handler for */dotslash-status* so that a request for *http://host.domain/dotslash-status* can retrieve the current DotSlash status for the web server *host.domain.*

6 Evaluation

For a web server, we use two performance metrics *D* and *R,* where *D* is the maximum data rate of HTTP responses delivered to clients, and *R* is the maximum request rate supported. Our goal is to improve a web server's *D* and *R* by using DotSlash rescue services. For a web server without using DotSlash, its *D* and *R* can be estimated as λ_d^m and $\lambda_d^m/(F+H)$, respectively, where *F* is the average size of requested files, and H is the average HTTP header size of responses, assuming the CPU is not a bottleneck. For any web server, the maximum rate of HTTP redirects it can support can be estimated as λ_d^m/A_r, where A_r is the accounting size for an HTTP redirect. Thus, a web server can improve its *R* and *D* by using DotSlash as follows. If it only uses HTTP redirect to offload client requests, its *R* is bounded by λ_d^m/A_r, and its *D* is bounded by *R(F + H).* However, a web server can use DNS round robin to overcome this scaling limitation so as to further improve its *R* and *D.*

We performed experiments in our local area networks (LANs) and on PlanetLab [21] for two goals. First, given a web server with a constraint on its outbound bandwidth, we want to improve its *R* and *D* by using DotSlash rescue services, and aim to achieve an improvement close to the analytical bound, i.e., the web server can handle a request rate close to λ_d^m/A_r when only HTTP redirect is used. Second, we want to confirm that our workload control algorithm works as expected.

6.1 Workload Generation

We use *httperf* [18] to generate workloads. If the request rate to be generated is high, multiple httperf clients are used, each running on a separate machine. To simulate web hotspots, a small number of files are requested repeatedly from a web server. Each request uses a separate TCP connection. Thus, the request rate equals the connection rate. We made two enhancements to httperf to facilitate experiments on DotSlash. First, we extended httperf to handle HTTP redirects automatically since an httperf client needs to follow HTTP redirects in order to complete workload migrations from an origin server to its rescue servers. Second, we wrote a shell script to support workload profiles. A workload profile specifies a sequence of request rates and their testing durations, which is convenient for describing workload changes.

For a web server, its *R* and *D* are determined as follows. We use httperf clients to issue requests to the web server, starting at a low request rate, and increasing the request rate gradually until the web server gets overloaded. A client uses 7 seconds [9] as the timeout value for getting each response. If more than 10% [9] of issued requests time out, a client declares the web server as being overloaded. For a sequence of testing request rates that are monotonically increasing, $r_1 < r_2 < \cdots$, if the web server gets overloaded at r_i, then $R = r_{i-1}$. For all testing request rates, up to *R,* the maximum data rate delivered to clients is *D.*

6.2 Experimental Setup

In our LANs, we use a cluster of 30 Linux machines, which are connected using 100 Mb/s fast Ethernet. These machines have two different configurations, *CLIC* and *iDot.*

The former has a 1 GHz Intel Pentium III CPU, and 512 MB of memory, whereas the latter has a 2 GHz AMD Athlon XP CPU, and 1 GB of memory. They all run Redhat 9.0, with Linux kernel 2.4.20-20.9. PlanetLab consists of more than 300 nodes distributed all over the world, each with a CPU of at least 1 GHz clock rate, and has at least 1 GB of memory. PlanetLab nodes have four types of network connections: DSL lines, Internet2, North America commodity Internet, and outside North America. They all run Redhat 9.0, with Linux kernel 2.4.22-r3_planetlab, and PlanetLab software 2.0.

We set up the DotSlash software in three steps. First, we compile Apache 2.0.48 with the *worker* multi-processing module, the proxy modules, the cache modules, and our DotSlash module. We configure Apache as follows. Since reverse proxying is taken care of by DotSlash automatically, no proxy configuration is needed. Caching is configured with 256 KB of memory cache, and 10 MB of disk cache, and the maximum file size allowed in memory cache is 20 kB. For the DotSlash module, we only configure λ_d^m. Second, we use BIND 9.2.2 as the DNS server software, and set up a DNS domain *dot-slash.net.* All rescue servers register their virtual host names in this domain via dynamic DNS updates. Currently, we have tested DotSlash workload migrations via HTTP redirect, without using DNS round robin. Third, we set up a DotSlash service registry using an mSLP DA. Each web server registers itself with this well-known service registry, and discovers other web servers by looking up this registry.

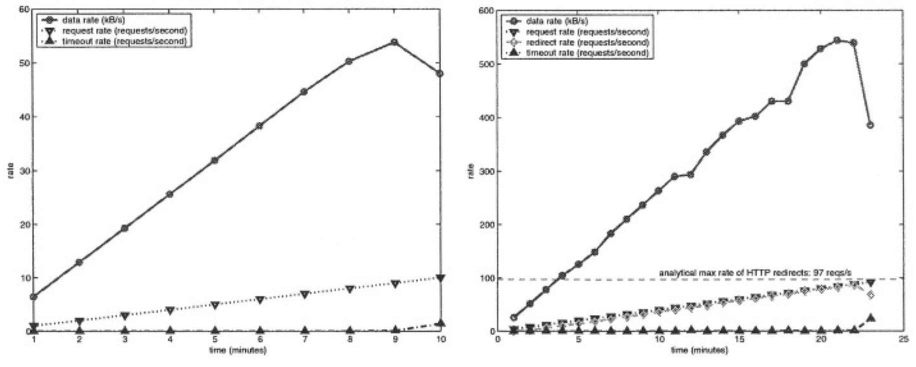
6.3 Experimental Results on PlanetLab

We run a web server on a PlanetLab DSL node, *planetlab1.gti-dsl.nodes.planet-lab.org* (referred to as *gtidsl1*), for which the outbound bandwidth is the bottleneck. We run httperf on a local CLIC machine. Ten files are requested repeatedly from *gtidsl1,* with an average size of 6 KB [30]. Our goal is to measure, from the client side, *gtidsl1*'s R and D in two cases, namely without using DotSlash versus using DotSlash.

For the first case, DotSlash is disabled. The request rate starts at 1 request/second, increases to 20 requests/second, with a step size of 1, and each request rate lasts for 60 seconds. Fig. 7(a) shows the experimental results. In this figure, *gtidsl1* gets overloaded at 10 requests/second, where 14% of requests, 84 out of 600, time out. Thus, R is 9 requests/second. The measured D is 53.9 kB/s, attained when the request rate is R.

For the second case, DotSlash is enabled. We set *gtidsl1*'s λ_d^m to 53.9 kB/s. To provide needed rescue capacity for *gtidsl1,* we run another web server on a local *iDot* machine (named *maglev*), and its λ_d^m is set to 2000 kB/s. The request rate starts at 4 requests/second, increases to 200 requests/second, with a step of 4, and each request rate lasts for 60 seconds. Fig. 7(b) shows the experimental results. In this figure, when the request rate reaches 8 requests/second, the origin server *gtidsl1* starts to redirect client requests via HTTP redirects to the rescue server *maglev.* As the request rate increases, the redirect rate increases accordingly. Eventually, *gtidsl1* redirects almost all clients requests to *maglev.* In this experiment, *gtidsl1* gets overloaded at 92 requests/second, where 25% of requests, 1404 out of 5520, time out. Thus, R is 88 requests/second. The measured D is 544.1 kB/s, attained when the request rate is 84 requests/second.

Comparing the results obtained from the above two cases, we have 88/9 = 9.78, and 544.1/53.9 = 10.1, meaning that by using DotSlash rescue services, we got about an order of magnitude improvement for *gtidsl1* on its R and D, even if only HTTP redirect is



(a) Without using DotSlash rescue services (b) Using DotSlash rescue services

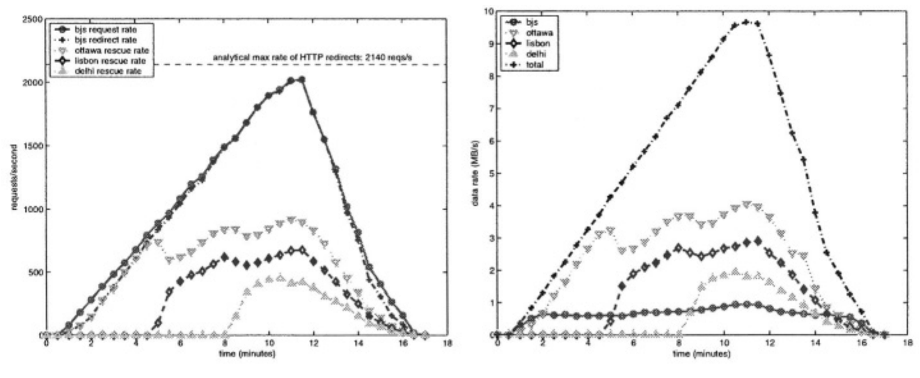
Fig. 7. The data rate and request rate for a PlanetLab DSL node *gtidsl1* in two cases, note different scales of ordinates

used. To show the effectiveness of DotSlash, we also compare *R* with its analytical bound λ_d^m/A_r below. In this experiment, we only measured λ_d^m at *gtidsl1*, without knowing its outbound bandwidth *B*. To be conservative, we use $U = (F{+}H)/(F{+}H{+}O) = 95\%$, where $F = 6$ KB, $H = 250$ bytes, and $O = 358$ bytes. Here the header overhead *O* for a single-request HTTP transaction is the same as that for an HTTP redirect (calculated in Section 4.3). Since the size of an HTTP redirect response is $n = 227$ bytes in the experiment, we have $A_r = (n + O)U = 556$ bytes. As a result, *R* is bounded by $\lambda_d^m/A_r = 53.9 * 1000/556 = 97$ requests/second, and we achieved 88/97 = 91% of its analytical bound.

6.4 Experimental Results in LANs

In the previous section we have shown the performance improvement, measured from the client side, for a web server by using DotSlash rescue services in a wide area network setting. In this section we will show, via an inside look from the server side, how workload is migrated from an origin server to its rescue servers. The workload monitoring component in DotSlash maintains a number of counters for outbound HTTP traffic, including total bytes served, the number of client requests served, the number of client requests redirected, and the number of requests served for rescuing others. The values of these counters for a web server *host.domain* can be obtained from *http://host.domain/dotslash-status ?auto*. By sampling these counters at a desired interval, we can calculate the needed average values of outbound data rate, request rate, redirect rate, and rescue rate.

In this experiment, four machines, *bjs, Ottawa, lisbon,* and *delhi,* run as web servers, where *bjs* is an *iDot* machine, and the other three are *CLIC* machines. To emulate a scenario where *bjs* works as an origin server with a bottleneck on its outbound bandwidth, and the rest web servers work as rescue servers, we configured their λ_d^m as 1000, 7000, 5000, and 3000 kB/s, respectively. We run httperf on five *CLIC* machines, which issue requests to *bjs* using the same workload profile. The maximum request rate is $400 * 5 =$



(a) The request rate and redirect rate at *bjs*, and the rescue rates at its rescue servers

(b) The data rate at each web server, and the total data rate of all web servers

Fig. 8. The request rates and data rates at the origin server *bjs* and its rescue servers

2000 requests/second, and the duration of the experiment is 15 minutes. Ten files are requested repeatedly, with an average size of 4 KB. We run a shell script to get the DotSlash status from the four web servers at an interval of 30 seconds. The retrieved status data are stored in round-robin databases using RRDtool [22], with one database for each web server. Fig. 8 shows the data rates and request rates for the four web servers in a duration of 17 minutes.

We observe the following results from Fig. 8(a). First, *bjs* can support a request rate of 2000 requests/second, which is close to λ_d^m/A_r, the analytical maximum rate of HTTP redirects at *bjs*. Since $A_r = (n + O)U = 468$ bytes in this experiment, where $n = 227$ bytes, $O = 358$ bytes, and U takes its default value 80%, we have $\lambda_d^m/A_r = 2140$ requests/second. Second, the redirect rate at *bjs* increases as the request rate increases, and it is roughly the same as the request rate once it is above 1500 requests/second. The reason is that *bjs* increases redirect probability P_r as its load increases. When the rate of HTTP redirects is greater than $\lambda_d^m \rho_n^u/A_r = 1603$ requests/second, P_r will stay at 1, that is all client requests are redirected from *bjs* to its rescue servers. Third, *bjs* allocates one rescue server at a time, and uses the one with the largest rescue capacity first. When a new rescue server is added in, the rescue rates at the existing rescue servers decrease. Also, the rescue rates at rescue servers are proportional to their rescue capacities because of the WRR at *bjs*.

Comparing Fig. 8(b) and 8(a), we observe that rescue servers have similarly shaped curves for their data rates and rescue rates. In contrast, as we expected, the origin server *bjs* have quite different shapes for its the request rate and data rate curves: its the request rate increases significantly from 200 requests/second at 1.5 minutes to 2000 requests/second at 11 minutes, but its data rate is roughly unchanged, staying at $\lambda_d^m \rho_n^u = 750$ kB/s for the most part. This indicates that *bjs* has successfully migrated its workload to its rescue servers under the constraint of its outbound bandwidth. Also, we observe that when the request rate is between 1600 and 2000 requests/second, the data rate at *bjs* goes above 750 kB/s, but still stays below $\lambda_d^m = 1000$ kB/s. This is because *bjs* can only support a rate of 1600 requests/second for HTTP redirects with a data rate of 750 kB/s. Furthermore, we observe that the total data rate of all web servers has a maximum value of 9.7 MB/s, which is higher than 9.2 MB/s, the maximum data rate measured from the httperf clients. The difference is due to our special accounting for HTTP redirects. As described in Section 4.3, an HTTP redirect is 227 bytes, but is counted as 468 bytes, which results in a rate increase of $241 * 2000 = 0.482$ MB/s for 2000 HTTP redirects.

7 Conclusion

We have described the design, implementation, and evaluation of DotSlash in this paper. As a rescue system, DotSlash complements the existing web server infrastructure to handle web hotspots effectively. It is self-configuring, scalable, cost-effective, easy to use, and transparent to clients. Through our preliminary experimental results, we have demonstrated the advantages of using DotSlash, where a web server increases the request rate it supported and the data rate it delivered to clients by an order of magnitude, even if only HTTP redirect is used.

We plan to perform trace-driven experiments on DotSlash by using log files from web hotspot events, and incorporate DNS round robin in the performance evaluation. Also, we plan to investigate load migration for dynamic content, which will extend the reach of DotSlash to more web sites. Our prototype implementation of DotSlash will be released as open source software.

References

1. T. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service (IWQoS),* London, England, June 1999.
2. Stephen Adler. The slashdot effect: An analysis of three Internet publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.
3. Akamai homepage. http://www.akamai.com/.
4. Apache. HTTP server project. http://httpd.apache.org/.
5. M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable context-aware request distribution in cluster-based network servers. In *Annual Usenix Technical Conference,* San Diego, California, June 2000.
6. A. Barbir, Brad Cain, Raj Nair, and Oliver Spatscheck. Known content network (CN) request-routing mechanisms. RFC 3568, Internet Engineering Task Force, July 2003.
7. BitTorrent homepage. http://bitconjurer.org/BitTorrent/.
8. V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys,* 34(2):263–311, June 2002.
9. V. Cardellini, M. Colajanni, and P.S. Yu. Geographic load balancing for scalable distributed web systems. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS),* San Francisco, California, August 2000.
10. E. Coffman, P. Jelenkovic, J. Nieh, and D. Rubenstein. The Columbia hotspot rescue service: A research plan. Technical Report EE2002-05-131, Department of Electrical Engineering, Columbia University, May 2002.

11. Mark Day, Brad Cain, G. Tomlinson, and P. Rzewski. A model for content internetworking (CDI). RFC 3466, Internet Engineering Task Force, February 2003.
12. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications,* 15(3), 2001.
13. E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.
14. J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *International World Wide Web Conference (WWW),* Honolulu, Hawaii, May 2002.
15. K. Kong and D. Ghosal. Mitigating server-side congestion in the Internet through pseudoserving. *IEEE/ACM Transactions on Networking,* 7(4):530–544, August 1999.
16. W. LeFebvre. CNN.com: Facing a world crisis. Invited talk at USENIX LISA, December 2001.
17. Q. Li and B. Moon. Distributed cooperative Apache web server. In *International World Wide Web Conference,* Hong Kong, May 2001.
18. D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. In *Workshop on Internet Server Performance (WISP),* Madison, Wisconsin, June 1998.
19. Netcraft. Web server survey. http://news.netcraft.com/archives/web_server_survey.html.
20. V. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *International Workshop on Peer-to-Peer Systems (IPTPS),* Cambridge, Massachusetts, March 2002.
21. PlanetLab. http://www.planet-lab.org/.
22. RRDtool homepage. http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/.
23. P. Rzewski, Mark Day, and D. Gilletti. Content internetworking (CDI) scenarios. RFC 3570, Internet Engineering Task Force, July 2003.
24. Stan Schwarz. Web servers, earthquakes, and the slashdot effect. http://pasadena.wr.usgs.gov/office/stans/slashdot.html.
25. T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *International Workshop on Peer-to-Peer Systems (IPTPS),* Cambridge, Massachusetts, March 2002.
26. A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust P2P system to handle flash crowds. In *IEEE International Conference on Network Protocols (ICNP),* Paris, France, November 2002.
27. A. Vakali and G. Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing,* 7(6):68–74, December 2003.
28. Paul Vixie, Sue Thomson, Y. Rekhter, and Jim Bound. Dynamic updates in the domain name system (DNS UPDATE). RFC 2136, Internet Engineering Task Force, April 1997.
29. Jia Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Review (CCR),* 29(5), October 1999.
30. L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Symposium on Operating Systems Design and Implementation (OSDI),* Boston, Massachusetts, December 2002.
31. M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In *USENIX Conference on Internet Technologies and Systems (USITS),* Seattle, Washington, March 2003.
32. W. Zhao and H. Schulzrinne. Selective anti-entropy. In *ACM Symposium on Principles of Distributed Computing,* Monterey, California, July 2002.
33. W. Zhao, H. Schulzrinne, and E. Guttman. Mesh-enhanced service location protocol (mSLP). RFC 3528, Internet Engineering Task Force, April 2003.
34. W. Zhao, H. Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. Select and sort extensions for the service location protocol (SLP). RFC 3421, Internet Engineering Task Force, November 2002.

Dynamic Content Placement for Mobile Content Distribution Networks

Wagner M. Aioffi, Geraldo R. Mateus, Jussara M. Almeida, and Raquel C. Melo

Department of Computer Science
Federal University of Minas Gerais - Brazil
{aioffi,mateus,jussara,raquelcm}@dcc.ufmg.br

Abstract. As wireless networks increase in popularity, the development of efficient content distribution techniques to meet the growing and constantly changing client demand becomes a necessity. Previous content distribution network proposals, targeting mainly wired networks, are not adequate to handle the high temporal and spatial variability in client demand caused by user mobility. This paper proposes and analyzes a wireless dynamic content distribution network framework that replicates content in response to changes in client demand in order to reduce the total traffic over the network backbone. We propose a mathematical programming model to determine the offline optimal solution as well as an online algorithm based on demand forecasting. Using a previously developed mobility simulator, we show our new online algorithm outperforms the traditional centralized and distributed static placement approaches, reducing the total traffic over the network backbone in up to 78%, compared to the best previous approach analyzed.

Keywords: Dynamic content placement, mobile networks, demand forecasting.

1 Introduction

As the number of wireless network users increases and with the advent of high bandwidth third-generation mobile networks, it becomes necessary to develop new system design techniques to meet the increasing demand. Next generation wireless telecommunication systems should be designed as a high-capacity system, able to cope with the envisaged overwhelming traffic volume.

A Content Distribution Network, or simply CDN, replicates popular content at a number of servers, placed closer to high demand sites, in order to reduce network bandwidth requirements and latency observed by the end user. CDNs have been shown to be effective for managing content distribution to a large number of users in wired networks. In those cases, the assumption of a static client demand distribution is usually made [3,13,7,10,15].

In wireless networks, client demand is expected to vary significantly not only over time but also over space, due to user mobility. For example, a city downtown area may have a higher demand for business content during working hours. However, one might expect the demand to drop significantly as people return to their homes after work. Thus, the design of a content distribution system for wireless networks must rely on dynamic replica placement strategies in order to cope with the possibly high variability of client demand.

To the best of our knowledge, content replication for variable client demand has only been studied recently. In [3,5], the authors address the problem for specific network topologies. A heuristic for dynamic replication of Internet applications, in more general topologies was recently proposed in [14]. Optimization models targeting system availability [18] and for providing lower bounds on the cost-effectiveness of different replica placement heuristics [8] have also been proposed. Nevertheless, these studies propose and evaluate solutions for wired networks.

This work proposes and analyzes a Wireless Dynamic Content Distribution Network Model (WDCDNM). In WDCDNM, the CDN should be dynamically reconfigured in response to, possibly high, temporal and spatial variations in client demand with the goal of reducing total traffic over the network backbone. Reconfiguring the CDN implies that new replicas of currently popular objects should be added to the network. Similarly, previously created replicas of objects that have just become unpopular should be removed from the network.

We formulate a mathematical programming model, based on well-known combinatorial optimization models, which represents the offline optimal version of the problem. We also propose an online heuristic algorithm, which uses demand forecasting to reconfigure the network for future demand, to approximate the optimal solution given by the mathematical model. Since the offline model requires *a priori* knowledge of all future demand, it cannot be implemented in a real system. However, it offers an ideal lower-bound that can be used to analyze the competitive efficiency of the online algorithm [1].

An extensive performance analysis of our online algorithm, using a previously developed mobility simulator [11], showed that it produces solutions with a total network traffic that is only up to four times the offline optimal. Furthermore, compared to existing centralized and distributed static content placement approaches, our heuristic saves up to 78% of network traffic over the best static approach analyzed.

This paper makes the following contributions:

- An offline optimization model that provides a lower bound for the total traffic over the network backbone for Wireless Content Distribution Networks. The offline model can be implement total and partial content replication.
- A novel and efficient online algorithm for dynamic content placement in wireless networks that reduces network traffic in up to 78%, compared to static content placement.

The remaining of this paper is organized as follows. Section 2 discusses related work on CDNs. Section 3 describes the mathematical programming model and the online heuristic algorithm. Section 4 reports a performance evaluation of our solutions. Conclusions and future work are offered in Section 5.

2 Content Distribution Networks

A Content Distribution Network (CDN) is a system that allocates replicas of its contents over servers geographically scattered, placed, typically, close to high demand sites in order to reduce client latency and network bandwidth requirements. Content placement is a key problem in the design of efficient CDNs. Determining the number and location of replicas that should be created for a certain content is a non-trivial problem that depends on the solution of a complex tradeoff. Associated with each new replica is a replication cost which is proportional to the bandwidth required for transferring a copy of the content from the closest server where it is currently stored to the new replica server. There is also an extra cost associated with maintaining the local replica up-to-date. On the other hand, a new replica contributes to reducing the bandwidth required for serving client requests and, thus, reducing client latency.

The content placement problem, previously shown to be NP-Complete [4], has been studied mostly for static client demands. Optimal solutions for specific topologies [9,10] and heuristics for more general topologies [6,13,15] have been proposed. Content placement for dynamic client demand has been studied only very recently [3,5,8,18,14]. In [3], the authors address the problem with the goal of minimizing the number of replicas deployed in a tree topology while meeting client QoS and server capacity constraints. Dynamic content placement heuristics for replicating content inside a cluster-based cache are proposed in [5]. In [8], the authors propose an offline integer programming model for choosing the most cost-effective heuristic for a given system, workload and performance goal. By using specific constraints, the proposed model offers lower bounds for the cost associated with a number of different heuristics. A replication cost optimization models targeting system availability is proposed and formalized in [18]. Finally, Rabinovich *e*t al [14] study the problem in the context of Internet application replication in general topologies. They propose a heuristic, called ACDN, for dynamic replica placement based on the observed past demand that aims at minimizing the network bandwidth requirements. Like the solutions proposed in [3,5,8,18], the ACDN algorithm was proposed and evaluated for wired networks.

CDN design solutions targeting, specifically, wireless networks receive little attention so far. Previous work addresses mainly the server selection problem [16,17]. In [16] the authors address the server selection problem in a streaming media mobile CDN with servers arranged hierarchically, with the goal of reducing the number of mobile client handoffs. In [17], the authors propose a mobile streaming media CDN controlled by SMIL (Synchronized Multimedia Integrated Language) with the goals of improving streaming media quality, supporting client mobility and efficiently utilizing network resources.

This paper addresses the content placement problem in the specific context of wireless networks, where client demand may vary significantly over time and space. It proposes and evaluates not only an online dynamic content placement heuristic but also an offline optimal solution. Unlike the optimization models proposed in [8] and in [18], which target latency and system availability in wired networks, respectively, our offline optimal solution aims at minimizing total traffic over the backbone of wireless content distribution networks.

3 Wireless Dynamic Content Distribution Networks

This section defines our Wireless Dynamic Content Distribution Network Model (WDCDNM), a framework for managing content replication and placement in a wireless network aiming at reducing the total traffic in the network backbone. It also presents the mathematical programming model, which provides an offline optimal solution, and the online heuristic algorithm.

WDCDNM uses the general term "content" to represent a collection of objects (for instance, the collection of objects of a given site). Each content is stored in at least one server in the system, which is referred to as its origin server. Mobile users, or simply clients, request individual objects of a content. A request to a certain object is always sent to the *currently* closest server, due to server wireless coverage limitation. This server may or may not have a replica of the requested content. If it has, the request is served locally, incurring no traffic over the network backbone. Otherwise, it forwards the request to the closest server that has the content replica, and relays the response to the client. In this case, the indirect request service generates traffic over the network backbone between the two servers involved in the operation. Similarly, replication and replica maintenance operations also generate traffic over the backbone. A maintenance operation occurs whenever a content is modified in its origin server.

WDCDNM characterizes each content c by three size parameters: number of bytes transmitted during a replication (sr^c), number of bytes transmitted when a request for an object of the content is indirectly served (sm^c), and number of bytes transmitted during a maintenance operation. Note that WDCDNM assumes all objects within the same content have equal sizes, although different contents may have different total sizes. Extensions for allowing heterogeneous object sizes within the same content are straightforward. Partial and total content replication can be implemented in the WDCDNM framework by simply setting sr^c and si^c appropriately. In case of partial replication, indirect request service and replication are performed on a per-object basis and, thus, for a given content c, $sr^c = si^c$. Otherwise, replication is performed for the whole content at once. Thus, $sr^c \geq si^c$. The value assigned to sm^c depends on whether a maintenance operation is performed by transmitting a new copy of the content or simply "patches" with the changes to be performed on the old replica.

In response to the current spatial distribution of client demand (i.e., number of client requests per unit of time) for objects of different contents, WDCDNM performs content replication and/or removal of previously created replicas in the servers, with the goal of minimizing the total traffic over the network backbone. This traffic is calculated as the sum of the total traffic generated by partial/total content replication operations, the total traffic generated by replica maintenance operations and the total traffic generated by indirected responses to client requests. Each traffic component is estimated as the product of the number of bytes transferred in each operation (given by sr^c, si^c and sm^c), the "distance" (i.e., number of hops, geographical distance, etc) between the two servers involved in the operation and the number of operations performed.

The decision of adding a new content replica to a server or removing an existing one is based on the local demand for the content, and should be revisited periodically to absorb spatial and temporal changes in client demand. The following sections describe our two implementations of the WDCDNM framework: the offline optimal solution and the online (heuristic) algorithm.

3.1 Offline Optimal Solution

The offline optimal solution of WDCDNM is formulated as a mathematical programming model. The model is based on the well-known Uncapacitated Location Problem [12], a classical NP-Complete combinatorial optimization problem, extended to represent the spatial and temporal variations in client demand caused by user mobility. The following notation is used in the model formulation:

C: Set of contents available in the network;
T: Set of reconfiguration periods;
S: Set of servers;
$dist_{i,j}$: Distance between servers $i \in S$ and $j \in S$;
o^c: Origin server of content $c \in C$;
d_i^{tc}: Total demand, expressed as the number of requests sent by local clients to server $i \in S$ for objects of content $c \in C$ during period $t \in T$;
m^{tc}: Flag that indicates whether content $c \in C$ is modified at its origin server during period $t \in T$;
bi_{ij}^c: Traffic generated over the backbone if server $j \in S$ indirectly serves a request, originally sent to server $i \in S$, to an object of content $c \in C$;
br_{ij}^c: Traffic generated over the backbone if server $i \in S$ (totally or partially) replicates content $c \in C$ from server $j \in S$;
bm_i^c: Traffic generated over the backbone if server $i \in S$ has to update a local replica of content $c \in C$;
y_{ij}^{tc}: Binary variable that indicates whether server $i \in S$ replicates content $c \in C$ from server $j \in S$ during period $t \in T$;
a_i^{tc}: Binary variable that indicates whether server $i \in S$ has the content $c \in C$ during period $t \in T$;
x_{ij}^{tc}: Number of times that server $i \in S$ forwards a request for an object of content $c \in C$ to server $j \in S$ during period $t \in T$.

The problem is then formulated as follows:

$$\min \sum_{t\in T}\sum_{s\in S}\sum_{i\in I}\sum_{j\in I} x_{ij}^{tc} bi_{ij}^{c} +$$
$$\sum_{t\in T}\sum_{s\in S}\sum_{i\in I}\sum_{j\in I} y_{ij}^{ts} br_{ij}^{c} + \sum_{t\in T}\sum_{s\in S}\sum_{i\in I} a_i^{tc} m^{tc} bm_i^{c}$$

Subject to:

$$
\begin{array}{rlr}
bi_{ij}^{c} = si^{c} \times dist_{i,j} & \forall c \in C, \forall i \in S, \forall j \in S & (1) \\
br_{ij}^{c} = sr^{c} \times dist_{i,j} & \forall c \in C, \forall i \in S, \forall j \in S & (2) \\
bm_{i}^{c} = sm^{c} \times dist_{io^{c}} & \forall c \in C, \forall i \in S & (3) \\
\sum_{i \in S} a_{i}^{tc} \geq 1, & \forall t \in T, \forall c \in C & (4) \\
\sum_{j \in S} x_{ij}^{tc} + d_{i}^{tc} \cdot a_{i}^{tc} = d_{i}^{tc}, & \forall t \in T, \forall c \in C, \forall i \in S & (5) \\
x_{ij}^{tc} \leq d_{i}^{tc} \cdot a_{j}^{tc}, & \forall t \in T, \forall c \in C, \forall i \in S, \forall j \in S & (6) \\
a_{i}^{(t+1)c} - a_{i}^{tc} \leq \sum_{j \in S} y_{ij}^{tc}, & \forall t \in T, \forall c \in C, \forall i \in S & (7) \\
y_{ij}^{tc} \leq a_{j}^{tc}, & \forall t \in T, \forall c \in C, \forall i \in S, \forall j \in S & (8) \\
x_{ij}^{tc} \geq 0, & \forall t \in T, \forall c \in C, \forall i \in S, \forall j \in S & (9) \\
y_{ij}^{tc} \in \{0,1\}, & \forall t \in T, \forall c \in C, \forall i \in S, \forall j \in S & (10) \\
a_{i}^{tc} \in \{0,1\}, & \forall t \in T, \forall c \in C, \forall i \in S & (11)
\end{array}
$$

The objective function minimizes the total traffic over the network backbone. The first set of summations represents the total traffic generated bye request indirect attendance, the second set of summations represents the total traffic generated by replication operations and, finally, the last set of summations represents the total traffic generated by replica maintenance operations.

Constraints (1), (2), and (3) computes the traffic generated by each indirect request service, replication and maintenance operation, respectively. The set of constraints (4) ensures that there is at least one server for each content on the network. The set of constraints (5) ensures that client requests are served, either locally or by remote servers. The set of constraints (6) ensures that a server only forwards a request to a server containing the requested object. Constraints (7) guarantee that each replication does result in a new content replica. The set of constraints (8) ensures that the content being copied during a replication is stored in the originating server. Constraints (9), (10) and (11) ensure that variables x_{ij}^{tc} are non-negative and variables y_{ij}^{tc} and a_{ij}^{tc} are binary, respectively. For simplicity, the model does not impose constraints on server storage requirements. However, it could be easily extended to include such constraints.

The model defined above implements the offline version of our WDCDNM and can be solved to optimality. Unlike previous static content placement optimization models [9,10,13], it uses the set of parameters d_{i}^{tc} to model demand variation over time and space, since the spatial location of each mobile user at a given time determines the server to which its requests are sent. Because it needs a priori knowledge of all future demand, the model can not be applied to a real system. Nevertheless, it provides a lower-bound on the total network traffic generated for efficient content placement in mobile networks and thus, can be used to assess the efficiency of our online approximate algorithm, described next.

3.2 Online Algorithm

In order to solve large and practical problems, we propose and evaluate a new online heuristic algorithm for dynamic placement of content in wireless CDNs. The algorithm uses a statistical forecasting method, called Double Exponential Smoothing [2], to predict future demand for different contents on each server, and uses the predictions to decide whether to add a new content replica or to remove an existing one.

The Double Exponential Smoothing algorithm works as follows. Let y_t be the actual demand, in number of client requests, observed during period t at a server. Let α be a smoothing factor. The method calculates the prediction $\hat{y}_{t+\tau}$, made at the beginning of period t for the expected demand for period $t + \tau$ at the server, as follows:

$$\hat{y}_{T+\tau}(T) = (2 + \tfrac{\alpha\tau}{1-\alpha})S_T - (1 + \tfrac{\alpha\tau}{1-\alpha})S_T^{[2]}$$

where:

$$S_T = \alpha y_T + (1-\alpha)S_{T-1}$$
$$S_T^{[2]} = \alpha S_T + (1-\alpha)S_{T-1}^{[2]}$$

The α parameter is used to control how quickly recent demand observations are incorporated in the prediction. In other words, α is the weight given to recent observations in the prediction of future demand.

Our new online heuristic is a distributed greedy algorithm, which is executed simultaneously and independently by each server. At the beginning of each reconfiguration period, each server independently decides whether it should locally replicate any of the contents available in the system and whether it should remove local replicas, based on the predictions of future *local* demand for the following δ periods. More precisely, each server keeps track of the number of local client requests to each content received during each period. It then uses the Double Exponential Smoothing method to predict future local demand for each content. The predictions are then used to estimate the total network traffic that would be generated from replicating the content and maintaining the new replica, or from simply maintaining the replica, if it already exists, for the following δ periods. It also estimates the total network traffic that would be generated during those periods if it had to forward local client requests to the current closest replica. The content is replicated in the server only if the estimated traffic incurred by replication and future maintenance is lower than the estimated traffic incurred by indirect request service. Similarly, the local replica is removed from the server if the estimated maintenance traffic is higher than the estimated traffic caused by indirect request service. In summary, each server independently makes a decision that, at the time, minimizes its share of the total traffic generated for serving its *predicted* future demand. The algorithm is shown in Algorithm 1.

Note that our algorithm takes a different approach from the previously proposed ACDN dynamic content placement algorithm [14]. In particular, there are four key points that distinguish them. First, the ACDN algorithm makes replication decisions based on the demand observed in the past, whereas our algorithm

Algorithm 1 New Online Content Placement Algorithm for Wireless Networks

```
for $\forall$ content $c \in C$ do {Executed by server $i \in S$ at beginning of each period $t \in T$}
  for $k = 1, \cdots, \delta$ do
    $\hat{y}_{t+k} = Forecast\_Local\_Demand(c, i, t, k)$ {Use Double Exponential Smoothing Method to forecast local demand for period $t + k$ in the future.}
  end for
  $d^{ic} = \sum_{k=1}^{\delta} \hat{y}_{local_{t+k}}$ {Compute predicted local demand for the next $\delta$ periods.}
  {Let $j$ be the server with the closest replica of content $c$ to server $i$.}
  {Estimate total traffic over the backbone for indirect service in the next $\delta$ periods.}
  $bi^c = d^{ic} \times si^c \times dist_{i,j}$
  {Estimate total traffic over the backbone for replicating content $c$ in server $i$. }
  $br^c = sr^c \times dist_{i,j}$
  {Estimate total traffic over the backbone for maintaining new replica of content $c$ in server $i$. Flag $m^{t,c}$ indicates whether content $c$ changes during period $t$.}
  $bm^c = \sum_{k=1}^{\delta} m^{t+k,c} \times sm^c \times dist_{i,o^c}$
  if $i$ does not have a replica of content $c$ then
    if $bi^c > (br^c + bm^c)$ then
      Add new replica of $c$ in server $i$
    end if
  else if $i$ has the content $c$ and is not the origin server of $c$ then
    if $bi^c < bm^c$ then
      Remove replica of $c$ from server $i$
    end if
  end if
end for
```

uses predictions of future demand, which in turn, are based on the *evolution* of past demand. Second, the ACDN algorithm adopts a "push" strategy. In other words, at each reconfiguration period, a server that currently has a replica decides as to whether it should send it to other servers from which it received a large number of indirect requests in the past. In contrast, our algorithm follows a "pull" strategy. Third, the "push" approach makes an implementation of the ACDN algorithm in space-constrained servers more difficult. The server that is initiating a content replication operation should know whether the destination server has enough local space to store the new replica. Finally, our online algorithm is more efficient than the ACDN algorithm. Whereas our algorithm has time complexity equals to $O(|C|)$, their algorithm has complexity $O(|S| \times |C|)$, where $|C|$ and $|S|$ are the numbers of contents and servers in the system. A quantitative performance comparison of both algorithms is left for future work.

4 Performance Evaluation

This section evaluates the performance of our new online dynamic content placement algorithm, comparing it with the offline optimal solution and with the previous centralized and distributed static placement approaches. Section 4.1 briefly describes the mobility simulator used to evaluate the algorithms. The most relevant performance results are presented in section 4.2.

4.1 Mobility Simulator

The mobility simulation and demand generation are performed using a mobility simulator developed in [11]. The simulator models a twenty-kilometer radial city, divided in area zones. The division is based on population density and natural limits (e.g., rivers, highways, railway tracks). Taking mobile telecommunication requirements into account, it seems reasonable to assume that each area zone is equal to a network area (e.g., macrocell, local exchange area). The area zones are connected via high-capacity routes, which represent the most frequently selected streets for movement support, and are grouped into four area types: city center, urban, suburban and rural. The simulator also includes a number of content servers with fixed locations. Figure 1 shows a representation of the modeled city with the server locations indicated by small dark circles. The city has 32 area zones (eight per city area types), four peripheral routes (one per area type) and four radial high-capacity routes.

The simulator models residences, workplaces, schools and other points such as shopping centers and parks as movement attraction points, i.e., locations where people spend considerable amounts of time. Figure 2 shows the frequencies of different types of movement attraction points in each city area type.

The user population is divided into four mobile groups according to the mobility characteristics of the individuals and to the demand they generate for different content types. The four user groups are defined as follows: 5% of the users are 24-hour delivery boys, 60% are common workers, 30% are housekeepers and the remaining users are taxi drivers. The typical mobility behavior of each user group is defined in a movement table. In this table, the day is divided into time periods and a probability of a user of the group being at a certain location is associated to each period. Typical movement tables are given in [11]

Each user within a group generates a number of calls during simulation. The per-user call inter-arrival times are exponentially distributed with means 14, 7, 14 and 18 minutes, for 24-hour delivery boys, common workers, housekeepers and taxi drivers, respectively. Once connected, a user issues a number of requests at rate 1 request per second. The number of requests issued by a user during a call depends on the call duration, which is also exponentially distributed with mean 60 seconds, for all groups. Within each user group, relative content popularity follows a Zipf-like distribution (Prob(request content c) $= K/c^{\alpha}$, where $\alpha > 0$ and K is a normalizing constant [19]), with parameter $\alpha = 0.84$.

4.2 Results

We evaluate the performance of the new online dynamic content placement algorithm comparing it to two traditional static placement strategies: centralized and distributed. In the centralized approach, each content is stored in only one server in the network, whereas the distributed approach allows for each content to be replicated in a fixed number of servers. In both cases, the location of the content replicas do not change over time, although the content may itself change, triggering maintenance operations, in case of multiple replicas. We also compare our algorithm with the ideal lower-bound provided by the offline optimal solution.

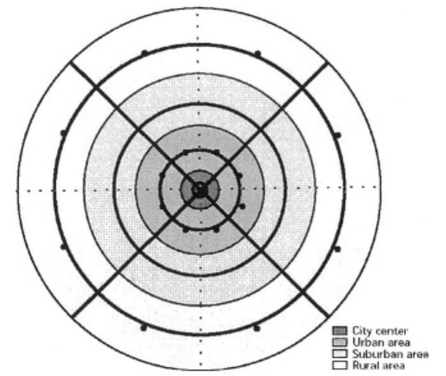


Fig. 1. The Simulation Model: City Areas and Server Locations

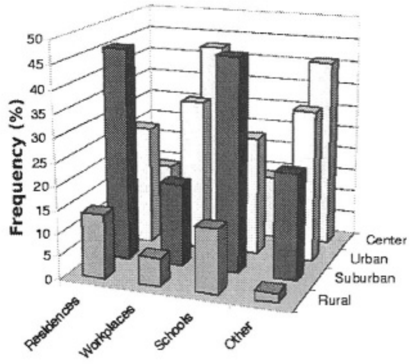


Fig. 2. Frequency of Movement Attraction Points over the City Areas

In our simulations, we experiment with a distributed approach with 2 and 4 replica servers. Furthermore, in both centralized and distributed static approaches, the content replicas are placed in the more central servers (see Figure 1). We make the assumption that the distance between two servers is the linear distance shown in Figure 1. Furthermore, unless otherwise stated, in all experiments, we set the number of contents to 3 and the number of mobile users to 5000. The reconfiguration period is set to 10 minutes and the simulation runs for 90 periods. The α and δ parameters of the demand forecasting method are set to 0.2 and 7, respectively. We also make the conservative assumption that the content modification rate is one per period, for all contents. Thus, maintenance operations must be performed over the deployed content replicas at each period.

We run a large number of experiments varying several system parameters and covering a large design space. The next sections present the most relevant results obtained in our experiments.

Replication, Indirect Service, and Maintenance Content Sizes
This section analyzes the impact of the three content size parameters, namely, the replication size (sr^c), the indirect service size (si^c) and the maintenance size (sm^c), which represent the number of bytes transferred in each specific operation. Since the traffic generated in each operation is proportional to the number of bytes transferred, the relative content sizes impact directly a server decision for replicating a content or removing an existing replica and thus, the performance of our algorithms. In the following experiments, we assume a baseline configuration where $sr^c = si^c = sm^c = 1KByte$, for contents $c \in C$.

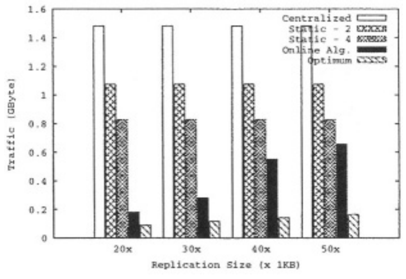
We start by evaluating the impact of the replication size on the total network traffic generated over the backbone if the indirect service and maintenance sizes are fixed at the baseline value and the replication size increases to 20, 30, 40 and 50 times the baseline. Figure 3 shows the total traffic generated for the online algorithm and offline optimal solution. For comparison purposes, it also shows the traffic for the centralized and distributed static approaches. Note that the variation in the replication size has no impact on the static approaches. However, as replication size increases, the replication component of the total traffic starts to dominate. In response, the online and offline optimal approaches tend to reduce the number of replications. As expected, the offline optimal solution, having the knowledge of all future demand, makes replication decisions that will lead to significant reductions in the future traffic, especially given the low maintenance cost. On the other hand, the online algorithm uses predictions of future demand for the following δ period and thus, may make some sub-optimal replication decisions which will not pay-off in the long run. Nevertheless, it is interesting to note that in the worst case, for a replication size equal to 50 times the baseline, the online algorithm produce a solution with total network traffic which is only four times larger than the offline optimal. Furthermore, compared to the best static approach (distributed with 4 servers), our online algorithm reduces total network traffic in 78%, for a replication size equal to 20 times the baseline, and in 21% for a replication size equal to 50 times the baseline.

Fixing the replication size at 50 times the baseline and the indirect service size at the baseline, Figure 4 shows the total traffic generated by each algorithm as the maintenance size increases. Recall that we assume a content modification rate of one per period, for every content. Thus, as maintenance size increases, it becomes less cost-effective to maintain a replica in a server with low client demand. In this scenario, total network traffic for the dynamic algorithms increases significantly. However, the performance of the static distributed approaches degrades even more significantly, especially for large number of replica servers, because fixed location replicas must be maintained even during low demand periods.Compared to deploying four fixed-location replica servers, our online algorithm reduces total network traffic in up to 50%. In particular, for maintenance sizes larger than 20 times the baseline, the online dynamic algorithm chooses not to replicate any content and, thus, performs similarly to the centralized approach. Furthermore, our online algorithm results in an increase in the total traffic that is only 14% higher than the offline optimal. If maintenance size is very large, both algorithms decide for not replicating. Thus,the knowledge of future demand does not give a significant advantage to the offline optimal.

Finally, we now fix both replication and maintenance sizes equal to 50 times the baseline, and vary the indirect service size. Figure 5 shows the results. As the content indirect service size increases, it becomes more cost-effective for a server to replicate the content locally than to forward their client requests to the closest replica. Thus, the dynamic algorithms generate significantly better results than the static approaches. In particular, our online algorithm, which produces solutions within 18% of the offline optimal, reduces the total network traffic generated by the best static approach analyzed by up to 61%.

Number of Simultaneous Mobile Users

This section analyzes the impact of the number of simultaneous mobile users simulated on the total network traffic generated over the backbone by our new online and offline optimal solutions and by the traditional static centralized and distributed approaches. We use homogeneous size configurations with all three size parameters equal to 10 KBytes. Figure 6 shows that the dynamic content



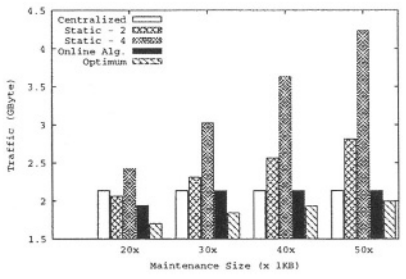


Fig. 3. Total Network Traffic as a Function of Replication Size sr^c. $(si^c = sm^c = 1KB, \forall c \in C)$

Fig. 4. Total Network Traffic as a Function of Maintenance Size sm^c. $(si^c = 1KB, sr^c = 50KB, \forall c \in C)$

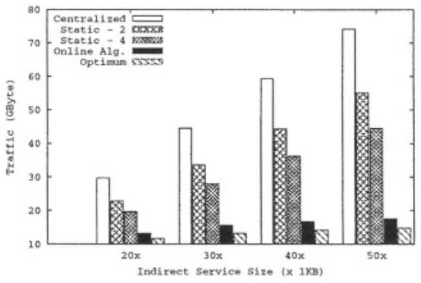


Fig. 5. Total Network Traffic as a Function of Indirect Request Service Size si^c. $(sr^c = sm^c = 50KB, \forall c \in C)$

placement solutions scale much better with the number of concurrent mobile users than the static approaches. With 500 thousand mobile users, the online algorithm reduces total traffic in 75%, over static placement with 4 replicas.

Simulation Time

This section evaluates the increase in the cumulative network traffic as a function of simulation time. As shown in Figure 7, for a configuration with all three size parameters equal to 10 KBytes and 2000 mobile users, the dynamic solutions outperforms the static approaches significantly, as time progresses.

Traffic Overhead Due to Management Operations

Content replication and maintenance operations incur a management traffic overhead, which must be payed off by a reduction in the traffic of indirect client responses in order to be beneficial to the system. Otherwise, total network traffic increases and dynamic content placement may perform worse than static content placement. Figure 8 shows, for each algorithm analyzed, the portion of the total traffic due to management operations, for a configuration with all three size pa-

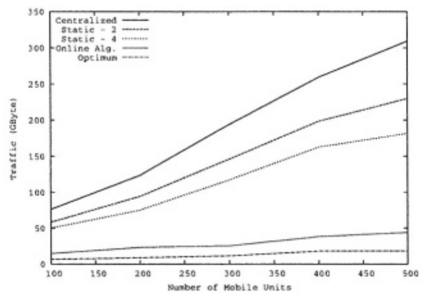


Fig. 6. Total Network Traffic as a Function of the Number of Users

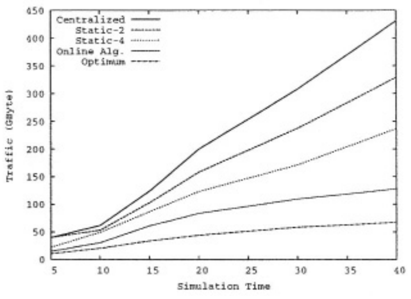


Fig. 7. Total Network Traffic as a Function of Simulation Time

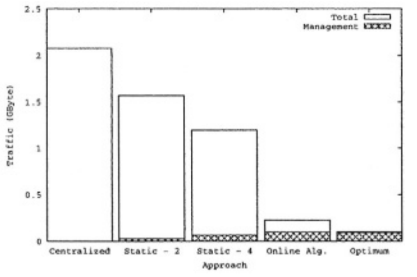


Fig. 8. Portion of Traffic due to Management Operations

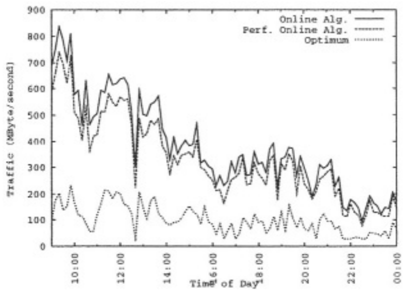


Fig. 9. Precision of Forecasting Method

rameters equal to 10 KBytes and 2000 mobile users. For the static approaches, the management traffic, due to replica maintenance operations, is a tiny fraction of the total traffic. The dynamic approaches reduce total traffic significantly by performing replication and thus, have a higher management traffic overhead. In particular, practically all the traffic generated in the offline optimal solution is due to management operations. Thus, the replication decisions made are payed off by significant reductions in indirect request service. The online dynamic algorithm generates approximately the same management overhead, with a total traffic that is only 126% higher than the offline optimal.

Forecasting Method

Finally, we evaluate the accuracy of the Double Exponential Smoothing forecasting method by comparing it with perfect forecasting, where the real client demands for the future δ periods are known *a priori*. We compare the performance of the online algorithm using either method, for a configuration with 12000 users and replication, indirect service and maintenance sizes equal to 2MBytes, 20KBytes and 900KBytes, respectively. We run the simulator for 150 periods, covering almost a whole day, and use δ equal to 7, for both methods. Figure 9 shows the total network traffic obtained as a function of the time of the simulated day. For comparison purposes, it also shows the results obtained with the offline optimal solution. Perfect forecasting reduces total traffic in only up to 9.7%, compared to the Double Exponential Smoothing method. Thus, this demand forecasting method is highly accurate. Note that, as time goes by, the online algorithm approximates to the offline optimal, being within 122% of optimal after 150 periods. Similar results were obtained with other values of δ.

5 Conclusions and Future Work

This papers proposed WDCDNM, a Wireless Dynamic Content Distribution Network Model that takes both temporal and spatial variations in client demand into account to dynamically replicate content with the goal of minimizing total traffic over the backbone of wireless networks. Our WDCDNM decides to replicate a certain content or remove an existing replica based on the solution of a tradeoff between minimizing the total traffic generated by replication and replica maintenance operations and minimizing the traffic resulting from indirect request service.

We proposed and evaluated two implementations of WDCDNM: an offline optimal solution and an online heuristic, based on demand forecasting. We used a previously developed mobility simulator to evaluate the performance of our solutions, contrasting it with the performance of previous static centralized and distributed approaches. Compared to the best static approach, with 4 replica servers, our online algorithm reduces total network traffic in up to 78%, producing solutions that are within four times the ideal offline optimal.

Directions for future work include introducing capacity constraints in WDCDNM, performing a quantitative comparison of our online solution and the ACDN algorithm [14] and further optimizing the online dynamic placement algorithm.

References

1. S. Albers. Competitive Online Algorithms. *Mathematical Programming,* 5:3–26, 2003.
2. B. L. Bowerman and R. T. O'Connell. *Forecasting and Time Series an Applied Approach.* Wadsworth Company, 1993.
3. Y. Chen, R. Katz, and J. Kubiatowicz. Dynamic Replica Placement for Scalable Content Delivery. In *Proc. International Workshop on Peer-to-Peer Systems,* Cambridge, MA, March 2002.
4. M.R. Garey and D. S. Jonhson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W H Freeman, 1979.
5. Y. Guo, Z. Ge, B. Urgaonkar, P. Shenoy, and D. Towsley. Dynamic Cache Reconfiguration Strategies for a Cluster-Based Streaming Proxy. In *Proc. Web Caching and Content Distribution Workshop,* Hawthorne, NY, September 2003.
6. S. Jamin, C. Jin, A. R. Kurc, D. Raz, and Y. Shavitt. Constrained Mirror Placement on the Internet. In *Proc. IEEE INFOCOM,* Anchorage, AK, April 2001.
7. J. Kangasharju, J. Roberts, and K. Ross. Object Replication Strategies in Content Distribution Networks. In *Proc. Web Caching and Content Distribution Workshop,* Boston, MA, June 2001.

8. M. Karlsson and C. Karamanolis. Choosing Replica Placement Heuristics for Wide-Area Systems. In *Proc. International Conference on Distributed Computing Systems (ICDCS),* Hachioji, Tokyo, Japan, March 2004.
9. P. Krishnan, D. Raz, and Y. Shavitt. The Cache Location Problem. *IEEE/ACM Transactions on Networking,* 8(5):568–582, October 2000.
10. B. Li, M. J. Golin, G. F. Italiano, X. Deng, and K. Sohraby. On the Optimal Placement of Web Proxies in the Internet. In *Proc. IEEE INFOCOM,* New York, NY, March 1999.
11. G. R. Mateus, O. Goussevskaia, and A. A. F. Loureiro. Simulating Demand-Driven Server and Service Location in Third Generation Mobile Systems. In *Proc. Europar 2003,* Klagenfurt, Austria, August 2003.
12. P. Mirchandani and R. Francis. *Discrete Location Theory.* John Wiley and Sons, 1990.
13. L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the Placement of Web Server Replicas. In *Proc. IEEE INFOCOM,* Anchorage, AK, April 2001.
14. M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the Edge: A Platform for Replicating Internet. In *Proc. Web Caching and Content Distribution Workshop,* Hawthorne, NY, September 2003.
15. P. Radoslavov, R. Govindan, and D. Estrin. Topology-Informed Internet Replica Placement. In *Proc. Web Caching and Content Distribution Workshop,* Boston, MA, June 2001.
16. M. Tariq, R. Jain, and T. Kawahara. Mobility-Aware Server Selection for Mobile Streaming Multimedia Content Distribution Networks. In *Proc. Web Caching and Content Distribution Workshop,* Hawthorne, NY, September 2003.
17. T. Yoshimura, Y. Yonemoto, T. Ohya, M. Etoh, and S. Wee. Mobile Streaming Media CDN Enabled by SMIL. In *Proc. WWW Conf.,* Honolulu, HI, May 2002.
18. H. Yu and A. Vahdat. Minimal Replication Cost for Availability. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC),* Monterey, CA, June 2002.
19. G. K. Zipf. *Human Behavior and the Principal of Least-Effort.* Addison-Wesley, Cambridge, MA, 1949.

Overhaul
Extending HTTP to Combat Flash Crowds

Jay A. Patel and Indranil Gupta

Department of Computer Science
University of Illinois at Urbana - Champaign
Urbana, IL 61801

Abstract. The increasing use of the web for serving http content, for database transactions, etc. can place heavy stress on servers. Flash crowds can occur at a server when there is a burst of a large number of clients attempting to access the service, and an unprepared server site could be overloaded or become unavailable. This paper discusses an extension to the http protocol that allows graceful performance at web servers under flash crowds. We implement our modifications for the Apache web server, and call the new system as *Overhaul.* In Overhaul mode, a server copes with a stampede by offloading file transfer duties to the clients. Overhaul enables servers to chunk each requested document into small sections and distribute these partial documents to clients. The clients then share the sections amongst themselves to form a complete document. Overhaul enables a web server to remain responsive to further requests from other clients and at the same time helps conserve the amount of bandwidth utilized by a flash crowd. We present detailed experimental results comparing the benefits of using Overhaul under flash crowds and under normal operating situations. Although we restrict our studies to static content, the Overhaul architecture is applicable to improving web services in general.

1 Introduction

There have been numerous reports of flash crowds ruining the performance of web sites. Flash crowds can bring a web server to a screeching halt. For example, an order of magnitude increase in traffic can be expected as a result of linkage from a highly popular news feed. Most web servers are not designed to cope with such a large spike in traffic. There are not many reasonable, economically-sound solutions to control a massive surge in traffic. There are only a few choices a web master can make to combat the sudden influx of traffic: invest in extra resources (be overly insured), temporarily shut down the web site, or change the content of the site. The first solution is not a viable alternative for many enthusiast web masters or not-for-profit organizations as buying surplus resources is an expensive proposition. Additionally, since most flash crowds aren't malicious, the later two solutions could pose a problem to the regular visitors of the site. If the site is shut down, one may alienate the regular visitors and, at the same time, fail to attract new ones.

The performance of a web server degrades as the number of concurrent requests increases. Additionally, a pending document request blocks the server from addressing requests from other clients. Flash crowds create both these problems and therefore present a challenging research question. Previous research to address flash crowds, which is discussed in detail in Section 2, can be broadly categorized as solutions that either propose modifications to the server architecture, institute protocol changes, or depend on client collaboration for data sharing.

In this paper, we propose a systematic yet simple change to HTTP that reduces the resources required to serve a flash crowd. The primary goal of this paper is to present a technically feasible solution with minimal changes to the current HTTP standard. It is important that the solution be deployable on today's most-popular web servers and clients. The process presented by this paper is seamless to the end-user and requires no special input from the web master once it is deployed. We strive to minimize the economic costs of a flash crowd from the server administrator's perspective. The main contribution of this paper is the massive reduction of resources (primarily bandwidth) utilized by the server during a flash crowd.

Overhaul is a modification to HTTP that involves both the client and the server. During a flash crowd, the server steps into Overhaul mode and serves only portions of requested documents. Each file is divided into multiple portions and distributed amongst the clients. The clients collaborate together (independent of the server) to share the various portions to form a complete document.

2 Related Work

Previous research can be divided into three different broad categories: architectural changes, protocol changes, and cooperative sharing.

SEDA [12] is a staged event-driven architecture used to support massively concurrent requests for well-conditioned services like HTTP. Capriccio [11] is a highly scalable thread package that can be used with high-concurrency processes. The thread package is implemented as a user-level process, making it easily portable across platforms and far removed from inconsistencies of the underlying operating system. Another paper [10] examines the impact of various connection-accepting strategies used to boost web server performance. Web booster [7] is a dedicated machine that intercepts the TCP connection from clients to the server during peak times and only bothers the server if the document is not present in its cache.

DHTTP [8] is a replacement for the TCP-based HTTP protocol. Client requests are initiated over UDP streams, which frees up the server from the overhead of maintaining expensive and non-cacheable TCP connections.

Backslash [9] is a collaborative web mirroring system. Squirrel [3], based on Pastry, implements a web caching mechanism over a P2P overlay in lieu of a centralized proxy server. Another collaborative web caching mechanism [5] based on Kelips performs well during heavy churn (rapid arrival and departure of member nodes). Content distribution networks (CDN) [4] like Akamai are utilized by many well established sites. CDNs are used to minimize the number of hops required to fetch a document. FastReplica [1] is an algorithm used to efficiently replicate a large file using subfile propagation. BitTorrent [2] is a protocol for distributing large files over a P2P network. The file provider makes the file available by providing a meta-info (torrent) file and a tracker URL. Clients exchange blocks of the file with other peers connected to the tracker.

3 Design

Overview. Overhaul is set of changes to HTTP that lets overloaded servers distribute content through a P2P network. As described in this paper, Overhaul assumes a fully collaborative environment. A web server under the rampage of a flash crowd enters into Overhaul mode and splits up the requested document into n chunks. Each request results in a response that includes the i^{th} chunk and the IP addresses of m other clients accessing the document. A signature for each of the n chunks is also provided in the header. A client supporting the Overhaul extension connects to other clients to retrieve the remaining chunks.

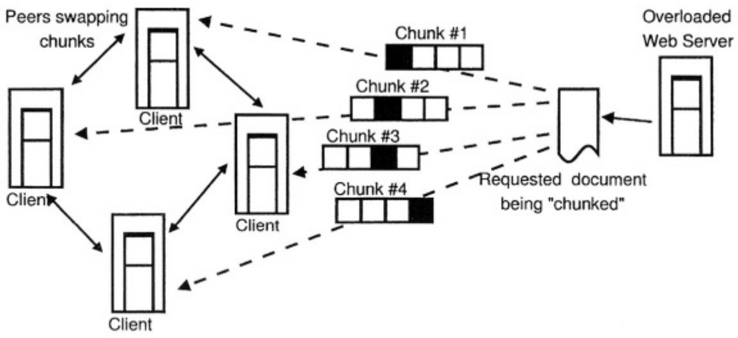


Fig. 1. Overhaul: An overview

A simple Overhaul model is shown in Fig. 1. A document is divided into chunks and distributed amongst four unique clients. The clients collaborate to merge the chunks together and form a coherent document. The Overhaul process enables the server to save approximately three-fourths of the bandwidth utilized by a regular fetch. By transferring only a small portion of the document, the Overhaul process frees up the server to satisfy requests from other clients.

Overhaul provides an HTTP-integrated solution for collaborative fetching and sharing of documents. This differs from the approach taken by BitTorrent, which excels as a specialized tool for distributing large files over a separate P2P network. Unlike Overhaul, BitTorrent requires a dedicated meta-info file for each requested document. This results in additional traffic to a server. BitTorrent also requires a dedicated tracker (basically a server which is not compatible with HTTP clients). Moreover, BitTorrent is not geared towards short-time downloads, as it depends on peers sharing documents after completing a document fetch.

Details. The Overhaul solution requires modification to regular HTTP client-server interaction. To maintain backwards compatibility, clients inform the server of their Overhaul functionality by sending a `Supports` tag with a regular HTTP request. This allows the server to maintain a list of the clients that will engage in the Overhaul process, and additionally send an error message to legacy clients. A sample tag takes the form of:

```
Supports: Overhaul <Speed> <Port>
```

In most cases the server will ignore this tag and respond normally. However if the server is overloaded, it will respond in Overhaul mode with only one chunk of the document. Three special tags: `Overhaul-chunk`, `Overhaul-hosts`, and `Overhaul-md5sum` are sent in addition to the header. The first tag provides the sequence number of the current chunk being sent to the client. The second tag includes a list of peers (and their respective server ports) that are currently engaged in the Overhaul process. The last tag is a list of md5 signatures of the chunks. The md5 hash is provided to verify the contents of the chunks. The server will need to return HTTP code 206 to indicate partial content and the corresponding `Content-Range` tag with the byte range of the transmitted chunk.

If a client gets an Overhaul mode response back from the server, it opens the aforementioned port to allow requests from other peer-clients. Meanwhile, the client establishes connections to other peer-clients to gather more chunks. After establishing a connection to another peer-client, a client sends a request to fetch a missing chunk in the given form:

```
GET /file.html CHUNK <#> HTTP/1.0
HOST www.host.com
Overhaul-port: <Port>
Overhaul-hosts: <List of known hosts:port>
Overhaul-chunks: <List of available chunks>
```

The requesting peer-client provides the server port, a list of known peer-clients and available chunks with the request. In response to an Overhaul request, the serving peer-client sends a HTTP return code of `206` and the relevant data if it possesses the requested chunk. Otherwise, a HTTP return code of `404` is sent. The response header also includes `Overhaul-hosts` and `Overhaul-chunks` tags to exchange information with the requesting peer-client.

After the initial interaction, both peer-clients learn about new peer-clients and the available chunks on the other peer-client. This enables clients to grow their list of known peer-clients. The clients repeat the process with other peer-clients until the document is completely assembled.

If the peer-clients list is fully exhausted without successful completion of the document, a client may re-request chunks from known peer-clients. If the client fails to collect the document chunks even after successive tries, the client may retry the server. The server will respond either with a new list of peer-clients or the document in its entirety depending on the current server load.

To further reduce server workload, a client may query known peer-clients for other documents being Overhauled on a particular host with the `INFO` tag. This tag helps reduce the number of active TCP connections and the amount of bandwidth utilized by affected hosts as HTML documents generally include many embedded objects. A query seeking information about documents being served takes the form of:

```
INFO www.host.com HTTP/1.0
```

In response to this tag, the peer-client indicates that there is no associated body content with HTTP response code `204`. Additionally, the following set of header tags is sent for each document currently being fetched in the Overhaul mode:

```
Overhaul-URL: <An Overhaul URL>
Overhaul-md5sum: <List of md5 hashes>
Overhaul-hosts: <List of host:port>
Overhaul-chunks: <List of available chunks>
```

Once the document fetch process is complete, the peer-client may shut down the port and decline further connections.

4 Implementation

The modifications required by Overhaul on a server were implemented as a module to the Apache/2.0 web server. Apache was chosen because of its popularity [6], extensibility, and openness. Apache web server can be extended by loading modules at run time for added functionality. `mod_overhaul` implements the Overhaul extension for the Apache web server. The module can be customized by specifying the minimum chunk size, the maximum number of chunks, and the number of peers (to keep track of) per document.

The client was not implemented directly: a proxy server was developed using the Java programming language. A proxy was deemed to be a superior solution as it can be used with most web browsers without any source modifications to the browser. The proxy can also be run on a vast multitude of platforms that have a Java virtual machine. The proxy normally fetches HTTP documents but acts as an Overhaul client when a server responds in Overhaul mode.

5 Results

Server. The performance of mod_overhaul was tested by executing a few controlled experiments. Apache/2.0 web server was compiled with the worker MPM, which uses a single process with multiple child threads to handle requests. The tests were performed on a medium-range web server with a 2.5 GHz AMD Athlon XP+ processor and 1 GB RAM powered by RedHat's Fedora GNU/Linux operating system. The client machine utilized for the experiments was a low-end computer powered by a 650MHz Intel Pentium III processor and 320 MB RAM. Both machines were placed on the same 100 Mbps switch as the web server to minimize network latency and maximize throughput. Testing was performed using the ApacheBench utility. All experiments were performed with 25 concurrent accesses from the client machine. The CPU utilization of the client machine was minimum. However, network bandwidth was saturated on both the server and client machines when performing some large experiments.

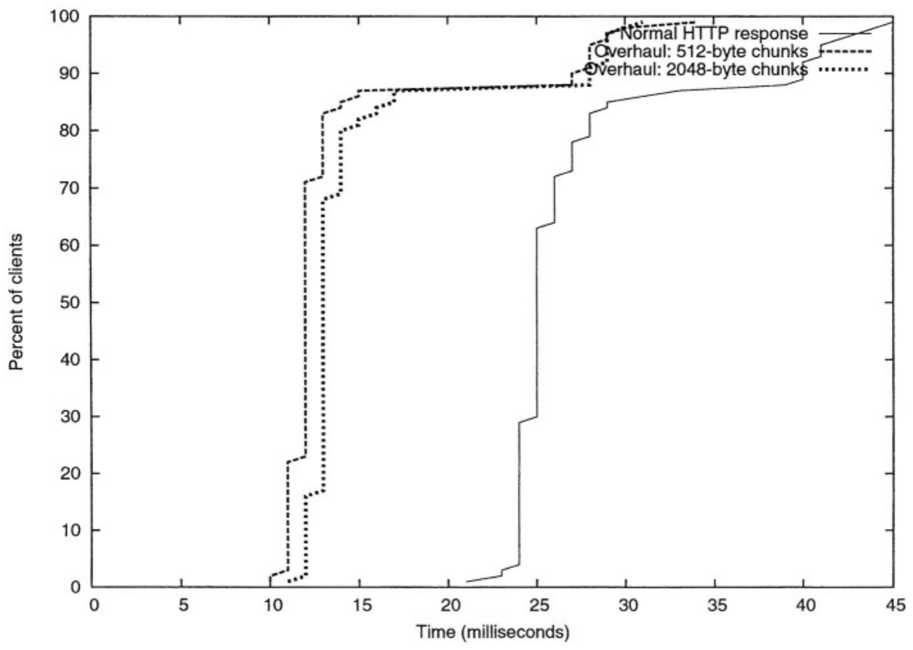


Fig. 2. Time to serve or a medium-sized document (10,000-bytes)

Overhaul's chunked responses lead to a substantial performance gain over regular responses to a medium-sized static document as shown in Fig. 2. Smaller chunks lead to better performance. However, 512-byte chunks only provide a minor performance gain over 2048-byte chunks.

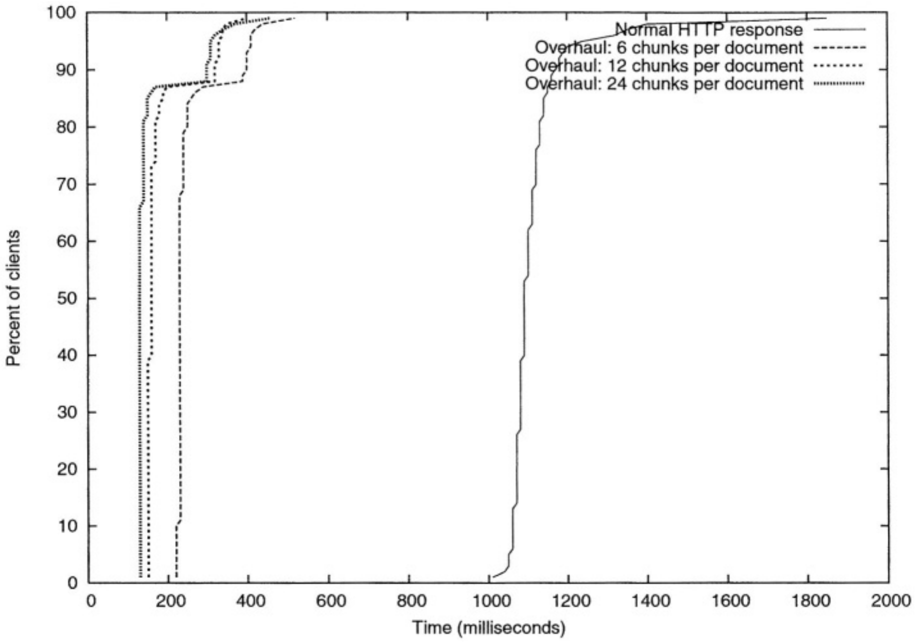


Fig. 3. Time to serve a large document (50,000-bytes)

With larger documents, a constant chunking size creates numerous chunks. Fig. 3 maps the performance of the server when configured to split a document in a fixed number of chunks. This experiment verifies the the diminishing benefits of smaller chunk sizes, possibly due to the constant overhead incurred for establishing a TCP connection. Additionally, the work load increases by having small chunks as clients need to fetch more chunks from other peer-clients. A better solution would be to have a fixed-number of chunks per document, with limitations on the minimum chunk-size.

An experiment to benchmark performance of Apache while serving different size files was setup. The server was configured to have minimum chunk size of 512-bytes with the maximum number of chunks set to twelve. Fig. 4 shows the result of the experiment. In Overhaul mode, the requests are satisfied at a much higher rate than regular responses. Apache is up to eight times more responsive when using as few as twelve chunks for medium sized files. The solution also scales well as file sizes increase.

In other experiments, chunking dynamically generated documents was not beneficial. This behavior is not surprising because the bottleneck in dynamic content generation process is the CPU and not the network. Moreover, dynamic content creates incompatible chunks as many documents (or parts there of) are uniquely created for specific clients.

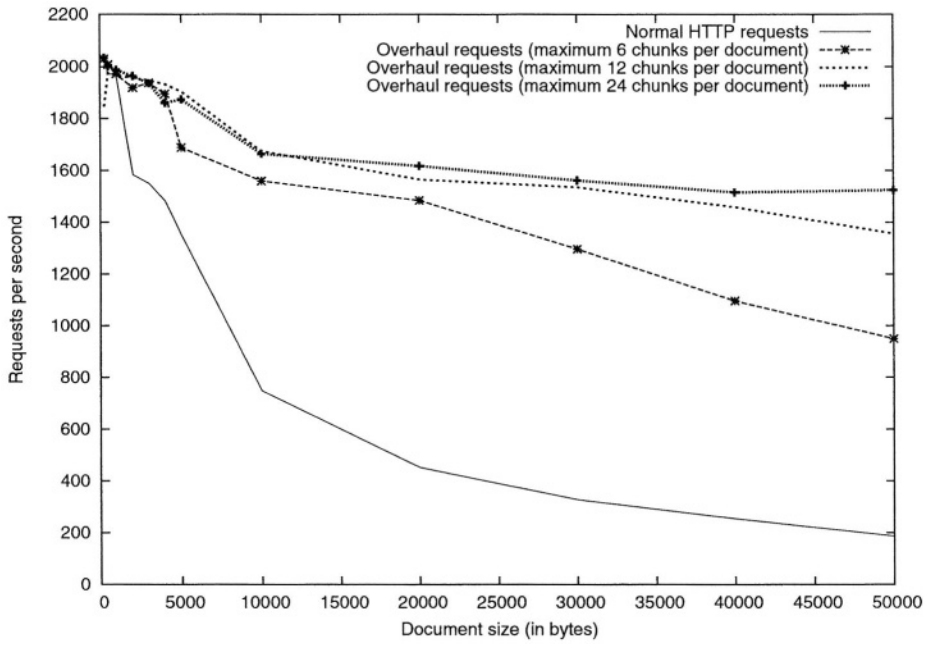


Fig. 4. Request satisfaction rate of static documents

Client. Successful deployment of the Overhaul extension depends on user experience and satisfaction. To gauge the performance of the Overhaul client process, lab experiments were performed on a cluster of 25 high-end workstations. Each machine in the cluster was equipped with a Pentium 4 processor running at 2.8 GHz with 1 GB RAM running the RedHat 9 GNU/Linux operating system. The machines were all part of the same subnetwork. One of the workstations was used as a server and the remaining were utilized as client machines.

For the first client-side experiment, 24 clients simultaneously fetched a 50,000-byte document in Overhaul mode. The server was configured to distribute each document in 12 chunks of 4,167-bytes each. This result was compared against regular HTTP fetches from a server under the flash crowd effect, simulated by requesting the same document concurrently from 150 clients for a brief period. The results are presented in Table 1 show that a fetch in Overhaul mode is more responsive than a regular HTTP request on an overloaded server. Additionally, the bandwidth utilized by the server in the Overhaul mode is approximately one-twelfth of a normal request.

The clients fetched multiple files in the next experiment. Eight files were used: one file of 30,000-bytes, three files of 20,000-bytes each and four files of 5,000-bytes each. The server was configured to send a minimum of 2048-bytes per chunk, with a maximum of 12 chunks per document. The requests were

Table 1. Time required to fetch a large document

	Regular request in flash crowds	Overhaul mode experiment
Fastest	1 sec	3 secs
Slowest	184 secs	8 secs
Average	13 secs	5 secs

staggered in two batches to take advantage of the `INFO` tag: the first twelve clients requested all eight files, whereas the next batch were restricted to requesting only the 30,000-byte file. The regular requests under a flash crowd were simulated as in the previous experiment. Table 2 shows that Overhaul mode fetches are also more responsive than regular HTTP requests when fetching multiple files under a flash crowd.

Table 2. Time required to fetch multiple documents

	Regular request in flash crowds	Overhaul mode experiment
Fastest	1 sec	14 secs
Slowest	355 secs	24 secs
Average	26 secs	17 secs

In the last experiment, the amount of bandwidth utilized by the server in Overhaul mode was approximately only one-eighteenth of the bandwidth consumed by the normal HTTP requests. The `INFO` tag reduces the number of direct TCP connections to the server and therefore the amount of bandwidth required to serve clients decreases as the flash crowd grows.

6 Conclusion

Final Thoughts. A server can benefit tremendously from chunking. In our experiments, the Apache web server is able to serve medium-sized files up to eight times faster in Overhaul mode. Since most of HTTP bandwidth is utilized on transferring medium-sized static objects (usually images), Overhaul can greatly increase the responsiveness of a server. Additionally, the amount of per capita bandwidth utilized by the server decreases as the flash crowd grows. These two properties work in tandem to lengthen the liveliness of a web site being visited by a flash crowd.

Overhaul presents a two-fold benefit to the server administrator: increased server responsiveness and decreased utilization of bandwidth. Additionally, the user is presented with the desired content in a timely manner instead of a scenario involving timed out requests, sluggish server response or complete unavailability.

Future Work. Currently, Overhaul works under the assumption that every user will be cooperative. There are no strong safeguards against freeloaders and malicious users. However, malicious peers can not corrupt chunks because of md5 hash verification. To counter freeloaders, a Overhaul client needs to maintain a trust matrix for each peer-client. The trust score for each peer-client can be a function of chunks shared, reputation amongst other peer-clients, and the bandwidth and latency of network connection. A further modification can be placed in the Overhaul extension to let clients swap trust matrices with other peer-clients.

Besides user behavior issues, many problems are associated with the structure of the Internet. There are many clients that reside behind firewalls and NATs. These clients cannot make their Overhaul port accessible to other peer-clients easily. Another issue is the heterogeneity of network connections. A future implementation can use the speed provided by the client to organize peer-groups.

References

1. L. Cherkasova and J. Lee. FastReplica: Efficient large file distribution within content delivery networks. In *USENIX Symposium on Internet Technologies and Systems,* Seattle, WA, USA, March 2003.
2. Bram Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems,* Berkeley, CA, USA, May 2003.
3. S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *ACM Symposium on Principles of Distributed Computing,* Monterey, CA, USA, July 2002.
4. J. Kangasharju, J. Roberts, and K. Ross. Object replication strategies in content distribution networks. In *Proceedings of Web Caching and Content Distribution Workshop,* Boston, MA, USA, June 2001.
5. P. Linga, I. Gupta, and K. Birman. A churn-resistant peer-to-peer web caching system. In *ACM Workshop on Survivable and Self-Regenerative Systems,* Fairfax, VA, USA, October 2003.
6. Netcraft. April 2004 web server survey.
7. V. V. Panteleenko and V. W. Freeh. Instantaneous offloading of transient web server load. In *Proceedings of Web Caching and Content Distribution Workshop,* Boston, MA, USA, June 2001.
8. M. Rabinovich and H. Wang. DHTTP: An efficient and cache-friendly transfer protocol for web traffic. In *INFOCOM,* Anchorage, AK, USA, April 2001.
9. T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *International Peer To Peer Systems Workshop (IPTPS 2002),* Cambridge, MA, USA, March 2002.
10. D. Pariag T. Brecht and L. Gammo. accept()able strategies for improving web server performance. In *USENIX Annual Technical Conference,* Boston, MA, USA, June 2004.
11. R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *ACM Symposium on Operating Systems Principles,* Lake George, NY, USA, October 2003.
12. M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *ACM Symposium on Operating Systems Principles,* Banff, Canada, October 2001.

ShortCuts: Using Soft State to Improve DHT Routing

Kiran Tati and Geoffrey M. Voelker

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{ktati, voelker}@cs.ucsd.edu

Abstract. Distributed hash tables are increasingly being proposed as the core substrate for content delivery applications in the Internet, such as cooperative Web caches, Web index and search, and content delivery systems. The performance of these applications built on DHTs fundamentally depends on the effectiveness of request routing within the DHT. In this paper, we show how to use soft state to achieve routing performance that approaches the aggressive performance of one-hop schemes, but with an order of magnitude less overhead on average. We use three kinds of hint caches to improve routing latency: local hint caches, path hint caches, and global hint caches. Local hint caches use large successor lists to short cut final hops. Path hint caches store a moderate number of effective route entries gathered while performing lookups for other nodes. And global hint caches store direct routes to peers distributed across the ID space. Based upon our simulation results, we find that the combination of the hint caches significantly improves Chord routing performance: in a network of 4,096 peers, the hint caches enable Chord to route requests with average latencies only 6% more than algorithms that use complete routing tables with significantly less overhead.

1 Introduction

Peer-to-peer overlay networks provide a distributed, fault-tolerant, scalable architecture on which wide-area distributed systems and applications can be built. An increasing trend has been to propose content delivery services on peer-to-peer networks, including cooperative Web caches [8], Web indexing and searching [13,15], content delivery systems [11,2], and Usenet news [5]. Popular designs of these overlay networks implement a distributed hash table (DHT) interface to higher level software. DHTs map keys in a large, virtual ID space to associated values stored and managed by individual nodes in the overlay network. DHTs use a distributed routing protocol to implement this mapping. Each node in the overlay network maintains a routing table. When a node receives a request for a particular key, it forwards the request to another node in its routing table that brings the request closer to its destination.

A natural trade off in the design of these routing protocols is the size of the routing table and the latency of routing requests. Larger routing tables can reduce routing latency in terms of the number of hops to reach a destination, but at the cost of additional route maintenance overhead. Because the performance and overhead of DHT overlay networks fundamentally depend upon the distributed routing protocol, significant work has focused on the problem of balancing the degree of routing state and maintenance with route performance.

Initial systems like Chord [25], Pastry [24], Tapestry [27], and CAN [22] use routing tables of degree $O(\log n)$ to route requests in $O(\log n))$ hops, where n is the number of hosts in the network. Newer algorithms improve the theoretical bounds on routing state and hops. Randomized algorithms like Viceroy [16] and Symphony [17] achieve small, constant-degree routing tables to route requests on average in $O(\log n)$ and $O(\log \log n)$ hops, respectively. Koorde [9] is a tunable protocol that can route requests with a latency ranging from $O(\log n)$ to $O(\log n / \log \log n)$ hops for routing tables of constant size to $O(\log n))$ size, respectively. Other approaches, such as Kelips [7], Structured Superpeers [20], Beehive [21], and CUP [23] focus on achieving constant-time $O(1)$ hops to route requests at the expense of high degree routing tables, hierarchical routing, tailoring to traffic distributions, or aggressive update protocols to maintain consistency among the large routing tables in each peer.

In this paper, we argue that the appropriate use of cached routing state within the routing protocol can provide competitive improvements in performance while using a simple baseline routing algorithm. We describe and evaluate the use of three kinds of *hint caches* containing route hints to improve the routing performance of distributed hash tables (DHTs): *local hint caches* store direct routes to successors in the ID space; *path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests; and *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space.

These hint caches require state similar to previous approaches that route requests in constant-time hops, but they do not require the complexity and communication overhead of a distributed update mechanism to maintain consistency among cached routes. Instead, the hint caches do not explicitly maintain consistency in response to peer arrivals and departures other than as straightforward extensions of the standard operations of the overlay network. We show that hint cache inconsistency does not degrade their performance benefits.

We evaluate the use of these hint caches by simulating the latest version of the Chord DHT [4] and extending it to use the three hint caches. We evaluate the effectiveness of the hint caches under a variety of conditions, including highly volatile peer turnover rates and relatively large network sizes. Based upon our simulation results, we find that the combination of the hint caches significantly improves Chord routing performance. In networks of 4,096 peers, the hint caches enable Chord to route requests with average latencies only 6% more than algorithms like “OneHop” that use complete routing tables, while requiring an order of magnitude less bandwidth to maintain the caches and without the complexity of a distributed update mechanism to maintain consistency.

The remainder of the paper is organized as follows. In Section 2, we discuss related work on improving routing performance in peer-to-peer overlays. Section 3 describes how we extend Chord to use the local, path, and global hint caches. Section 4 describes our simulation methodology for evaluating the hint caches, and presents the results of our evaluations. Finally, Section 5 summarizes our results and concludes.

2 Related Work

Initial distributed routing protocols for DHT peer-to-peer overlay networks were designed to balance routing state overhead and route maintenance overhead while providing provably attractive routing performance. Chord [25], Pastry [24], Tapestry [27], and Kademlia [18] use routing tables with degree $O(\log n)$ to route requests in $O(\log n)$ hops through the networks.

Since the performance and overhead of these networks fundamentally depend upon the distributed routing protocol, significant research has focused on the problems of improving route maintenance and performance. As a result, newer algorithms have improved the theoretical bounds on routing degree and routing hops. The most efficient of these algorithms achieve either constant degree routing state or constant hop routing latency, often, however, at the price of additional system complexity.

A number of algorithms seek to minimize route maintenance overhead by using only constant-size $O(1)$ routing tables per node, such as Viceroy [16], Symphony [17], and [9]. Several efforts have also been made to achieve constant-time $O(1)$ hops to route requests at the cost of high-degree routing tables. These approaches use gossip protocols to propagate route updates [7], hierarchical routing through structured superpeers [20], complete routing tables consistently maintained using a hierarchical update protocol [6], and reliance on traffic distributions [21].

Chord/DHash++ [4] exploits the fact that lookups for replicated values only need to reach a peer near the owner of the key associated with the value (since the peer will have a replica). Although this is appropriate for locating any one of a number of replicas, many applications require exact lookup. The “OneHop” approach uses a very aggressive update mechanism to route requests in only a single hop [6]. However, this approach requires a hierarchical update distribution tree overlayed on the DHT, and requires significant communication overhead to distribute updates to all nodes in the system. Beehive exploits the power-law property of lookup traffic distributions [21] to achieve constant-time lookups. However, a large class of applications induce different types of lookup traffic.

Perhaps the most closely related work is the Controlled Update Protocol (CUP) for managing *path caches* on peers [23]. CUP uses a query and update protocol to keep path caches consistent with peer arrivals and departures. CUP was implemented in the context of the CAN overlay network, and evaluated relative to straightforward path caches with expiration. Although the CUP path caches are analogous to the path hint caches in our work, our work differs in a number of ways with the CUP approach. Rather than providing an update mechanism to keep caches consistent, we instead combine the use of local hint caches with path and global hint caches to improve performance and tolerate inconsistency. We also evaluate hint caches with a baseline DHT routing algorithm that routes in $O(\log n)$ hops (rather than a range of coordinate dimensions).

3 Design

Distributed hash tables (DHT) increasingly serve as the foundation for a wide range of content delivery systems and applications. The DHT lookup operation is the fundamental operation on which applications base their communication. As a result, the performance of these applications directly depends on the performance of the lookup operation, and improving lookup performance improves performance for all applications layered on DHTs.

The primary goal of our work is to reduce lookup performance as close to direct routing with much less overhead than previous approaches and without relying upon specific traffic patterns. We also integrate the cache update mechanism to refresh cached route entries into the routing protocol to minimize the update complexity as well as overhead. To achieve this goal, each peer employs three *hint caches*. *Local hint caches* store direct routes to neighbors in the ID space. *Path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests. Finally, *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space. We call them hint caches since the cached routes are hints that may potentially be stale or inconsistent. We also consider them soft-state hints since they can be reconstructed quickly at any time and they are not necessary for the correctness of the routing algorithm.

The following sections describe the behavior of each of the three hint caches. Although these caches are applicable to DHTs in general, we describe them in the context of integrating them into the Chord DHT as a concrete example. So we start with a brief overview of the Chord lookup operation and routing algorithm as background.

3.1 The Chord DHT

In Chord, all peers in the overlay form a circular linked list. Each peer has one successor and one predecessor. Each peer also maintains $O(\log n)$ successors and $O(\log n)$ additional peers called *fingers*. The owner of a key is defined as a peer for which the key is in between the peer's predecessor's ID and its ID. The lookup operation for a given key returns the owner peer by successively traversing the overlay. Peers construct their finger tables such that the lookup operation traverses progressively closer to the owner in each step. In recursive lookup, the initiator peer uses its routing table to contact the closest peer to the key. This closest peer then recursively forwards the lookup request using its routing table. Included in the request is the IP address of the initiating peer. When the request reaches the peer that owns the key, that peer responds directly to the initiator. This lookup operation contacts $O(\log n)$ application level intermediate peers to reach the owner for a given key.

We augment Chord with the three hint caches. Chord uses these hint caches as simple extensions to its original routing table. When determining the next best hop to forward a request, Chord considers the entries in its original finger table as well as all entries in the hint caches.

3.2 Local Hint Caches

Local hints are direct routes to neighbors in the ID space. They are extensions of successor lists in Chord and leaf nodes in Pastry, except that their purpose is to improve routing performance. With a cache of local hints, a peer can directly reach a small fraction of peers directly and peers can short cut the final hops of request routing.

Local hints are straightforward to implement in a system like Chord using its successor lists. Normally, each peer maintains a small list of its successors to support fault-tolerance within Chord and upper layer applications. Peers request successor lists when they join the network. As part of a process called *stabilization* in Chord, each peer also periodically pings its successor to determine liveness and to receive updates of new successors in its list. This stabilization process is fundamental for maintaining lookup routing correctness, and most DHT designs perform similar processes to maintain successor liveness.

We propose enlarging these lists significantly — on the order of a thousand entries — to become local hint caches. Growing the successor lists does not introduce any additional updates, but it does consume additional bandwidth. The additional bandwidth required is $\frac{S}{H}$ entries per second where S is the number of entries in local hint cache, and H is the half life time of peers in the system. Each peer change, either joining or leaving, requires two entries to update. Similar to [14], we define the half life as the time in seconds for half of the peers in the system to either leave or join the DHT. For perspective, a study of the Overnet peer-to-peer file sharing system measured a half life of four hours [1].

The overhead of maintaining the local hint cache is quite small. For example, when S is 1000 entries and H is four hours, then each peer will receive 0.07 extra entries per second during stabilization. Since entries are relatively small (e.g., 64 bytes), this corresponds to only a couple of bytes/sec of overhead.

Local hint caches can be inconsistent due to peer arrivals and departures. When a peer fails or a new peer joins, for example, its immediate predecessor will detect the failure or join event during stabilization. It will then update its successor list, and start propagating this update backwards along the ring during subsequent rounds of stabilization. Consequently, the further one peer is from one its successors, the longer it takes that peer to learn that the successor has failed or joined.

The average amount of stale data in the local hint cache is $\frac{R*S*(S+1)}{4*H}$, where R is the stabilization period in seconds (typically one second). On average a peer accumulates $\frac{1}{2*H}$ peers per second of stale data. Since a peer updates its

x'th successor every $x * R$ seconds, it accumulates $\frac{x*R}{2*H}$ stale entries from its x'th successor. If a peer has S successors, then on average the total amount of stale data is $\sum_{i=1}^{S} \frac{i*R}{2*H}$. If the system half life time H is four hours and the local hint cache size is 1000 peers, then each peer only has 1.7% stale entries. Of course, a peer can further reduce the stale data by using additional update mechanisms, introducing additional bandwidth and complexity. Given the small impact on routing, we argue that such additions are unnecessary.

3.3 Path Hint Caches

The distributed nature of routing lookup requests requires each peer to process the lookup requests of other peers. These lookup requests are generated both by the application layered on top of the DHT as well as the DHT itself to maintain the overlay structure. In the process of handling a lookup request, a given peer can get information about other peers that contact it as well as the peer that initiated the lookup.

With Path Caching with Expiration (PCX) [23], peers cache path entries when handling lookup requests, expiring them after a time threshold. PCX caches entries to the initiator of the request as well as the result of the lookup, and the initiator caches the results of the lookup. In PCX, a peer stores routes to other peers without considering the latency between itself and these new peers. In many cases, these extra peers are far away in terms latency. Using these cached routes to peers can significantly add to the overall latency of lookups (Figure 4(a)). Hence PCX, although it reduces hops (Figure 4(b)), can also counter-intuitively increase lookup latency.

Instead, peers should be selective in terms of caching information about routes to other peers learned while handling lookups. We propose a selection criteria based on the latency to select a peer to cache it. A peer x caches a peer y if the latency to y from x is less than the latency from x to peer z, where (1) z is in x's finger table already and (2) its ID comes immediately before y's ID if x orders the IDs of its finger table peers. For example, assume y falls between a and b in x's finger table and then peer x contacts a to perform the lookup request for an ID between $(a, b]$. If we insert y, then x would contact y for the ID between $(y, b]$. Since the latency to y from x is less than the latency a from x, the lookup latency may reduce for IDs between $(y, b]$. As a result, x will cache the hop to y. We call the cache that collects such hints the path hint cache.

We would like to maintain the path hint cache without the cost of keeping entries consistent. The following cache eviction mechanism tries to achieve this goal. Since a small amount of stale data will not affect lookup performance significantly (Figure 3), a peer tries to choose a time period to evict entries in the path hint cache such that amount of stale data in its path cache is small, around 1%. The average time to accumulate d percentage of stale data in the path hint cache is $2 * d * h$ seconds, where h is the halving time [14]. Hence a peer can use this time period as the eviction time period.

Although the improvement provided by path hint caches is somewhat marginal (1–2%), we still use this information since it takes advantage of existing communication and comes free of cost.

3.4 Global Hint Caches

The goal of the global hint cache is to approximate two-hop route coverage of the entire overlay network using a set of direct routes to low-latency, or *nearby,* peers. Ideally, entries in the global hint cache provide routes to roughly equally distributed points in the ID space; for Chord, these nearby routes are to peers roughly equally distributed around the ring.

These nearby peers work particularly well in combination with the local hint caches at peers. When routing a request, a peer can forward a lookup to one of its global cache entries whose local hint cache has a direct route to the destination. With a local hint cache with 1000 entries, a global hint cache with a few thousand nodes will approximately cover an entire system of few million peers in two hops.

A peer populates its global hint cache by collecting route entries to low-latency nodes by walking the ID space. A peer x contacts a peer y from its routing table to request a peer z from y's local hint cache. The peer x can repeat this process from z until it reaches one of its local hint cache peers. We call this process *space walking.*

While choosing peer z, we have three requirements: minimizing the latency from x, minimizing x's global hint cache size, and preventing gaps in coverage due to new peer arrivals. Hence, we would like to have a large set of peers to choose from to find the closest peer, to choose the farthest peer in the y's local hint cache to minimize the global hint cache size, and to choose the closer peer in y's local hint cache to prevent gaps. To balance these three requirements, when doing a space walk to fill the global hint cache we use the second half of the successor peers in the local hint cache.

Each peer uses the following algorithm to maintain the global hint cache. Each peer maintains an index pointer into the global hint cache called the *refresh pointer.* Initially, the refresh pointer points to the first entry in the global hint cache. The peer then periodically walks through the cache and examines cache entries for staleness. The peer only refreshes a cache entry if the entry has not been used in the previous half life time period. The rate at which the peer examines entries in the global hint cache is $\frac{g}{2*d*h}$, where d is targeted percentage of stale data in the global hint cache, g is the global hint cache size, and h is the halving time. This formula is based on the formula for stale data in the path hint cache (Section 3.3).

d is a system configuration parameter, and peers can estimate h based upon peer leave events in the local hint cache. For example, if the halving time h is four hours, the global hint cache size g is 1000, and the maximum staleness d is 0.125%, then the refresh time period is 3.6 seconds. Note that if a peer uses an entry in the global hint cache to perform a lookup, it implicitly refreshes it as well and consequently reduces the overhead of maintaining the hint cache.

Scaling the system to a very large number of nodes, such as two million peers, the global hint cache would have around 4000 entries and peers would require one ping message per second to maintain 0.5% stale data in very high churn situations like one-hour halving times. Such overheads are small, even in large networks.

Peers continually maintain the local and path hint caches after they join the DHT. In contrast, a peer will only start space walking to populate its global hint cache if it receives a threshold explicit lookup requests directly from the application layer (as opposed to routing requests from other peers). The global hint cache is only useful for the lookups made by the peer itself. Hence, it is unnecessary to maintain this cache for a peer that is not making any lookup requests. Since a peer can build this cache very quickly (Figure 6), it benefits from this cache soon after it starts making application level lookups. A peer maintains the global hint cache using the above algorithm as long as it receives lookups from applications on the peer.

3.5 Discussion

Our goal is to achieve near-minimal request routing performance with significantly less overhead than previous approaches. Local hint caches require $\frac{S}{H}$ entries/sec additional stabilization bandwidth, where S is the number of entries in the local hint cache and H is the half life of the system. Path hint caches require no extra bandwidth since they incorporate information from requests sent to the peer. And, in the worst case, global hint caches require one ping message per $\frac{2*d*h}{g}$ seconds to refresh stale entries.

For comparison, in the "OneHop" approach [6] each peer periodically communicates $\frac{N}{2*H}$ entries to update its routing table, an order of magnitude more overhead. With one million peers at four hour half life time, for example, peers in "OneHop" would need to communicate at least 35 entries per second to maintain the state consistently, whereas the local hint cache requires 0.07 entries per second and one ping per 28 seconds to maintain the global hint cache.

4 Methodology and Results

In this section we describe our DHT simulator and our simulation methodology. We also define our performance metric, average space walk time, to evaluate the benefits of our hint caches on DHT routing performance.

4.1 Chord Simulator

Although the caching techniques are applicable to DHTs in general, we chose to implement and evaluate them in the context of Chord [25] due to its relative simplicity. Although the Chord group at MIT makes its simulator available for external use [10], we chose to implement our own Chord simulator together with our hint caching extensions. We implemented a Chord simulator according to the recent design in [4] that optimizes the lookup latency by choosing nearest fingers. It is an event-driven simulator that models network latencies, but assumes infinite bandwidth and no queuing in the network. Since our experiments had small bandwidth requirements, these assumptions have a negligible effect on the simulation results.

We separated the successor list and finger tables to simplify the implementation of the hint caches. During stabilization, each peer periodically pings its successor and predecessor. If it does not receive an acknowledgment to its ping, then it simply removes that peer from it tables. Each peer also periodically requests a successor list update from its immediate successor, and issues lookup requests to keep its finger table consistent. When the lookup reaches the key's owner, the initiating peer chooses as a finger the peer with the lowest latency among the peers in the owner's successor list.

For our experiments, we used a period of one second to ping the successor and predecessor and a 15 minute time period to refresh the fingers. A finger is refreshed immediately if a peer detects that the finger has left the DHT while performing the lookup operation. These time periods are same as ones used in the Chord implementation [10].

To compare different approaches, we want to evaluate the potential performance of a peer's routing table for a given approach. We do this by defining a new metric called *space walk latency*. The space walk latency for a peer is the average time it takes to perform a lookup to any other peer on the DHT at a given point of time. We define a similar metric, *space walk hops,* in terms of hops rather than latency. The space walk time is a more complete measurement than a few thousands of random sample lookups because space walk time represent lookup times to all peers in the network.

We simulate experiments in three phases: an initial phase, a stabilization phase, and an experiment phase. The initial phase builds the Chord ring of a specified number of nodes, where nodes join the ring at the rate of 50 nodes per second. The stabilization phase settles the Chord ring over 15 minutes and establishes a stable baseline for the Chord routing data structures. The experimental phase simulates the peer request workload and peer arrival and departure patterns for a specified duration. The simulator collects results to evaluate the hint caching techniques only during the experimental phase.

Because membership churn is an important aspect of overlay networks, we study the performance of the hint caches using three different churn scenarios: twenty-four-hour, four-hour, and one-hour half life times. The twenty-four-hour half life time represent the churn in a distributed file system with many stable corporate/university peers [26]. The four-hour half life time represent the churn in a file sharing peer-to-peer network with many home users [19]. And the one-hour half life time represent extremely aggressive worst-case churn scenarios [6].

For the simulations in this paper, we use an overlay network of 8,192 peers with latency characteristics derived from real measurements. We start with the latencies measured as part of the Vivaldi [3] evaluation using the King [12] measurement technique. This data set has approximately 1,700 DNS servers, but only has complete all-pair latency information for 468 of the DNS servers. To simulate a larger network, for each one of these 468 DNS servers we create roughly 16 additional peers to represent peers in the same stub networks as the DNS servers. We create these additional peers to form a network of 8,192 peers. We model the latency among hosts within the group as zero to correspond to the minimal latency among hosts in the same network. We model the latency among hosts between groups according to the measured latencies from the Vivaldi data set. The minimum, average, and maximum latencies among groups are 2, 165, and 795 milliseconds, respectively. As a timeout value for detecting failed peers, we use a single round trip time to that peer (according to the optimizations in [4]).

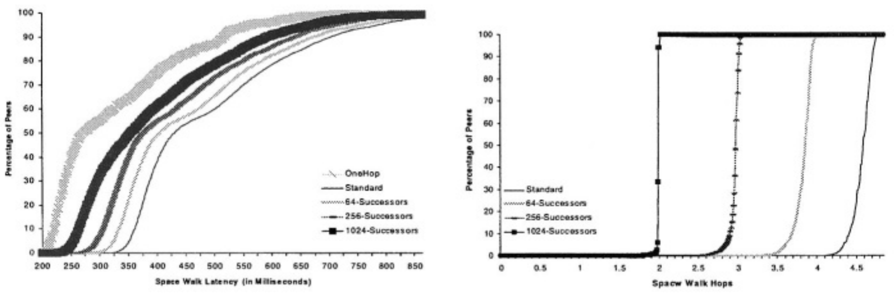
Using measurements to create the network model adds realism to the evaluation. At the same time, though, the evaluation only scales to the limits of the measurements. To study the hint caches on systems of much larger scale, we also performed experiments using another network model. First, we created a matrix of network latency among 8,192 groups by randomly assigning a latency between two groups from the range of 10 to 500 milliseconds. We then created an overlay network of 65,536 peers by randomly assigning each peer to one group, keeping the groups balanced. We do not present the results here due to space limitations. But the results using the randomly assigned latencies are qualitatively similar to the results using the Vivaldi data set, and we prefer instead to report in detail the results using the network model derived from the Vivaldi data set.

4.2 Local Hint Caches

In this experiment we evaluate the performance of the local hint cache compared with two baseline routing algorithms, "Standard" and "OneHop." "Standard" is the default routing algorithm in Chord++ [4] that optimized for lookup latency by choosing nearest fingers. "OneHop" maintains complete routing tables on all peers [6].

Figures 1(a) and 1(b) show the cumulative distributions for the space walk latencies and hops, respectively, across all peers in the system. Since there is no churn in this experiment, we calculate the latencies and hops after the network stabilizes when reaching the experimental phase; we address churn in the next experiment. Figure 1(a) shows results for "Standard" and "OneHop" and local hint cache sizes ranging from 64–1024 successors; Figure 1(b) omits "OneHop" since it only requires one hop for all lookups with stable routing tables.

Figure 1(a) shows that the local hint caches improve routing performance over the Chord baseline, and that doubling the cache size roughly improves space walk latency by a linear amount. The median space walk latency drops from 432 ms in Chord to 355 ms with 1024 successors in the local hint cache (a decrease of 18%). Although an improvement, the local hint cache alone is still substantially slower than "OneHop", which has a median space walk latency of 273 ms (a decrease of 37% is needed).



(a) Latency distributions (b) Hop count distributions

Fig. 1. Sensitivity of local hint cache size on lookup performance.

Figure 1(b) shows similar behavior for local hint caches in terms of hops. A local hint cache with 1024 successors decreases median space walk latency by 2.5 hops, although using such a cache still requires one more hop than "OneHop".

From the graph, we see that doubling the local hint cache size improves the number of hops by at most 0.5. Doubling the local hint cache size reduces hop count by one for half of the peers, and the remaining half does not benefit from the increase. For example, consider a network of 100 peers where each peer maintains 50 other peers in its local hint cache. For each peer, 50 peers are one hop away and the other 50 peers are two hops away. As a result, the space walk hop distance is 1.5 hops. If we increase the local hint cache to 100 peers, then each peer reduces the hop distance for only the 50 peers that were two hops away in the original scenario. In this case, the space walk hop distance is 1.

When there is no churn in the system, the lookup performance when measured in terms of hops either remains the same or improves when we double the local hint cache size. The results are more complicated when we measure lookup performance in terms of latency. Most peers improves their lookup latencies to other peers and, on average, increasing local hint cache improves the space walk latency. However, lookup latency to individual peers can increase when we double the local hint cache size in some cases. This is because Internet routing does not necessarily follow the triangular inequality: routing through multiple hops may have lower latency than a direct route between two peers. Since we derive our network latency model from Internet measurements, our latency results reflect this characteristic of Internet routing.

4.3 Staleness in Local Hint Caches

The previous experiment measured the benefit of using the local hint cache in a stable network, and we now measure the cost in terms of stale entries in the local hint cache and the effect of staleness on lookup performance.

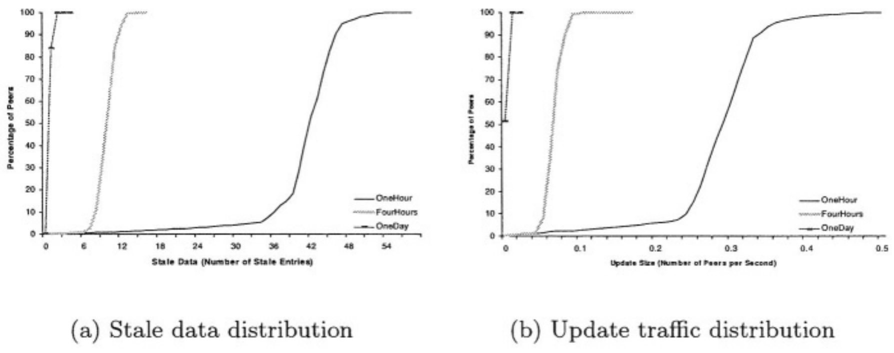


Fig. 2. Stale data and update traffic under various churn situations.

In this experiment, we use a local hint cache size of 1024 successors. To calculate stale data in local hint caches, we simulated a network of 4096 peers for an experimental phase of 30 minutes. During the experimental phase, the network experiences churn in terms of peer joins and leaves. We vary peer churn in the network by varying the half life of peers in the system from one hour to one day; we select nodes to join or leave from a uniform distribution.

Figure 2(a) shows the fraction of stale entries in local hint caches for various system half life times as a cumulative distribution across all peers. We calculated the fraction of stale entries by sampling each peer's local hint cache every second and determining the number of stale entries. We then averaged the samples across the entire simulation run. Each point (x, y) on a curve indicates that y percentage of peers have at most $x\%$ stale data in their local hint caches. As expected, the amount of stale data increases as the churn increases. Note that the amount of stale data is always less than the amount calculated from our analysis in Section 3.2 since the analysis conservatively assumes worst case update synchronization.

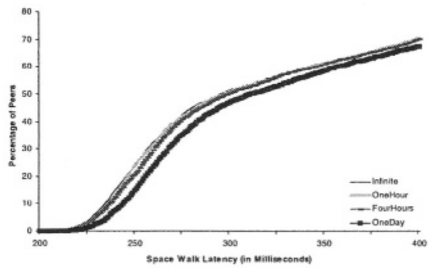


Fig. 3. Space walk latency distributions under various churn situations.

Figure 3 shows the effect of stale local hint cache entries on lookup performance across all peers. It shows results for the same system half life times as Figure 2(a) and adds results for an “Infinite” half life time. An “Infinite” half life means that there is no churn, no stale entries in the hint cache, and therefore represents the best-case latency. At the end of the experiment phase in the simulation, we used the state of each peer’s routing table to calculate the distribution of space walk latencies across all peers. Each point (x, y) in the figure indicates that y percentage of peers have at most x space walk latency. We cut off the y-axis at 75% of peers to highlight the difference between the various curves.

The space walk latencies for a four hour half life time are similar to the latencies from the ideal case with no churn (medians differ by only 1.35%). From these results we conclude that the small amount of stale data (1–2%) does not significantly degrade lookup performance, and that the local hint cache update mechanism maintains fresh entries well. As the churn rate increases, stale data increases and lookup performance also suffers. At an one hour half life, lookup performance increases moderately.

Note that the “Infinite” half life time curve in Figure 3 performs better than the 1024 successors curve in Figure 1(a) even though one would expect them to be the same. The reason they differ is that the finger table entries in these two cases are different. When we evaluated the local hint cache, we used a routing table with 13 successors and added the remaining successors to create the local hint cache without changing the finger table. When we evaluated the stale data effects in the local hint cache we have 1024 successors from which to choose “nearest” fingers. As a result, the performance is better.

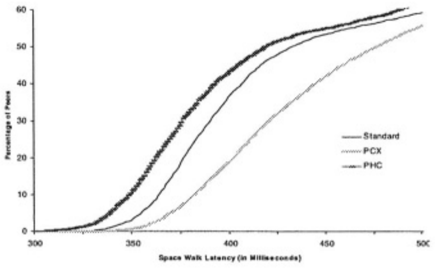
4.4 Update Traffic

In the previous section we evaluated the effect of stale data on lookup performance under various churn scenarios. In this section we evaluate the update traffic load under various churn scenarios to evaluate the update traffic bandwidth required by large local hint caches.

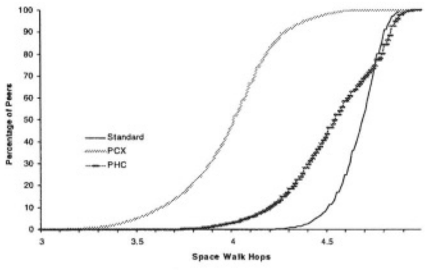
We performed a similar experiment as in Section 4.3. However, instead of measuring stale data entries we measured the update traffic size. We calculated the average of all update samples per second for each peer over its lifetime in terms of the number of entries communicated. Figure 2(b) presents this average for all peers as cumulative distributions. Each curve in Figure 2(b) corresponds to a different churn scenario. A point (x, y) on each curve represent the y percentage of peers that have at most x entries of average update traffic. The average update traffic closely matches the estimate from our analysis in Section3.2. The average update traffic (0.4 entries/second) is extremely low even under worst case conditions. Hence, this traffic does not impose a burden on the system.

4.5 Path Hint Caches

Next we evaluate the performance of the path hint cache (PHC) described in Section 3.3 compared to path caching with expiration (PCX) as well as Chord. PCX



(a) Latency Distribution



(b) Hop count distributions

Fig. 4. Lookup performance of path caching with expiration (PCX), path hint cache (PHC), and standard Chord.

is the technique of caching path entries described in [23]. When handling lookup requests on behalf of other nodes, PCX caches route entries to the initiator of the request as well as the result of the lookup.

In this experiment, we simulate a network of 4096 peers with a 30-minute experimental phase. We study the lower-bound effect of the path hint caches in that we do not perform any application level lookups. Instead, the path hint caches are only populated by traffic resulting from network stabilization. We did not simulate application level lookup traffic to separate its effects on cache performance; with applications performing lookups, the path hint caches may provide more benefit, although it will likely be application-dependent. Since there is no churn, cache entries never expire. To focus on the effect of path caches only, we used a local hint cache size of 13, the size of the standard Chord successor list, and no global hint cache. We collected the routing tables for all peers at the end of the simulations and calculated the space walk latencies and hops.

Figure 4(a) shows the cumulative distribution of space walk latencies across all peers at the end of the simulation for the various path caches and standard Chord. Each point (x, y) in this figure indicates that y percent peers have at most x space walk latency. From these results we see that, as expected, the path hint cache improves latency only marginally. However, the path hint cache is essentially free, requiring no communication overhead and a small amount of memory to maintain.

We also see that PCX performs worse even than Chord. The reason for this is that PCX optimizes for hops and caches routing information independent of the latency between the caching peer and the peer being cached. The latest version of Chord and our path hint caches use latency to determine what entries to place and use in the caches and in the routing tables. For peers with high latency, it is often better to use additional hops through low-latency peers than fewer hops through high-latency peers.

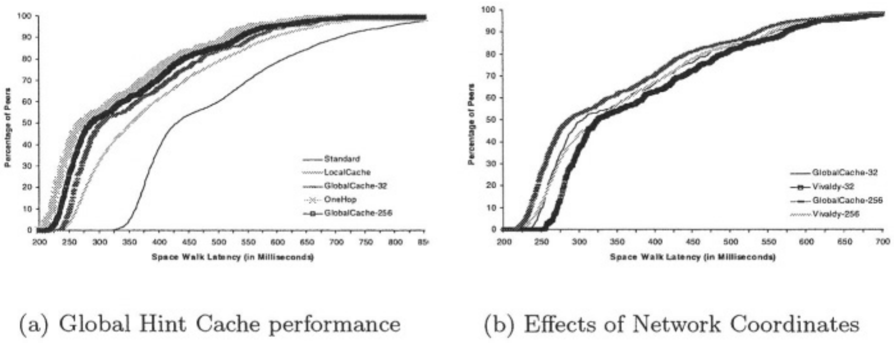


Fig. 5. Global hint cache performance in ideal and practical situations.

Figure 4(b) shows this effect as well by presenting the cumulative distribution of space walk hops across all peers for the various algorithms. Each point (x, y) in this figure indicates that y percent peers have at most x space walk hops. Using the metric of hops, PCX performs better than both Chord and PHC. Similar to results in previous work incorporating latency into the analysis, these results again demonstrate that improving hop count does not necessarily improve latency. Choosing routing table and cache entries in terms of latency is important for improving performance.

The path hint cache are small and each peer aggressively evicts the cache entries to minimize the stale data. Hence the effects of stale data on lookup request performance is marginal.

4.6 Global Hint Caches

Finally, we evaluate the performance of using the global hint cache together with the local and path hint caches. We compare its performance with "Standard" Chord and "OneHop". In this experiment, we simulated 4092 peers with both 32 and 256 entries in their local hint caches. We have two different configurations of local hint caches to compare the extent to which the global hint cache benefits from having more candidate peers from which to select nearby peers to place in the global hint cache. (Note that the global hint cache uses only the second half of the nodes in the local hint cache to select nearest nodes; hence, the global hint cache uses only 16 and 128 entries to choose nearest peers in the above two cases.) Each peer constructs its global cache when it joined the network as described in Section 3.4. We collected the peer's routing tables once the network reached a stable state during the experimentation phase, and calculated the space walk latencies for each peer from the tables.

Figure 5(a) shows the cumulative distributions of space walk latencies across all peers for the various algorithms. The "Standard", "OneHop", "LocalCache", "GlobalCache-32", and "GlobalCache-256" curves represent Chord, the "One-Hop" approach, a 1024-entry local hint cache, a 32-entry global hint cache with a 256-entry local hint cache, and a 256-entry global hint cache with a 32-entry local hint cache. Comparing the size of local hint caches used to populate the global hint cache, we find that the median space walk latency of "GlobalCache-256" is 287 ms and "GlobalCache-32" is 305 ms; the performance of the global hint cache improved only 6% when it has more peers in the local hint cache to choose the nearest peer.

Comparing algorithms, we find that the median latency of the global hint cache comes within 6% of the "OneHop" approach when the global hint cache uses 128 of 256 entries in the local hint cache to choose nearby peers. Although these results are from a stable system without churn, the global hint cache performance under churn is similar to the local hint cache performance under churn because both of them use a similar update mechanism. As a result, the effect of stale data in the global hint cache is negligible for a one-day system half life time and four-hour system half life time. Overall, our soft-state approach approaches the lookup latency of algorithms like "OneHop" that use significantly more communication overhead to maintain complete routing tables.

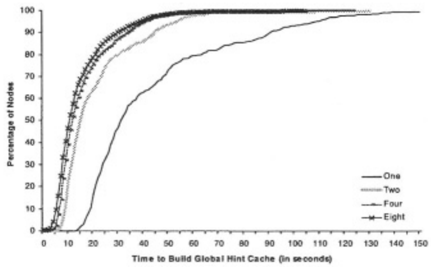


Fig. 6. Global hint cache build time

Since we contact closer peers while constructing the global hint cache, one can build this cache within a few minutes. To demonstrate this, we calculated the time to build the global hint cache for 4096 peers. Figure 6 presents the results of this experiment as a distribution of cache build times. Each point (x, y) on a curve indicates that y percentage of peers needs at most x seconds to build the cache. Each peer has around 500 peers in its the global hint caches. On the average it took 45 seconds to build the global hint cache. A peer can speed up this process by initiating walks from multiple peers from its routing table in parallel. The curves labeled "Two", "Four", and "Eight" represent the cache build times with two, four, and eight parallel walks, respectively. As expected, cache build time reduces as we increase the number of parallel walks. The median reduces from 32 seconds for single walk to 12 seconds for four parallel walks. We see only a small benefit of increasing the parallel walks after four parallel walks.

So far we have assumed that, when populating the global hint caches, peers are aware of the latencies among all other peers in the system. As a result, the results represent upper bounds. In practice, peers will likely only track the latencies of other peers they communicate with, and not have detailed knowledge of latencies among arbitrary peers. One way to solve this problem is to use a distributed network coordinate system such as Vivaldi [3]. Of course, network coordinate systems introduce some error in the latency prediction. To study the effect of coordinate systems for populating global hint caches, we next use Vivaldi to estimate peer latencies.

In our simulation we selected the nearest node according to network latency estimated according to the Vivaldi network coordinate system, but calculated the space walk time using actual network latency. We did this for the "GlobalCache-32" and "GlobalCache-256" curves in Figure 5(a). Figure 5(b) shows these curves and the results using Vivaldi coordinates as "Vivaldi-32" and "Vivaldi-256". The performance using the coordinate system decreases 6% on average in both cases, showing that the coordinate system performs well in practice.

5 Conclusions

In this paper, we describe and evaluate the use of three kinds of *hint caches* containing route hints to improve the routing performance of distributed hash tables (DHTs): *local hint caches* store direct routes to neighbors in the ID space; *path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests; and *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space.

We simulate the effectiveness of these hint caches as extensions to the Chord DHT. Based upon our simulation results, we find that the combination of hint caches significantly improves Chord routing performance with little overhead. For example, in networks of 4,096 peers, the hint caches enable Chord to route requests with average latencies only 6% more than algorithms like "OneHop" that use complete routing tables, while requiring an order of magnitude less bandwidth to maintain the caches and without the complexity of a distributed update mechanism to maintain consistency.

Acknowledgments. We would like thank the anonymous reviewers for their valuable feedback for improving this paper. We would also like to express our gratitude to Marvin McNett for system support for performing our simulation experiments, and to Frank Dabek for providing us with the network latency information used in all simulations. Support for this work was provided in part by AFOSR MURI Contract F49620-02-1-0233 and DARPA FTN Contract N66001-01-1-8933.

References

1. R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *2nd International Workshop on Peer-to-Peer Systems,* Feb. 2003.

2. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *19th ACM Symposium on Operating Systems Principles,* Oct. 2003.
3. R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, distributed network coordinates. In *proceedings of Second Workshop on Hot Topics in Networks,* Nov. 2003.
4. F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *ACM/USENIX Symposium on Networked Systems Design and Implementation,* Mar. 2004.
5. F. D. Emil Sit and J. Robertson. Usenetdht: A low overhead Usenet server. In *3rd International Workshop on Peer-to-Peer Systems,* Feb. 2004.
6. A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *A CM/USENIX Symposium on Networked Systems Design and Implementation,* Mar. 2004.
7. I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *2nd International Workshop on Peer-to-Peer Systems,* Feb. 2003.
8. S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized and peer-to-peer web cache. In *21st ACM Symposium on Principles of Distributed Computing,* July 2002.
9. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal hash table. In *2nd International Workshop on Peer-to-Peer Systems,* Feb. 2003.
10. M. F. Kaashoek and R. Morris. http://www.pdos.lcs.mit.edu/chord/.
11. D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *19th ACM Symposium on Operating Systems Principles,* Oct. 2003.
12. S. S. Krishna P. Gummadi and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *2nd Internet Measurement Workshop,* Nov. 2002.
13. J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *2nd International Workshop on Peer-to-Peer Systems,* Feb. 2003.
14. D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *First International Workshop on Peer-to-Peer Systems,* Mar. 2002.
15. B. T. Loo, S. Krishnamurthy, and O. Cooper. Distributed web crawling over dhts. Technical Report UCB/CSD-4-1305, UC Berkeley, 2004.
16. D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *21st ACM Symposium on Principles of Distributed Computing,* July 2002.
17. G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *4th USENIX Symposium on Internet Technologies and Systems,* Mar. 2003.
18. P. Maymounkov and D. Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In *1st International Workshop on Peer-to-Peer Systems,* Mar. 2002.
19. J. McCaleb. http://www.overnet.com/.
20. A. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *IEEE Workshop on Internet Applications,* June 2003.

21. V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *ACM/USENIX Symposium on Networked Systems Design and Implementation,* Mar. 2004.
22. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *proceedings of ACM SIGCOMM,* Aug. 2001.
23. M. Roussopoulos and M. Baker. Cup: Controlled update propagation in peer-to-peer networks. In *USENIX Annual Technical Conference,* June 2003.
24. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware),* Nov. 2001.
25. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *proceedings of ACM SIGCOMM,* Aug. 2001.
26. D. E. William J. Bolosky, John R. Douceur and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of SIGMETRICS,* June 2000.
27. B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.

Distributed Hashtable on Pre-structured Overlay Networks*

Kai Shen[1] and Yuan Sun[2]

[1] Department of Computer Science, University of Rochester, Rochester, NY 14627, USA
kshen@cs.rochester.edu
[2] Department of Computer Science, University of California, Santa Barbara, CA 93106, USA
suny@cs.ucsb.edu

Abstract. Internet overlay services must adapt to the substrate network topology and link properties to achieve high performance. A common overlay structure management layer is desirable for enhancing the architectural modularity of service design and deployment. A shared substrate-aware overlay structure can also save redundant per-service link-selection probing when overlay nodes participate in multiple services. Despite the benefits, the concept of building services on a common structure management layer does not work well with recently proposed scalable distributed hashtable (DHT) protocols that employ protocol-specific overlay structures. In this paper, we present the design of a self-organizing DHT protocol based on the Landmark Hierarchy. Coupled with a simple low-latency overlay structure management protocol, this approach can support low-latency DHT lookup without any service-specific requirement on the overlay structure. Using simulations and experimentation on 51 PlanetLab sites, we measure the performance of the proposed scheme in terms of lookup latency, load balance, and stability during node chums.

1 Introduction

Internet overlay services may suffer poor performance when they ignore the topology and link properties of the substrate network. Various service-specific techniques have been proposed to adapt to Internet properties by selecting overlay routes with low latency or high bandwidth. Notable examples include the unicast overlay path selection [1] and measurement-based end-system multicast protocols [5]. The employment of a common software layer that maintains overlay connectivity structure can greatly ease the design and deployment of overlay services. For instance, a more effective link probing technique or a new partition repair protocol can be incorporated into the common structure layer such that a large number of overlay services can benefit transparently. Furthermore, a common substrate-aware overlay structure layer can reduce redundant service-specific link-selection probing when overlay nodes participate in multiple services. Early experience on PlanetLab [2] indicates that link-selection probing can consume significant network resources. With a common structure management layer, upper-level services built on top of it can share the cost of structure maintenance.

* This work was supported in part by the National Science Foundation grants CCR-0306473 and ITR/IIS-0312925.

The key for overlay services to take advantage of a common structure management layer is that they must be able to function on pre-structured overlay networks. In other words, these services must not dictate how overlay links are structured in any service-specific way. This requirement fits well with services such as unstructured peer-to-peer search (*e.g.,* Gnutella and random walks [12]). This layer can also benefit unicast or multicast overlay path selection services (*e.g.,* RON [1] and Narada [5]). For instance, Narada employs a DVMRP-style multicast routing protocol running on top of a low-latency pre-structured overlay network. It achieves high performance without additional link-selection probing beyond the structure management layer.

Despite these benefits, it is unclear how a given substrate-aware overlay structure can assist the construction of the distributed hashtable (DHT) service. A DHT is a self-organizing overlay network of hosts that supports insertion, deletion, and lookup with hash keys. The heart of a DHT protocol is a distributed lookup scheme that maps each hash key into a deterministic location with well balanced object placement and runtime overhead. Recently proposed scalable DHT protocols such as Chord [21], CAN [16], and Pastry [19] are all *strongly structured, i.e.,* they place protocol-specific requirements on overlay connectivity structures. As a result, substrate-aware link-selection enhancements designed for specific DHT protocols [3,17,26,27] cannot benefit other services.

This paper examines the construction of a scalable DHT service on top of a service-independent structure management layer. The rest of this paper is organized as follows. Section 2 describes a low-latency structure management layer that our DHT construction can rely on. We then present the design of a hierarchical DHT protocol that operates on pre-structured overlays in section 3. Section 4 describes evaluation results based on simulations. Our implementation and experimentation on the PlanetLab testbed is reported in section 5. Section 6 discusses related work and section 7 concludes the paper.

2 Service-Independent Overlay Structure Management

The proposed DHT construction is based on our earlier design of a common structure management layer, called *Saxons* [20]. The Saxons overlay structure management layer contains six components, as illustrated in Figure 1. The bootstrap process determines how new nodes join the overlay structure. The structure quality maintenance component maintains a high quality overlay mesh while the connectivity support component actively detects and repairs overlay partitions. They run periodically to accommodate dynamic changes in the system. The above Saxons components are all supported by the membership management component that tracks a random subset of overlay members. The structure quality maintenance is further supported by two other components responsible for acquiring performance measurement data for overlay links and finding nearby overlay hosts. Below we briefly describe the low-latency structure quality maintenance component that is most related to our DHT construction in this paper. A more complete description of the Saxons structure management layer can be found in [20].

The quality maintenance component runs at a certain link density, specified by a node degree range $<d_a - d_t>$. Each node makes d_a active overlay links and it can also accept a number of passive links as long as the total degree does not exceed d_t. The

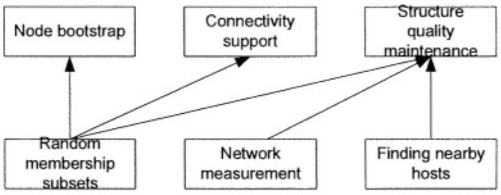


Fig. 1. Saxons components.

degree upper-bound is maintained to control the stress on each node's physical access links and limit the impact of a node failure on the Saxons structure. The heart of the Saxons structure quality maintenance is a periodic routine that continuously adjusts for potentially better structure quality. Specifically, it checks the distance to hosts in the local random membership subset and replaces the longest existing links if new hosts are closer. In addition to quality-oriented link adjustments, each node also periodically pings its Saxons neighbors. A neighbor link will be replaced when several consecutive pings fail.

The overlay structure stability is important for the performance of overlay services that maintain link-related state, including the DHT service we describe in the next section. To avoid frequent structure changes, we require that a link adjustment occurs only when a new link is shorter than the existing one for more than a specified threshold. It is also possible to disable the quality-oriented structure changes after a node bootstrap to further enhance the structure stability. It should be noted that runtime structure changes cannot be completely avoided at the presence of overlay membership changes.

A critical problem for latency-oriented structure optimization is to efficiently acquire the latency performance data. Previous studies have proposed various techniques for estimating Internet host distances [7,13] or locating nearby hosts [17]; Saxons can utilize any of them for latency estimation. In particular, we point out a *landmark-based Cartesian distance* approach for its simplicity. This approach requires a set of l well-known landmark hosts spread across the network and each landmark defines an axis in an l-dimensional Cartesian space. Each group member measures its latencies to these landmarks and the l-element latency vector represents its coordinates in the Cartesian space. For nearby host selection, a node chooses the one to which its Cartesian distance is minimum. This approach has been shown to be competitive to a number of other landmark-based schemes [17]. Using landmark-based latency estimation for structure quality maintenance, link probing is only necessary at node startup to measure latency to a few designated landmark nodes. No additional runtime network overhead would be needed for the structure quality maintenance. Other Saxons components incur a low per-node runtime network overhead of around 1.3Kbps [20].

3 DHT on Pre-structured Overlay Networks

A common structure management layer, such as Saxons, can greatly ease the construction of overlay services with better design modularity and shared structure maintenance cost. In this section, we investigate how distributed hashtable can be constructed based on this layer. Our design considers common DHT service objectives as in other DHT protocols [16,19,21]: self-organization and automatic adaptation; scalability in supporting large overlay groups; high-performance DHT lookup; balanced key placement and lookup routing overhead. Note that our key contribution is that the proposed DHT can function on pre-structured overlay networks while the previous solutions require protocol-specific structures.

Distributed hashtable on pre-structured networks resembles network routing in subtle ways. A DHT lookup can be considered as a network routing request with its destination labeled with a hash key instead of a host ID. Our DHT design can draw upon earlier efforts in designing scalable network routing protocols. In particular, we choose to base our design on the Landmark Hierarchy [22][1], due to its potential of self-organization and automatic adaptation to overlay membership changes. Under this context, the load balance objective is especially difficult to achieve because of the hierarchical nature of such scheme.

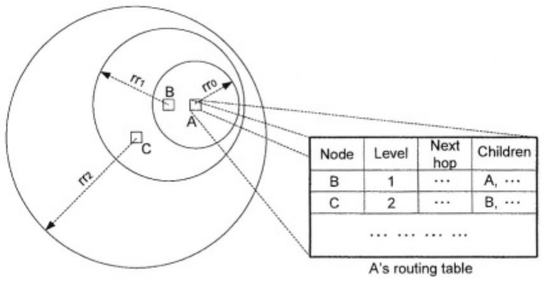
3.1 Landmark Hierarchy on Overlay Networks

Our concept of Landmark Hierarchy mostly follows that of the original Landmark Hierarchy [22], with necessary changes to support the DHT service. A *Landmark* is a node whose neighbors within a certain number of hops (called *routing radius*) contain a routing entry for it. This is usually achieved by having each landmark periodically flood a route advertisement message along the overlay structure for up to a hop-count bound of its routing radius. For our DHT protocol, all overlay nodes form a hierarchy of landmarks, with level 0 being the lowest level, and level H being the highest level. Every overlay node is at least a level 0 landmark. All landmarks in the same level have the same routing radius. A higher-level routing radius is always larger than a lower-level one and the level H landmarks flood their route advertisements to the complete overlay. Let rr_l be the level l routing radius. Therefore we have $rr_{l+1} > rr_l$ for each $0 \le l < H$; and $rr_H = \infty$.

We call a child-parent relationship exists between nodes A and B when A is one-level lower than B in the Landmark Hierarchy and they are within each other's routing radii, *i.e.,* they have routing entries for each other. We require that all nodes except those at the top level have at least a parent. Note that a node may have multiple parents in the hierarchy. We also require each node carry IDs of all its children in its route advertisements and subsequently they are stored as part of the routing entry at nodes within the routing radius. This information is critical for our DHT construction, though it is not needed for network routing in the original Landmark Hierarchy [22]. Figure 2 illustrates an example of our Landmark Hierarchy and its routing table layout.

Having small routing tables is a key benefit for hierarchical routing schemes. Kleinrock and Kamoun found that the average number of routing table entries in an H-level hierarchy with a single top-level node is at best $H \times N^{\frac{1}{H}}$, where N is the total number of nodes in the hierarchy [10]. This may only be achieved when every landmark (except level 0 nodes) has the same number of children and no landmark has more than one

[1] Do not confuse the *Landmark* Hierarchy with the *landmark-based* Cartesian distance approach we use for latency estimation.



Node	Level	Next hop	Children
B	1	···	A, ···
C	2	···	B, ···
···	···	···	···

Fig.2. An exemplar Landmark Hierarchy and its routing table. Nodes *A, B,* and *C* are level 0, 1, and 2 landmarks respectively.

parent. In practice, the routing table sizes are larger and we will examine this in the performance evaluation in section 4.3.

3.2 Hierarchy Construction and Adaptation

We now describe an automatic hierarchy construction and adaption scheme. The goal is to dynamically maintain a balanced hierarchy with exponentially smaller population at higher levels. All overlay nodes start at level 0 with a routing radius of rr_0 hops. Each node sends out periodic routing advertisements at interval t_{int} to other nodes within its routing radius. Routing entries are kept as soft state and are refreshed upon the reception of these advertisements; they expire after not being refreshed for several rounds. Periodically, every node checks the existence of an unexpired routing entry for a parent. If a parent routing entry does not exist and the hierarchy level-bound has not be reached, it schedules a *promotional* procedure at a random delay and increases its landmark level by one. This random delay is chosen uniformly from $[t_{int} + t_{delay}, (1 + \alpha N_{peer})t_{int} + t_{delay}]$, where t_{delay} is the estimated message propagation delay upper-bound in the overlay network, α is a constant, and N_{peer} is the number of same-level peers in the local routing table. The linear back-off component on N_{peer} is employed to prevent many nodes in a densely connected area to promote themselves simultaneously. The scheduled promotional procedure is canceled when a routing advertisement from a parent is later received.

When the hierarchy level-bound is large, it is desirable to stop the hierarchy buildup when there is only a single top-level landmark node. Following an idea presented in [11], a node without any same-level peer in its routing table never promotes itself. This scheme works when each node sees at least one same-level peer if any exists, which can be ensured by having $rr_{l+1} > 2 \times rr_l$ for all $l \geq 0$. However, one drawback with this approach is that the routing radii increase too fast for high levels, resulting in large number of children nodes at high levels. We attempt to avoid this problem by introducing a peer notification radius (denoted by pr_l at level l) independent of the routing radius. Each level l landmark floods the overlay network with a peer notification announcement for up to pr_l hops. We require $pr_{l+1} > 2 * rr_l$ such that each node sees at least one same-level peer if any exists.

A node may want to lower its hierarchy level after some other nodes depart from the overlay or an earlier promotion has been pre-mature. We employ two demotion rules in the automatic hierarchy adaptation.

Rule 1: Each node periodically checks the existence of an unexpired routing entry for a child. When discovering no child is present, it schedules a *demotional* procedure at the delay of $t_{int} + t_{delay}$. We use a constant scheduling delay because no back-off is necessary in this case.

Rule 2: Each node also checks its routing table for whether a hierarchy peer can serve as a parent if it demotes itself. This is the case when the hop-count distance to one of the peers is within the routing radius of the hierarchy level after the demotion. If so, a demotional procedure is scheduled at a random delay between $[t_{int} + t_{delay}, (1 + \beta N_{peer})t_{int} + t_{delay}]$. The linear back-off component on N_{peer} is employed to prevent all peers to demote themselves simultaneously. This demotion rule ensures that no two level l landmarks are within the distance of rr_{l-1} from each other.

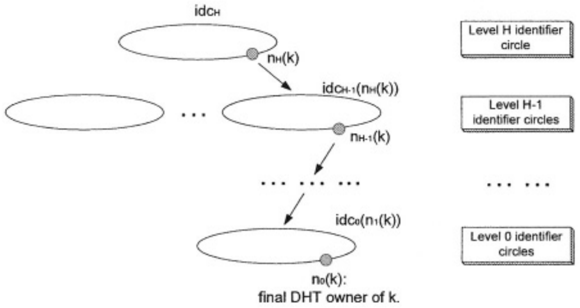
3.3 DHT on Landmark Hierarchy

Based on the constructed landmark hierarchy, we present the design of the proposed DHT protocol in this section. We first describe the mapping scheme between each hash key and a deterministic host in our DHT protocol. We then present a distributed algorithm for any node to find such location with a given hash key.

One of the building blocks in our DHT mapping is the Chord identifier circle [21]. In Chord, each overlay node and key is assigned an *identifier* in <0 $- ID_{max}$> using an ID assignment function such as SHA-1 or MD5. A node's identifier is chosen by hashing the node's IP address, while a key identifier is generated by hashing the key. All identifiers are ordered in an identifier circle modulo $ID_{max} + 1$. Key k is assigned to the first node whose identifier is equal to or follows k's in the identifier space, called $owner(k.id)$. If identifiers are represented as a circle of numbers from 0 to ID_{max}, then $owner(k.id)$ is the first node clockwise from k.

Instead of a single identifier circle, our protocol employs a hierarchy of identifier circles to map hash keys to overlay nodes. First, all top level (*e.g.,* level *H*) landmarks form a level *H* identifier circle (denoted by idc_H). In addition to the top level identifier circle, all children of each level $l + 1$ landmark X $(0 \leq l < H)$ form a level l identifier circle (denoted by $idc_l(X)$). Note that there are typically multiple identifier circles in each level below level *H*. Each hash key k is first mapped to a level *H* landmark node (denoted by $n_H(k)$) in the top level identifier circle. It is then subsequently mapped to a level *H* – 1 landmark in $n_H(k)$'s children identifier circle. This process continues until the hash key is eventually hashed into a level 0 landmark $n_0(k)$, which is considered as the key's final owner. A disadvantage of this scheme is that hash keys mapped to a particular landmark are close to each other in the identifier circle. Therefore these keys would always map into the same region in subsequent lower-lever identifier circles, resulting in unbalanced key placement. To avoid this problem, we use different key-identifier assignment functions at each level such that keys with close identifiers in one level are spread out in the identifier circles for all other levels. We use $\mathrm{MD5}_l()$ to denote the ID assignment function at level l. Equation (1) and Figure 3 illustrate our DHT mapping scheme at each level. Note again that the level 0 DHT owner $n_0(k)$ is considered as k's final owner.

$$n_l(k) = \begin{cases} idc_H.owner(MD5_H(k)) & \text{if } l = H, \\ idc_l(n_{l+1}(k)).owner(MD5_l(k)) & \text{if } 0 \leq l < H. \end{cases} \tag{1}$$



idcH
nH(k)
idcH-1(nH(k))
nH-1(k)
idc0(n1(k))
n0(k): final DHT owner of k.
Level H identifier circle
Level H-1 identifier circles
Level 0 identifier circles

Fig.3. Illustration of the proposed DHT mapping scheme.

We now describe a distributed lookup algorithm to implement the DHT mapping described above. For each given hash key k, the lookup initiator node A first finds k's level H owner $n_H(k)$ in the top level identifier circle. This can be performed locally at every node since the identifiers of top level landmarks are known to all through their route advertisements. Because all children identifiers are carried in each route advertisement, A is also able to locally find k's level $H-1$ owner $n_{H-1}(k)$ in n_H's children identifier circle. If A has a routing entry for $n_{H-1}(k)$ (and therefore the identifiers of all its children), A would continue to perform lookup for lower-level DHT owners. When A's local lookup stops because it does not have the routing entry for k's level $l-1$ DHT owner $n_{l-1}(k)$, A forwards the lookup query to its next hop node toward $n_l(k)$, which it has a routing entry for. This process continues until $n_0(k)$ is located. Figure 4 illustrates this algorithm in recursive form. This algorithm is invoked at the lookup initiator node with DHT_LOOKUP(*key*, *H*, $n_H(key)$).

Note that lookup queries normally do not go through high-level DHT owners before finding the final level 0 owner. This is because a lookup query aiming at the level l DHT owner shifts toward the level $l-1$ owner as soon as it moves within its routing radius. This is more so when rr_i is much larger than rr_{i-1}. This behavior is essential for offloading higher-level landmarks in terms of lookup routing overhead.

4 Simulation Results

Our performance evaluation consists of simulations and Internet experiments. The goal of simulation studies is to assess the effectiveness of proposed techniques for large-scale overlays while Internet experiments illustrate the system performance under a small but practical real-world environment. In this section, we evaluate the performance of our

Algorithm 3.1: DHT_LOOKUP(key, l, n_l)

```
Input: $key$: the hash key. $l$: the current lookup level. $n_l$: the DHT owner at level $l$.

// Local lookup for lower-level DHT owners.
while $l > 0$ do
    $n_{l-1} \leftarrow idc_{l-1}(n_l).owner(MD5_{l-1}(key))$;
    if $n_{l-1}$ is not in the local routing table then break;
    $l \leftarrow l - 1$;
enddo;

if $l = 0$ then return $(n_0)$;    // Finding $n_0$ − global termination.

// Proceed to the next hop and perform recursive lookup.
$m \leftarrow$ the next hop node toward $n_l$;
return ($m$.DHT_LOOKUP($key, l, n_l$));
```

Fig. 4. The distributed lookup algorithm in recursive form.

Table 1. Backbone networks.

Backbone	Node count	Link latency
ASmap	3,104	1 − 40ms
Inet	3,050	1 − 40ms
TransitStub	3,040	1 − 20ms for stub links; 1 − 40ms for other links
AMP-all	118	measurement
AMP-US	108	measurement

Saxons-based DHT protocol using simulations. Section 5 presents experimental results on 51 PlanetLab sites.

4.1 Simulation Methodology and Setup

We use a locally-developed discrete-event simulator that simulates all packet-level events at overlay nodes in our evaluations. We do not simulate the packet routing at the substrate network routers. Instead, we assume shortest-path routing in the substrate network and use that to determine overlay link latency. This model does not capture packet queuing delays or packet losses at routers and physical links. However, such a tradeoff is important to allow us achieve reasonable simulation speed for large networks.

The substrate networks we use in the simulations are based on four sets of backbone networks listed in Table 1. First, we use Internet Autonomous Systems maps extracted from BGP routing table dumps, available at NLANR [14] and at the University of Oregon Route Views Archive [18]. Second, we include topologies generated by the Michigan *Inet*-3.0 [24]. We also use some transit-stub topologies generated using the GT-ITM toolkit [25]. For ASmap and Inet topologies, we assign a random link latency of 1 – 40ms. For TransitStub topologies, we assign a random link latency of 1 – 20ms for

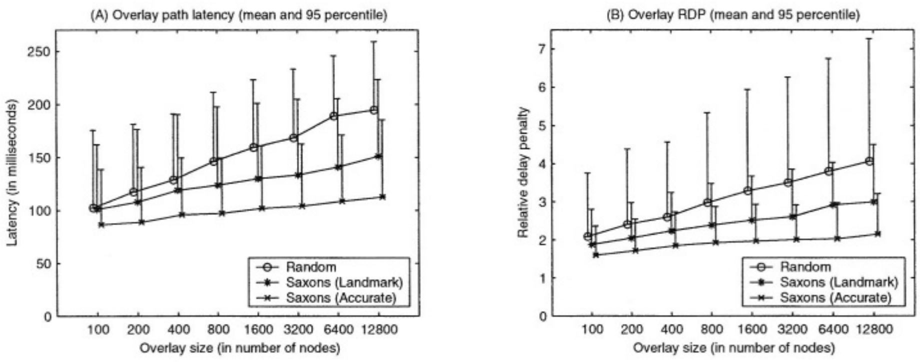


Fig. 5. Structure quality at various overlay sizes.

stub links and 1 – 40ms for other links. The last set of backbone network is based on end-to-end latency measurement data among 118 Internet nodes, reported by the NLANR Active Measurement Project (AMP) [15]. The AMP-US network excludes 10 non-U.S. hosts from the full AMP dataset. These 10 hosts have substantially larger latencies to other hosts than others. We use both AMP-all and AMP-US in the evaluation. With a given backbone network, each overlay node in our simulations is randomly attached to a backbone node through an edge link. We assign a random latency of 1 – 4ms for all edge links.

In all simulations, the Saxons overlay structure is configured with a node degree range of <4 – 16> and the periodic structure quality maintenance routine runs at 30-second intervals. Except explicitly evaluating the impact of different backbone networks, most results shown here are based on the ASmap network. Each data point represents the average value of five runs.

4.2 Overlay Structure Quality

This set of simulations examine the quality of overlay structure constructed using Saxons, upon which our proposed DHT protocol is built. We show the overlay structure quality in two metrics: 1) *overlay path latency,* defined as the end-to-end latency along the shortest overlay path for each pair of nodes; and 2) *relative delay penalty* (or *RDP*), defined as the ratio of the overlay path latency to the direct Internet latency. We compare three different overlay structure construction schemes. First, we consider the Saxons protocol with the landmark-based Cartesian distance approach for latency estimation (denoted by *Saxons (Landmark)*). Second, we examine Saxons with an accurate latency estimation between any two nodes (denoted by *Saxons (Accurate)*). Although it might not be practical, the results for Saxons (Accurate) are useful in indicating the performance potential of a Saxons-like structure management protocol. We finally consider the degree-bounded random structure construction (denoted by *Random*).

Figure 5 illustrates the overlay path latency and RDP at various overlay sizes. For each overlay size, nodes join the network at the average rate of 10 joins/second with exponentially distributed inter-arrival time. Node joins stop when the desired overlay size is reached and the measurement results are taken after the system stabilizes. For 12800-node overlays, results show that Saxons (Accurate) achieves 42% lower overlay path latency and 47% lower RDP compared with Random. The saving for Saxons (Landmark) is 22% on overlay path latency and 26% on RDP. The performance results of the Saxons indicate that it can benefit overlay services built on top of it by providing low-latency overlay structures.

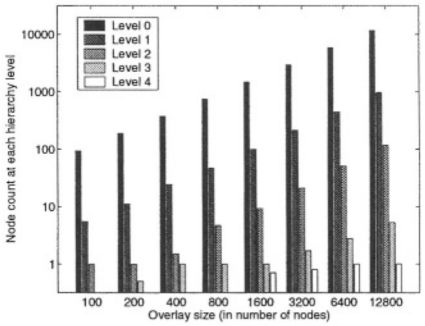


Fig. 6. Node count at each hierarchy level. Y-axis is on the logarithmic scale.

4.3 Statistics on the Landmark Hierarchy

Our proposed DHT is based on a self-organizing landmark hierarchy. In this set of simulations, we further explore the statistics on the landmark hierarchy construction over the Saxons overlay structure. The routing radii (starting from level 0) for the landmark hierarchy are set as 2, 4, 8, 16, 32, 64, ... The peer notification radii (starting from level 0) are 2, 5, 9, 17, 33, 65, ... Note that we require $pr_{l+1} > 2 * rr_l$ such that each node sees at least one other same-level peer if any exists.

Figure 6 shows the overlay node count at each hierarchy level for up to 12800 overlay nodes. The results indicate an exponentially larger population at lower hierarchy levels with around ten times more nodes at each immediate lower level.

Figure 7 illustrates the mean routing table size at each hierarchy level. A **level-l** routing entry at a node corresponds to a **level-l** landmark whose advertisements are received. We observe that the number of routing entries at each level remains around or below 80 for up to 12800 overlay nodes. The reason is that the large advertisement flooding hops of high-level landmarks are compensated by their sparse presence in the overlay structure. Note that the routing table sizes can be controlled by adjusting the routing radii in the hierarchy generation. Our adaptive promotion and demotion schemes result in the automatic construction of balanced hierarchies.

Since the list of children is included in the landmark route advertisement, a large children population at overlay nodes may result in excessive route advertisement overhead. Figure 8 shows the average number of children for nodes in each hierarchy level. There are no level-0 results because level-0 landmarks have no child. We observe that the number of children at each node remains around or below 50 for up to 12800 overlay nodes. Again, the large advertisement flooding hops of high-level landmarks are compensated by their sparse presence in the overlay structure.

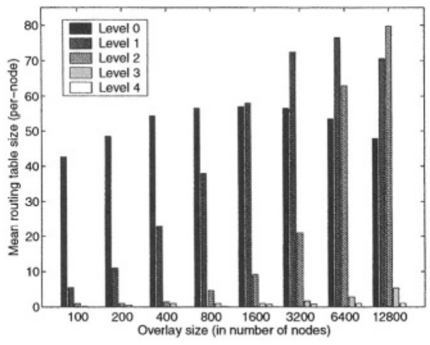


Fig. 7. Mean routing table size at each hierarchy level.

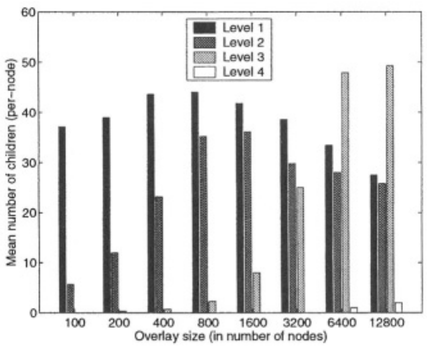


Fig. 8. Mean number of children at each hierarchy level.

4.4 DHT Performance

This section investigates the performance of our proposed DHT protocol on the Saxons structure management layer (denoted by *SaxonsDHT*). We examine the DHT performance in terms of lookup latency, fault tolerance, the balance of key placement and lookup routing overhead. We assess SaxonsDHT performance in relation to that of Chord [21], a well-known DHT protocol. A previous study [8] shows that Chord performs competitively against other strongly-structured DHT protocols such as CAN [16] and Pastry [19] in terms of lookup latency and load balance. We implemented the SaxonsDHT and Chord protocols in our simulator and both schemes are configured at the same link density in our evaluation. For SaxonsDHT, the node degree range of <4 – 16> results in an average degree of 8. For Chord, each node maintains an 8-entry finger table supporting DHT lookups. Higher-level finger entries in Chord point to nodes with exponentially larger distances in the identifier circle.

It should be noted that the purpose of our evaluation is to assess the performance of SaxonsDHT. We do not intend to make claim on its performance superiority over strongly structured DHT protocols. In particular, the comparison between SaxonsDHT and Chord at the same link density is not strictly fair for at least two reasons. First, Chord can freely structure the overlay network in order to achieve the best performance while SaxonsDHT has to function on top of a service-independent structure. On the other hand, SaxonsDHT requires a fairly large routing table at each node while Chord's finger table size is the same as the number of outgoing links. Further, recent enhancements on Chord have considered the latency of fetching DHT data in addition to the lookup latency [6]. We do not consider the data fetch latency in this paper.

DHTLookup Latency. The performance metrics for the DHT lookup include both lookup latency and hop-counts. Figure 9(A) illustrates the DHT lookup hop-count for SaxonsDHT and Chord at various overlay sizes. For each configuration, we measure the average performance of 100,000 DHT lookups on randomly chosen initiator nodes and hash keys. A quick analysis finds that the mean lookup hop-count for a Chord protocol with d finger entries is $d(\sqrt[d]{N}-1)/2$. Results in Figure 9(A) show that SaxonsDHT achieves slightly better performance (around 12% fewer lookup hops for 12800-node overlays) due to its hierarchical lookup routing scheme.

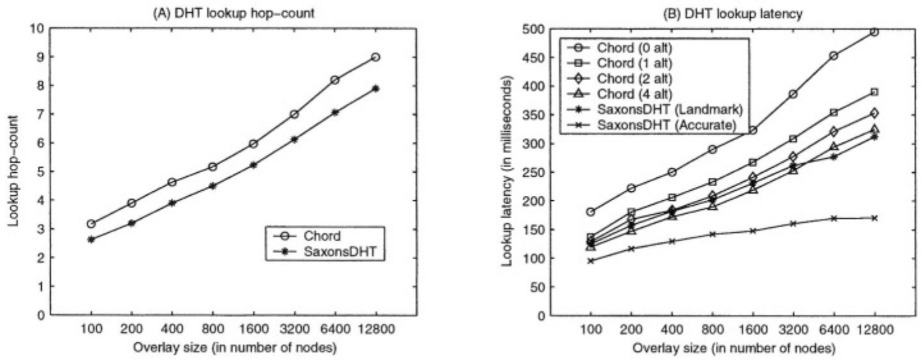


Fig. 9. DHT lookup performance at various overlay sizes.

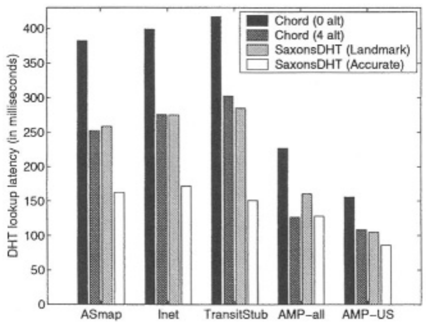


Fig. 10. DHT lookup latency over different backbone networks.

Figure 9(B) shows the DHT lookup latency at various overlay sizes. We introduce variations of the SaxonsDHT and Chord protocols that may have significant impact on the lookup latency. For SaxonsDHT, we examine two variations: one with the landmark-based Cartesian distance approach for latency estimation and another with an accurate latency estimation. As we discussed earlier, accurate latency estimation may not be practical for large-scale overlays, but it is useful in indicating the performance potential of the SaxonsDHT protocol. We also examine variations of the Chord protocol with a substrate-aware link-selection enhancement, called *proximity neighbor selection* [6]. In this enhancement, instead of simply picking the first node in each finger table entry's interval in the identifier ring, a few alternative nodes in each interval are probed and then the closest node is chosen to fill the finger table entry. We use *Chord (n alt)* to denote the Chord protocol with n alternative link probings for each finger table entry. In particular, Chord (0 alt) stands for the basic Chord protocol without the link-selection enhancement. For 12800-node overlays, results in Figure 9(B) show that SaxonsDHT (Landmark) achieves 37% less lookup latency than the basic Chord protocol and its performance is close to that of Chord (4 alt). Results also show that SaxonsDHT (Accurate) outperforms Chord (0 alt) and Chord (4 alt) at 65% and 31% respectively, indicating the vast performance potential of SaxonsDHT.

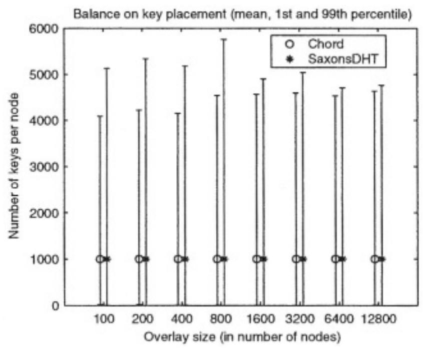


Fig. 11. DHT load balance on key placement.

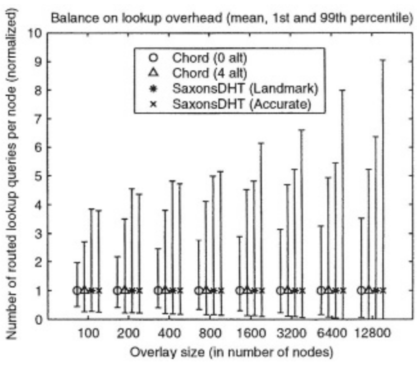


Fig. 12. DHT load balance on lookup routing overhead.

Figure 10 shows the DHT lookup latency over different backbone networks. Results indicate that the performance difference is not significantly affected by the choice of backbone networks. Savings are smaller for the two measurement-based backbones due to their small sizes.

DHT Load Balance. Load balance is another essential goal for a distributed hashtable and it is particularly challenging for hierarchical schemes. In our DHT protocol, we employ different key-identifier assignment functions at each hierarchy level to achieve balanced key placement. The balance on lookup routing overhead is supported by the property that queries often shift toward lower-level DHT owners before actually reaching any high-level DHT owner.

Figure 11 illustrates the DHT load balance on key placement over various overlay sizes. $1000 \times N$ (N is the overlay size) random keys are generated and mapped into overlay nodes. Results show that the balance of key placement for SaxonsDHT is close to that of Chord. We do not show results for variations of the SaxonsDHT and Chord protocols because they do not have significant impact on the balance of key placement.

Figure 12 shows the DHT load balance in terms of the lookup routing overhead. Note that the results in Figure 12 are normalized to the mean values. We observe that protocols with lower lookup latency typically exhibit less desirable load balance. For 12800-node overlays, the normalized 99-percentile lookup routing overhead for Chord (0 alt) is 44% and 61% less than those of SaxonsDHT (Landmark) and SaxonsDHT (Accurate) respectively. The difference between the substrate-aware Chord (4 alt) and the two SaxonsDHT schemes is much less. The inferior load balance of substrate-aware DHT protocols can be explained by their tendency of avoiding nodes that have long latency to other overlay nodes.

DHT Performance under Unstable Environments. Figure 13 shows the DHT lookup failure rate of SaxonsDHT and Chord under frequent node joins and departures at various average node lifetimes. The results are for 3200-node overlays. Individual node lifetimes are picked following an exponential distribution with the proper mean. In these experiments, a lookup is considered a success if it reaches the current level 0 DHT owner of the desired key. In a real system, however, there might be delays in which the current owner has not yet acquired the data associated with the key from the prior owner. We do not consider this factor since it is highly dependent on higher-level service implementation while we are primarily concerned with the DHT protocol. Results in Figure 13 show that SaxonsDHT and Chord deliver similar lookup success rate during overlay membership changes. Following an argument in [21], the lookup success rate under a certain frequency of membership changes mainly depends on the average lookup hop-counts. The similar success rate between SaxonsDHT and Chord can be explained by their similar lookup hop-counts.

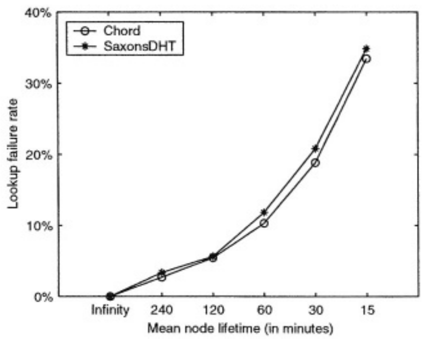


Fig. 13. SaxonsDHT lookup failure rate with biased node failures.

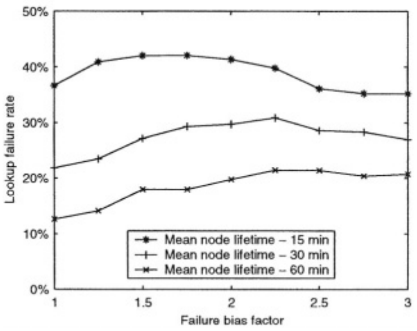


Fig. 14. SaxonsDHT lookup failure rate with biased node failures.

A hierarchical scheme like SaxonsDHT may suffer poor performance under targeted attacks against nodes of high importance. We examine the lookup failure rate when nodes at higher hierarchy levels have shorter lifetime. To quantify such scenarios, we introduce the*failure biasfactor,* defined as the ratio of the average node lifetime at each hierarchy level to that of the immediate higher level. In other words, for a failure bias factor of 2, level 1 nodes are twice likely to fail than level 0 nodes while level 4 nodes are 16 times more likely to fail than level 0 nodes. Figure 14 illustrates the SaxonsDHT lookup failure rate under biased node failure rates for 3200-node overlays. The results show that the lookup failure rate increases initially at the increase of the failure bias factor. However, this increase tapers off quickly and it may even decrease as the failure bias factor continues to grow. This is because the SaxonsDHT lookups are mainly based on local routing entries and high-level nodes are often not visited. This result shows that SaxonsDHT lookups are not particularly susceptible to targeted attacks despite the nature of its hierarchical design.

Due to its structure-sensitive DHT mapping scheme, SaxonsDHT tends to produce more key reassignments after overlay membership changes. Figure 15 shows the proportion of key assignment changes of SaxonsDHT and Chord with certain number of random node joins and leaves in 3200-node overlays. We consider the SaxonsDHT performance with varying failure bias factors to model the impact of targeted failures of high-level landmarks. Comparing to Chord, SaxonsDHT produces two to three times key reassignments (186% more in average) after overlay membership changes. Targeted failures of high-level landmarks may result in more key reassignments. Such a performance difference suggests that our DHT design may be better suited for applications that do not require large data migration after the DHT mapping changes, such as those that involve time-sensitive information or require repetitive refreshment. We should point out that the DHT mapping in Chord is based on *consistent hashing,* a technique with provable minimal key reassignments after membership changes [9].

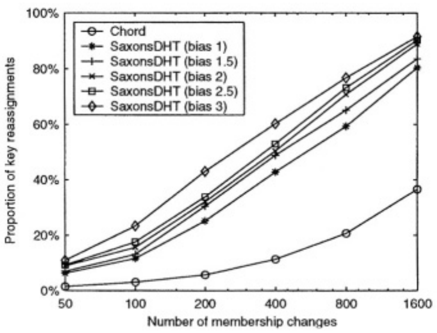


Fig. 15. Key reassignments due to overlay membership changes.

5 Internet Experimentation

The implementation of our proposed DHT design is based on a Saxons prototype [20]. In our DHT implementation, every node maintains a TCP connection with each of its overlay neighbors. The route advertisements and lookup queries flow through these TCP connections on the overlay structure. The DHT service periodically queries the Saxons kernel for up-to-date overlay structure information. Link-related state such as the TCP connections to neighbors and routing table entries may have to be adjusted when directly attached overlay links change. For the purpose of comparison, we also made a prototype implementation of Chord. Our prototype can correctly form the Chord finger tables at the absence of node departures. We did not implement the full Chord stabilization protocol for simplicity. Each node in our Chord prototype maintains a TCP connection with each of the nodes listed in its finger table and lookup queries flow through these connections. For both DHT implementations, node IDs are assigned using MD5-hashed IP addresses.

We conducted experiments on the PlanetLab testbed [2] to evaluate the performance of the proposed DHT service. Our experiments involve 51 PlanetLab nodes, all from unique wide-area sites. Among them, 43 are in the United States, 5 are in Europe. The other three sites are in Australia, Brazil, and Taiwan respectively. The round-trip latency between a U.S. site and a non-U.S. site is often much higher than that between two U.S. sites. Due to the small number of sites in the experiments, we are able to employ direct runtime latency measurements for the Saxons structure quality maintenance. Specifically, a node pings the other 10 times and measure the round-trip time. It then takes the average of the median 6 measurement results. This scheme is close to *Saxons (Accurate)* examined in the simulation studies.

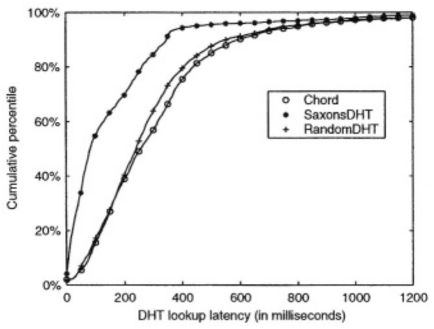


Fig. 16. DHT lookup latency CDFs on 51 PlanetLab sites.

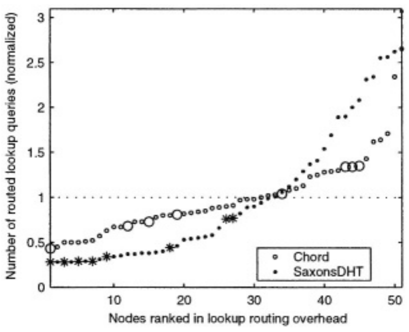


Fig. 17. Load balance of DHT lookup routing overhead on 51 PlanetLab sites. Larger markers represent non-U.S. sites.

In order to compensate the small size of our testbed, we use a relatively sparse overlay structure in our experimentation. For SaxonsDHT, the Saxons overlay structure is configured with a node degree range of <2 – 8>, and consequently an average node degree of 4. The settings for routing radii and peer notification radii are the same as those in the simulation study. A typical run shows that the Landmark Hierarchy contains one level 3 landmark, three level 2 landmarks, and eight level 1 landmarks. The remaining nodes are at level 0.

We compare the performance of SaxonsDHT against Chord with a 4-entry finger table at each node. For the purpose of comparison, we also consider the performance of our proposed DHT service running on a degree-bounded random overlay structure (denoted by *RandomDHT*). In each run of our experiments, 1000 DHT lookups are initiated at each participating node with random hash keys. Figure 16 illustrates the cumulative distribution functions of the 51,000 DHT lookup latency measurements taken out of a typical test run. We observe that the lookup latency of RandomDHT is close to that of Chord while SaxonsDHT significantly outperforms them. In average, SaxonsDHT achieves about 48% latency reduction compared with Chord (335.5ms *vs.* 643.0ms).

We also examine the DHT load balance on lookup routing overhead for SaxonsDHT and Chord. Figure 17 shows the number (normalized to the mean value over all nodes) of routed lookup queries over each node. Results in the figure are increasingly ranked and larger markers represent 8 non-U.S. sites in the testbed. The results show that Chord exhibits better load balance than SaxonsDHT. We also observe that SaxonsDHT tends to avoid the non-U.S. sites in query routing while Chord is oblivious to network distances between participating sites. Such a behavior helps SaxonsDHT to achieve better lookup performance at the expense of load balance.

6 Related Work

Previously proposed scalable DHT protocols such as Chord [21], CAN [16], and Pastry [19] all function on protocol-specific overlay structures to support DHT lookups. A recent work [8] suggests that measurement-based overlay structures often have much lower latency than structures provided by Chord, CAN, or Pastry. However, it did not explain how a DHT service can be built on top of a low-latency measurement-based overlay structure. Several studies have proposed substrate-aware techniques to enhance particular DHT protocols. Zhao *et al.* proposed to construct a secondary overlay (called Brocade) on top of existing DHT structures to exploit unique network resources available at each overlay node [27]. Ratnasamy *et al.* introduced a distributed binning scheme for CAN such that the overlay topology resembles the underlying IP network [17]. In Mithos [23], Waldvogel and Rinaldi proposed an efficient overlay routing scheme based on an energy-minimizing node ID assignment in a multi-dimensional ID space. Zhang *et al.* suggested a random sampling technique is effective for incrementally reducing lookup latency in DHT systems [26]. These approaches are valuable in improving the performance of specifically targeted protocols. However, substrate-aware techniques built for particular DHT structures cannot benefit other services.

Kleinrock and Kamoun proposed hierarchical routing protocols to achieve low routing latency with small routing table sizes [10]. Landmark Hierarchy was later introduced by Tsuchiya to allow minimal administration overhead and automatic adaptation to dynamic networks [22]. Recent studies (SCOUT [11] and L^+ [4]) employed the Landmark Hierarchy-based routing and location schemes for sensor and wireless networks. Our design draws upon results and experience of these work. New techniques are introduced in our design to construct a distributed hashtable service and satisfy its performance requirements. For instance, balanced key placement is a unique performance objective for DHT and it has not been addressed in previous studies on hierarchical routing.

7 Conclusion

This paper presents a distributed hashtable protocol that operates on pre-structured overlays, and thus is able to take advantage of a common structure management layer such as Saxons [20]. Compared with Chord [21] at the same overlay link density[2], simulations and Internet experiments find that the proposed scheme can deliver better lookup performance at the cost of less load balance on query routing overhead. Evaluation results also show that the balance of key placement and fault tolerance for our approach are close to those of Chord. In addition, we find that the proposed scheme is not particularly susceptible to targeted attacks despite its hierarchical nature. Due to the structure-sensitive DHT mapping scheme, however, the proposed approach may produce significantly more key reassignments at high node churn rates.

Overall, our effort supports the broader goal of providing a common overlay structure management layer that can benefit the construction of a wide range of overlay services. While it is well understood that this model works well with services like unstructured peer-to-peer search and unicast/multicast path selections, our work is the first to examine its applicability on the distributed hashtable service.

Acknowledgment. We would like to thank anonymous referees for their valuable comments.

References

1. D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. *In Proc. of SOSP,* pages 131–145, Banff, Canada, Oct. 2001.
2. A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. of NSDI,* pages 253–266, San Francisco, CA, Mar. 2004.
3. M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting Network Proximity in Peer-to-Peer Overlay Networks. In *Proc. of the FuDiCo Workshop,* Bertinoro, Italy, June 2002.
4. B. Chen and R. Morris. L+: Scalable Landmark Routing and Address Lookup for Multi-hop Wireless Networks. Technical Report MIT-LCS-TR-837, Laboratory for Computer Science, MIT, 2002.
5. Y.-H. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *Proc. of SIGMETRICS,* pages 1–12, Santa Clara, CA, June 2000.
6. F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for Low Latency and High Throughput. In *Proc. of NSDI,* pages 85–98, San Francisco, CA, Mar. 2004.
7. P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin. An Architecture for a Global Internet Host Distance Estimation Service. In *Proc. of INFOCOM,* New York, NY, Mar. 1999.
8. S. Jain, R. Mahajan, and D. Wetherall. A Study of the Performance Potential of DHT-based Overlays. In *Proc. of USITS,* Seattle, WA, Mar. 2003.

[2] Note that at the same link density, our DHT protocol requires a larger routing table compared with Chord's finger table.

9. D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistency Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of the ACM Symp. on Theory of Computing,* pages 654–663, El Paso, TX, May 1997.
10. L. Kleinrock and F. Kamoun. Hierarchical Routing for Large Networks. *Computer Networks,* 1:155–174, 1977.
11. S. Kumar, C. Alaettinoglu, and D. Estrin. SCalable Object-tracking Through Unattended Techniques (SCOUT). In *Proc. of ICNP,* Osaka, Japan, Nov. 2000.
12. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proc. of the ACM International Conference on Supercomputing,* pages 84–95, New York, NY, June 2002.
13. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-based Approaches. In *Proc. of INFOCOM,* New York, NY, June 2002.
14. BGP Routing Data at the National Laboratory for Applied Network Research. http://moat.nlanr.net/Routing/rawdata.
15. Active Measurement Project at the National Laboratory for Applied Network Research. http://amp.nlanr.net.
16. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of SIGCOMM,* pages 161–172, San Diego, CA, Aug. 2001.
17. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proc. of INFOCOM,* New York, NY, June 2002.
18. University of Oregon Route Views Archive Project. http://archive.routeviews.org.
19. A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proc. of the IFIP/ACM Middleware,* Heidelberg, Germany, Nov. 2001.
20. K. Shen. Structure Management for Scalable Overlay Service Construction. In *Proc. of NSDI,* pages 281–294, San Francisco, CA, Mar. 2004.
21. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of SIGCOMM,* pages 149–160, San Diego, CA, Aug. 2001.
22. P. F. Tsuchiya. The Landmark Hierarchy: A New Hierarchy for Routing in Very Large Networks. In *Proc. of SIGCOMM,* pages 35–42, Stanford, CA, Aug. 1988.
23. M. Waldvogel and R. Rinaldi. Efficient Topology-Aware Overlay Network. In *Proc. of the HotNets Workshop,* Princeton, NJ, Oct. 2002.
24. J. Winick and S. Jamin. Inet-3.0: Internet Topology Generator. Technical Report CSE-TR-456-02, Dept. of EECS, University of Michigan, 2002.
25. E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of INFOCOM,* San Francisco, CA, Mar. 1996.
26. H. Zhang, A. Goel, and R. Govindan. Incrementally Improving Lookup Latency in Distributed Hash Table Systems. In *Proc. of SIGMETRICS,* San Diego, CA, June 2003.
27. B. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiatowicz. Brocade: Landmark Routing on Overlay Networks. In *Proc. of the Workshop on Peer-to-Peer Systems,* Cambridge, MA, Mar. 2002.

Application Networking – An Architecture for Pervasive Content Delivery

Mu Su and Chi-Hung Chi

School of Computing
National University of Singapore
{sumu, chich}@comp.nus.edu.sg

Abstract. This paper proposes Application Networking (App.Net) architecture that allows Web server deploying intermediate response with associated partial workflow logic to middle proxies. The recipient proxy is allowed to instantiate the workflow using local services or download remote application modules. The final response is the output from the workflow fed with the intermediate response entity. In this paper, we define a workflow modulation method to organize service workflow and manipulate them in uniform operations. Furthermore, we developed one App.Net simulator to measure the performance of sample applications existed in the Web under different deployment methods and delivery environments. Our simulation results show that using workflow and dynamic application deployment, App.Net can achieve better performance than conventional methods.

1 Introduction

Recently the Web has witnessed the increasing heterogeneity of content delivery context. The emergence of diverse user devices, last-mile network connections, and more service personalization requirements makes content and service adaptations necessities to achieve pervasive service availability. However, the enhanced adaptation processes may compromise the efficiency and scalability of conventional content delivery methods, which fixes their processing points either to content server or the client. With server-side adaptation, servers always serve fully adapted content. Hence proxies may cache adapted content only. The drawback of caching fully adapted content is a significant reduction of reuse benefits. On the other hand, pure client-side programs are prone to cause the service unavailability to devices that don't have the particular processing capability. Therefore, necessary architecture change of web content generation and delivery is in tandem.

Researchers have proposed various methods that allow network intermediaries take part in the content generation or adaptation processes like active proxy structures [1, 2, 3, 4, 5], OPES general architecture [6], TACC active proxy cluster [7], collaborative proxies [8], Content Service Network (CSN) [9], Application CDN (ACDN) [10], and active cache [11, 12]. However, the effectiveness of these architectures may be limited in a heterogeneous and dynamic Web environment because they may have one or both of two weaknesses: (i) static application deployment and (ii) logic indivisibility.

Static application deployment includes two issues: (i) all application modules have to be *preinstalled* in advance of service time, and (ii) the deployed application modules *cannot be redistributed* according to the change of delivery environment. Requiring all application modules preinstalled will significantly limit the system scalability, because we expect content providers and end users will randomly deploy new application modules to the proxy nodes according to particular service requirement at the runtime. Secondly, the static deployment strategy will be error-prone and compromise the system performance under dynamically changing environment.

Logic indivisibility means that original service logic is deployed at an atomic unit. Indivisible logic is a coarse deployment granularity and will restrict inferring optimal service deployment strategy. We look forward a divisible logic would be helpful to more effectively deploy the service by distributing segments of partial logic to different network nodes.

Hence, a more effective delivery scheme should support features as: (i) dynamic application deployment that allows mobile application modules to be loaded on the proxy at runtime; (ii) divisible logic that allows deploying service logic at a fine granularity.

We thus propose Application Networking (App.Net) architecture, which allows deploying intermediate response data together with one associated workflow to cache proxies. In App.Net, a workflow is aggregated by a sequence of mobile applications modules, each of which implement specific task in the workflow. When receiving a request, the content server can partition the original workflow and execute the front part on the server to generate an intermediate response. The intermediate response with the associated rear partial workflow will be delivered to the proxy. The recipient proxy can instantiate the workflow by loading locally installed applications or downloading new modules from remote sites. Finally, the proxy feeds the intermediate response to the instantiated workflow to generate the final response entity for the clients.

Consequently, in virtue of workflow, App.Net can manipulate and deploy partial logic to intermediate nodes to achieve optimal or sub-optimal system performance under specific delivery context. Secondly, implementing workflow segment as mobile application allows recipient proxy dynamically load new modules from Internet nodes.

The rest of this paper is organized as follows. Section 2 proposes the architecture of Application Networking. A modulation method that facilitates workflow manipulation is defined in section 3, which is followed by detailed system operations in section 4. Then, a preliminary performance study is given in section 5. In section 6, we survey the related work. Finally, we conclude our current research in section 7.

2 Architecture Overview

The general architecture of Application Networking (App.Net) is depicted in Figure 1. An App.Net platform can reside on content server, intermediate proxies, or even end users. App.Net Porc is the central process of the platform. App.Net Cache stores the received response message, including response data and its associated workflow specification. And Application Pool stores various applications, including modules locally installed in advance and ones downloaded on the fly.

To publish content on App.Net server, the content provider prepares the original content and associates it with a particular workflow specification. When the server receives a request forwarded from an intermediate App.Net proxy, the server-side App.Net Proc may partition the workflow associated with the requested content into two parts according to particular system policy or performance consideration. The front part is executed on the server to generate an intermediate response entity. Then content server delivers the intermediate response with the rear part of the partitioned workflow to the proxy.

The proxy-side App.Net platform will load necessary application modules to instantiate the received workflow specification. If the required application module is not available on proxy local site, App.Net Proc can download it from a remote site and store in the local Application Pool. After instantiating the workflow, the proxy will execute applications sequentially and generate the final response presentation for the client. At the same time, App.Net Proc will cache the response message, including the intermediate response and associated workflow specification, into App.Net Cache.

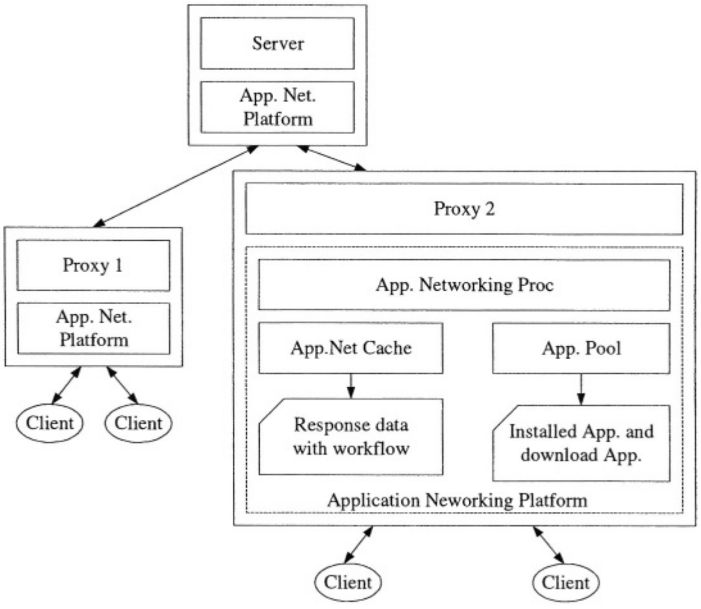


Fig. 1. Application Networking Architecture

Furthermore, application networking platform allows any intermediate proxy node to recursively partition the received workflow in the same manner as original server. Therefore, the original workflow segments will be distributed to multiple proxies on the content delivery path. However, different deployment strategies may induce different system performance, which will be discussed in section 5.

From above, the workflow specification is a crucial issue in App.Net. It should not only facilitate logic partitioning, but also supply information to direct the recipient proxy downloading appropriate remote applications. Hence, in section 3 we will describe the *workflow modulation* scheme and a workflow specification method in HTTP response message.

3 Workflow Modulation

3.1 Formulation

Workflow modulation defines a formulized scheme to organize and manipulate one workflow. In modulation, a workflow is well is defined as the aggregation of sequential segments (SEG). Each segment is composed by one configuration rule (CR), a particular task (TSK), one input response entity ($\mathbf{RE_{in}}$), and one modified output entity ($\mathbf{RE_{out}}$). In addition, each task is also fed with one input parameter vector (v) and additional data entries (d) if necessary. Once conditions in the CR are fulfilled, the specified task will be executed, which modifies the input RE according to corresponding parameter vector and generate an output RE. Otherwise, the TSK should not be executed, and SEG will relay the input RE to the subsequence segment without any modification. Equation (1) gives the formulized definition of SEG.

$$SEG{:=}RE_{out}{=}TSK^{CR}(RE_{in},v,d) \tag{1}$$

In order to implement operations of the task, a TSK should always be instantiated by specific code base, a particular application module. Hence, equation (1) can be revised as equation (2).

$$SEG{:=}RE_{out}{=}TSK^{CR,codebase}(RE_{in},v,d) \tag{2}$$

In the workflow, segments are linked sequentially in that the output RE of the previous SEG will be input RE to subsequent SEG, as shown in equation (3).

$$SEG_i := RE_i = TSK_i^{CR_i,codebase_i}(RE_{i-1}, v_i, d_i),\ (1\leq i\leq L) \tag{3}$$

A workflow (WF) can be rephrased as equation (4).

$$\begin{aligned} WF^L &= SEG_1 \bullet SEG_2 \bullet \cdots \bullet SEG_i \bullet SEG_{i+1} \bullet \cdots \bullet SEG_L \\ &= (\bullet)_{i=1}^{L} SEG_i \end{aligned} \tag{4}$$

The equation presents one workflow of L segments, which is composed by L tasks, and has one original entity RE_0, L-1 intermediate response entities from RE_1 to $RE_{L\text{-}1}$, and one ultimate response entity RE_L. In practice, the RE_0 may be the web object stored on server, such as HTML page, image, and so on. Also we can view a server-side program, like JSP and PHP, as an application module TSK_1; and the backend database queried by this program can treat as RE_0. Any intermediate response RE constructed from the origin RE_0 may be formulized in equation (5), and the final response RE_L can be generated from any intermediate response entity, as shown in equation (6).

$$RE_t = (\bullet)_{i=1}^{t} SEG_i(RE_{i-1}, v_i, d_i), 1 \le t \le L \tag{5}$$

$$RE_L = (\bullet)_{i=t}^{L} SEG_i(RE_{i-1}, v_i, d_i), 1 \le t \le L \tag{6}$$

Consequently, the modulated workflow can be illustrated as figure 2.

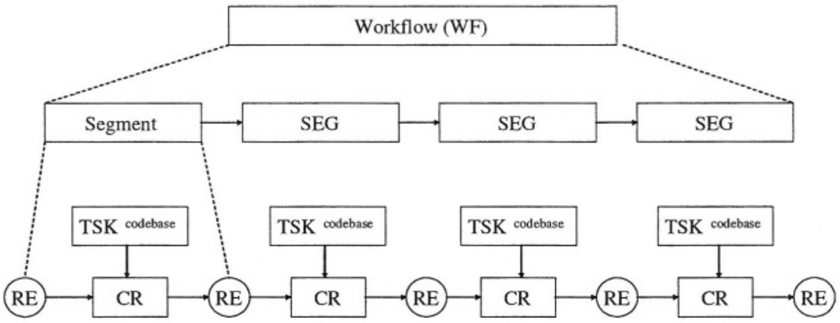


Fig. 2. Workflow Modulation

In addition, workflow modulation also defines a set of operations to facilitate uniform manipulation of a workflow. The operations are defined as the follows.

(1) *Selection*

Selection ($F^{l=t}(WF^L) := WF^{t/L}$) defines an operation to partition a workflow and select its front segments, as shown in equation (7).

$$\begin{aligned} F^{l=t}(WF^L) &= SEG_1 \bullet SEG_2 \bullet \cdots \bullet SEG_t \\ &= (\bullet)_{i=1}^{t} SEG_i \\ &= WF^{t/L} \end{aligned} \tag{7}$$

Since $WF^{t/L}$ is composed by the first t segments from original workflow WF^L, we can conclude from equation (5) that selection $WF^{t/L}$ can generate the intermediate response RE_t from original RE_0.

(2) *Inverse-selection*

Inverse-selection ($\tilde{F}^{l=t+1\sim L} := WF^{t+1\sim L/L}$) defines an operation to partition a workflow and select its rear segments, as shown in equation (8).

$$\begin{aligned} \tilde{F}^{l=t+1\sim L}(WF^L) &= \tilde{F}^{l=t+1\sim L}((\bullet)_{i=1}^{L} SEG_i) \\ &= (\bullet)_{i=t+1}^{L} SEG_i \\ &= WF^{t+1\sim L/L} \end{aligned} \tag{8}$$

Since $WF^{t+1\sim L/L}$ is composed by the last L-t segments from origin workflow WF^L, we can conclude equation (6) that inverse-selection $WF^{t+1\sim L/L}$ can generate the final response RE_L, given intermediate response entity RE_t.

(3) *Replicated Inverse-selection*

Replicated inverse-selection ($\tilde{F}^{l=t\sim L} := WF^{t\sim L/L}$) defines an operation to duplicate one segment to both parts of a partitioned workflow. The difference between inverse-selection and replicated inverse-selection is that the latter has one replicated segment existing on both front workflow partition and the rear workflow partition, equation (9).

$$\begin{aligned} &\tilde{F}^{l=t\sim L}(WF^L) = \tilde{F}^{l=t\sim L}((\bullet)_{i=1}^{L} SEG_i) \\ &= (\bullet)_{i=t}^{L} SEG_i \\ &= WF^{t\sim L/L} \end{aligned} \tag{9}$$

Replicated inverse-selection $WF^{t\sim L/L}$ is composed by the last L-t+1 segments from origin workflow WF^L, while the front selected part is composed by the first t segments, with SEG_t duplicated on both partitions.

(4) *Appending*

Appending ($F^{L\to L+t} := WF^{L+t/L}$) is an operation used to append additional segments to original workflow WF^L, in that clients or intermediaries can attach customization service to original workflow.

$$\begin{aligned} &F^{L\to L+t}(WF^L) = ((\bullet)_{i=1}^{L} SEG_i) \bullet ((\bullet)_{j=L+1}^{L+t} SEG_j) \\ &= (\bullet)_{i=1}^{L+t} SEG_i \\ &= WF^{L+t/L} \end{aligned} \tag{10}$$

(5) *Substitution*

Substitution ($F^{codebase'}(SEG)$) defines an operation to substitute original code base of one SEG with another one, given by equation (11).

$$\begin{aligned} &F^{codebase'}(SEG) = F^{codebase'}(TSK^{CR,codebase}(RE_{in}, v, d)) \\ &= TSK^{CR,codebase'}(RE_{in}, v, d) \end{aligned} \tag{11}$$

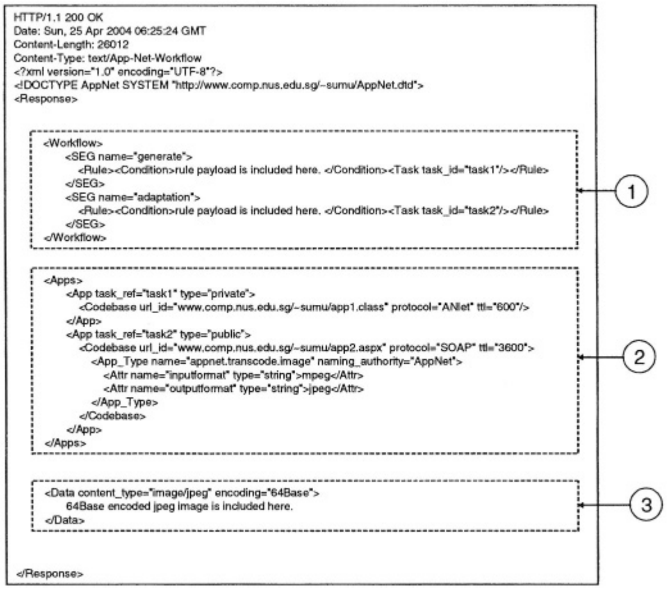
Substitution lets intermediaries substitute original application module specified in workflow with another application, such as a locally installed general service. However, after substitution, the semantics of origin task should not be changed. The node that carries out the substitution should maintain the semantic consistency.

3.2 Data Format

To specify modulated workflows, different methods can be taken according to specific use domains such as cache proxy, CDN, server-cluster, and so on. In this paper, we propose one workflow specification and message encapsulation method used for the cache proxy architecture in figure 1.

We notice active cache systems [12] use extended HTTP response headers to carry server directions, which refer to the target applet to be delivered. Although easy to implement, extending HTTP header is deficient to describe workflow logic and identify applications precisely in App.Net.

Therefore, we plan to use XML to specify modulated workflows and the associated input RE. The markup response will be included in the HTTP response body, tagged with content type, App-Net-Workflow. One sample response message is given in figure 3.



```
HTTP/1.1 200 OK
Date: Sun, 25 Apr 2004 06:25:24 GMT
Content-Length: 26012
Content-Type: text/App-Net-Workflow
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AppNet SYSTEM "http://www.comp.nus.edu.sg/~sumu/AppNet.dtd">
<Response>

  <Workflow>
      <SEG name="generate">
         <Rule><Condition>rule payload is included here. </Condition><Task task_id="task1"/></Rule>
      </SEG>
      <SEG name="adaptation">
         <Rule><Condition>rule payload is included here. </Condition><Task task_id="task2"/></Rule>
      </SEG>
  </Workflow>

  <Apps>
      <App task_ref="task1" type="private">
         <Codebase url_id="www.comp.nus.edu.sg/~sumu/app1.class" protocol="ANlet" ttl="600"/>
      </App>
      <App task_ref="task2" type="public">
         <Codebase url_id="www.comp.nus.edu.sg/~sumu/app2.aspx" protocol="SOAP" ttl="3600">
            <App_Type name="appnet.transcode.image" naming_authority="AppNet">
               <Attr name="inputformat" type="string">mpeg</Attr>
               <Attr name="outputformat" type="string">jpeg</Attr>
            </App_Type>
         </Codebase>
      </App>
  </Apps>

  <Data content_type="image/jpeg" encoding="64Base">
      64Base encoded jpeg image is included here.
  </Data>

</Response>
```

Fig. 3. Example Response Message

The formatted response message includes three parts: (1) Workflow Identification part, (2) Application Identification part, and (3) Input Data part. Workflow identification part describes the modulation structure of the workflow. And the application identification part specifies the URLs of application modules for each task in the workflow. To facilitate application substitution, necessary properties can also be included. Finally, the input RE to the workflow is encapsulated in Data part.

4 System Operations

Based on definitions and data format above, a typical application networking system works as the follows, assuming both server and proxy are App.Net enabled. (In practice, the server can verify that the requesting proxy supports App.Net by detecting specifically extended request header in HTTP message.)

Step 1. To publish content on server, the content provider prepares the original content RE_0 and associates it with a particular workflow specification WF^L. The workflow can transform the original content RE_0 to the final presentation RE_L through multiple intermediate response entities RE_i, shown in figure 4.

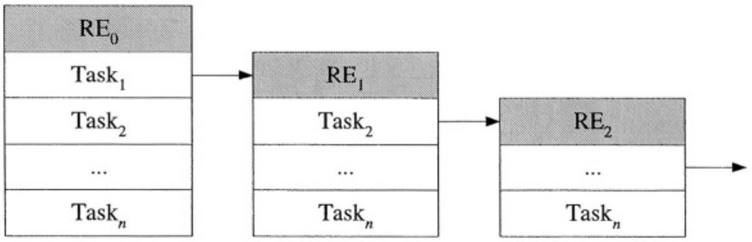


Fig. 4. Workflow and Intermediate REs

Step 2. When receiving one request forwarded from the proxy, the server operates as the following:

a. If necessary, server-side App.Net Proc partitions the workflow associated to the requested content according to the particular system policy or performance strategy.

b. The selected front part segments $WF^{t/L}$ will be executed on the server and the inverse-selected segments $WF^{t+1\sim L/L}$ or replicated-inverse-selected segments $WF^{t\sim L/L}$ will be delivered to the proxy.

c. Server prepares the response message, tagged with "Content-Type: App-Net-Workflow". The generated intermediate response and partitioned rear partial workflow will be encapsulated into the response message.

Step 3. Once the proxy receives response message, it operates as the follows.

a. The proxy App.Net Proc tries to instantiate the deployed workflow by loading installed modules from local Application Pool and downloading other application modules from remote site.

b. It executes the initiated workflow and generates the final presentation RE_L.

c. At the same time, the proxy App.Net Cache will store the received response message from server.

d. Similarly, Proxy App. Pool will cache downloaded applications till they are replaced or invalidated.

Step 4. When receiving a new request, the proxy will match the request message against entries in the App.Net Cache (for details see section 5). If it is a cache hit, the process goes to step 5. Otherwise, if there is a cache miss, the proxy will forward the request to upstream nodes and process goes to step 2.

Step 5. In the case of cache hit, proxy's App.Net Proc operates as the follows:

a. If necessary, the proxy-side App.Net Proc may also partition the cached workflow according to the particular system policy or performance strategy. The selected segments will be executed on the proxy, and the inverse-selected or replicated-inverse-selected segments will be delivered to downstream nodes together with associated intermediate response entity.

b. As an extreme case, the proxy will execute the whole cached workflow to generate the final response.

Step 6. The App.Net Proc output entity is delivered to downstream nodes.

5 Performance

In this section, we present our preliminary performance evaluation for a sample Device Independence (DI) service to explore the potential benefit and overhead of the different application distributions in App.Net.

In our simulation, network communication and bandwidth consumption are treated as the major factors to determine the system performance, while assuming the effect of computation load is negligible. The assumption is reasonable because the effect of computation load can be mitigated by enriched hardware and local clustering. However, network communication between nodes cannot be easily bypassed.

In addition, the transmission overhead of mobile applications is negligible, because the size of our applications, which are called ANlets, in simulation range from 1KB to 7KB, which is trivial compared with accumulated traffic of content transmission.

Based on Jigsaw 2.2.4 edge server, we developed an App.Net simulator composed by a server, a proxy, and a client component. The simulation environment is built on three desktops interlinked by 10MB Ethernet local network. The server is located on one P4-2.8GHz PC with 1GB memory, the proxy is located on one P4-2.0GHz PC with 512MB memory, and the client simulator is installed on one P3-600MHz PC with 256MB memory. The networking characters between these two components are simulated using Linux CBQ utility.

Currently, there are lots of DI service solutions [13,14,15,16]. However, little attention has been taken to the performance of a DI authoring system that serves numerous clients with heterogeneous devices. In our simulation, a sample DI service workflow is designed and it is composed by two independent ANlets. The first one is a content trimming ANlet that selects appropriate fragments of the original document according to client's device display size and his/her navigation position; and the second one is a page rendering ANlet that generates final web page using particular markup language accepted by user devices.

In simulation, the size of original document is set to 15KB. We suppose the first ANlet can generate 10 diverse fragments from original document according to the input parameter that takes one of 10 different values. The average size of the trimmed fragments is set to 4KB. The second ANlet is set to generate web pages in three kinds of markup languages. The average size of final web pages is set to 6KB. Additional, original document, intermediate fragments, and final pages can be cached for 2 sec.

Obviously, there are three kinds of application deployments. The first extreme case, named wf_1, deploys both ANlets to origin server and is equivalent to server-side DI service. The second deployment, name wf_2, partitions the workflow in the middle and distributes the first ANlet on the server and the second one on the proxy. wf_2, is a new case inferred by App.Net workflow. Another extreme case, named wf_3, deploys both ANlets to proxy and is equivalent to proxy-side DI service. The experiment results are given in figure 5.

Figure 5(a) shows that wf_2 consumes the least bandwidth at low request rate. However, when the request rate increases above 12/sec, wf_3 consumes the least bandwidth because the bandwidth saving due to original document reuse dominates its transmission overhead. On the contrary, the sever-side solution wf_1 consumes the most bandwidth continually because the final web pages have the least reusability.

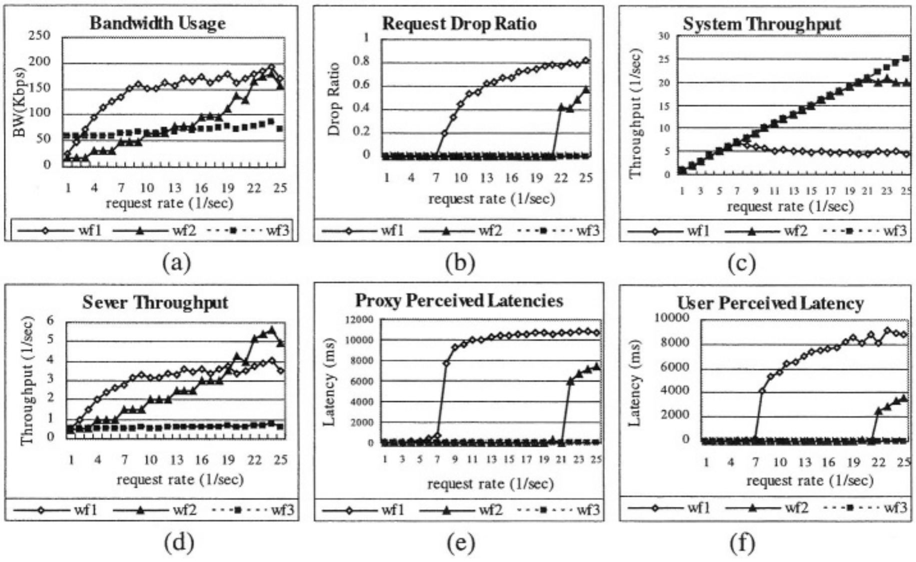


Fig. 5. Performance of a sample DI service workflow

Figure 5(c) and (d) show that the throughput of wf_3 can increase linearly with request rate up to 30/sec. In contrast, wf_1 and wf_2 reach their throughput upper bounds at request rate 8/sec and 21/sec respectively.

In addition, figure 5(e) and (f) show that all three deployment strategies have similar system latency when the request rate is low. However, with the increase of request rate, the system latency of wf_1 and wf_2 increase sharply. However, wf_3 keeps low service latency continually.

In summary, wf_2 is the best distribution method at low request rate, while wf_3 is the best one at high request rate. Hence, we can see that for popular documents, we can use proxy based strategy; while for unpopular documents we can use partial deployment strategy. Consequently, App.Net scheme can explore more service deployment methods dynamically to achieve higher system performance.

6 Related Work

In past years, researchers have proposed many active proxy architectures including MOWSER[1], CONCA[3], TranSquid[17], PTC[4], SEE[5], and OPES general model [2, 6]. Any active proxy system allows an edge proxy modify the passing data flow by executing value-add service, such as image transcoding, language translation, and so on. However, active proxy systems rely on preinstalled application modules before service time. This feature limits the system scalability to adopt new applications at the run time. Moreover, most active proxy systems use static application deployment method, which is error-prone under dynamic Web environment, where users request pattern and variance of devices changes on the fly.

[7] proposed active proxy cluster architecture TACC. It allows service module replication among cluster servers, and servers load balance by a cluster manager. TACC achieves better system scalability and performance than single active proxy because of hardware richness and sophisticated clustering mechanism.

Furthermore, [8] proposes collaborative proxy architecture allowing hierarchical proxies finish the content adaptation in a collaborative manner. By scattering adaptation tasks among collaborative proxies the system performance can be increased due to enhance load balance. However, the proxy collaboration and load balance are based on condition that required service modules are already installed on the proxies. A worst-case distribution that replicates all possible applications to each proxy would be costly and infeasible.

[9] proposed one overlay service network architecture, called Content Service Network (CSN), based on conventional CDN. CSN deploys Application Proxies (AP) at the edge of CDN. AP can be remotely invoked by OPES cache proxies located in CDN. The services of a CSN is subscribed and used by content provider, end user, ISP or even CDNs using Internet Service Delivery Protocol (ISCP). Similar to CDN mechanisms, CSN uses Service Distribution and Management Server to distribute service modules, and uses Redirection Servers to distribute client requests.

[10] describes architecture of a CDN for applications (ACDN). The difference between CSN and ACDN is that the former is one overlay service network on CDN, while the latter is a CDN by itself. New applications can be plunged into ACDN by publishing them on CDN servers. Furthermore, ACDN includes novel application placement algorithms based on access proximity and replica load. However, in ACDN each application-logic is treated as one atomic unit, specified by a metafile. This feature may restrict ACDN inferring optimal application deployment strategy as we have shown in the simulation.

[11], [12] proposed active cache architectures, that use applets, located in the proxy cache, to customize objects that could otherwise not be cached. When a request for personalized content is first issued, the original server provides the object and any associated cache applets. Once subsequent requests are made for that same content, the cache applets perform functions locally that would otherwise more expensively be performed at the original server. Thus, applets enable customization while retaining the benefits of caching. However, there is not much performance consideration on active cache yet. Moreover, active cache deems applet as one atomic unit, which may restrict the system inferring efficient application deployment strategy.

7 Conclusion

This paper describes an Application Networking architecture that facilitates dynamic application deployment to intermediate proxy nodes through workflow. We propose a modulation method to define workflow in formulized structure and manipulate it in uniform operations. Our simulation results show that using workflow and dynamic application deployment, App.Net can achieve better performance than conventional methods. However, a complete and feasible application deployment strategy will be a part of our future work.

References

[1] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul, "An Active Transcoding Proxy to Support Mobile Web Access", *Proceedings of IEEE Symposium on Reliable Distributed Systems,* 1998.

[2] A. Beck, M. Hofmann, "Enabling the Internet to Deliver Content-Oriented Services", *Proceedings of 6th International Workshop on Web Caching and Content Distribution,* June 2001.

[3] W. S. Shi, V. Karamcheti, "CONCA: An Architecture for Consistent Nomadic Content Access", *Proceedings of Workshop on Cache, Coherence, and Consistency,* 2001.

[4] A. Singh, A. Trivedi, K. Ramamritham, P. Shenoy, "PTC: Proxies that Transcode and Cache in Heterogeneous Web Client Environments", *Proceedings of the 3rd International Conference on Web Information System Engineering,* 2002.

[5] V. Mastoli, V. Desai, W. S. Shi, "SEE: A Service Execution Environment for Edge Services", *Proceedings of the 3th IEEE Workshop on Internet Applications,* June 2003.

[6] A. Barbir, R. Chen, M. Hofmann, etc., "An Architecture for Open Pluggable Edge Services", http://www.ietf.org/internet-drafts/draft-ietf-opes-architecture-04.txt

[7] A. Fox, S. D. Gribble, Y. Chawathe, "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives", *in 'A Special Issue of IEEE Personal Communication on Adaptation',* 1998.

[8] V. Cardellini, P. S. Yu, Y Huang, "Collaborative Proxy System for Distributed Web Content Transcoding", *Proceedings of the 9th International Conference on Information and Knowledge Management,* Nov. 2000, pp.520-527.

[9] W. Ma, B. Shen, J. Brassil, "Content Services Network: The Architecture and Protocols", *Proceedings of 6th International Workshop on Web Caching and Content Distribution,* June 2001.

[10] M. Rabinovich, Z. Xiao, A. Aggarwal, "Computing on the Edge: A Platform for Replicating Internet Applications", *Proceedings of 8th International Workshop on Web Caching and Content Distribution,* Sept. 2003.

[11] P. Cao, J. Zhang, K. Beach, "Active Cache: Caching Dynamic Contents on the Web", *Proceedings of IFIP International Conf. on Distributed Systems Platforms and Open Distributed Processing,* Sept. 1998.

[12] B. Knutsson, H. Lu, J. Mogul, "Architecture and Performance of Server-directed Transcoding", *Proceedings of ACM Transaction on Internet Tech.,* Vol.3 Iss.4, 2003, pp. 392-424.

[13] M. Hori, G. Kondoh, etc., "Annotation-Based Web Content Transcoding". *Proceedings of 8th International World Wide Web Conference,* May 1999.

[14] J, Chen, B. Zhou, J. Shi, H-J. Zhang, Qiufeng Wu, "Function-based Object Model towards Website Adaptation", *Proceedings of 10th International World Wide Web Conference,* May 2001.

[15] Y. Chen, W. Ma, H. Zhang, "Detecting Web Page Structure for Adaptive Viewing on Small Form Factor Devices", *Proceedings of 12th International World Wide Web Conference,* May 2003.

[16] R. Hanrahan, R. Merrick, "Authoring Techniques for Device Independence", http://www.w3.org

[17] A. Maheshwari, A. Sharma, K. Ramamritham, P. Shenoy, "TranSquid: Transcoding and Caching Proxy for Heterogenous E-Commerce Environments", *Proceedings of 12th IEEE Workshop on Research Issues in Data Engineering,* 2001.

Data Integrity Framework and Language Support for Active Web Intermediaries

Chi-Hung Chi[1*], Xiao-Yan Yu[1], Wenjie Zhang[1], Chen Ding[2], and Weng-Fai Wong[1]

[1]School of Computing
National University of Singapore

[2]School of Computer Science
Ryerson University

Abstract. With the maturity of content adaptation technologies (such as I-CAP and OPES) in web intermediaries and of content personalization in the network, data integrity has already become a key concern when such technologies are deployed. To address this issue, we propose a data integrity framework with the following functionalities: (i) a server can specify its authorizations, (ii) active web intermediaries can provide value-added services in accordance with the server's intentions, and more importantly, (iii) a client is facilitated to verify the received message with the server's authorizations and intermediaries' footprints. With the actual implementation of our data integrity framework in web (content and proxy) servers and clients, we show the feasibility and practicability of our proposed design through its low performance overhead and possible data (re)use.

Keywords: Content Delivery, Content Adaptation, Web Intermediaries, Data Integrity.

1 Introduction

World Wide Web has already emerged from a simple homogeneous environment to an increasingly heterogeneous one. In today's pervasive computing world, users access information sources on the web through a wide variety of mobile and fixed devices. One key direction to provide better web quality services in such heterogeneous web environment is the real-time adaptive content delivery [19]. Basically, the research of focus is to investigate technologies and practical systems to provide more efficient and effective value-added services through real-time content transformation in the network. With the numerous efforts of technology development to handle real-time content transformation in proxy and wireless gateway in pervasive computing environment, working groups in the Internet Engineering Task Force (IETF) start to engage in the related protocol and API definition and standardization.

* Contact Author: `chich@comp.nus.edu.sg`

These include the Internet Content Adaptation Protocol (I-CAP) [18] and Open Pluggable Edge Services (OPES) working group [1].

The prosperity of the research on real-time content transformation by active web intermediaries draws great attention to the problem of data integrity. Since the same set of technologies supports the modification of a message on its way from a server to a client, independent of the authorization of the web intermediaries by the content server, how can the client trust the receiving message and how can the server ensure that what the client receives is what it intends to respond? In this paper, we would like to address this data integrity problem with our data integrity framework. With the actual implementation of our data integrity model and system framework in web (content and proxy) servers and clients, we demonstrate the feasibility and practicability of our proposed framework through its low performance overhead and accurate data (re)use.

2 Related Works

In this section, we review the development in real-time web content transformation and outline the existing mechanisms proposed to handle the data integrity problem. Most efforts in active web intermediaries are focused on the actual deployment of content adaptation technologies in an active web intermediary. [11] presents evidence that on-the-fly adaptation by active web intermediaries is a widely applicable, cost-effective, and flexible technique. [6] designs and implements a Digestor, which modifies requested web pages dynamically to achieve the best visual effect for a given display size. Mobiware [2] aims to provide a programmable mobile network that meets the service demand of adaptive mobile applications and addresses the inherent complexity of delivering scalable audio, video and real time services to mobile devices. [5] proposes a proxy based system MOWSER to facilitate mobile clients visiting web pages via transcoding of HTTP streams. [7] makes use of the bit-streaming feature of JPEG2000 to support scalable layered proxy-based transcoding with maximum (transcoded) data reuse.

With respect to data integrity, [3] analyzes most threats associated with the OPES environment (or most real-time content adaptation intermediaries in the network). Based on the dataflow of an OPES application, major threats to the content can be summarized as: i) unauthorized proxies doing services on the object, 2) authorized proxies performing unauthorized services on the object, and 3) inappropriate content transformation being applied to the object (e.g. advertisement flooding due to local advertisement insertion service). These threats may cause chaos to the content delivery service because the clients cannot get what they really request.

There have been proposed solutions for data integrity but their context is quite different from the new active web intermediary environment here. In HTTP/1.1, integrity protection [15] is a way for a client and a server to verify not only each other's identity but also the authenticity of the data they send. When the client wants to post data to the server such as paying bills, he will include the entire entity body of his message and personal information in the input of the digest function and send this digest value to the server. Likewise, the server will response its data with a digest value calculated in the same way. The pre-condition for this approach, however, is that the server knows who the client is (i.e. the user id and password are on the server). Moreover, if an adversary intercepts the user's information, especially its password, it can take advantage of it to attack the server or the client. Secure Sockets Layer [16] does a good job for the integrity of the transferred data since it ensures the security of the data through encryption. However, these methods do not meet the need of active web intermediary services because they do not support legal content modification in the data transmission process, even by the authorized intermediaries.

Recently, there are new proposals being put forward in the area of content delivery and real-time content adaptation on the web. To meet the need of data integrity for delta-encoding [12], [13] defines a new HTTP header "Delta-MD5" to carry the digest value of the reassembling HTTP response from several individual messages. However, this solution is proposed only for delta-encoding exclusively and is not suitable for active web intermediaries. VPCN [4] is another proposal to solve the integrity problem brought by OPES. It makes use of the concept similar to Virtual Private Networks [9] to ensure the integrity of the transport of content among network nodes. It also supports transformation on content provided that the nodes are inside the virtual private content network. Its main problems are the potential high overhead and the restriction of performing value-added web services by a small predefined subset of proxy gateways only. Other proposals [14] draft the requirements of data integrity solution in the active web intermediary environment. [8] proposes a XML-based service model to define the data integrity solution formally. However, these works are at their preliminary stages; they are just drafts or proposals without actual implementation to demonstrate the system feasibility and performance.

3 Data-Integrity Versus Security

Traditionally, data integrity is defined as the condition where data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed. However, in the context of active web intermediaries, we extend this definition to "the preservation of data for their intended use, which includes content transformation by the authorized, delegated web intermediaries during the data retrieval process". Note that the aim of data integrity is not to make the data secret to the client and the server. Usually, this is done with the help of XML and digital signature technologies. Data is embedded in the XML structure and is signed by XML digital signature to form a data-integrity message.

It is obvious that strong security methods such as encryption can keep data more secure than data integrity mechanisms do. Then why do we employ data integrity but not very strong traditional security methods? It stems from three aspects of consideration:

- *Value-Added Services by Active Web Intermediaries*

Once data transferred between a client and a server is encrypted, value-added services will no longer be possible by any web intermediaries. This reduces the potentials of content delivery network services.

- *Data Reusability*

  Since current encryption along the network link is an end-to-end mechanism, it is also impossible for any encrypted data to be reused by multiple clients. This has great negative impact to the deployment and efficiency of proxy caching.
- *Cost-Performance*

  A large proportion of data on the internet are not content sensitive. That is, there is no harm if the data is visible to anyone. In this case, it is not necessary to keep the data invisible via very strong security methods because of the high performance cost of traditional encryption processes.

4 Basic Data Flow Model

In this paper, we focus on the data integrity problem for messages that are transferred based on the HTTP protocol. Hence, a client can be either a proxy or a web browser so long as it is an end point of a HTTP connection. Detail study shows that data integrity problem can actually occur in both the HTTP request and the HTTP response. Here, we mainly focus on the latter situation in the rest of the paper because the former situation can be considered as a simple case to the latter one. In the HTTP request, the type of requests that should be interested to data integrity research is one using the POST method, where a message body is included in the request. Compared to the HTTP response, it should be much easier to construct a data-integrity message embedded in an HTTP request. There are much fewer scenarios for web intermediaries to provide value-added services to the request and the construction method should be similar to those for the HTTP response. More importantly, there is no need to consider the reuse of the POST request. As it will be shown later, the feasibility for the reuse of data-integrity message is a key design consideration in our system framework. Furthermore, a data-integrity message that we study here must be in "text/xml" MIME type because this is the only data type that web intermediary services might work on.

To understand the data integrity message exchange model, we need to firstly understand the basic dataflow process for the server to response to a client's request. In the discussion below, we assume the most general case in which the requested object is not found in the web intermediaries (or proxy caches) along the data transfer path.

Given a HTTP request and response, there are five stages along its round trip path. The first two stages depict the situation from a client to a server whereas the next three stages refer to the situation from the server back to the client:

0. (Pre-Stage): A content server decomposes a given object such as a HTML page container object into parts (or fragments) and specifies a subset of the parts to predefined web intermediaries for real-time content adaptation. Note that this is done offline and can be considered as the initial preparation stage.
1. A client submits a HTTP request to the content server for an object retrieval.

2. The HTTP request reaches the server untouched. This is the assumption that we make here to ease our discussion (i.e. we focus on the discussion for the HTTP response).
3. The server responds with a data-integrity message over HTTP. The message contains the decomposed object and the server's authorization information for content modification.
4. The authorized web intermediaries that are on the return path from the server to the client provide value-added services to the object according to the server's specification. They will also describe what they have done in the message, but they will not validate the received message (for the sake of simplicity and low performance overhead).
5. The client has the option to verify the received data-integrity message via the specifications of server's authorization and active web intermediary's footprints. If any inconsistency between the server's authorization and the footprints is found, the client will handle this by his local rules. Some possible actions are: discarding the content with errors and showing users with the content left, or resending the request to the server.

From the above discussion, we can see that there will be three main components in the data integrity framework. The first one is the markup language specification. It should allow a server to specify its authorization intention for web intermediaries to take the appropriate content transformation action. It should also support web intermediaries to leave their footprints about the actions they have done. The second and third ones are related to the system model for the framework. The system architecture should have the data integrity modification component which supports an authorized web intermediary to provide its service. It should also have a data-integrity verification component to allow clients who concern about data integrity to verify the receiving message. Of course, on top of all these, attention should be paid on the performance overhead incurred. This is to make sure the feasibility and practicability of our proposal.

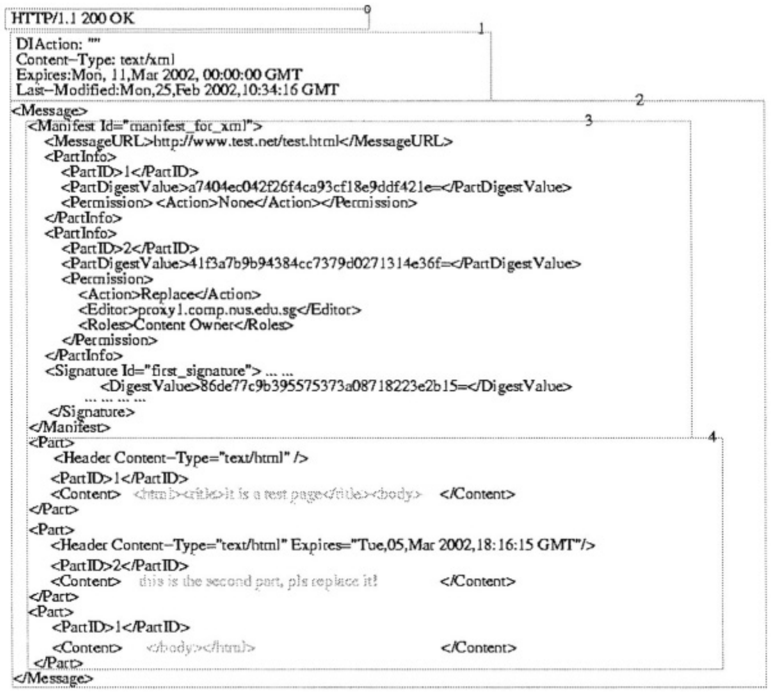
4.1 Examples of Data-Integrity Messages

Let us first start with a simple example to illustrate what might happen in the active network environment with web intermediaries for value-added features. Figure 1 shows a sample HTML object and Figure 2 and 3 show the two typical data integrity message examples as the object is transferred along the return retrieval path. There are two parts of the object that the server would like to send to a client. The first part is a (untouched) data-integrity message as a HTTP response to the client. The second part is a message to be modified by web intermediary as it is sent from the server to the client.

```
  <html><title>it is a test page</title>
  <body>this   is   the   second   part,   pls   replace
it!</body></html>
```

Fig. 1. A Sample HTML Object

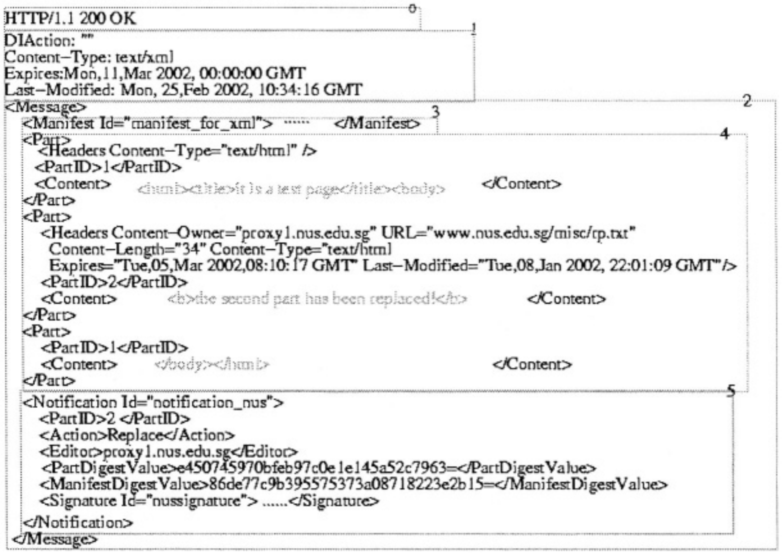
When the data integrity technique is applied to the object, the server might convert it into one shown in the Box-2 of Figure 2. The content of the original `HTML` object is now partitioned into two parts (shown in Box-4). Its authorization intention for content modification is specified in Box-3: While no one is authorized to modify the first part, a web intermediary, `proxy1.comp.nus.edu.sg` might adapt the content of the second part to some local information.



```
HTTP/1.1 200 OK
DIAction: ""
Content–Type: text/xml
Expires:Mon, 11,Mar 2002, 00:00:00 GMT
Last–Modified:Mon,25,Feb 2002,10:34:16 GMT
<Message>
  <Manifest Id="manifest_for_xml">
    <MessageURL>http://www.test.net/test.html</MessageURL>
    <PartInfo>
      <PartID>1</PartID>
      <PartDigestValue>a7404ec042f26f4ca93cf18e9ddf421e=</PartDigestValue>
      <Permission> <Action>None</Action></Permission>
    </PartInfo>
    <PartInfo>
      <PartID>2</PartID>
      <PartDigestValue>41f3a7b9b94384cc7379d0271314e36f=</PartDigestValue>
      <Permission>
        <Action>Replace</Action>
        <Editor>proxy1.comp.nus.edu.sg</Editor>
        <Roles>Content Owner</Roles>
      </Permission>
    </PartInfo>
    <Signature Id="first_signature"> ... ...
          <DigestValue>86de77c9b395575373a08718223e2b15=</DigestValue>
        ... ... ... ...
    </Signature>
  </Manifest>
  <Part>
     <Header Content–Type="text/html" />
     <PartID>1</PartID>
     <Content>  <html><title>It is a test page</title><body>    </Content>
  </Part>
  <Part>
     <Header Content–Type="text/html" Expires="Tue,05,Mar 2002,18:16:15 GMT"/>
     <PartID>2</PartID>
     <Content>  this is the second part, pls replace it!           </Content>
  </Part>
  <Part>
     <PartID>1</PartID>
     <Content>    </body></html>                                  </Content>
   </Part>
</Message>
```

Fig. 2. Data-Integrity Message in a Server

When the server receives a client's `HTTP` request for the object, it will combine the message body (in Box-2) with the message headers (in Box-0 and Box-1) into a data integrity message shown in Figure 2 and send it to the client as the `HTTP` response. As the message passes through the proxy `proxy1.comp.nus.edu.sg`, this intermediary will take action as specified in the message. It modifies the second part of the message and then adds a notification to declare what it has done. This is shown in Figure 3. The transformed data-integrity message that the client receives will now consist of the original message headers (in Box-0 and Box-1), the original server's intentions (in Box-3), the modified parts (in Box-4) and the added notification (in Box5) as the web intermediary's footprint.



```
HTTP/1.1 200 OK
DIAction: ""
Content-Type: text/xml
Expires:Mon,11,Mar 2002, 00:00:00 GMT
Last-Modified: Mon, 25,Feb 2002, 10:34:16 GMT
<Message>
  <Manifest Id="manifest_for_xml"> ······   </Manifest>
  <Part>
    <Headers Content-Type="text/html" />
    <PartID>1</PartID>
    <Content>   <html><title>it is a test page</title><body>     </Content>
  </Part>
  <Part>
    <Headers Content-Owner="proxy1.nus.edu.sg" URL="www.nus.edu.sg/misc/rp.txt"
     Content-Length="34" Content-Type="text/html"
     Expires="Tue,05,Mar 2002,08:10:17 GMT" Last-Modified="Tue,08,Jan 2002, 22:01:09 GMT"/>
    <PartID>2</PartID>
    <Content>      <b>the second part has been replaced!</b>        </Content>
  </Part>
  <Part>
    <PartID>1</PartID>
    <Content>     </body></html>                                  </Content>
  </Part>
  <Notification Id="notification_nus">
    <PartID>2 </PartID>
    <Action>Replace</Action>
    <Editor>proxy1.nus.edu.sg</Editor>
    <PartDigestValue>e450745970bfeb97c0e1e145a52c7963=</PartDigestValue>
    <ManifestDigestValue>86de77c9b395575373a08718223e2b15=</ManifestDigestValue>
    <Signature Id="nussignature"> ......</Signature>
  </Notification>
</Message>
```

Fig. 3. Data-Integrity Message after the Modification by a Web Intermediary

5 Language Support in Data Integrity Framework

In this section, we will briefly describe the basic format/structure of the language for our data integrity framework. Since this paper focuses more on the system architecture and the performance issues, readers should refer to [20] for the details of the formal schema and the detailed description of the language syntax.

Our data integrity framework follows naturally the HTTP response message model to transfer data-integrity messages. Under this framework, a data-integrity message contains an entity body so that a server can declare its authorization on a message, active web intermediaries can modify the message and clients can verify it. However, it should also be backward compatible such that a normal HTTP proxy can process the non-integrity part of the response without error.

The format for the data-integrity message in our framework is shown in Figure 4. Their details are as follows:

```
HTTP Status_Line
General \ Response \ Entity Headers
CRLF
(Manifest)+
(Part Headers
  Part Body)+
(Notification)*
```

Fig. 4. Message Format (where "+" denotes one or more occurrences and "*" denotes zero or more occurrences)

- *Status Line*

The status line in a data-integrity message is defined in the same way as that in a normal `HTTP` response message. The semantics of the status codes also follows those in `HTTP/1.1` for status communication. For example, a `200` status code indicates that the `HTTP` request is received successfully and a data integrity message is responded by the server to the client. Note that in the active web intermediary environment, the status code might not reflect errors that occur in the network (such as the abuse of information by the web intermediaries).

- *Headers*

Generally speaking, the message headers are consistent with those defined in `HTTP/1.1` [10]. However, some headers might lose their original meanings due to the change of the operating environment from object homogeneity to heterogeneity. Take "`Expires`" as an example. This header shows the expired time of the entire (homogeneous) object. However, when different web intermediary proxies do services on different parts of the same object, each of the parts within the object might have its own unique "`Expires`" date. This results in ambiguity in some of the "global" header fields under this heterogeneous environment for fragments. Furthermore, we also need "DIAction", an extended HTTP response header field to indicate the intent of a data-integrity message (See [20] for details).

- *Message Body*

The entity body consists of one or more "manifests", one or more "parts", and zero or more "notifications". They are the important components of our language.

Manifest: A server should provide a manifest to specify its intentions for authorizing intermediary proxies to perform value-added services on the message. A manifest might also be provided by an intermediary proxy who is authorized by the server for further task delegation.

Part: A part is the basic unit of data content for manipulation by an intermediary. The party who provides a manifest should divide the object (or fragment of an object) into parts, each of which can be manipulated and validated separately from the rest. An intermediary proxy should modify content in the range of an authorized part, and a client might verify a message in the unit of a part. A part consists of a `PartID` (a sequence number used to identify which part or fragment in an object), part headers and a part body. Note that a part can be defined recursively, i.e. a part can be divided further into sub-parts.

Notification: A notification is the footprint about the content modification of a part that an authorized proxy performs..

Note that the entity body of a message body might be encoded via the method specified in the "Transfer-Encoding" header field (See [10] for details).

6 System Architecture Model

In this section, we will propose a system architecture model for our Data Integrity Framework. Our system architecture consists of the components shown in Figure 5. The description of the modules of each component is depicted as follows.

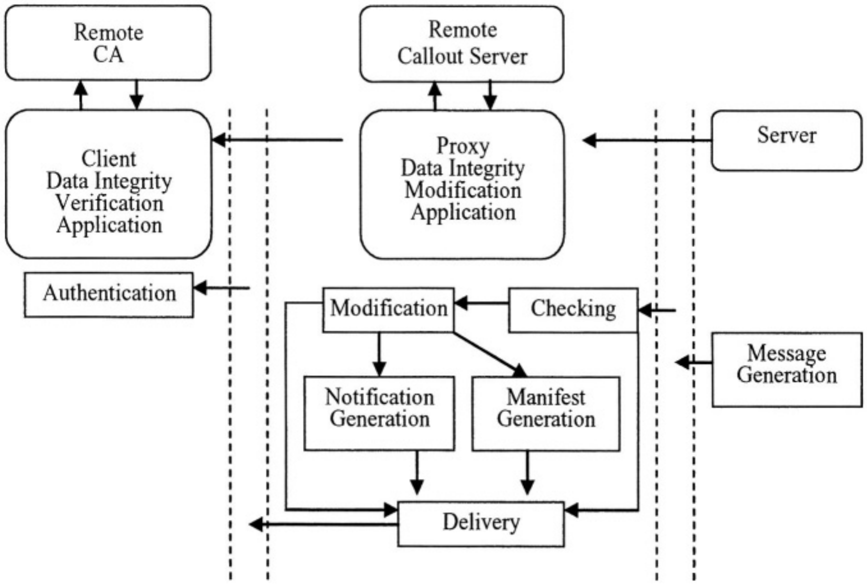


Fig. 5. System Architecture for Data-Integrity Framework

6.1 Message Generation Module

When a server receives a web request, it will generate the HTTP response message header and will transfer the header followed by the data-integrity message body.

6.2 Data-Integrity Modification Application

The data-integrity modification application is made up of five modules: scanning, modification, notification generation, manifest generation, and delivery.

- *Scanning Module.* This module is to recognize a data-integrity message - to look for authorizations, and to find the authorized part.
- *Modification Module.* After a proxy knows what it should do on a part, it will accept some of them according to its local rules. For a part that is authorized to and accepted by the proxy, the proxy will perform its service on that part through two steps: i) to modify the headers of the part; and ii) to do the authorized action.
- *Notification Generation Module.* When it is time to count the digest value of a modified part, the process comes to this module. After the digest value is obtained, it will also generate a notification to declare what the proxy has done.
- *Manifest Generation Module.* When the proxy intends to give a child manifest, this module will be called to generate a manifest. The manifest will be put at the end of all the manifests by now.

- *Delivery Module.* The proxy delivers all the ready packets. If it is time to deliver the last packet, it will append the generated notifications (if any) to the last packet and deliver it.

If errors occur in some module (which we call them “application errors”), the intermediary proxy will specify them in the `Warning` header of the parts where errors occur.

6.3 Data-Integrity Verification Application

One important feature of our Data Integrity Framework is that a client can verify what it receives is what the server intends to give to it. If the client wants to verify data-integrity responses, it should employ a data-integrity verification application. Of course, the client can choose not to install the application and treat a data-integrity response as a normal response.

Data-Integrity Verification Application has one module, Authenticating Module. It consists of the following functions:

- *Data-integrity responses differentiation:* The application can differentiate data - integrity responses with other responses via “`DIAction`” header field of the response. Furthermore, it does not affect the process of other responses.
- *Certificate authentication:* To verify that a server’s manifest is signed by the server, or some proxy’s manifest and notification are signed by the proxy, we need its real public key. To check the authenticity of the public key in its certificate as one of the elements of the XML Digital Signature, the application will verify the certificate authority (CA) locally or remotely that provides the trustworthy certificate. This method is a typical public key verification [17].
- *Manifest authentication:* The digital signature on a manifest can be verified with the help of its public key so that it is easy to find out if the manifest from a server or from some delegated proxies is modified or replaced by malicious intermediaries during the transmission. It also checks if a delegatee gives a child manifest within its authority.
- *Declaration authentication:* In order to verify that a proxy declares to do what it is authorized to do on a part, the application needs to find the notification of the proxy and all the authorizations on the part to the proxy from the manifest first. Then it matches what the proxy declares to do with the authorizations gotten.
- *Part authentication:* The application can make known the authenticity of the received part after it knows who finally touches the part with what kinds of action. Through the proxy’s host name in the last header line of a part, the corresponding notification can be found easily. As for “`Delete`” or “`Replace`” action, the received one is authentic if its digest value is the same as the one declared in the notification. It is a nested procedure to verify a part on which the final action is “`Transform`”. Besides verifying the digest values of the counted and the declared, the application should verify the input of the transformation. It can identify who touches the part before the proxy transforms it and verifies if the input is provided by the server or a verified proxy. For “`Delegate`” action, the part without sub-parts is authentic if its digest value matches the one in the notification. The part with sub-parts is authentic if all its sub-parts are authentic.

7 Conclusions

In this paper, we proposed a data integrity framework with the following functionalities that a server can specify its authorizations, active web intermediaries can provide services in accordance with the server's intentions, and more importantly, a client is facilitated to verify the received message with the server's authorizations and intermediaries' traces. We actually implemented the proxy-side of the framework on top of the Squid proxy server system and its client-side with Netscape Plug-in SDK. With this prototype, we illustrate the practicability of our proposal through its low performance overhead and the feasibility of data reuse.

References

[1] [Online]. Available: http://www.ietf-opes.org

[2] O. Angin, A. Campbell, M. Kounavis, and R. Liao, "The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking," *IEEE Personal Communications Magazine,* August 1998.

[3] A. Barbir, O. Batuner, B. Srinivas, M. Hofmann, and H. Orman, "Security threats and risks for OPES," Feb 2003. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-opes-threats-02.txt

[4] A. Barbir, N. Mistry, R. Penno, and D. Kaplan, "A framework for OPES end to end data integrity: Virtual private content networks (VPCN)," Nov 2001. [Online]. Available: http://standards.nortelnetworks.com/opes/non-wg-doc/draft-barbir-opes-vpcn-00.txt

[5] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul, "An active transcoding proxy to support mobile web access," in *Proc. the IEEE Symposium on Reliable Distributed Systems,* 1998.

[6] T. W. Bickmore, and B. N. Schilit, "Digestor: Device-independent access to the World Wide Web," *Computer Networks and ISDN Systems,* vol. 29, no. 8–13,1997, pp. 1075–1082.

[7] C.-H. Chi, and Y. Cao, "Pervasive web content delivery with efficient data reuse," in *Proc. International Workshop on Web Content Caching and Distribution,* August 2002.

[8] C.-H. Chi, and Y. Wu, "An XML-based data integrity service model for web intermediaries," in *Proc. International Workshop on Web Content Caching and Distribution,* August 2002.

[9] B. G. et al, "A framework for IP based Virtual Private Networks," Feb 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2764.txt

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L.Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol–HTTP/1.1," 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[11] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer, "Adapting to network and client variation using active proxies: Lessons and perspectives," in a special issue of *IEEE Personal Communications on Adaptation,* 1998.

[12] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for HTTP," in *Proc. SIGCOMM,* 1997, pp. 181–194.

[13] ——, "Potential benefits of delta encoding and data compression for HTTP (corrected version)," Dec 1997. [Online]. Available: http://citeseer.nj.nec.com/mogul97potential.html

[14] H. K. Orman, "Data integrity for mildly active content," Aug 14 2001. [Online]. Available: http://standards.nortelnetworks.com/opes/non-wg-doc/opes-data-integrityPaper.pdf

[15] S. Thomas, *HTTP Essentials.* New York: Wiley, John and Sons, Incorporated, 2001, Chapter on Integrity Protection, pp. 152 – 156.
[16] ——, *HTTP Essentials.* New York: Wiley, John and Sons, Incorporated, 2001, Chapter on Secure Sockets Layer, pp. 156 – 169.
[17] ——, *HTTP Essentials.* New York: Wiley, John and Sons, Incorporated, 2001, Chapter on Public Key Cryptography, pp. 159 – 161.
[18] I-CAP: Internet Content Adaptation Protocol [Online]. Available: http://www.i-cap.org
[19] X. Li, "Quality-Based Content Delivery on Internet," Ph.D Thesis, School of Computing, National University of Singapore, 2004.
[20] C. H. Chi, X.Y. Yu, "Data Integrity for Active Web Intermediaries," Internal Report, School of Computing, National University of Singapore, 2004

Xeja: A Scalable Channel-Based Multi-source Content Distribution System

Pei Zheng[1,2] and Chen Wang [2]

[1] Department of Computer Science, Arcadia University
Glenside, PA 19038, USA
zheng@arcadia.edu

[2] Department of Computer Science and Engineering, Michigan State University,
East Lansing, MI 48823, USA
{zhengpei, wangche7}@cse.msu.edu

Abstract. Popular large-scale P2P content distribution systems employ a multi-source parallel downloading scheme to allow peers serve each other in a horde. Usually those systems rely on a central server to provide a directory service, which may become a bottleneck while the population of peers increases rapidly. We propose Xeja, a scalable P2P multi-source content distribution system to solve the problem. Xeja has a flexible channel-based distributed directory service to provide each peer an aggregate of content index automatically collected from peers who share some common interest. In addition, Xeja leverages the coordination between peers in a channel hierarchy in order to improve content availability. Simulation results show that Xeja can be built as an overlay on top of existing multi-source content distribution systems, and the distributed directory service effectively extends the available time of the entire file due to sub-seeding.

1 Introduction

The Internet in a sense is a large content distribution system. Unlike commercial proprietary content delivery networks such as Inktomi, an open distributed large scale content distribution overlay can be built in the Internet, which enables any user to share and distribute any content among a vast number of peers at any time. There has been a great amount of effort on improving the performance of such systems in terms of overall file download speed and bandwidth consumption. Well-known systems such as BitTorrent [1] and eDonkey/Overnet [2] are able to dramatically increase file download speed by leveraging multi-source parallel download of file pieces among a group (the horde) of peers.
A major drawback of existing large scale content distribution systems is the directory service, which generally consists of an index server and a tracker server. The index server (e.g., a web site) hosts all the metadata files of shared content. A tracker server is a central host to maintain state information of all clients. In effect, such a directory service does not scale well as it cannot accommodate a large number of requests

when the population of the system increases rapidly, as described in [3]. Moreover, since there is no collective search service available, a user has to visit numerous index server sites to look for a specific content.

Another drawback of existing large scale content distribution systems is the seeding problem due to the asynchronous arrival of download requests: A seed, a peer with the complete piece set of the file, may leave the horde after serving for some time. When all the seeds are offline, it is very likely that there are some missing pieces among all the active peers in the horde. Accordingly, those active peers will not be able to download the entire file, even if they just miss one or two pieces. Since there is an opportunity that a peer can choose pieces to download/upload from/to others, some piece distribution algorithms are designed as an effort to evenly distribute all the pieces based on the peer's local knowledge of piece distribution. However, such schemes require a peer to maintain piece state information, which maybe largely biased, thereby making the algorithm hard to design and less effective.

In this paper, we propose Xeja, a scalable P2P multi-source content distribution system to solve those problems. Xeja employs a flexible channel-based distributed directory service scheme to provide each client an aggregate of content index automatically collected from users who share some common interest. There is no directory server in Xeja. Additionally, Xeja leverages the coordination among peers in the channel hierarchy in order to improve availability of a complete piece set (the probability that a new peer is able to download the entire file from a horde). Rather than statistically optimizing piece distribution algorithms to evenly distribute all pieces of a file among all downloading peers, Xeja's download component is designed to guarantee the availability of a complete piece set. It dynamically distributes all the pieces of a file from a seed to a number of carefully selected sub-seeds, which act as subtrackers to maintain horde state information and direct peers to other horde members. As a result, even unpopular files will be available for download for a fairly long period of time.

The organization of the paper is as follows: Section 2 summarizes related literature and compares with our approach. In Section 3 and Section 4, the architecture of Xeja and important design issues are presented and discussed. The evaluation of Xeja is outlined in Section 5, followed by a conclusion of future work in the last section.

2 Related Work

The issue of large scale content distribution over the Internet has been actively researched lately due to the dramatic increase of population of broadband users. Traditional FTP-like approaches will fail as the high bandwidth consumption makes it impossible to accommodate hundred of thousands of users. Server mirroring can help enterprise content distribution only, where access to mirrored servers is restricted and unavailable to general public. Overlay multicast [4, 5] replaces IP multicast as a major scheme to address the bandwidth issue, largely because of IP multicast's lack of deployment in the current Internet. On the other hand, P2P based content distribution schemes emerged along with popular real working systems. To this end, a significant amount of effort has been made on file search using either structured [6, 7] or unstructured architecture [8-10]. A few P2P systems or content distribution networks are proposed to address the issue of scalable and efficient file download with either a stateful overlay [11, 12] or erasure coding [13]. But to our knowledge no open large scale content distribution architecture has been introduced in literature that provides both scalable distributed directory service and multi-source self-organized download service.

Slurpie [12] is cooperative multi-source data transfer system that aims to make the horde adaptable to network size and link bandwidth. Its major design goal is improving download performance with the assumption that the seed is always available. In contrast, Xeja's design goal is to provide a distributed directory service to improve file availability and system scalability, which are different aspects of the whole issue.

Unlike [13], Xeja takes a stateful approach to form a download horde, and effectively addresses the limitations of stateful systems such as seeding and tracker server problems by Proactive Piece Seeding and sub-tracking.

BitTorrent [1, 3] is a file distribution system that relies on an index web server and a tracker server for content location and downloader hoard state maintenance. The tracker server collects and maintains the state information of all the hordes in order to direct downloaders to others for piece exchange. Thus the tracker may not be able to service when the peer population increases rapidly. Furthermore, in BitTorrent, the availability of a complete file is largely determined by the popularity of the file. Xeja is designed to offer a channel-based distributed directory service without using a central tracker server. Within a channel the seeding problem is solved by sub-seeding and sub-tracking, which will be discussed in the next section.

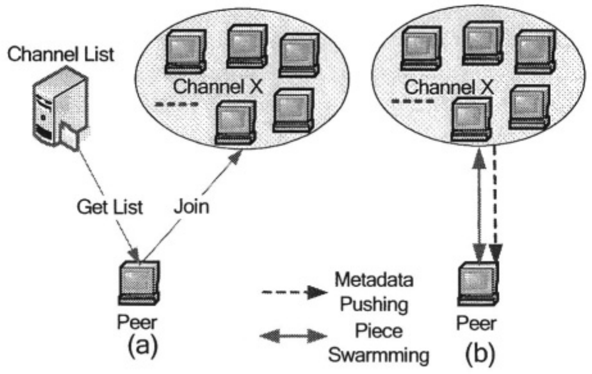


Fig. 1. Xeja architecture

3 Xeja Architecture

Xeja is large scale, channel-based content distribution overlay utilizing self-organized multi-source parallel download and state maintenance. It is designed for one-to-many, large file distribution over the Internet. There are three components in the Xeja architecture: Channel Directory Service (CDS), Proactive Piece Seeding (PPS), and sub-tracking.

CDS is the service to automatically locate content a peer may be interested in the peer's subscribed channels. A channel is a closely maintained online community of peers who share some common interest. As shown in Fig. 1, a peer chooses to join Xeja by first identifying some channels she is interested with. The list of available channels can be published on a web server. The namespace of channels follows a hierarchy rather than a flat list. Essentially each peer in Xeja subscribes to a number of channels, within which she is able to download content metadata and interested files published in that channel with the help of other peers in the same channel. In effect, a centralized search namespace has been structured into channels. File search and download are conducted only in those channels. Accordingly, Xeja will show better scalability in handling large number of users.

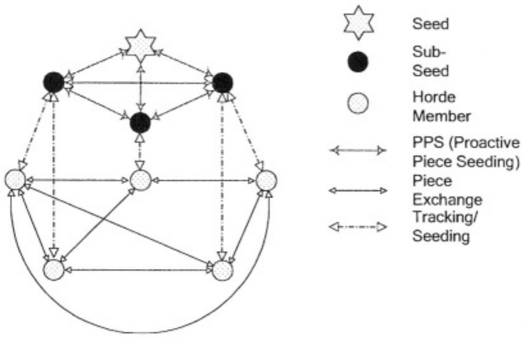


Fig. 2. PPS in Xeja

The basic idea of PPS is that pieces of a file will be evenly distributed to some carefully selected peers in the channel, namely Xeja core. The sub-seeds will provide multiple copies of each piece of the file. As a result, even if no seeds are available at some time, a peer can still download the entire file successfully with the help of those sub-seeds. Disk spaces used by PPS on sub-seeds can be reclaimed after a period of time specified by a user. At each horde member the locally maintained upload/download rates of its connected horde members are used to determine corresponding outgoing data upload rate. In addition, Xeja employs cross-horde coordination among horde members and sub-seeds in order to further improve individual download rate. Being a sub-seed substantially increases the effective upload rate of the peer, making it possible to achieve higher download rate while being served by other peers.

Sub-tracking is a mechanism to eliminate the central tracker server by distributing horde state maintenance task over sub-seeds. A horde member, once connected to a sub-seed, will report state information to the sub-seed. The sub-seed, also a sub-tracker now, will periodically broadcast a list of all its connected horde members to other sub-trackers. An example of PPS is shown in Fig. 2. The seed and three sub-seeds serve five horde members, each connecting to a sub-seed for horde state update.

4 Design Issues

4.1 CDS (Channel Directory Service)

Xeja adopts a channel hierarchy similar to Google groups [14], formerly known as Deja. A channel is uniquely identified by a string in the format of "a.b.c.d.e.f...", whereby each letter representing a special keyword. As the ID implies, a channel always resides in a hierarchical namespace. Peers in a lower-level channel are also members of its upper-level channel. A peer only maintains a persistent bilateral relationship with some number of peers in a channel, resulting in a loosely connected channel member overlay. When a new peer chooses to subscribe to a channel, it will initiate a *join* operation, which takes a simple broadcasting approach if the peer does not know anyone in the channel; otherwise, the peer will just ask those peers who are already in the channel. Note that during bootstrapping, a new peer is always connected to some peers who are members of high-level channels. When a peer chooses to unsubscribe from the channel, only its neighbors in the channel will be notified and updated accordingly. It is possible that two clusters of peers are not directly connected even if there are in the same channel. However, the join operation serves as a remedy for this problem: it is very likely that when a new peer broadcasts a join channel request, she will be able to receive lists of peers in the same channel from multiple separated clusters if any. Therefore the two clusters of the same channel are connected through the joining peer.
All the peers in a channel are ready to receive publishing announcement in the form of a metadata file. To publish a new content in a channel, a peer (the initial seed) will create a small metadata file and transmit it to other peers in the channel. The metadata file consists of two parts: the fist part is the necessary descriptive information of the shared content; the second is a summary of all the pieces for the file, including the number of pieces and each piece's hash code. Within the channel, the metadata file will be distributed to all the peers the initial seed can reach.

4.2 Sub-seed Selection

A peer in Xeja has two working modes for each channel: the content mode and the metadata mode. In the content mode, a peer is willing to receive whatever content published in the channel without prompting the user for approval; in the metadata mode, a peer will only receive metadata files published in the channel. After that the user has an opportunity to decide whether or not to download the file after reviewing the metadata file. In reality, the content mode is designed for Xeja users who are optimistic about the content published in her subscribed channels.
To publish a new content in a specific channel, after publishing the metadata file, the initial seed will try to find a number sub-seeds to conduct PPS. Sub-seeds are chosen among peers who are operating in content mode in the channel. The initial seed will send a *sub-seeding request* to each content-mode peer in the channel for a report of local machine information, including CPU, memory size, network connection types, data upload rate, a vector of time of last *m* online durations, recent data upload rate,

and previous seeding and sub-seeding history. The sub-seeding request is further forwarded at each peer in order to reach many content-mode peers. To avoid too much traffic overhead, a TTL will be set for the sub-seeding request to limit the scope of the sub-seeds searching. Then those candidates with a higher serving capability will be selected as sub-seeds. If enough number of sub-seeds is available for sub-seeding, the initial seed can proceed to conduct PPS. Otherwise, the initial seed simply gives up PPS at this time. The initial seed or other seeds may try to conduct such a sub-seeding request operation on a timely basis for a number of times until there is enough number of sub-seeds available for PPS.

4.3 PPS (Proactive Piece Seeding)

PPS (Proactive Piece Seeding) is the piece distribution algorithm among sub-seeds based on a piece distribution tree. Let *r* be the *redundancy vector,* which denotes the number of available copies of each piece of the file among the seed and sub-seeds. Depending on *r* and the number of complete copies of the file the initial seed needs to maintain, the initial seed will find some number of sub-seeds to form a Xeja core, and copy *r* number of pieces to each of them. The Xeja core keeps an updated PPS table as who has which pieces at each sub-seed. Between the sub-seeds, a check-piece message is exchanged in a certain interval to exchange information of available pieces. If a sub-seed leaves the core and the initial seed is still working, the initial seed will have to find another sub-seed, and inform other sub-seeds of this change. If the initial seed has left the core and then a sub-seed is about to leave, one of the sub-seeds will conduct a sub-seed searching operation among content-mode peers for a new sub-seed.

A sub-seed will have a PPS table of available pieces at each sub-seed in the horde. The reason to maintain a global PPS table among sub-seeds is based on two observations. One is that sub-seeds are quite stable peers that do not frequently go online/offline (they are in content-mode anyway), so maintenance cost of a complete PPS table is not very high. The other observation is that generally the size of a PPS table is not large as the number of pieces is at most several thousands in reality with a piece size of less than 1 Mbytes. In order to adapt to increasing popularity of the file, the seed or some sub-seeds may decide to increase the redundancy of pieces. To achieve this, a sub-seed may simply act as a seed and start to build a PPS distribution tree. The only update information other sub-seeds and the seed need to know is that there are more pieces being serviced by the underlying sub-seed.

A sub-seed is also a sub-tracker in a Xeja core. It is responsible for maintaining download horde state information, which is a list of peers who are active downloading the content. Once asked, a sub-seed will tell interested peers a list of other horde members for piece exchange. A sub-seed maintains the entire horde state by exchanging horde member update information between each other in a gossip manner on a timely basis. Note that a state update will be gradually passed to all sub-seeds after a number of gossip rounds, resulting in some inconsistency of horde state on different sub-seeds during this process. However, as long as a horde member is able to receive a list of some active members, it can begin to exchange piece immediately while frequently checking for new active members at its connected sub-seeds.

4.4 Sub-tracking and File Download

In Xeja, file download operation is actually carried out in two stages, as shown in Fig. 1: swarming (piece exchange) among horde members, and downloading pieces unavailable among horde members form sub-seeds. In the first stage, by asking some sub-seeds, a horde member will be able to know other horde members who are now downloading the file. Those sub-seeds being asked then update their horde member list to reflect the update of horde state and gossip the change in the Xeja core. At the same time, the horde member can proceed to exchange bit filters of file pieces with each other and exchange pieces accordingly. If the first stage turns out to be successful, there will not be the second stage.
When the horde member cannot find any new pieces available for download from its connected peers, and the file is not complete, it can ask one of its sub-seed, which serves as an agent for the horde member to find out where to get the desired pieces from the Xeja core. The agent sub-seed will then consult its PPS table for a number of sub-seeds that can provide all the required pieces. Note that the PPS algorithm ensures that load of piece download will be evenly distributed among sub-seeds. Then the asking peer can proceed to download those missing pieces from those sub-seeds.
Designed for system scalability and content availability rather than overall performance, Xeja is able to adopt any piece scheduling algorithms such as the choking algorithm used in BitTorrent for overall performance. In addition, a special incentive mechanism is devised to encourage peers to be sub-seeds. Thus the sub-seed will enjoy higher data download rate compared with other non sub-seed peers.

5 Trace Analysis and Simulation

In this section, we first show an analysis on the traces we collected from BitTorrent. Based on that we conduct the simulation of Xeja to explore the performance and effectiveness of the channel-based distributed directory service and PPS mechanism.

5.1 Trace Analysis

The network dynamics and overall service characteristics of a large scale multi-source content distribution overlay are very sophisticated, and thus difficult to model and simulate. In order to conduct meaningful simulation of Xeja, we first collect traces from BitTorrent for our trace-driven simulation. We design and implement a BitTorrent SuperTracker program that acts as a dummy peer using the scraping interface defined BitTorrent protocol [1] to trace horde dynamics and tracker server statistics. The traces are collected for 88 hours consecutively in a 15-minute interval from a tracker server site from February 15 to February 18, 2004.
Given a tracker server hosting many hordes, the number of peers in each horde varies in a very wide range. Our traces on a tracker server show 93% of hordes have an average size (the number of members) of less than 20 members over the tracing pe-

riod, and 96% of hordes have an average number of seeds less than 3, as shown in Fig. 3. The traces further show that, among all those hordes, 296 of them (85%) fail to provide a seed at least once during our tracing period. The result implies even with rare-first piece distribution algorithms, a horde still has a high probability of suffering the seeding problem when it becomes unpopular at some point, thereby limiting the availability of those files. However, from the traces we notice that, although the number of peers across many hordes varies dramatically during the tracing period, the number of seeds serving those hordes seems quite stable. In fact, those seeds are believed to be the ones who have a strong intention to serve others in a multi-source content distribution overlay such as Xeja, in which they are likely to be those peers working in content-mode in a channel. Consequently, we argue that, based on the stable number of seeds in the traces, PPS among sub-seeds in Xeja is feasible, as there seems to be a large number of content-mode peers in a channel.

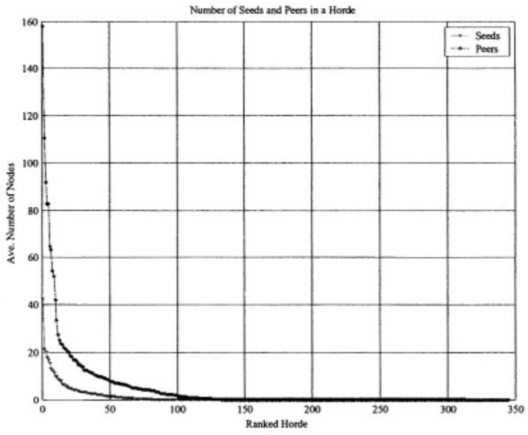


Fig. 3. Average number of seeds and peers across all hordes over the tracing period

5.2 Trace-Driven Simulation

The simulation of Xeja is based on the collected traces. For simplicity a horde is mapped into a channel in Xeja, although in reality a channel may host multiple hordes. The number of seeds and peers of each horde is obtained from our traces. Each shared file has 1024 pieces. The sub-seeds are selected by randomly choosing a specific number of seeds in other hordes, since we expect those seeds are likely to work in content mode in the channel. The redundancy of each piece is 2 to achieve a "mirror" effect. Pieces are evenly distributed among those selected sub-seeds. Note that each Xeja network serving many channels may need specific configuration of these system parameters.

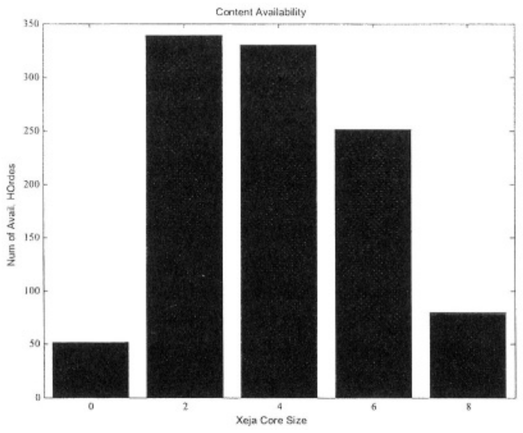


Fig. 4. Content availability against Xeja core size

The issue of improving content availability in Xeja is a combination of two factors: during the creation phase, the possibility of having enough sub-seeds to perform PPS; and during the seeding phase, the possibility of having enough qualified seeds and sub-seeds for PPS (in case some sub-seed leaves the horde) and seeding. In the simulation, we use 3-day as the interval of disk space reclamation on each sub-seed. Fig. 4 shows the availability of Xeja compared with BitTorrent, which can be considered to have a Xeja core size of 0. Compared with BitTorrent, Xeja with PPS shows a higher availability for shared files. It is also clear that when the core size is small, most hordes will be able to maintain at least one complete piece set. When the core size becomes larger, enough number of sub-seeds may not be always available for all hordes, resulting in failure of building a Xeja core. Thus the size of a Xeja core has to be carefully determined and dynamically adjustable based on the number of horde numbers. We are in the process of investigating a "slow-start" Xeja core scheme to achieve these goals.
The overhead traffic of PPS, which includes piece distribution cost and the maintenance cost due to leaving sub-seeds, is minimal compared with the traffic on a traditional tracker server. Result of overhead traffic is skipped due to space limit.

6 Conclusions and Future Work

We propose Xeja, a scalable P2P multi-source content distribution system that addresses both the issue of distributed directory service and the issue of content availability. Xeja employs a flexible channel-based distributed directory service scheme to provide each client an aggregate of content index automatically collected from users who share some common interest. In addition, Xeja leverages the coordination among peers in the channel hierarchy by utilizing the Proactive Piece Seeding scheme in order to improve availability of a complete set of file pieces.

We are in the process of investigating optimization issues of system parameters such as Xeja core size and redundancy number. Currently in Xeja a channel index server is used to publish a list of available channels in the system. We plan to eliminate the channel index server to allow any peer to create new channels and publish them in a fully distributed manner.

References

[1] The BitTorrent Protocol Specification v1.0, http://wiki.theory.org/index.php/BitTorrentSpecification, 2004

[2] eDonkey2000/Overnet, http://www.edonkey2000.com/, 2004

[3] B. Cohen, "Incentives Build Robustness in BitTorrent," *Proc. P2P Economics Workshop,* 2003.

[4] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole, "Overcast: Reliable Multicasing with an Overlay Network," *Proc. OSDI 2000,* San Diego, CA, 2000.

[5] Y. H. Chu, S. Rao, and H. Zhang, "A Case for End System Multicast," *Proc. ACM SIGMETRICS,* Santa Clara, CA, 2000.

[6] B.Y.Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-resilient wide-area location and routing," Technical Report UCB//CSD-01-1141, U.C.Berkeley 2001.

[7] R. M. Stoica, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," *Proc. ACM SIGCOMM,* 2001.

[8] P. Ganesan, Q. Sun, and H. Garcia-Molina, "Yappers: A Peer-to-Peer Lookup Service over Arbitrary Topology," *Proc. IEEE INFOCOM,* 2003.

[9] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," *Proc. the 16th ACM International Conference on Super-computing,* 2002.

[10] C. Wang, L. Xiao, Y. Liu, and P. Zheng, "Distributed Caching and Adaptive Search in Multilayer P2P Networks," *Proc. ICDCS'04,* 2004.

[11] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Split-stream: High-bandwidth Content Distribution in Cooperative Environments," *Proc. IPTPS'03,* 2003.

[12] R. Sherwood, R. Braud, and B. Bhattacharjee, "Slurpie: A Cooperative Bulk Data Trans-fer Protocol," *Proc. IEEE INFOCOM,* Hong Kong, 2004.

[13] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed Content Delivery Across Adaptive Overlay Networks," *Proc. SIGCOMM'02,* 2002.

[14] Google Groups/Deja, http://groups.google.com or http://www.deja.com

Segment-Based Adaptive Caching for Streaming Media Delivery on the Internet

ShaoHua Qin[1,3], ZiMu Li[2], QingSong Cai[1], and JianPing Hu[1]

[1] School of Computer Science, Beijing University of Aeronautics and Astronautics, Beijing 100083,China
shqin@buaa.edu.cn
[2] Network Research Center of Tsinghua University, Beijing 100084,China
zmli@cernet.edu.cn
[3] School of Physics & Information Engineering, Guangxi Normal University, Guilin, 541004, China

Abstract. Delivering streaming media objects on the Internet consume a significant amount of network resources. Segment-based proxy caching schemes have been an effective solution. In this paper, we investigate the bandwidth consumption of streaming multiple objects from a remote server through a proxy to multiple clients. We propose a caching strategy to realize the suffix segment data adaptively caching at proxy, and integrate it with multicast-based batched patching transmission strategy to develop a proxy—assisted scheme that can efficiently support to deliver media objects in the Internet-like environment. We then introduce a simple transmission cost model to quantify the overall usage of network bandwidth of our framework. Simulation results show that our scheme can reduce the consumption of aggregate transmission cost significantly under the small amount cache size at proxy.

Keywords: Streaming media, proxy caching, prefix caching, multicast, streaming media delivery

1 Introduction

The emergence of the Internet as a pervasive communication medium, and a mature digital video technology have resulted in a rapid growth of various networked streaming media applications. Example applications include video-on-demand, distance learning, video game, interactive television and video conferencing. However, due to the high bandwidth requirements and long-lived nature of the streaming objects, streaming such medium over the Internet needs to consume a significant amount of network and server bandwidths. Also, different clients are likely to be asynchronously issuing requests to receive their chosen media streams, this fact make the problem further complicated for efficiently distributing streaming media object across wide area networks.

This work is supported by the NNSF of China under grant 60103005.

Existing research has focused on developing transmission schemes that use multicast or broadcast connections in innovative ways to reduce server and network loads, for serving a popular video to multiple asynchronous clients. Batching [1], Patch [2], [8], Optimized Batch Patching [3] and HMSM[9] are reactive in that the server transmits video data only on demand, in response to arriving client requests. More recently work [4],[5],[10] extends the above techniques to streaming content distribution system consisting of remote servers and proxies close to clients, where proxies cache some media object locally and assist in transmitting the data streams from the server to clients. Much of the existing research has an underlying requirement that the multicast or broadcast connectivity between the server and the clients is available, they focused on optimizing the usage of the server bandwidth or the network bandwidth in an environment of multicast-capable network. However, IP multicast deployment in the Internet has been slow and even today remains severely limited in scope and reach. Therefore, transmission schemes that can support efficient delivery in such predominantly unicast settings need to be developed.

There has been much less work on optimizing network bandwidth usage when transmitting multiple multimedia objects to asynchronous clients in wide area Internet-like settings (the wide area server-proxy paths are only unicast capable while the local area proxy-client paths might be multicast capable). Work in [7] explores the combination of proxy prefix caching with proxy-assisted patching, streaming merging schemes to reduce the transmission cost over multiple heterogeneous videos. This work presented a technique to determine, for a given proxy-assisted transmission scheme, the optimal proxy prefix caching for a set of videos that minimizes the aggregate transmission cost. But it does not take into consideration the dynamic behavior of proxy suffix caching. In addition, the patching scheme still consumes significant bandwidth in the condition of high request arrival rate, and streaming merging scheme need client to switch among multiple data streams in order to dynamically aggregate clients into larger and larger groups that share streams while it is more efficient than patching.

In this paper, we investigate the combination of proxy prefix caching, segment-based suffix dynamically caching with proxy-assisted batched patching transmission scheme to reduce the transmission cost over multiple multimedia objects. An adaptive segment-based caching scheme is proposed to implement the adaptive suffix caching in terms of the request arrival distribution in each time interval, and accordingly achieves much more transmission cost reduction on the server-proxy path. In addition, the multicast-based batched patching method batches the requests to joint the patching stream so as to efficiently save proxy bandwidth. Our evaluation demonstrates that integrating such two schemes can efficiently distribute streaming media in the Internet-like environment.

The remainder of this paper is organized as follows. Section 2 presents the system model, and introduces some concepts and terminology used in the remainder of the paper. Section 3 presents the adaptive segment-based caching scheme in detail. Section 4 presents the transmission cost under the multicast batched patching with prefix caching and segment-based suffix caching. Our evaluations are shown in Section 5. Finally we present our conclusions and ongoing work in section 6.

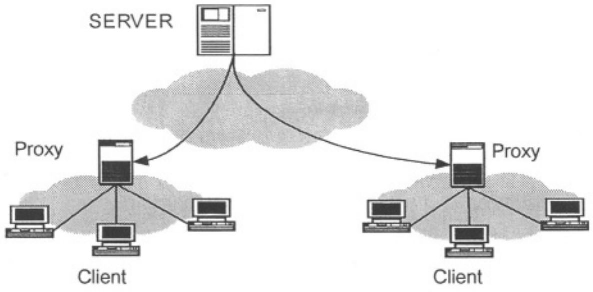


Fig. 1. Distributed architecture for streaming media delivery

2 System Architecture for Deliverying Streaming Media

In this section, we consider architecture of delivering streaming media object, which composed of origin server, proxy, and clients. A group of clients receive the multimedia data streamed across the Internet from a server via a single proxy (Fig.1). We assume reliable transmissions and only unicast capable over the server-proxy path, and the access network (proxy-client) is lossless and multicast enabled. We further assume that the clients are always request playback from the beginning of the media object. Moreover we impose the proxy to play the role of a client for server. That is, all the streaming media object data streamed out of the server are requested by the proxy and are thereby forwarded through it. A proxy streams the prefix directly to its clients if a prefix of the requested media object is present locally, and contacts the server for the remainder (suffix) of this object, and then relays this incoming data to the client. In order to shield the client-perceived startup latency, the proxy immediately sends each client the first segment data by unicast channel once a request arrives, then at the boundary of each batching interval, the clients will join the corresponding complete stream and the patching stream that started by the proxy via multicast channel.

We next introduce the notations used in this paper, as presented in Table 1. We consider a server with a repository of M Constant-Bit-Rate (CBR) media objects. We assume the aggregate access rate to all the objects are known a priori. That can be obtained by monitoring the system in a real scenario. Let f_m be the access probability of video m, $\sum_{m=1}^{M} f_m = 1$. Let λ be the aggregate access rate to the object repository and λ_m be access rate of object m, $\lambda_m = \lambda f_m$, $1 \le m \le M$.

Let media object $m \in M$ is characterized by its playback rate r_m, duration L_m, and average access rate λ_m. As like in the work [7], we also introduce a caching grain of size u to be the smallest unit of cache allocation and all allocations are in multiples of this unit. The caching grain can be one bit or one minute of data. Steaming media object m has playback rate r_m, length L_m seconds, and size n_m units, $L_m r_m = u n_m$, and the batching interval b_m is an integer multiple of u. Finally, the proxy can store S units where $S = \sum_{m=1}^{M} n_m$. We use α and β respectively representing the costs associated with transmission one bit of streaming media data on the server-proxy path and on the proxy-client path. The total transmission cost per unit time is $\sum_{m=1}^{M} Cost_m$, where $Cost_m$ represent the transmission cost per unit time for object m. Our objective is to develop appropriate transmission and caching schemes that minimize the average transmission cost per unit time over all the streaming media object in the system.

Table 1. Parameters used in this paper

Parameter	Definition
M	Number of the media objects
L_m	Length of media object m (sec.)
$L_{p,m}$	The prefix length of media object m (sec)
r_m	Playback rate of media object m (bps)
S	The cache size of proxy
R_m	Backbone transmission rate
u	Caching grain
λ_m	Average request arrival rate for object m (req/min)
λ	Aggregate request arrival rate (req/min)
W_m	Threshold window size for object m
μ_m	Total patch data for object m
b_m	The batching interval (sec)
α	Transmission cost on server-proxy link (per bit)
β	Transmission cost on proxy-client link (per bit)
$Cost_m$	Transmission cost per unit time for media object m

We differ from all the existing works in that we develop a new mechanism to cache the upcoming segment data at proxy from the ongoing entire stream along the unicast connection of the server and the proxy, which depends on the current batching interval has requests or not. These cached segment data can serve the requests arrived in the same threshold window. By this method, the more popular the object is, the more buffering space it will take. This makes the limited storage be used efficiently. Moreover, by varying the size of the threshold window according to popularity of the streaming media object, the minimum consumption of the backbone bandwidth can achieve.

3 Adaptive Segment-Based Caching Scheme

This section describes our proposed adaptive segment-based caching method. A streaming media object is always segmented into two portions: the prefix segment [6] and the suffix segment. The suffix section is further divided into multiple segments by uniform size b_m. For simplicity, we assume the prefix length of object m is equal to the uniform size b_m, that is $L_{p,m} = b_m$.

3.1 Scheme Description

The basic idea of the segment-based adaptive caching is that whenever a full stream is started, the proxies that receive the data from this stream allocate a buffer size of b_m units to cache the upcoming data. And at the end of each batching interval in the threshold window, the proxy checks to see whether or not it has requests. If it has requests, the proxy adds b_m units to the buffer and cache the ongoing stream continuously. Otherwise it stops caching. Since we do not cache the data segments at the end of each batch of zero requests, the proxy will need to start one extra channel to get them if it has requests in the subsequence batching intervals.

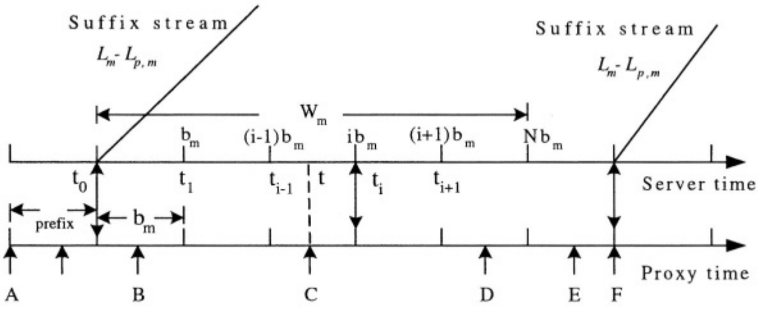


Fig. 2. Timing Diagram for adaptive batched patch caching

Our scenario is illustrated in Fig.2, The proxy divides the time axis into intervals $[t_{i-1}, t_i)$ of duration b_m, Assume a request arrives at the proxy at time $t\in[t_{i-1}, t_i)$, and the most recent suffix stream was started at time t_0. If t_i is such that $t_i < t_0 + W_m$, the proxy need to transmit patch data of duration $t_i - t_0 = ib_m$ to the client at time t_i, also the proxy joins the suffix stream at time t_i, and multicasts it to the clients. However, if $t_i > t_0 + W_m$, a new suffix stream will be started at time t_i. Since the first segment data $[0, b_m)$ of the streaming media object has been stored at the proxy at time t_1 after the suffix stream is started, whether the subsequent segment data need to be cached depends on the batching intervals that have no-zero requests before time t_i. This is better explained with the following example.

Suppose there are six batching intervals in the threshold window, that is $W_m=6b_m$. The suffix stream was started at time t_0. Also at time t_0 the proxy began to buffer the first segment data from this steam. Suppose $[t_0, t_1]$ had requests, then at time t_1 the proxy adds b_m units to the buffer and caches the ongoing stream continuously. This makes the proxy can serve the requests arrive at time intervals $[t_0, t_1]$ and $[t_1, t_2]$, and does not request server for any patch bytes. Suppose the next three intervals do not have any requests and the fifth interval $[t_4, t_5]$ has requests. At time t_5, the buffer of the proxy has contained two segment data $[0, b_m]$ and $[b_m, 2b_m]$. The requests arrive in the fifth time interval require the patch to be $5b_m$-long, and so the proxy adds $4b_m$ units to the buffer and then fetches the missing patch data $[2b_m,3b_m)$, $[3b_m,4b_m)$ and

$[4b_m, 5b_m)$ from the server by the extra channel while storing the segment data $[5b_m, 6b_m)$ from the ongoing stream in the suffix stream. Whether the sixth interval $[t_5, t_6)$, has requests or not, the proxy will not store the next segment data from the suffix stream at time t_6.

Our scheme has a remarkable feature. That is, the more probability of the request arrival each batching interval has (That means the request arrival rate is very high), the little patch data need to be transmitted afresh become. When all of the batched intervals in threshold window have no-zero requests, the pre-caching mechanism can assure that the demanded patching data just obtaining from suffix stream can satisfy all the request arrived in the threshold window. On the other hand, if only the last batching interval has requests, the proxy will need to afresh obtain the patch data up to $(N-1)b_m$ by the extra channel.

3.2 Formulation of Adaptive Caching Problem

For simplicity of exposition, we ignore network propagation latency. In order to make a qualitative analysis to our scheme, we first make abstraction as follows:

- Assume the access rate of media object m is modeled by Poisson process with parameter λ_m such that $p = e^{-\lambda_m b_m}$ is the probability to have an empty batch (zero request) of duration b_m.
- The proxy stores the patch data in the buffer at least for a period of $W_m = Nb_m$, so that it is available for the all requests arrived in the same threshold window.
- Suppose x_i denotes the requests that arrive in the ith batching interval $[t_{i-1}, t_i]$. Let $x_1, x_2, \cdots, x_N$ be a sequence of independent random variables with common probability distribution.
- Whether the proxy need to fetch the patch data by extra channel at time t_i depends on the values of x_1, x_2, $\cdots, x_{i-1}$ and x_i. In particular if $x_1 = x_2 = \cdots = x_i = 0$ or $x_1 = x_2 = \cdots = x_i \neq 0$, the proxy does not request server for any patch bytes.
- Assume the proxy request server for the patches size is η in the threshold window. It is obviously η will likely take one among the values $0, 1b_m, 2b_m, \cdots, (N-1)b_m$, we might as well let p_i denotes the probability of $\eta = ib_m$, *that is* $p(\eta = ib_m) = p_i$, $i = 0, 1, 2, \cdots, N-1$.

With the above assumption, we can achieve the mean value of η, namely

$$E\eta = b_m \sum_{i=1}^{N-1} i \cdot p_i \tag{1}$$

Now we need to determine p_i. Obviously,

$$\begin{aligned}
p_0 &= p(\eta = 0) \\
&= p(x_1 = \cdots = x_N \neq 0) + p(x_1 = \cdots = x_{N-1} \neq 0, x_N = 0) \\
&\quad + p(x_1 = \cdots = x_{N-2} \neq 0, x_{N-1} = x_N = 0) + \cdots \\
&\quad + p(x_1 \neq 0, x_2 = \cdots = x_N = 0) + p(x_1 = \cdots = x_N = 0) \\
&= (1-p)^N + (1-p)^{N-1} p^1 + (1-p)^{N-2} p^2 + \cdots \\
&\quad + (1-p)^2 p^{N-2} + (1-p) p^{N-1} \\
&= C^0_{N-0}(1-p)^N + C^0_{N-1}(1-p)^{N-1}p^1 + C^0_{N-2}(1-p)^{N-2}p^2 \\
&\quad + \cdots + C^0_1(1-p)^1 p^{N-1} + C^0_0 p^N \\
&= \sum_{j=0}^{N} C^0_{N-j}(1-p)^{N-j} p^j
\end{aligned}$$

$$\begin{aligned}
p_1 &= p(\eta = 1b_m) \\
&= C^1_{N-1}(1-p)^{N-1} p + C^1_{N-2}(1-p)^{N-2} p^2 + \cdots \\
&\quad + C^1_2(1-p)^2 p^{N-2} + C^1_1(1-p) p^{N-1} \\
&= \sum_{j=1}^{N-1} C^1_{N-j}(1-p)^{N-1-j+1} p^{1+j-1}
\end{aligned}$$

$$\begin{aligned}
p_2 &= p(\eta = 2b_m) \\
&= C^2_{N-1}(1-p)^{N-2} p^2 + C^2_{N-2}(1-p)^{N-3} p^3 + \cdots \\
&\quad + C^2_3(1-p^m_k)^2 p^{N-2} + C^2_2(1-p) p^{N-1} \\
&= \sum_{j=1}^{N-2} C^2_{N-j}\ (1-p)^{N-2-j+1} p^{2+j-1}
\end{aligned}$$

$$\begin{aligned}
p_3 &= p(\eta = 3b_m) \\
&= C^3_{N-1}(1-p)^{N-3} p^3 + C^3_{N-2}(1-p)^{N-4} p^4 + \cdots \\
&\quad + C^4_3(1-p)^2 p^{N-2} + C^3_3(1-p) p^{N-1} \\
&= \sum_{j=1}^{N-3} C^3_{N-j}(1-p)^{N-3-j+1} p^{3+j-1}
\end{aligned}$$

Thus the expression of p_i can be written as:

$$p_i = \sum_{j=1}^{N-i} C^i_{N-j}(1-p)^{N-i-j+1} p^{i+j-1}\ ,\ i = 1,2,\cdots,N-1. \tag{2}$$

Substituting equation (2) into equation (1), by computing all the different possibilities of batching interval along the patching window, we get the average number of patched data segments, μ_m at proxy. It is given by

$$\mu_m = E\eta = b_m \sum_{i=1}^{N-1}\sum_{j=1}^{N-i} i \cdot C^i_{N-j}(1-p)^{N-i-j+1} p^{i+j-1} \tag{3}$$

The aggregate transmission rate on the backbone link, R_m, includes the transmission of patches (μ_m) and the suffix stream of duration L_m–$L_{p,m}$ from the server. It is given by

$$R_m = \frac{(L_m - L_{p,m} + \mu_m)r_m}{I_m} \tag{4}$$

Where I_m represents the interval duration between two adjacent suffix streams:

$$I_m = (N+1)b_m + 1/\lambda_m \tag{5}$$

4 Proxy-Assisted Caching and Transmission Scheme

In this section, we develop a proxy-assisted caching and transmission scheme that use proxy prefix caching and adaptive segment-based suffix caching as integral part for bandwidth-efficient delivery in the Internet-like environments, where the server-proxy network connections provide unicast-only service, and the proxy-client path can offer multicast communication. Since such one-to-many intra-domain multicast is much simpler to deploy and manage, and the proxy is located close to the clients, we assume the transmission cost required to send one bit to multiple clients using multicast is still equal to that using unicast. In order to shield the client-perceived startup latency, the prefix segment caching must take precedence over all other segments for an object.

Suppose the first request for object *m* arrive at time t_0, as shown in Fig.3. The proxy immediately begins transmitting the prefix $L_{p,m}$ to the client. If the suffix of object m does not cache at proxy, the first segment of suffix stream is scheduled to reach proxy at time t_1. For any request arriving in batching interval $[t_0, t_1]$, the proxy just forwards the single incoming suffix stream (of length $L_m - L_{p,m}$) via multicast to the new client, and no other data segments transmission is needed from the server. At time t_2, we exploit the adaptive segment-based suffix caching scheme to realize the suffix segment dynamically caching at proxy. For the request arriving in batching interval $[t_1, t_2]$, the proxy firstly stream the prefix segment to each client via unicast, and then batch them together at time t_2 to join the patching stream (of length b_m) started at time t_2. In the other batching interval within the suffix caching threshold window, the transmission schedule process is similar to that in interval $[t_1, t_2]$. We name the above transmission scheme multicast batched patching with segment-based adaptive caching (MBP).

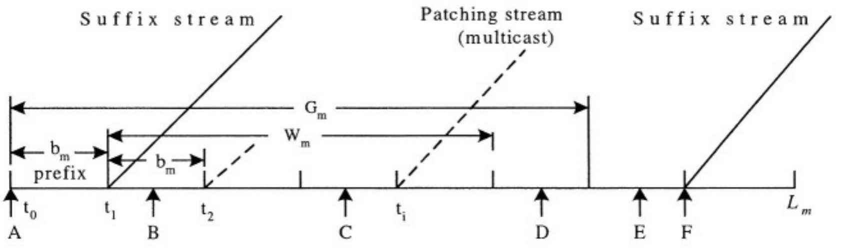


Fig. 3. Batched patching multicast with prefix caching and suffix caching

Let W_m be the suffix caching threshold window to the object m, $W_n=ib_m$, $i=0,1,2,\ldots.N$. The data segment outside this threshold window will not cache in the proxy. Also, Let G_m be the threshold to regulate the frequency at which the complete suffix stream is transmitted. Suppose the cached suffix segment will be evicted from the proxy after the time go over $\max(G_m,W_m+b_m)$. If a request arrives in time interval $[t_0, t_0+G_m]$, data stream delivery to this client can be classified into the following cases depending on the relationship of W_m+b_m and G_m:

- *Case 1:* $t_0 \leq W_m+b_m < G_m$. This is show in Fig.3. If a client arrives in time interval $[t_0, t_0+b_m+W_m]$, for example client B receives the segment $[t_0, t_1]$ (prefix segment) from a separate channel via unicast and receives segment $[t_1, t_2]$ from patching stream via multicast from the proxy and receives segment $[t_2, L_m]$ via the ongoing suffix stream forwarded by proxy. If a client arrives in time interval $[t_0+W_m+b_m, t_0+G_m]$, for example, beside receives segment $[t_0,t_1)$, $[t_1,t_1+W_m)$ and $[t_0+G_m, L_m]$, client D need receive data segment $[t_0+b_m+W_m, t_0+G_m)$ from server via proxy relay. Assume a Possion arrival, the transmission cost in this case C_1 is

$$C_1 = \frac{r_m}{G_m + 1/\lambda_m}[(L_m - L_{p,m} + \mu_m + \tfrac{\lambda_m(G_m-W_m-b_m)^2}{2})\alpha + (L_m - L_{p,m} + \lambda_m G_m b_m + \tfrac{\lambda_m W_m^2}{2} + \tfrac{\lambda_m(G_m-W_m-b_m)^2}{2})\beta] \tag{6}$$

- *Case 2:* $G_m \leq W_m+b_m \leq L_m$. In such a situation, the client receive data segment that is same as client B in case *1*. The transmission cost in case *2* is

$$C_2 = \frac{r_m}{G_m + 1/\lambda_m}[(L_m - L_{p,m} + \mu_m)\alpha + (L_m - L_{p,m} + \lambda_m G_m b_m + \tfrac{\lambda_m W_m^2}{2})\beta] \tag{7}$$

For a given prefix $L_{p,m}=b_m$, the average minimum transmission cost is

$$Cost_m = \min(C_1, C_2) \tag{8}$$

5 Performance Evaluation

In this section, we examine the normalized transmission cost under our previously proposed scheme (MBP), and then compare with Mpath in [7]. We consider a repository of 100 CBR media objects with access probabilities characterized by a Zipf distribution with parameter θ=0.271[11], that is, the request probability for object m is $f_m=\frac{1}{m^{1-\theta}}/\sum_{m=1}^{M}\frac{1}{m^{1-\theta}}$, $m=1,2,\cdots,M$. Therefore, the average request arrival rate from media object m is given by $\lambda_m=\lambda f_m$. For simplicity, we assume all objects are 120 minutes long, and have the same playback rate. We normalize the transmission cost by both the media object playback rate and the value of α. That is, the normalized transmission cost is $\sum_{m=1}^{M} Cost_m/\alpha b_m$. Let $\gamma=\beta/\alpha$, and we assume $\gamma\in[0,1]$. Obviously, this is a reasonable assumption because bandwidth resource on the server-proxy path is usually more valuable than that on the proxy-client path. Note that $\gamma=0$ corresponds to $\beta=0$, while $\gamma=1$ corresponds to $\beta=\alpha$.

In segment-based adaptive suffix caching scheme, the size of the proxy cache allocated to an object is proportional to its popularity, under the constraint that the allocated buffer is no larger than the size of the object. By vary the threshold window W_m, we let the most popular objects caching completely at proxy, the intermediate popular ones caching partially and the unpopular ones should be streamed directly from server via proxy relay. By this mean, we may achieve much more the bandwidth saving under the limited storage at proxy, and consequently reduce the aggregate transmission cost. In order to get the exact information of popularity distribution among the media objects online, we use the distribution status of the request arrival in each batching interval to measure the popularity of the object.

Fig.4 plots the normalized transmission cost for MPatch and MBP when $\gamma=0$ and the aggregate arrival rate λ equals to 100 or 200 requests per minute, respectively. It depicts the normalized transmission cost usage on server-proxy path under the MBP scheme decreases as size of the proxy cache augments. The two schemes under a small amount of proxy buffer size (25%) result in substantial cost savings. MPatch incurs lower costs than MBP, because the former does not take into account the transmission cost of the cached portion, while in the MBP scheme, we only ignore the transmission cost of the first segment (prefix), and consider the dynamic behavior of caching for the suffix segment data.

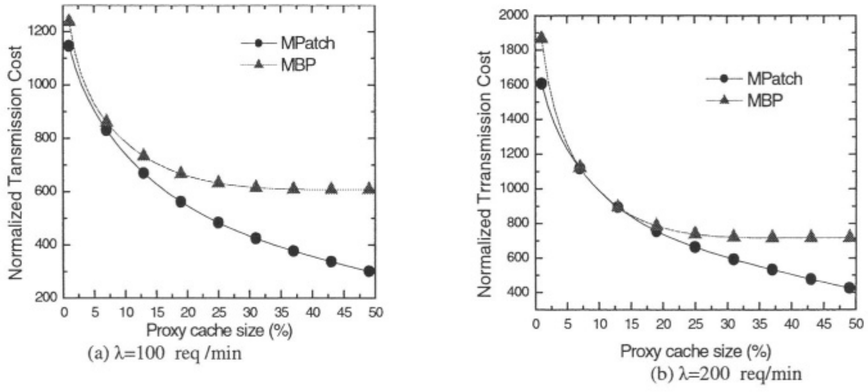


Fig. 4. Normalized transmission cost v.s. proxy cache size, $\gamma=0, b_m=1$[min].

Fig.5 depicts the normalized transmission cost as a function of proxy cache size, when $\gamma=0.5$, for aggregate arrival rates $\lambda=50$ req/min, $\lambda=100$ req/min, $\lambda=200$ req/min, and $\lambda=300$ req/min. We see that, in conditions of high aggregate request arrival rate, MBP incurs much more transmission cost savings than MPatch, while the latter outperforms the former in conditions of low request arrival rate. The reason behind this is that the MBP batches the request together to join the patching stream (multicast), thus it can save much more bandwidth on proxy-client path.

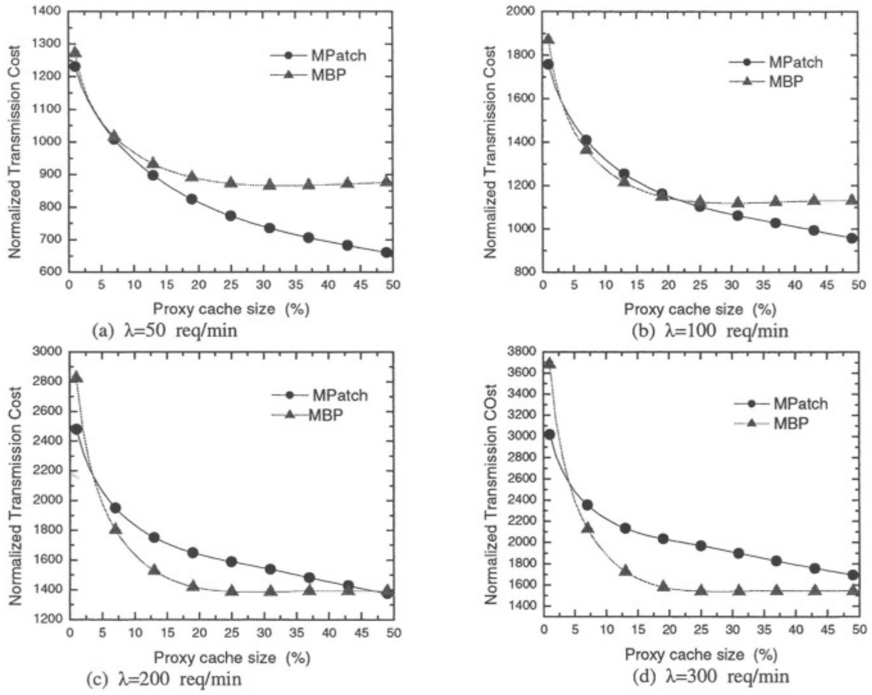


Fig. 5. Normalized transmission cost v.s. proxy cache size, γ=0.5,b_m=1[min].

In Fig.6 (a) ~ (d), We can observe that when the system goes into stable state, that is all the cached segments are not be replaced again, the MBP achieves much more savings of transmission cost than the MPatch whatever the request arrival rate is high or low.

6 Conclusions and Ongoing Work

Segment-based proxy caching is one of the attractive solutions to improve the performance of multimedia service systems on the Internet. In this paper, we study multimedia object streaming in the settings where the server-proxy paths only offer unicast connection and the proxy-client paths are multicast capable. We proposed a segment-based adaptive caching scheme, which combine the transmission scheme of multicast-based batched patching aimed at reducing the aggregate transmission cost. By storing the frequently accessed objects at proxy and sending the requesting clients the first segment data of the requested object via unicast by proxy, immediately, client perceived latency, server load and backbone network traffic can be reduced significantly. Our performance evaluation shows that: (1) our scheme is sufficient to realize substantial savings in transmission cost under a relatively small proxy cache (10% of the object repository); (2) Integrating the segment-based adaptive caching with the multicast-based batched patching can reduce the transmission cost on both the backbone link and local network; (3) compare with the MPatch scheme, the performance of our scheme is better, especially in the conditions of high request arrival rates.

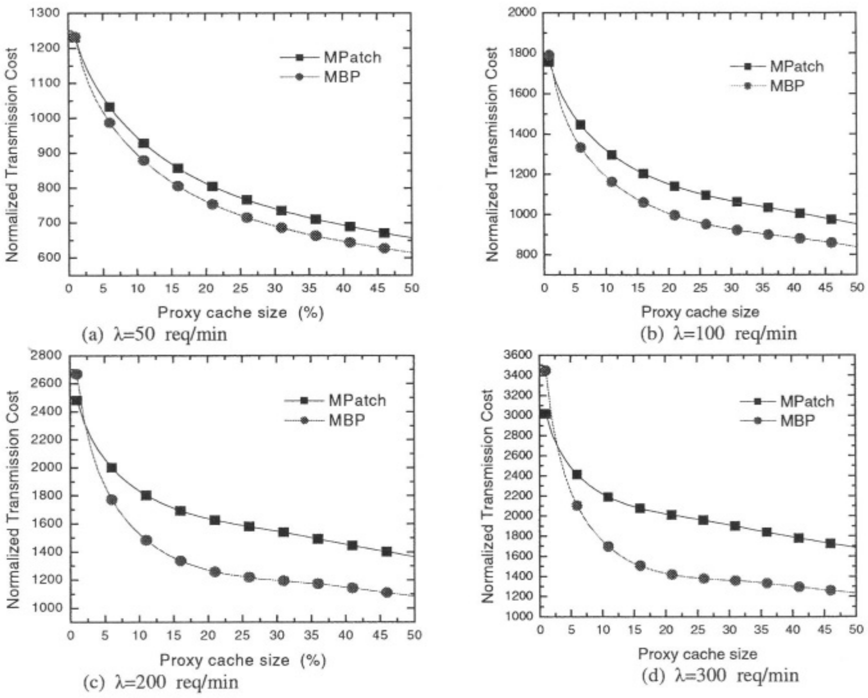


Fig. 6. Normalized transmission cost v.s. proxy cache size in steady situation, $\gamma = 0.5$, b_m=1[min].

As ongoing work, we are building a prototype implementation of MBP scheme for streaming media distribution, in order to evaluate the performance of our proposal under the realistic network. We are also pursuing the optimizing threshold window to maximize the aggregate transmission cost over all the media objects.

References

1. A. Dan, D. Sitaram,and P. Shahabuddin, “Schduling policies for an on-demand video server with batching”, in Proc. ACM Multimedia, San Francisco, California, pp.15–23, October 1994.
2. K. A. Hua,Y. Cai, and S. Sheu, “Patching: A multicast technique for true video-on-demand services”, in Proc. ACM Multimedia, Britstol UK, pp. 191–200, Sept. 1998.
3. P. P. White, and J. Crowcroft, “Optimized batch patching with classes of service”, ACM Communications Review, vol. 30, no.4, October. 2000.

4. O.Verscheure,C.Verkatramani, P. Froassard, and L. Amini, “Joint server scheduling and proxy caching for video delivery”, Computer Communications, vol.25, no. 4, pp.413–423, Mar. 2002.
5. C. Venkatramani, O. Verscheure, P. Frossard, and K. W. Lee, “Optimal proxy management for multimedia streaming in content distribution networks”, in Proc. of ACM *NOSSDAV*2002, Miami Beach, FL, USA, pp.147–154, May 2002.
6. S. Sen, J. Rexford, and D. Towsley, “Proxy prefix caching for multimedia steams”, in Proc. of IEEE Inforcom’99, New York, USA, no.3, pp.1310–1319, 1999.
7. B. Wang, S. Sen, M. Adler, and D. Towsley. “Optimal proxy cache allocation for efficient streaming media distribution,” IEEE Trans. on Multimedia, vol.6, no.2, April 2004
8. Y. Cai, K. A. Hua, and K. Vu, “Optimizing patching performance”, in Proc. ACM/SPIE Multimedia Computing and Networking, pp. 203–215 , January 1999.
9. Eager, D., M.Vernon, and J. Zahorjan, “Minimizing Bandwidth Requirements for On-Demand Data Delivery”, IEEE Trans. on Knowledge and Data Engineering, Vol. 13, No. 5, 2001
10. S. Ramesh, I. Rhee, and K. Guo, “Multicast with cache(mcache): An adaptive zero-delay video-on-demand service,” in Proc. IEEE *INFOCOM,* April 2001.
11. C. Aggarwal, J. Wolf, and P. Yu, “On optional batching policies for video-on-demand storage servers,” in Proc. IEEE International Conference on Multimedia Computing and Systems, June 1996.

A Client-Based Web Prefetching Management System Based on Detection Theory

Kelvin Lau and Yiu-Kai Ng

Computer Science Department
Brigham Young University
Provo, Utah 84602, U.S.A.
{ng, klau}@cs.byu.edu

Abstract. The two most commonly used Web browsers, Netscape Navigator and Internet Explorer, lack Web prefetching capability, which can reduce time latency for retrieving Web documents over the Internet. In this paper, we propose a client-based Web prefetching management system, called CPMS, which is based on the caching schema of Netscape Navigator. CPMS includes a prefetching engine, which predicts and prefetches Web documents and their linked documents that have high hit rate, minimizes traffic increase, and reduces time latency in accessing Web documents. We adopt detection theory to determine the threshold value that dictates which Web documents and their links should be prefetched. Besides the prefetching engine, CPMS also provides a graphical interface that allows the user to activate, delete, and modify any computed prefetching schedule.

1 Introduction

Prefetching Web documents in advance can minimize response time for retrieving documents over the Internet whose network load and access time have been increasing dramatically. In this paper, we present a client-based prefetching approach that caches Web documents and their linked documents that are likely to be accessed by the user at the client's site. Our targeted users come from diverse backgrounds who have "regular" Web accessing patterns, such as professional stock investors who access stock Web sites on a daily basis at various periods of time. We have considered different existing caching approaches: (i) Web-server caching [11], (ii) proxy caching [1,3,18], and (iii) client caching [15]. Unlike Web-server caches and proxy caches, client caches are available in client-based Web browsers, such as Netscape Navigator and Internet Explorer. Client caching is set up for a particular client that uses the Web browsers to retrieve Web documents. We adopt the client-based prefetching approach for caching Web documents, since (i) a Web user often has access only to the caches stored on his machine, and (ii) we use the caching schema of Netscape Navigator whose source code is available to the public.

We realize that Web prefetching, which is not available in existing client-based Web browsers, is a promising strategy in minimizing time latency for retrieving Web documents over the Internet [2,4,6,9,12,13,14,15]. Unlike [7], which prefetches objects according to their popularity and update rates, we propose a client-based Web prefetching management system (CPMS) that prefetches documents and their linked documents with high hit rate (or likely to be accessed) and minimizes traffic increase. We chose to develop CPMS on top of Netscape Navigator, a client-based Web browser and one of the commonly used browsers. We adopt the ranking approach in [10] to implement the prefetching engine in CPMS, which ranks each previously accessed document referenced in a Netscape history file. CPMS computes the ranking value of each previously accessed Web document to determine whether the (latest version of the) document should be prefetched, and we analyze the performance of our prefetching approach according to its prediction accuracy. In measuring the prediction accuracy, we collected a training set and a test set of proxy traces of different users and applied a technique in detection theory to determine which Web documents to prefetch. We use the training set data to determine the criteria on prefetching Web documents and the test set data to verify the prediction accuracy. Besides the prefetching engine, CPMS provides a graphical user interface that allows the user to view, activate, delete, or update computed prefetching schedules. CPMS is different from the prefetching system of [15], which is the only other client-based prefetching system that we are aware of, since CPMS does not require extra tools to create a user's access history file and runs faster and predicts more accurately.

We present our results as follows. In Section 2, we describe the system design of CPMS. In Section 3, we discuss how to apply detection theory in our prefetching approach. In Section 4, we analyze our experimental results. In Section 5, we introduce the user interface of CPMS. In Section 6, we give a conclusion.

2 System Design

CPMS consists of (i) a prefetching engine and (ii) a graphical user interface for manipulating prefetching schedules. In Section 2.1, we present the design of the prefetching engine, and in Section 2.2, we discuss the ranking value of a Web document *V* that determines whether *V* should be prefetched. In Section 5, we introduce the graphical user interface.

2.1 A Prefetching Engine

Web prefetching [9] loads Web documents that the user is likely to access in the near future into a Web cache. If the user requests a document that is already in the cache, the time latency perceived by the user improves compared with retrieving it through the Internet. When prefetching is employed *between a Web server and a proxy server,* the proxy server uses the past access history of Web documents requested by its clients to determine which Web documents to be prefetched from the Web server; this past access history is extracted from either the server's or proxy's data [4,9,12]. This approach eliminates the interactions between proxy's clients and the Web server, which usually have a higher network cost than the interactions between proxy's clients and the proxy server. The approach benefits clients who have a low-bandwidth or high-price network connection. Prefetching *between a Web server and a client,* on the other hand, is a client-based approach that prefetches Web documents by using either the server's or client's past access history [2,13,14,15]. In this approach, the client prefetches Web documents directly from the server into the local client machine. As a result, time latency for retrieving a cached document is reduced because there is no network latency for retrieving locally cached documents. Prefetching *between a proxy server and a client* is similar to prefetching *between a Web server and a client;* however, in the former the past access history in the proxy is used [6]. A proxy server handles Web document requests from its own users, whereas a Web server deals with all the requests from various users. A proxy server predicts which Web documents that reside at different servers may be accessed by its clients, whereas a Web server predicts which Web documents that reside at its own location may be accessed by its clients. We adopt a client-based prefetching approach *between a Web server and a client* as explained below.

We have examined different prefetching approaches as described in [2,4,6, 9,12,13,14,17]. With the exception of [17], which prefetches long-term steady-state objects according to their access rates and update frequencies, all of these prefetching approaches consider the server's or proxy's access history in predicting future Web accesses. They generally do not apply to client-based prefetching systems because a client machine normally cannot access the history file at a Web server's or proxy server's site unless the server grants access privileges to the client machine. We adopt the client-based approach on prefetching because our prefetching system targets only individual users, as Netscape Navigator does, and retrieving documents from a Web server using the client's past history bypasses a proxy server, which is impossible in the other two prefetching approaches.

Since our prefetching system is client-based and is built on top of Netscape Navigator, our prefetching engine considers the information stored in the client's access history file, called netscape.hst, in the Navigator (see Section 2.2 for details). Our prefetching engine uses this information to determine the ranking value of each previously accessed Web document V. The *ranking value* of V is the *estimated probability* that the user will access (the latest version of) V again, and ranking values determine whether the corresponding documents referenced in the Netscape history file should be prefetched or not. Using each document in a set S referenced in the Netscape history file whose ranking value exceeds the threshold value (determined by the training set as discussed in Section 3), the prefetching engine generates a *prefetching schedule* of S. This schedule is a 24-hour time table for prefetching Web documents in S, which are likely to be accessed by the user within a 24-hour period. The prefetching engine can be activated to prefetch Web documents in S according to the prefetching schedule. When the cache is full, the prefetching engine uses ranking values to determine which Web cache entries should be removed.

2.2 Ranking Values and Prefetching Schedules

To compute a prefetching schedule, our prefetching engine first extracts from a user's Netscape *history file,* i.e., netscape.hst, data of all previously accessed Web documents within the last 14 days. A Netscape history file is a binary file that contains a *hash table,* and the history file is located under the user profile directory in the "Users" directory of Netscape Navigator. Data of each previously accessed document are stored in the hash table as a history entry. Each history entry contains five different data fields, which include (i) the *last* visit date and time, (ii) the *first* visit date and time, (iii) the *visit count,* (iv) the *title,* and (v) *flags,* of a previously accessed Web document. (Flags are used to indicate some internal states, e.g., whether any image is embedded.) The cost function of our prefetching engine uses only (iii) and (i) to compute the ranking value of *each* previously accessed document.

We adopt and modify the equation in [10], as shown in Equation 1, to calculate the ranking value of a previously accessed Web document *V.* This equation computes the probability of reaccessing *V,* denoted P_v. P_v is calculated by using (i) all the visit counts of distinct Web documents in a Netscape history file in where the access history of *V* forms an entry, (ii) the *visit count* of *V,* denoted by i, and (iii) the *time interval* since the last access of *V,* denoted by t. Using their experimental data, [10] shows that i and t are almost independent of each other and approximates P_v as $P_v(i) \times P_v(t)$, where $P_v(i)$ ($P_v(t)$, respectively) is the probability of reaccessing *V* based on i (t, respectively).

$P_v(i)$ is calculated as $\frac{\|D_{i+1}\|}{\|D_i\|}$, where i is the *visit count* of *V* as shown in a Netscape history file, and $\|D_i\|$ ($\|D_{i+1}\|$, respectively) is the number of distinct documents in the Netscape history file that have been accessed at least i ($i+1$, respectively) times. Assume that 100 documents have the visit count of at least three, and 60 out of these 100 documents have the visit count of at least four, then the probability that a document with the visit count of three has been accessed again is $\frac{60}{100} = 0.6$. Unlike $P_v(i)$, $P_v(t)$ is computed by using the *cumulative distribution function* $D_v(t)$, which is the probability that *V* has been reaccessed in the elapsed time t, i.e., from the last access date and time until time t, and is calculated by using Equation 3. In their experiments, [10] first collected data on user accesses from a proxy server during a time period of 100 days, and then analyzed the statistical data of Web documents that had been accessed *again* during a second 100-day time period. Let $P_v(t)$ denote the probability that *V* has been reaccessed after t (e.g., up till the 100th day). Using $d_v(\tau)$, the *probability density function,* which is the probability of accessing *V* at time t, $D_v(t)$ is defined as $\{d_v(\tau) : 0 \leq \tau \leq t\}$ and $P_v(t) = \{d_v(\tau) : \tau > t\}$. Hence $P_v(t) = 1 - D_v(t)$, and P_v can be defined as

$$P_v = P_v(i) \times P_v(t) = \frac{\|D_{i+1}\|}{\|D_i\|} \times (1 - D_v(t)) \tag{1}$$

Our cost function, *CF,* which generalizes P_v, is defined as

$$CF = \frac{\|D_{i+1}\|}{\|D_i\|} \times (1 - D(t)) \tag{2}$$

where $D(t)$ is the cumulative distribution function.

[10] approximate $D_v(t)$ in Equation 1 using data collected from a proxy server, which is a set of previously accessed Web documents. $D_v(t)$ is the *integration* of $d_v(t)$, which, as discussed earlier, computes the probability of a Web

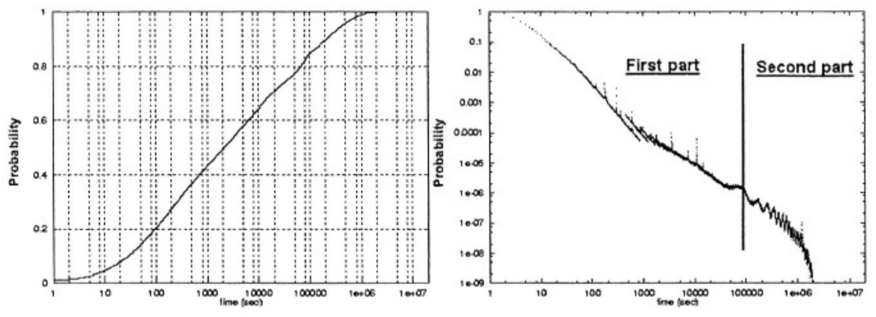


Fig. 1. Cumulative distribution function $D_v(t)$ (left) and probability density function $d_v(t)$ (right)

document V that has been reaccessed by time t (since the last access date and time of V) during the time period of 100 days. To approximate $D_v(t)$, [10] first obtain two curves, $D_v(t)$ and $d_v(t)$, as shown in Figure 1, and use their proxy traces to determine what the curves should look like. Using the shape of the curve of $d_v(t)$, $d_v(t)$ is approximated by using two parts that are determined by the slope of the curve as shown in Figure 1 such that the *first* part is approximated by the function k/t, and the *second* part is approximated by $e^{-\alpha t}$, where k and α are constants. The approximated $d_v(t)$ is defined as $w_1 \cdot k/t + w_2 \cdot e^{-\alpha t}$, where w_1 and w_2 are weight constants for adjusting the approximation of the curve and are determined by experiments. By integrating $d_v(t)$, $D_v(t)$ is calculated as

$$
\begin{aligned}
D_v(t) &= \int_{-\infty}^{t} d_v(t)\,dt = \int_{-\infty}^{0} d_v(t)\,dt + \int_{0}^{t} d_v(t)\,dt = \int_{0}^{t} d_v(t)\,dt && (3)\\
&= \int_{0}^{t} (w_1 \cdot k/t + w_2 \cdot e^{-\alpha t})\,dt = w_1 \cdot k \int_{0}^{t} \frac{1}{t}\,dt + w_2 \int_{0}^{t} e^{-\alpha t}\,dt && (4)\\
&= .35 \log(t+1) + .45(1 - e^{-\frac{t}{2E6}})
\end{aligned}
$$

where $\int_{-\infty}^{0} d_v(t)\,dt$ denotes the probability that the Web document V will be accessed again before the first time V is accessed, where 0 denotes the first time V is accessed. Since this probability is impossible, $\int_{-\infty}^{0} d_v(t)\,dt = 0$. There is also a discrepancy in deriving Equation 4 from Equation 3. Integrating $\frac{1}{t}$ should yield $\log(t)$; however, due to the boundary problem as log 0 is undefined, the result is approximated by incrementing t by one.

According to the ranking value P_v of each document V referenced in a Netscape history file, which is (re-)computed each day, and a *threshold value* $\mathcal{L}$, our prefetching engine determines whether V should be prefetched at a particular hour on a particular day. Using some of the methods developed in detection theory [5,16] and training set data, which is a proxy trace of different users' Web accesses, we determine the value of $\mathcal{L}$. (The fundamental concepts of detection theory will be discussed in Section 3.) According to the experimental data (as shown in Section 4.3), we set $\mathcal{L} = 0.245$. If $P_v > \mathcal{L}$, then the URL of V will be included in the prefetching set for next day. Note that prefetching Web documents can be changed from daily to hourly. (See Section 5 for details.)

After the prefetching set of Web documents (identified by their corresponding URLs) is determined, our prefetching engine will compute the hour to prefetch each document *V* in the prefetching set. The hour to prefetch *V* should be set so that the most recently updated *V* is available to the user instead of requiring the user to reload *V* manually. Because Netscape Navigator maintains only two data values referring to the access time of *V,* i.e., the *first* visit time and the *last* visit time of *V,* the appropriate time to prefetch *V* should be earlier than the two visit times. We have decided that one hour *before* (except the midnight hour, which is a special case) the smaller hour value of the two previous visit times of *V* (denoted in hour values between 0 and 23) is an ideal time for prefetching the latest version of *V* for the user. We compute the beginning time to prefetch *V* as the scheduled hour *SH* of *V,* which is defined as

$$SH = \begin{cases} 0 & \text{if } \min(h_1, h_2) = 0 \\ \min(h_1, h_2) - 1 & \text{otherwise} \end{cases} \tag{5}$$

where h_1 is the hour of the *first visit time* and h_2 is the hour of the *last visit time.* An exceptional case is presented in Equation 5; i.e., if the minimum hour value of either h_1 or h_2 is 0, which denotes midnight, then the scheduled prefetching hour *SH* of *V* should be 0 instead of -1, since we only deal with one-day schedules and hence will not prefetch Web documents at 11:00 p.m. on the previous day.

Our prefetching engine generates a prefetching schedule after the scheduled prefetching hour of each document in the prefetching set has been computed. Using the generated schedule, the prefetching engine prefetches Web documents by downloading them onto the user's machine at the prefetching scheduled time, which is during the first five minutes of the scheduled hour. If a Web document *V* has been cached, the prefetching engine still prefetches *V* to ensure that *V* is the latest version. If *V* is unavailable at the time of prefetching, e.g., the Web server where *V* resides is down, the prefetching engine will retry three more times with a 15-minute time lapse between each attempt within the scheduled hour. The prefetching engine will not prefetch *V* after all three retries fail. Unlike the prefetching system proposed by [8], which avoids interference between prefetch and demand requests at the server by all means possible, prefetching according to the hourly schedule has insignificant impacts on the Web server's load.

3 Applying Detection Theory

In many situations, we are required to consider among several choices to decide our next action in processing a particular task. For instance, in a radar detection system, whenever a radar signal is returned, it is analyzed to determine whether an actual object is present. Often, we might be involved in making a decision according to several possible choices. *Detection theory* [16], which provides a way for solving the decision problem, could guide us to make the appropriate decision. We adopt detection theory, along with training set data, to determine the *acceptable threshold value* $\mathcal{L}$ (as discussed in Section 2.2) and apply detection theory to analyze the performance of our cost function.

Table 1. The hypothesis in decision theory

Decision\True State	H_0 is true	H_1 is true
Accept H_0	Correct Decision	Type II Error (accept a false H_0)
Reject H_0	Type I Error (reject a true H_0)	Correct Decision

Hypothesis testing is the fundamental component of detection theory. We can think of a hypothesis as a statement or a claim that is either true or false. In detection theory, the *null* hypothesis, H_0, denotes the "current thinking" on a particular subject, whereas the *alternative hypothesis,* H_1, denotes the "alternative choice," and H_0 and H_1 are mutually exclusive. Table 1 shows each possible outcome for making a decision according to H_0 and H_1. Using the radar detection problem as an example, H_0 can be the hypothesis "There is no object present," whereas H_1 can be "There is an object present."

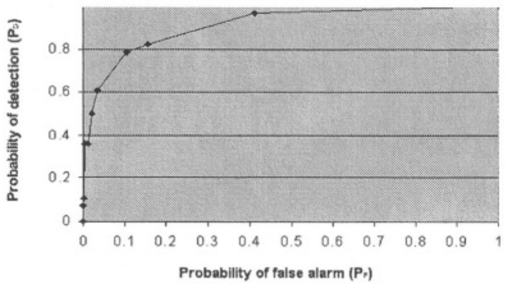


Fig. 2. An ROC curve, where P_D is the hit rate

The curve of *Receiver Operating Characteristics,* called ROC curve, is a graphical display for analyzing the performance of hypothesis testings in detection theory. The original ROC curve was developed during World War II to "calibrate" radar operators, and it is a graph which displays the probability of *detection* (P_D), i.e., *hit rate,* versus the probability of *false alarm* (P_F) (see Figure 2 for a sample ROC curve). In the hypothesis testing terminology, (i) a *detection* (a decision to accept an object that is indeed present or a decision to reject an object that is in fact absent) is a *correct* decision, (ii) a *false alarm* (a decision to accept an object that is not present) is the *type I error,* and (iii) a *miss* (a decision to reject an object that is present) is the *type II error.* An ROC curve, however, does not show the probability of *miss* (P_M), since P_M is already implicitly represented by the probability of detection (P_D) as $P_D + P_M = 1$, $P_D = 1 - P_M$. We create an ROC curve using the access history of different users on Web documents in the training set to determine the threshold value $\mathcal{L}$.

We establish the acceptable value $\mathcal{L}$ by (1) obtaining the proxy trace, [1] i.e., the training set data, (2) calculating the ranking values of Web documents, (3) determining the potential threshold values, and (4) computing the prefetching values and plotting the ROC curves from where $\mathcal{L}$ is obtained.

[1] The proxy trace, which is set up for experimental purposes only and is used mainly for determining the acceptable value $\mathcal{L}$, is not part of our prefetching system.

```
lunar.cs.byu.edu - - [24/Feb/2000:17:20:56 +0700] "GET http://www.yahoo.com/
  HTTP/1.0" 200 11997
lunar.cs.byu.edu - - [24/Feb/2000:17:20:56 +0700] "GET http://a32.g.a.yimg.com/7/
  32/31/000/us.ying.com/a/ya/yahoo-pager/messenger/messengermail.gif
  HTTP/1.0" 200 1897
lunar.cs.byu.edu - - [24/Feb/2000:17:20:56 +0700] "GET http://a1.g.a.yimg.com/7/
  1/31/000/us.ying.com /i/main4s3.gif HTTP/1.0" 200 4172
. . .
```

Fig. 3. A portion of the proxy-server access-log file collected in our experiment

Request command line format, "**Action URL HTTP-version**", where

- *Action*: The method type indicates how the content of the user input are sent to the query program resided at the Web server. The "GET" method type lists the attributes specified in the user input at the end of the URL address while the "POST" method type does not.
- *URL*: The URL of the requested Web document.
- *HTTP-version*: The version number of the HTTP. The current version is 1.0, and it is denoted as HTTP/1.0.

Fig. 4. The format of the request-command

3.1 Obtaining the Proxy Trace

We obtained the proxy trace of Web access requests that were made by different users during a 2-week period, which forms our training set data. In order to perform our experiments, we setup a proxy server on a C200 HP workstation running under HP-UX 10.20 operating system. Participating users remotely connected to our proxy server during the process of accessing Web documents on the Internet. At the end of the experimental period, we obtained the *proxy server access log file* from the proxy server. The file contains all the Web document requests made by each user who has connected to our proxy server for accessing Web documents on the Internet. A portion of the proxy-server access-log file collected in our experiments is shown in Figure 3, in which each line contains

remote-host – – '['*date*']' "*request-command*" *status bytes*

where *remote-host* is the host name of the remote machine, i.e., the user's machine, which initiates the request, *date* is the date and time of the request, *request-command* is the request command line submitted by the remote machine (with format of the request command line shown in Figure 4), *status* is the request status code returned by the Web server to the remote machine (see Table 2 for some of the status codes), and *bytes* is the size of the requested document.

Users submitted their Web access requests through their own machines to the proxy server during the experimental period, and our prefetching system CPMS separated entries in the access log file according to the remote host names and obtained an individual log file for each user. Having obtained each individual user log file, CPMS shranked each individual log file into a smaller log file that contains the log of each individual user's Web access requests within a 2-week consecutive time period. A 2-week time period is chosen because it is sufficient to show the Web access pattern of each individual user based on our observations

Table 2. List of request status codes

Code#	Description
200	The request was fulfilled, i.e., requested document has been sent to the user.
...	...
403	The request was forbidden by the server.
404	The server could not find the requested document.
...	...
500	The server encountered an unexpected internal error which prevented it from fulfilling the request.
501	The server does not implement the function specified in the request.
...	...

during the experimental period. We randomly chose a 2-week consecutive time period for each user to represent the access history of the user as long as the user accessed the Web on the fifteenth day, the day after the 2-week consecutive time period. We did this because the Web accesses on the fifteenth day determine which Web documents referenced in the proxy trace of the preceding 2-week time period have actually been reaccessed by the same user again on the fifteenth day. This information enables us to find out if the Web documents prefetched by our prefetching engine on the fifteenth day were correctly (or incorrectly) prefetched, i.e., was (or was not) actually reaccessed by the user on the fifteenth day. The 2-week log file was then converted into a log file of different format, called *converted access file,* which matches the Netscape history file that has a different format than the access log file created by our proxy server. Our prefetching engine uses the Netscape history file instead of the proxy access log file to calculate the ranking value (P_v) of each previously accessed document *V* referenced in the history file. Table 3 shows a portion of the converted access file of a user. Each line in the file consists of (i) the URL of each requested document *V,* (ii) the visit count of *V,* and (iii) the first visit date and time of *V* and (iv) the last visit date and time of *V* in terms of the number of seconds since January 1, 1970.

Table 3. A portion of an individual user's converted access file

http://my.yahoo.com/	27	951849382	952980156
http://my.yahoo.com/?myHome	32	951849383	952980161
http://us.yimg.com/i/my/top7.gif	24	951849384	952977418
http://us.yimg.com/i/my/persite3.gif	25	951849384	952977418
http://us.yimg.com/i/my/detach.gif	27	951849384	952977418
...	...	...	...

3.2 Calculating the Ranking Values

We ran our *values-calculating program* to obtain the ranking value of each document in the training set that has previously been accessed. The values-calculating program simulates the portion of our prefetching engine that calculates the ranking values of all the previously accessed Web documents referenced in the Netscape history file. The program first applies our cost function *CF* (as defined in Equation 2) on the training set to calculate the ranking value of each Web document in the training set. The inputs of the values-calculating program include (i) a converted access file that is created by using the corresponding 2-week proxy server access log file, (ii) an access log file for the fifteenth day, (iii) the starting date of the 2-week time period, and (iv) the number of days during the simulation time period, i.e., 14. [2]

3.3 Determining the Potential Threshold Values

The valid range of potential threshold values is between 0 and 1, and we chose 201 values within the range using an incremental value of 0.005, i.e., 0, 0.005, …, 0.995, 1, as the potential threshold values. These incremental values were chosen because this granularity should be sufficient for the measurement of $\mathcal{L}$.

3.4 Computing the Prefetching Values and Plotting the ROC Curves

After obtaining the ranking values of all the Web documents extracted from the 2-week converted access log file for each user, the values-calculating program uses (i) the obtained ranking values, (ii) one of the potential threshold values $\mathcal{L}$, and (iii) all the Web documents extracted from the fifteenth-day access log file to compute the prefetching values of $\mathcal{L}$. Note that each one of the 201 potential threshold values is associated with a set of prefetching values. The prefetching values of each potential threshold value $\mathcal{L}$ include (i) the potential number of Web documents referenced in the proxy trace that are correctly prefetched, denoted $N_{potentially\,correct}$, (ii) the potential number of Web documents referenced in the proxy trace that are incorrectly prefetched, denoted $N_{potentially\,incorrect}$, (iii) the actual number of correctly prefetched Web documents determined by $\mathcal{L}$, denoted $N_{correctly\,prefetched}$, and (iv) the actual number of incorrectly prefetched Web documents determined by $\mathcal{L}$, denoted $N_{incorrectly\,prefetched}$.

To illustrate how to compute the prefetching values of a particular threshold value $\mathcal{L}$ based on the ranking values of all the previously accessed Web documents, let's assume that a 2-week converted access log contains 1000 entries, i.e., previously accessed documents, and the fifteenth day access log, i.e., the day after the 2-week period, contains 100 entries. Further assume that 60 (previously accessed documents) out of the 100 entries in the fifteenth day access log also appear in the 2-week converted access log. Thus $N_{potentially\,correct} = 60$. Suppose, using the 2-week converted access log and a threshold value $\mathcal{L} = 0.5$, there are 300 entries (previously accessed documents) in the 2-week converted access log with a ranking value greater than 0.5, and 20 of the 300 entries appear in the

[2] Items (iii) and (iv) can be derived from item (i). Our values-calculating program requires items (iii) and (iv), since the time period of item (i) can be other than two weeks, and thus items (iii) and (iv) indicate the actual consecutive days that have been chosen within the time period of item (i).

fifteenth day access log, i.e., $N_{correctly\, prefetched} = 20$. Thus $N_{potentailly\ incorrect} = 1000 - 60 = 940$ and $N_{incorrectly\ prefetched} = 300 - 20 = 280$.

After obtaining all the prefetching values for each of the 201 potential threshold values, we calculated the P_D (i.e., hit rate) and P_F (i.e., false alarm) values of all the 201 potential threshold values for each user in five different groups and plotted the ROC curve to find an acceptable threshold value. Since P_D (P_F, respectively) of a potential threshold value denotes the probability of the correct decision (false alarm, respectively), which corresponds to the percentage of the Web documents referenced in the proxy trace that are prefetched correctly (incorrectly, respectively), $P_D \approx \frac{N_{correctly\, prefetched}}{N_{potentially\ correct}}$ ($P_F \approx \frac{N_{incorrectly\, prefetched}}{N_{potentially\, incorrect}}$, respectively). We plotted an ROC curve for each user so that each one of the 201 potential threshold values calculated earlier are represented as a dot (P_F, P_D) on the ROC curve, which shows how accurate our cost function can be used for prefetching Web documents for each user. To select an acceptable threshold value, we plotted one more ROC curve which is created by compiling the five groups of P_D and P_F user values, and the curve was used to determine an acceptable threshold value. The most ideal threshold value for the ROC curve is $P_D = 1$ and $P_F = 0$. However, P_D and P_F affect each other; i.e., when P_D is high, P_F is high, and when P_D is low, P_F is low. Thus we have to find the dot on the sixth ROC curve such that it achieves a good balance between P_D and P_F. Based on the training set data, the threshold values 0.24, 0.245, and 0.25, which have similar P_D ($\cong 0.86$) and P_F ($\cong 0.10$) values, are the most ideal threshold values. We choose 0.245 (with $P_D = 0.87$ and $P_F = 0.11$), the medium value.

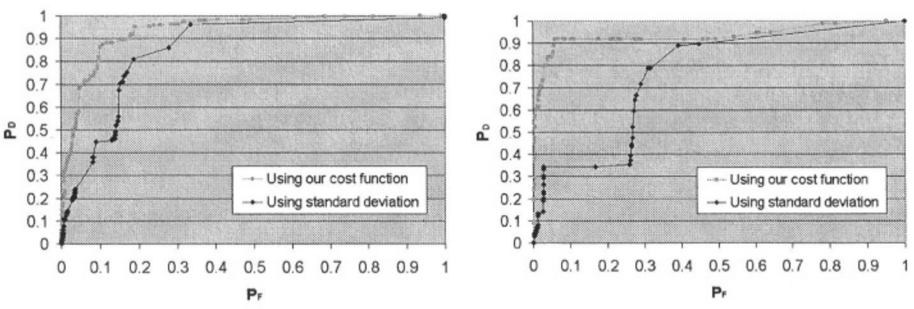
To further verify our results that the chosen threshold value is optimal, we gathered another proxy trace (i.e., the test set data) of Web accesses of different users (see Section 4.3 for details). We realize that the higher curvature of a ROC curve, the better the ROC curve is because the ROC curve can achieve the same P_D value while keeping the P_F value lower. The high P_D value, i.e., high hit rate, and low P_F value, i.e., low false alarm rate, translate into low traffic increase, since only a few number of prefetched documents are not actually accessed by the user. Since our prefetching engine achieves high hit rate and low false alarm rate, the waste of bandwidth in prefetching unused Web documents is relatively small compared with the number of Web documents that are actually retrieved upon requests without prefetching.

4 Experimental Results and Analysis

In this section, we present the results of the experiments performed on our prefetching engine and analyze the experimental results. In Section 4.1, we compare our cost function with the one used by [15]. In Section 4.2 we show the ROC curves computed by using the two cost functions against the test set data. In Section 4.3, we analyze the experimental results.

4.1 Cost Functions

We compare our cost function with the one in [15], which use *standard deviation* to calculate the deviation from the mean (average) [5] of sample data and is



(a) ROC curves based on training data (b) ROC curves based on test data

Fig. 5. ROC curves constructed by using the training and test set data

defined as

$$S = \sqrt{\frac{\sum_{i=1}^{n}(V_i - V_{average})^2}{n}} \tag{6}$$

where n is the number of days in the sampling time period (e.g., 14), i is the day number (e.g., $i = 1$ denotes the 1st day), V_i is the number of occurrences of Web document V referenced in the proxy access log file on the ith day, and $V_{average}$ is the mean of the number of occurrences of V on each day during the sampling time period.

Figure 5(a) shows the ROC curve of all the five groups of P_D and P_F user values compiled together using our cost function versus the ROC curve of the same five groups of P_D and P_F user values compiled together using standard deviation. As shown in the figure, the ROC curve of our cost function has a better curvature than the ROC curve of standard deviation. In general, our cost function achieves a 52%[3] reduction in prefetching incorrect Web documents compared with the standard deviation cost function.

We further compare the efficiency of the two cost functions by letting m denote the number of distinct Web documents referenced in a proxy access log and n denote the number of days in a sampling time period. The formula of our cost function (as shown in Equation 2) does not involve m nor n, and thus the time complexity of calculating the ranking value of a Web document using our cost function is $\mathcal{O}(1)$. Since our cost function is used m times, one for each distinct document referenced in the converted access log file, the time complexity of calculating all the ranking values of Web documents using our cost function is $m \times \mathcal{O}(1) = \mathcal{O}(m)$. For the standard deviation cost function, however, the time complexity for computing the ranking value of a Web document is $\mathcal{O}(n)$, since the function involves a summation, i.e., $\sum_{i=1}^{n}$ (as shown in Equation 6), of some calculations. Therefore, the time complexity of calculating all the ranking values using the standard deviation cost function is $m \times \mathcal{O}(n) = \mathcal{O}(m \times n)$.

[3] 52% is obtained by averaging the reduction ratios of all P_D values. In the graphical representation, it is the area residing between the ROC curves of our cost function and standard deviation.

4.2 Test Set Data

We obtained another proxy trace of Web accesses requested by five different groups of users, which forms our test set data. The test set data are collected in order to verify the accuracy of the experimental results compiled by using the training set data. The proxy trace recorded the Web accesses in one-month period, and the users involved are different from the ones who participated in the experiments to obtain the training set data. In order to ensure that the experimental results of the training set and test set are comparable, we used exactly the same machines and software. Figure 5(b) shows the ROC curve on all five groups of P_D and P_F user values computed by using the test set data and our cost function versus the ROC curve on the same five groups of P_D and P_F user values and standard deviation.

4.3 Analysis of the Results

The experimental results of our *training set data* show that using our cost function, we can achieve a high P_D value (= 0.87) while maintaining a low P_F value (= 0.11) on prefetching Web documents with threshold value $\mathcal{L} = 0.245$. The P_D (P_F, respectively) value suggests that using our prefetching engine, 87% (11%, respectively) of the Web documents referenced in a Netscape history file would be *correctly* (*incorrectly,* respectively) prefetched. Using our *test set data,* we further confirm that if the threshold value $\mathcal{L}$ is 0.245, our cost function can achieve a high P_D value (= 0.86) while maintaining a low P_F value (= 0.06) on prefetching Web documents.

5 A User Interface for Prefetching Schedules

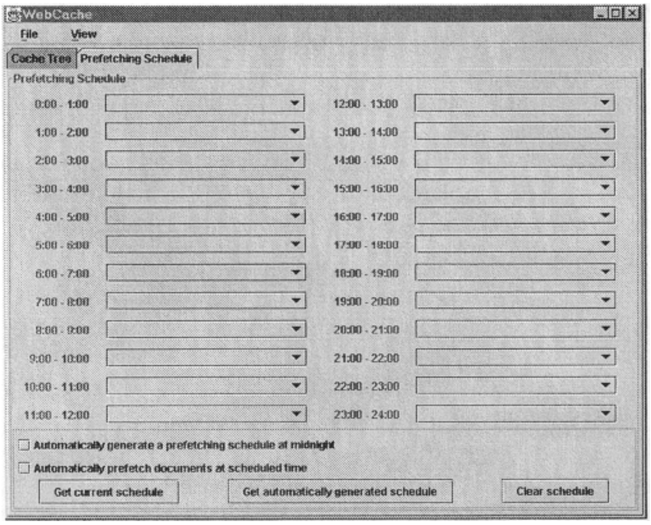


Fig. 6. The Prefetching Schedule pane

We have designed a graphical user interface for viewing and modifying any computed prefetching schedule. These features allow the user to (i) find out which Web documents will be prefetched and (ii) update a computed prefetching schedule as needed. As shown in Figure 6, the *Prefetching Schedule* pane consists of a time table, two check boxes, and three buttons. The time table is designed for displaying the URLs of documents to be prefetched at each scheduled hour, and it has a slot for each of the 24 hours. For each time slot, there is a *down arrow* for displaying all the documents (identified by their URLs) to be prefetched at the hour.

An enabled *Automatically generate a prefetching schedule at midnight* checkbox, as the name suggested, automatically (i) generates a prefetching schedule at midnight without actually prefetching the involved documents and (ii) displays the prefetching schedule in the time table at midnight. We choose midnight to be the first hour of the prefetching schedule template because midnight is the start of a day. Unlike the function of the upper checkbox, an enabled *Automatically prefetch documents at scheduled time* checkbox allows the actual prefetching to take place, i.e., actually prefetching the documents listed in the time table at the scheduled prefetching hour. The user can enable/disable any of these two functions by checking/unchecking the corresponding checkbox.

The three buttons in the *Prefetching Schedule* pane allow the user to change the display and/or the content of the prefetching schedule as shown in the time table. When the *Get current schedule* button is pressed, the prefetching engine generates and displays the most recent prefetching schedule starting at the current hour. The most updated prefetching schedule is generated by using the Netscape history file with history entries created up till the current hour when the user presses the button. If a prefetching schedule already exists, the newly computed prefetching schedule starting at the current hour will replace the existing prefetching schedule (of the current day). The content of the updated schedule will then be displayed in the time table from the current hour up till the 23:00 hour. The *Get automatically generated schedule* button, on the other hand, is used for retrieving the automatically generated schedule on the same day and displaying the schedule immediately in the time table. A click on this button causes the automatically generated prefetching schedule starting at midnight to be displayed in the time table that can be different from the schedule computed by the *Get current schedule* function. If the user presses the *Get current schedule* button before pressing the *Get automatically generated schedule* button, the automatically generated schedule at midnight will replace the current schedule generated by the *Get current schedule* function, and vice versa. Last but not least, the *Clear schedule* button is used for clearing the schedule in the time table so that the table becomes empty; i.e., it performs an "undone" operation. A click on the *clear schedule* button cleans up time tables generated by *Automatically generate a prefetching schedule at midnight, Get current schedule,* and *Get automatically generated schedule.*

6 Concluding Remarks

In this paper, we propose a client-based Web prefetching management system, called *CPMS,* which is developed according to the caching schema of Netscape Navigator and prefetches Web documents and their linked documents referenced in a history file of Netscape Navigator. We develop CPMS partially because Web prefetching is missing in Netscape Navigator, which is a commonly used Web browser. *CPMS* provides a 24-hour prefetching table and includes a prefetching engine that prefetches Web documents in advance to decrease time latency for accessing the documents, achieve high hit rate, and minimize traffic increase.

We have verified the performance of CPMS by using two proxy traces, a training set and a test set, of Web access requests made by different groups of users. We used (i) the training set data to identify a decision threshold and determine the prediction accuracy of our prefetching engine and (ii) the test set to verify the prediction accuracy of our prefetching engine. Experimental results show that CPMS correctly prefetches 87% of the training set documents and 86% of the test set documents.

References

1. M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox. Caching Proxies: Limitations and Potentials. In *Proc. of the 4th Intl. WWW Conf.,* 1995.
2. A. Bestavros and C. Cunha. A prefetching Protocol Using Client Speculation for the WWW. Technical Report 1995-011, Boston Univ., 1995.
3. P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proc. of the USENIX Symposium on Internet Technologies & Systems,* pages 193–206, 1997.
4. K. Chinen and S. Yamaguchi. An Interactive Prefetching Proxy Server for Improvement of WWW Latency. In *Proc. of the 7th Conf. of Internet Society,* 1997.
5. H. B. Christensen. *Introduction to Statistics.* Harcourt Brace Jovanovich, 1992.
6. Q. Jacobson and P. Cao. Potential and Limits of Web Prefetching Between Low-Bandwidth Clients and Proxies. In *Proc. of the Intl. Web Caching Workshop,* 1998.
7. Y. Jiang, M. Wu, and W. Shu. Web Prefetching: Costs, Benefits, and Performance. In *Proc. of the 7th Intl. Workshop on Web Caching & Distribution,* 2002.
8. R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin. NPS: A Non-Interfering Deployable Web Prefetching System. In *Proc. of the 4th USENIX Symposium on Internet Technologies & Systems,* 2003.
9. T.M. Kroeger and D.D. Long. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. of the UNENIX Symposium on Internet Technologies and Systems,* pages 319–328, December 1997.
10. P. Lorenzetti, L. Rizzo, and L. Vicisano. Replacement policies for a proxy cache. Technical Report LR-960731, Univ. di Pisa, July 1996.
11. E. P. Markatos. Main Memory Caching of Web Documents. In *Proc. of the 5th Intl. WWW Conf.,* May 1996.
12. E. P. Markatos and C. E. Chronaki. A Top-10 Approach to Prefetching the Web. In *Proc. of the Eighth Annual Conf. of Internet Society,* July 1998.
13. V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proc. of the ACM Computer Communication Review,* 1996.
14. T. Palpanas and A. Mendelzon. Web Prefetching Using Partial Match Prediction. In *Proc. of the 4th Intl. Web Caching Workshop,* 1999.
15. S. Peng, J. K. Flanagan, and F. Sorenson. Client-Based Web Prefetch Management. In *Proc. of the Eighth Intl. World Wide Web Conf.,* May 1999.
16. M. Srinath, P. Rajasekaran, and R. Viswanathan. *Introduction to Statistical Signal Processing with Applications.* Prentice-Hall, 1996.
17. A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. The Potential Costs and Benefits of Long-term Prefetching for Content Distribution. In *Proc. of the Intl. Workshop on Web Caching & Content Distribution,* 2001.
18. R. Wooster and M. Abrams. Proxy Caching that Estimates Page Load Delays. In *Proc. of the 6th Intl. WWW Conf.,* pages 250–252, 1997.

Structured Partially Caching Proxies for Mixed Media

Frank T. Johnsen, Carsten Griwodz, and Pål Halvorsen

Department of Informatics,
University of Oslo,
{frankjo,griff,paalh}@ifi.uio.no

Abstract. News on demand features user interaction, interdependent media and is used by different client types. These requirements are not yet accommodated by a single solution. We address this issue by utilizing a priori knowledge about news articles' structure encoded in a presentation plan. Using this, we investigate the gains that knowledge about structure can have for a caching proxy. Our simulations show that *structured partial caching* of media object segments achieves a higher byte hit ratio compared to a non-structured caching scheme when access patterns follow a Multi-Selection Zipf distribution.

1 Introduction

In recent years, we have seen an increase in services available on the Internet. More people peruse the Internet for information than ever before, and replace their modems with more capable ADSL lines. Service providers respond to this trend by providing even richer media for their customers. Some broadband providers now offer their customers true video on demand (VoD), a service one earlier could only dream of when using a modem. The next trend, which we anticipate will be even more popular than VoD, is news on demand (NoD). VoD requires a certain amount of bandwidth for a single stream, but is quite predictable in behavior. A user may perform simple VCR-like operations on a movie, such as playing, stopping or skipping. NoD, on the other hand, has interrelated files of different kind, including continuous and discrete media, and can offer more advanced interactivity to the clients. The clients themselves can have varying capabilities, and range from powerful PCs to PDAs and mobile phones. Also, the flexibility introduced by utilizing mixed media for NoD brings new requirements to proxies in distribution networks. We explore the concept of mixed media below when describing our NoD scenario.

A lot of research has been committed to VoD, and there exist several techniques which can be applied by the providers in order to make effective use of the resources in the network, such as caching proxies. Caching is a means to lower traffic at the origin server and save bandwidth between server and proxy, and it is our belief that efficient caching is a key to any content distribution scheme as it can lower the overall costs. In order to accommodate the special needs of NoD, we have devised our own variant of partial caching [1], which we call *structured partial caching*. Our scheme is specially tailored to make use of the structure inherent in mixed media, which probably will be utilized in the future for NoD. Our goal is to make a partial caching scheme that is specialized for mixed media content in a NoD scenario. To investigate possible gains we have written a simulator.[1] The results are promising, showing that using knowledge of structure can significantly increase byte hit ratio under certain conditions.

The rest of this paper is organized as follows: In section 2 we present the NoD scenario in detail. The differences between NoD and VoD are discussed. Further, important issues like structure and mixed media are defined. Some of the research done on VoD can be applied in a NoD scenario as well. We point out some of these important ideas in section 3 on related work. Our structured partial caching scheme is described in section 4, where we explain how knowledge of structure can be utilized in order to perform efficient caching of mixed media data in our NoD scenario. Details concerning the simulations are discussed in section 5. Our conclusion and summarized results are given in section 6 together with an outline of future work to be performed.

2 NoD Scenario

In recent years, much research has been performed in the area of VoD. Usage patterns have been investigated [2], streaming issues have been addressed [3] and servers optimized [4]. Also, distribution schemes have been devised and multimedia proxies [1,5,6] added to further enhance performance. However, there are other areas utilizing different media types that are similar to, but less explored than VoD. One such area is NoD. Important aspects of NoD is user interaction, a concept of structure through the use of mixed media content, and a vast array of different clients with different capabilities – ranging from powerful PCs to mobile phones – wanting to use the content. Since the characteristics of NoD differ significantly from those of VoD, old VoD distribution schemes cannot be used directly [7]. We need a system which heeds the needs of NoD.

VoD and NoD differ in a number of ways: VoD access follows the Zipf distribution, and popularity will often remain stable for a long period of time. Further, a video is one linear file, and the access pattern is totally independent of the other videos. NoD, on the other hand, is accessed as described by the Multi-Selection Zipf distribution [7]: Client requests tend to be for groups of articles rather than single items as in VoD. Rather than following a Zipf-distribution for all articles, the news items will be requested in groups, and these groups contain several articles. The popularity span is also much shorter, as an article is usually most popular during its first three days after publication [7]. When speaking of NoD, we assume that the structure of the media objects of news stories are described by some sort of meta data, in the following referred to as

[1] This work is part of the INSTANCE II project (147426/431) funded by the Norwegian Research Council.

a presentation plan. Such a plan or news item encompasses mixed media, for example text, video and audio objects.

We define two types of structure; *internal structure* and *external structure.* The presentation plan gives external structure to the media objects. This structure imposes certain restrictions on the objects, for example saying that two objects should always be retrieved together, after one another, or as an alternative to one another. There are other possibilities as well; a total of 13 were identified in [8]. Knowledge of external structure is important when servers and proxies perform rough synchronization between objects, as irregularities here will affect the user's perception [9] of the media. In addition to an external structure, media objects can also have an internal structure. This is the structure that is inherent in the media object's format, for example through the layering of video.

Mixed media has many similarities to hypermedia [8]; both have different media objects and presentation plans. However, the two differ on some key issues: Hypermedia is strongly focused on navigation issues, whereas mixed media focuses more on streaming and concurrent presentation of different media. Different objects have different spatial and temporal requirements, and objects with a temporal dimension (audio and video) can be streamed, while objects with only spatial requirements must be retrieved all at once. Further, we have some general knowledge about client interactivity and access patterns, as client requests tend to be for groups of articles [7] rather than single items as in VoD. Rather than following a Zipf-distribution for all articles, the news items will be requested in groups, and these groups contain several articles and are accessed following the Multi-Selection Zipf distribution.

In online newspapers, news presentations are usually constructed both from brand new material and from archives. We envision a scenario where news can be made using new media objects, whole or parts of old objects and presentations. This differs from HyNoDe [10], where one would not have reused the presentation plan, only a lot of raw material. In particular, HyNoDe would reuse only subsequences of audio or video clips. We assume a fair amount of content reuse in NoD, including connections between presentations. Using a presentation plan format that supports inclusion, like MHEG [11], might lead to the utilization of sub-presentations, while another presentation plan encoding like SMIL [12] would allow the use of files that are only logically related.

3 Related Work

Existing caching schemes such as prefix caching [6] for streaming media and the full object caching performed by many web proxies such as SQUID [13], though effective for their purposes, can be improved on to better suit a NoD scenario. This is due to the fact that rather than always requesting an entire object, or an object from start to finish, we may also have subset access. The most important issue here is that partial caching [1] should be employed, of which there are currently two main types: You can have partial caching in the time domain, an

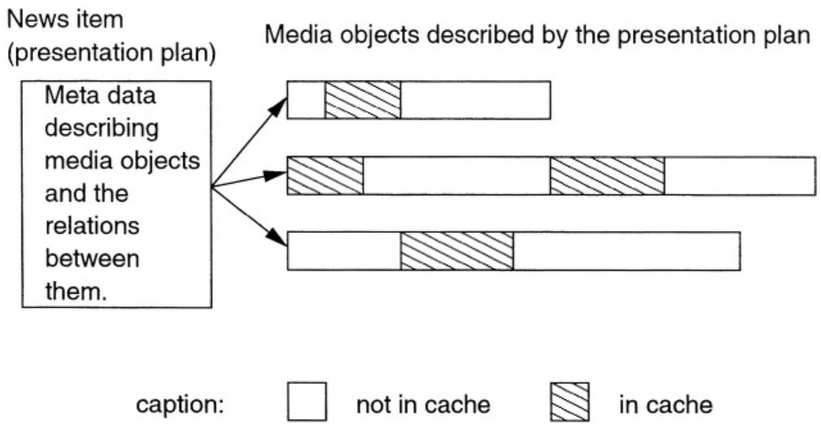

News item
(presentation plan)
Media objects described by the presentation plan
Meta data describing media objects and the relations between them.
caption:
not in cache
in cache

Fig. 1. An example showing a news item and its media objects.

example would be prefix caching [6], and in the quality domain; examples here are the layering of video [5] and the rescaling of video [1].

Our idea is different from existing work in that we are considering the needs of mixed media for NoD. To make even better caching decisions we believe that one should also look at the dependencies between media objects, i.e., the structure of a news story. This is what we call *structured partial caching.*

4 Structured Partial Caching

Figure 1 illustrates a news item and the three media objects it contains. The media objects and the presentation plan itself are all located at the origin server. When a user requests a certain plan, its client software will parse the plan and offer the options it enables to the user. Based on the user's interactions the news item may be downloaded and watched, either in part or in full. The figure shows a case where the plan uses only a few parts of existing media objects, so although the entire objects are present at the origin server, there is no need to cache more than is actually used. The figure illustrates how the proxy performs structured partial caching. Knowledge of internal structure is utilized here. However, the fact that the three media objects are part of the same presentation also imposes external structure on them, which implies that they will likely be retrieved together. This is important when performing rough synchronization between objects, and can be of use to the caching scheme. A caching scheme may guess that such a relation exists, but the synchronization mechanics must know.

We want to investigate if a proxy utilizing knowledge about structure will be more efficient than a proxy that does not, as the more information a proxy has about its cache elements, the more likely it is to make a good decision regarding cache replacement. There are several different ways to measure the efficiency of caching algorithms, but when speaking of efficiency here we mean the *byte hit ratio*. The byte hit ratio is the percentage of requested bytes that are served from the cache [14]. A high byte hit ratio means that there is an increase in saved bandwidth between server and proxy, an important issue when wanting to keep overall transmission costs low. However, making use of as much knowledge as possible about the objects eligible for caching will naturally be more CPU intensive than simpler existing caching schemes. The initial costs of deployment will therefore be higher, as we need more powerful machines for proxy servers. On the other hand, we save bandwidth.

When performing structured partial caching in the NoD scenario we have some extra knowledge about the media objects we are caching that can be used: The structure is known a priori, i.e., we have the presentation plan. Also, there may be a certain reuse of old elements, when old news items are linked and related to current events.

We propose that different media objects should be treated differently by the proxy, and that the meta data neccessary for this should be available in the presentation plans. More precisely, both the external and the internal structure should be described by the meta data. This enables the proxy to handle all content based on meta data, and does not render it useless in the case of a new media format becoming popular. Also, it saves processing power, since the proxy does not have to analyze the bulk of data passing through it, only the corresponding meta data.

If an object can be partially cached and the part that is stored at the proxy is usable in its own right, then the object has internal structure. An example here is layered video [5], where one can choose to cache only lower layers (at the expense of quality). Another example of internal structure is a progressive JPEG picture; here prefix caching of such a picture would render the prefix usable in its own right, rather than if one had the prefix of a picture without internal structure (gif, for example) where the prefix would be useless without the rest.

Our scheme differentiates between objects with and without internal structure in the following manner: Regular prefix caching is used if there is no internal structure. Structured partial caching is used in the cases where there is internal structure. For scalable, linear monomedia formats, this is the same as quality partial caching. In the case where one presentation plan uses the whole object, and clients request this, then the whole object is cached. But, if at some later point another presentation plan reuses only part of previous said object, then it has gained internal structure which the proxy has become aware of, and structured partial caching will be employed henceforth. This is our novel concept, and different from all other caching schemes. The proxy can "learn" about structure based on this new usage of the old material.

Mixed media consists of several different media objects that are structured by meta data in a presentation plan, which describes the relationships between the objects. For our work on structured partial caching, we do not choose a specific encoding format. Rather, we use our own, generic scheme in the simulations so

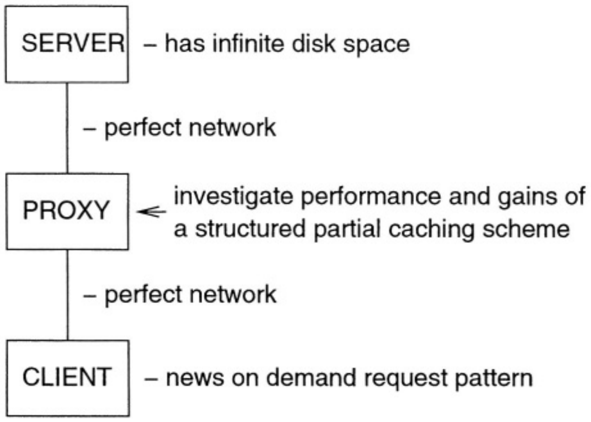

SERVER
– has infinite disk space
– perfect network
PROXY
investigate performance and gains of a structured partial caching scheme
– perfect network
CLIENT
– news on demand request pattern

Fig. 2. The simulator setup.

that the results will be applicable to whichever encoding format one may choose. Because of this, we limit the extent of external structure to a level that can be expressed by all the previously mentioned formats, i.e., we omit the concept of sub-presentations.

5 Simulations

To test our assumption that structured partial caching is indeed beneficial, we have simulated a proxy utilizing this technique. The simulator was implemented in C++ using the lagged fibonacci random generator from the boost library from "http://www.boost.org". The main parts are as Figure 2 illustrates. Client requests follow the Multi-Selection Zipf distribution [7]. The network is assumed to exhibit perfect behaviour, that is having zero delay, no packet loss and unlimited bandwidth. We want to investigate proxy internals only as we seek to determine if structured partial caching is beneficial or not. Thus, we do not need the added complexity of network simulations. The proxy gathers statistics about media object use and performs caching. We let the various modules communicate with each other through function calls, and collect statistics regarding cache replacement and algorithm bookkeeping only.

The news items, which are situated at the server, have been generated using very simple meta data. The data contains information about the number of media objects in the presentation and their attributes, such as size and internal structure. The news items may contain text, audio, video and pictures, and these objects may or may not have internal structure.

The algorithms tested were *LRU over entire objects* and *LRU over parts of objects*. The first of these was implemented as a reference only to see how well a scheme without special measures compares to our enhanced scheme. Caching

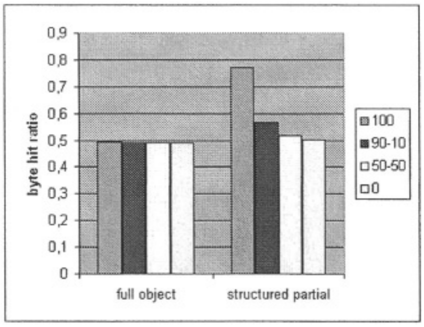


Fig. 3. Large cache, differing quality requests.

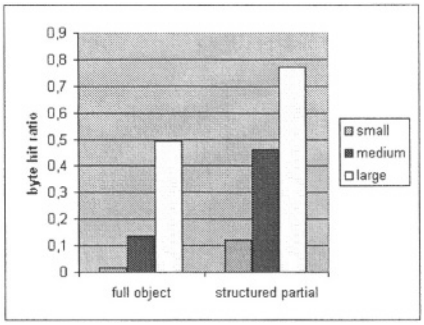


Fig. 4. Varying cache size, low quality requests only.

over parts of objects is partial caching in the quality domain using knowledge of structure.

For all algorithms, we ran tests both with and without an admission policy. The policy used in those cases was that presentation plans should be no older than three days for their objects to be eligible for caching. This was done to incorporate the knowledge that an article usually is most popular in the first three days after its release [7], so that if an older article is referenced it should not displace some of the newer ones in the cache. Old objects that are used in part or in full in new plans will be eligible for caching. Since all or part of an old object can become popular in this way, it is important to perform admission based on the presentation plan's age rather than object age.

We ran simulations using a total of 100000 presentation plans and 270000 media objects, for an average of 2.7 objects per plan. Simulations were performed with three different cache sizes. The small cache has room for an average of 200 whole objects, the medium cache 2000 and the large cache has room for 20000 objects. We distinguish clients that retrieve low quality from those that retrieve high quality objects. By low quality we mean a request for 10% of the media object's data, since progressive quality compression requires the client to retrieve 5–10% to recognize most of the features [15]. Figure 3 shows simulations for the largest cache size and clients exhibiting different quality requests. Clients browsing content have different capabilities. To reflect on this we ran several request combinations, namely such that all of the requests were for low quality content, 90% of the requests were low quality content, 50% of the requests were for low quality content, and all full quality. This is indicated in the figure by the numbers 100, 90–10, 50–50, and 0, respectively.

If only low quality is requested, we see significant gains as cache size increases. This is illustrated in Figure 4. We see that the scheme performs best when all clients request the lowest quality possible from the cache. This is due to the fact that the segments stored in the cache will be quite small, thus enabling many

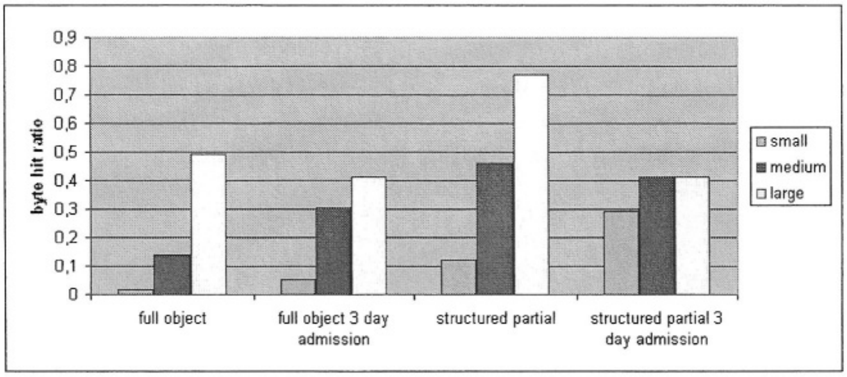


Fig. 5. Varying cache size, low quality requests and use of no vs 3 day admission policy.

of them to be stored compared to a non-partial caching scheme. A mixture of clients requiring various quality levels causes the scheme to perform worse, as is the case for our 90–10 and 50–50 low/high quality requests. Few high quality requests have a big impact on byte hit ratio. When all requests are for high quality, i.e., entire objects, the performance exhibited is slightly better than a non-structured scheme. This arises from the reuse of subsets of objects (we have a reuse probability of 10% in our simulations), where the structured scheme will save only the parts in use, as we saw in figure 1.

The different algorithms have varying CPU demands, and naturally an increase in complexity requires more processing power. The structured partial caching requires 5 times the processing power of whole object LRU when the requests are for low quality only, i.e., many segments in cache leading to much bookkeeping. When requests are for high quality only the processing power required diminishes to 3.5 times that of whole object LRU. We also tried a variant where we pinned all objects belonging to the same presentation plan as the object that was being requested. That variant requires 16 times the processing power for low quality requests and 5 times the processing power for high quality requests when compared to full object LRU. In our simulations there were no additional gains from performing this pinning, as results were identical to those of the structured partial caching which did not perform pinning.

Figure 5 illustrates the difference in byte hit ratio exhibited by the algorithms with no admission policy and our three day admission policy respectively. An interesting aspect to note is that for the smallest cache size the admission policy enhances performance by increasing the byte hit ratio. For the bigger caches, though, it can significantly lower performance by not allowing the cache to store older news as well.

Our results show that for our algorithm of choice, LRU, knowledge of structure has an impact on byte hit ratio. The scheme performs best when there are many requests for parts of objects, i.e., not full object requests. If we assume that the news service will be available for a large variety of clients such as PCs, PDAs and mobile phones, then there is obviously a need for different quality levels for the different clients, and structured partial caching will be particularly useful. For our mixed media NoD scenario the structured partial caching scheme will at all times exhibit a byte hit ratio equal to or better than the scheme not considering structure.

6 Conclusion

In this paper, we have investigated the usefulness of structured partial caching of mixed media content in a NoD scenario. We conclude that the proxies should utilize knowledge about structure provided that there is a mixture of client types. Structure expresses both relationships between media objects (external structure) as well as structure within objects (internal structure). Knowledge of structure can be extracted from the presentation plan, i.e., the meta data which describes each news item. We have investigated the benefits of utilizing knowledge of structure, and our results show that it will enhance a proxy's treatment of mixed media data.

What we have now is an extension of quality partial caching [1], in the future we want to look at extending this to incorporate a means of temporal partial caching [1] as well. Additionally, we want to investigate the additional gains possible by introducing cooperative caching [14]. It should be possible to save further on bandwidth costs when proxies cooperate by exchanging content rather than going all the way to the origin server.

Another important aspect of our simulations is the clients' interactions with the presentations. We want to investigate client patterns in order to refine that part of our simulations. The patterns described in [7] are not particularly fine grained. They are, however, the most detailed ones available at the time of writing, and we did use the knowledge therein for the evaluation of the structured partial caching scheme. At present we are in contact with various newscasters in order to try and obtain logs or other means of statistics that would enable us to pinpoint more realistic NoD client access patterns.

References

1. Peter Schojer, Laszlo Bözörmenyi, Hermann Hellwagner, Bernhard Penz, and Stefan Podlipnig. Architecture of a quality based intelligent proxy (QBIX) for MPEG-4 videos. In *The Twelfth International World Wide Web Conference.,* pages 394–402. ACM, 2003.
2. Carsten Griwodz, Michael Bär, and Lars C. Wolf. Long-term movie popularity in video-on-demand systems. In *Proceedings of the ACM International Multimedia Conference (ACM MM),* pages 340–357, Seattle, WA, USA, November 1997.
3. Michael Zink, Carsten Griwodz, and Ralf Steinmetz. KOM player - a platform for experimental vod research. In *IEEE Symposium on Computers and Communications (ISCC),* pages 370–375, July 2001.

4. Pål Halvorsen. *Improving I/O Performance of Multimedia Servers.* PhD thesis, Department of Informatics, University of Oslo, Norway, September 2001. Published by Unipub forlag, ISSN 1501-7710 (No. 161).
5. Reza Rejaie, Haobo Yu, Mark Handley, and Deborah Estrin. Multimedia proxy caching for quality adaptive streaming applications in the Internet. In *Proceedings of the Joint Conference of the IEEE Computer and Communications Societies (INFOCOM),* pages 980–989, Tel-Aviv, Israel, March 2000.
6. Subhabrata Sen, Jennifer Rexford, and Don Towsley. Proxy Prefix Caching for Multimedia Streams. In *Proceedings of the Joint Conference of the IEEE Computer and Communications Societies (INFOCOM),* pages 1310–1319, New York, NY, USA, March 1999. IEEE Press.
7. Y.-J. Kim, T. U. Choi, K. O. Jung, Y. K. Kang, S. H. Park, and Ki-Dong Chung. Clustered multi-media NOD: Popularity-based article prefetching and placement. In *IEEE Symposium on Mass Storage Systems,* pages 194–202, 1999.
8. Lynda Hardman, Dick C. A. Bulterman, and Guido van Rossum. The amsterdam hypermedia model: adding time and context to the Dexter model. *Commun. ACM,* 37(2):50–62, 1994.
9. Carsten Griwodz, Michael Liepert, Abdulmotaleb El Saddik, Giwon On, Michael Zink, and Ralf Steinmetz. Perceived Consistency. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications,* June 2001.
10. Carsten Griwodz, Michael Liepert, and The Hynode Consortium. Personalised News on Demand: The HyNoDe Server. In *Proc. of the 4th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'97),* volume 1309, pages 241–250. Springer-Verlag, Berlin, Heidelberg, New York, September 1997.
11. L. Rutledge, J. van Ossenbruggen, L. Hardman, and D. Bulterman. Cooperative use of MHEG and HyTime in hypermedia environments, 1997.
12. Lloyd Rutledge, Lynda Hardman, and Jacco van Ossenbruggen. Evaluating SMIL: three user case studies. In *Proceedings of the seventh ACM international conference on Multimedia (Part 2),* pages 171–174. ACM Press, 1999.
13. Squid web proxy cache. http://www.squid-cache.org/.
14. Michael Rabinovich and Oliver Spatscheck. *Web caching and replication.* Addison Wesley, 2002.
15. David Salomon. *Data Compression - The complete reference.* Springer-Verlag, 2 edition, 2000.

Performance Evaluation of Distributed Prefetching for Asynchronous Multicast in P2P Networks*

Abhishek Sharma[1], Azer Bestavros[2], and Ibrahim Matta[2]

[1] Elec. & Comp. Eng., Boston University, Boston, MA 02215, USA abhishek@bu.edu
[2] Comp. Sc., Boston University, Boston, MA 02215, USA {best,matta}@cs.bu.edu

Abstract. We consider the problem of delivering real-time, near real-time and stored streaming media to a large number of asynchronous clients. This problem has been studied in the context of asynchronous multicast and peer-to-peer content distribution. In this paper we evaluate through extensive simulations the performance of the distributed *prefetching* protocol, dPAM [20], proposed for scalable, asynchronous multicast in P2P systems. We show that the *prefetch-and-relay* strategy of dPAM can reduce the server bandwidth requirement quite significantly, compared to the previously proposed *cache-and-relay* strategy, even when the group of clients downloading a stream changes quite frequently due to client departures.

1 Introduction

On-demand media distribution is fast becoming an ubiquitous service deployed over the Internet. The long duration and high bandwidth requirements of streaming media delivery present a formidable strain on server and network capacity. Hence, scalable delivery techniques, both in terms of network link cost as well as server bandwidth requirement, are critical for the distribution for highly popular media objects.

For the delivery of real-time media to synchronous requests, multicast solutions (whether using network support in case of IP multicast or using end-system support through peer-to-peer networks) are attractive as they reduce both network link costs and server bandwidth requirements for serving a large number of clients [22,5,13,11]. However, a number of scenarios can be envisioned in which the client requests for streaming media objects are likely to be *asynchronous.* This is true for requests to stored streaming media objects (e.g., on-demand delivery of popular movie clips or news briefs to clients), as well as for requests to buffered live streams (e.g., playout of a webcast to a large number of clients requesting that webcast asynchronously but within a short interval).

To enable asynchronous access to streaming media objects, various IP multicast based periodic broadcasting and stream merging techniques [22,13,11,17]

* This work was supported in part by NSF grants ANI-9986397, ANI-0095988, EIA-0202067 and ITR ANI-0205294.

have been proposed. These techniques are scalable in terms of network link cost by virtue of multicast messaging. To achieve scalability in terms of server bandwidth requirement, they try to ensure that a relatively small number of multicast sessions (possibly coupled with short unicast sessions) are enough to cater to a large number of asynchronous client requests. The assumption about the availability of a network infrastructure supporting IP multicast may be practical within the boundary of a multicast-enabled intranet, but it is yet to become an ubiquitous alternative in today's Internet. This realization has led to an alternate approach of using application-layer (or end-system) multucast.

Application-layer multicast, or overlay multicast, can facilitate the deployment of multicast-based applications in the absence of IP multicast [5]. Multicast can be achieved in overlay networks through data relay among overlay members via unicast. Apart from elevating the multicast functionality to the application layer, this approach also provides a substantial degree of flexibility due to the fact that each node in an overlay network can perform more complicated application-specific tasks which might be too expensive to perform at the routers in case of IP multicast.

1.1 Paper Contribution

In this paper, we evaluate, through extensive simulations, the impact of streaming rate and the departure of nodes on the scalability of the *prefetch-and-relay* strategy employed in the dPAM protocol [20], proposed for scalable asynchronous overlay multicast. We highlight the importance of "prefetching" content in achieving a better playout quality in a scenario where client nodes participating in the overlay network depart from the network (or stop downloading the stream). We refer the reader to [20] for a detailed description of dPAM and its analysis.

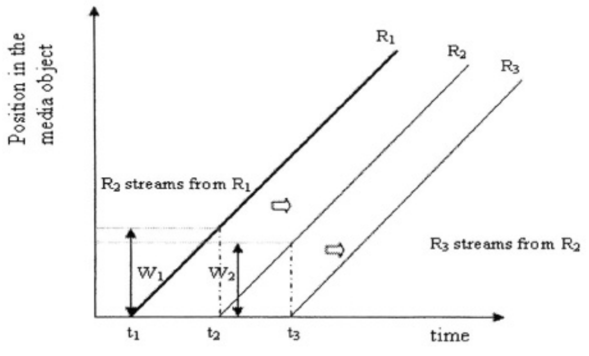


Fig.1. Overlay-based asynchronous streaming

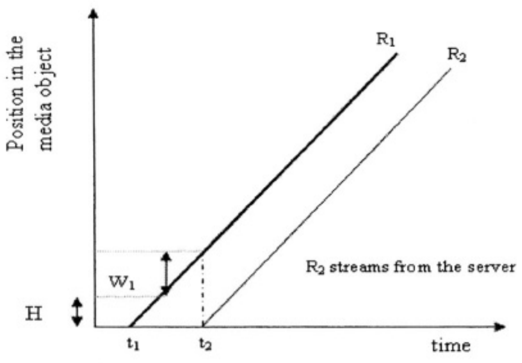


Fig. 2. Overlay-based aynchronous streaming

2 Prefetch-and-Relay

In this section, we review the *prefetch-and-relay* strategy employed in the distributed prefetching protocol, dPAM [20], proposed for scalable, asynchronous multicast in P2P networks. We illustrate the asynchronous delivery of streams through overlay networks using Figures 1 and 2. Assume that each client is able to buffer the streamed content for a certain amount of time after playback by overwriting its buffer in a circular manner. As shown in Figure 1, R_1 has enough buffer to store content for time length W_1; i.e. the data cached in the buffer is replaced by fresh data after an interval of W_1 time units. When the request R_2 arrives at time $t = t_2$, the content that R_2 wants to download is available in R_1's buffer and, hence, R_2 starts streaming from R_1 instead of going to the server. Similarly, R_3 streams from R_2 instead of the server. Thus, in Figure 1, leveraging the caches at end-hosts helps to serve three clients using just one stream from the server.

In Figure 2, by the time R_2 arrives, part of the content that it wants to download is missing from R_1's buffer. This missing content is shown as H in Figure 2. If the download rate is the same as the playout rate, then R_2 has no option but to download from the server. However, if the network (total) download rate is greater than the playback rate, then R_2 can open two simultaneous streams—one from R_1 and the other from the server. It can start downloading from R_1 at the playback rate (assuming that R_1's buffer is being overwritten at the playback rate) and obtain the content H from the server. After it has finished downloading H from the server, it can terminate its stream from the server and continue downloading from R_1. This stream patching technique to reduce server bandwidth was proposed in [12]. Assuming a total download rate of α bytes/second and a playback rate of 1 byte/second, the download rate of the stream from the server should be $\alpha - 1$ bytes/second. Hence, for this technique to work $\alpha - 1 \geq 1 \Rightarrow \alpha \geq 2$. Thus, we need the total download rate to be at least twice the playback rate for stream patching to work for a new arrival.

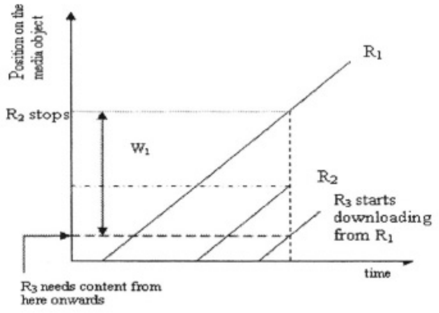

Position on the media object
R₂ stops
W₁
R₁
R₂
R₃ starts downloading from R₁
time
R₃ needs content from here onwards

Fig. 3. Switching streams on a departure: streaming from another client

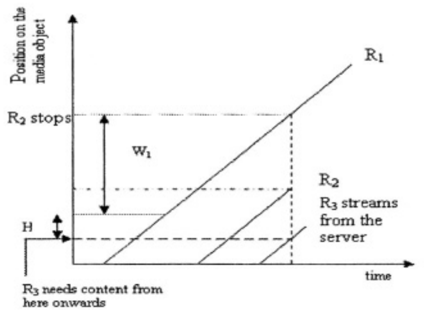

Position on the media object
R₂ stops
W₁
H
R₁
R₂
R₃ streams from the server
time
R₃ needs content from here onwards

Fig. 4. Switching streams on a departure: streaming from the server

A request may have to switch its streaming session under certain situations. As shown in Figure 3, R_3 initially streams from R_2 until R_2 leaves the overlay network. Since the content R_3 needs is still availabe in R_1'sbuffer, R_3 starts streaming from R_1 after R_2 departs. If the content needed by R_3 was missing from R_1's buffer, then R_3 could start streaming from the server after R_2's departute. In Figure 4, upon R_2 departure, the content that R_3 needs is not available in R_1's buffer and, hence R_3 is forced to stream from the server. The content missing from R_1's buffer is denoted as H in Figure 4. Similar to the case for a new arrival, if the total download rate is strictly greater than the playback rate, R_3 can open two simultaneous streams—one from R_1 and the other from the server. Once it has obtained the missing content H from the server it can terminate its stream from the server and continue to download from R_1. Unlike the case for a new arrival, as discussed in Section 2.3, the stream patching technique may work in this situation even when the total download rate is less than twice the playout rate.

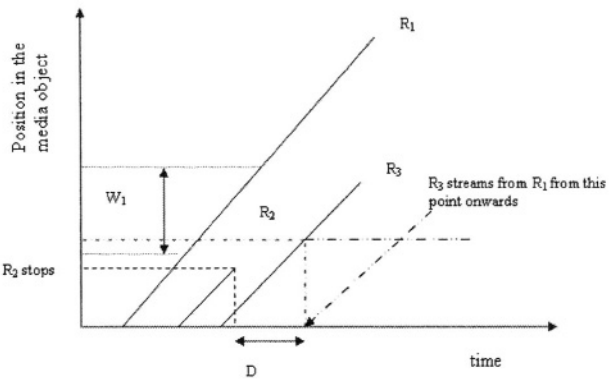

Position in the media object
R1
R3
R3 streams from R1 from this point onwards
W1
R2
R2 stops
D
time

Fig. 5. Delay in finding a new download source

When the download rate is greater than the playout rate, a client can *prefetch* content to its buffer before it is time to playout that content. Prefetching content can help achieve a better playout quality in overlay multicast. In a realistic setting, there would be a certain delay involved in searching for a peer to download from; for example, consider the situation depicted in Figure 5. R_3 starts streaming from R_2 on arrival. After R_2 departs, it takes R_3 D seconds (time units) to discover the new source of download R_1. If the prefetched "future" content in R_3's buffer, at the time of R_2's departure, requires more than D seconds (time units) to playout (i.e. the size of the future content is greater than D bytes, assuming a playout rate of 1 byte/second) then the playout at R_3 does not suffer any disruption on R_2's departure. If the size of the "future" content is smaller than D bytes, then R_3 will have to open a stream from the server, after it has finished playing out its prefetched content, until it discovers R_1. In a *cache-and-relay* strategy, clients do not prefetch content[1]. Thus, in the case of *cache-and-relay,* R_3 will have to open a stream from the server as soon as it realizes that R_2 has departed and continue downloading from the server for D seconds (until it discovers that it can download from R_1.) R_3 cannot know *a priori* when R_2 is going to depart. Due to the delays involved in opening a stream from the server, it is quite likely that the playout at R_3 would be disrupted

[1] It can be due to the fact that the playout rate is equal to the download rate or clients may choose not to prefetch content.

on R_2's departure in the case of *cache-and-relay.* In case of *prefetch-and-relay,* if the time required to playout the prefetched content is larger than the delay involved in finding a new source to download from, the playout at R_3 would not be disrupted upon R_2's departure from the peer-to-peer network. Prefetching content is also advantageous when the download rate is variable. A client can absorb a temporary degradation in download rate without affecting the playout quality if it has sufficient prefetched content in its buffer.

2.1 Control Parameters

In this paper, we use simulations to highlight the effect of the following three parameters in achieving scalable (in terms of server bandwidth), asynchronous delivery of streams in a peer-to-peer environment.

1. $\alpha = \frac{\text{Download rate}}{\text{Playout rate}}$

   Without loss of generality, we take the *Playout rate* to be equal to 1 byte/second and, hence the *Download rate* becomes α bytes/second. We assume $\alpha > 1$.
2. T_b : The time it takes to fill the buffer available at a client at the download rate.

   The actual buffer size at a client is, hence, $\alpha \times T_b$ bytes. The available buffer size at a client limits the time for which a client can download the stream at a rate higher than the playout rate.
3. $\beta = \frac{\text{Future Content}}{\text{Past Content}}$

   β represents the ratio of the content yet to be played out, "future content", to the content already played out, "past content", in the buffer. Given a particular α and T_b, it is easy to calculate that a client needs to download at the (total) rate α for $\left(\frac{\beta\alpha T_b}{(1+\beta)(\alpha-1)}\right)$ seconds to achieve the desired ratio β of "future" to "past" content in its buffer.

Next, we discuss the constraints, in terms of α, β and T_b, that must be satisfied for a client to be able to download the stream from the buffer of another client available in the peer-to-peer network.

2.2 Constraints in the Case of an Arrival

The following theorem is stated from [20]:
Theorem: A newly arrived client R_2 can download from the buffer of R_1 if the following conditions are satisfied:

– The inter-arrival time between R_1 and R_2 *is* less than T_b, or
– If the inter-arrival time between R_1 and R_2 is greater than T_b, then α should be greater than or equal to 2, R_1 must be over-writing the content in its buffer at the playout rate and the size of the content missing from R_1's buffer should be less than or equal to $\alpha \times T_b$.

The first condition ensures that the content needed by R_2 is present in R_1's buffer. The second condition defines the scenario in which the stream patching technique can be used by R_2 to "catch-up" with R_1.

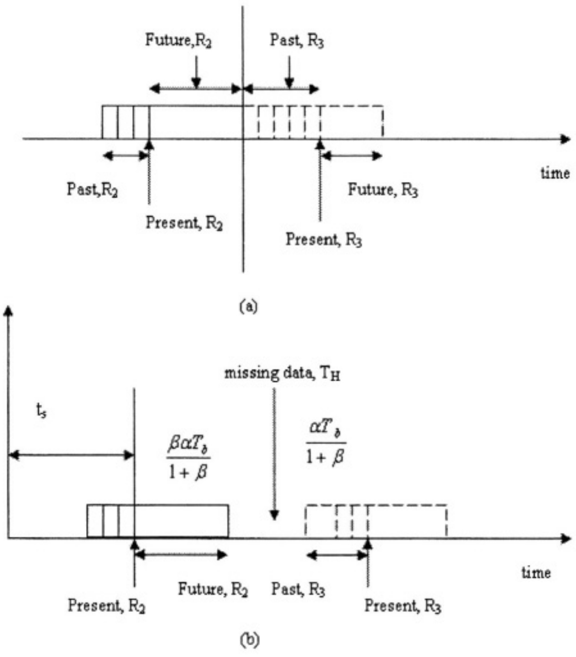


Fig. 6. Buffers of R_2 and R_3

2.3 Constraints in the Event of a Departure

Let us assume that R_2 was streaming from R_1's buffer. R_1 leaves the peer-to-peer network at time $t = t_d$. If the available buffer size at R_2 is $\alpha \times T_b$ bytes and at $t = t_d$, the ratio of "future" content to "past" content in R_2's buffer is β, then R_2 has $\left(\frac{\beta\alpha T_b}{1+\beta}\right)$ bytes of "future" content and $\left(\frac{\alpha T_b}{1+\beta}\right)$ bytes of "past" content in its buffer. At a *Playout rate* of 1 byte/time unit, R_2 has $\left(\frac{\beta\alpha T_b}{1+\beta}\right)$ time units to find a new source to download from after R_1 departs. Figure 6 presents the state of R_2's buffer after R_1 departs. The time axis represents the progression of the playout of the stream. The shaded portion labelled as "Past" refers to the $\left(\frac{\alpha T_b}{1+\beta}\right)$ bytes of "past" content and the portion labelled "Future" refers to the $\left(\frac{\beta\alpha T_b}{1+\beta}\right)$ bytes of "future" content in R_2's buffer. The arrow marking "Present" refers to the position in the media content that has been played out at R_2.

If $\alpha = 1$, then after R_1's departure, R_2 can download from another client R_3's buffer if and only if the contents in their buffers (partially) overlap. Figure 6(a) shows the situation when the buffers of R_2 and R_3 are contiguous. Any client that is ahead of R_3, in terms of playing out the stream, would have some content that R_2 needs to download missing from its buffer and hence, unsuitable for R_2 to download from. Figure 6(b) depicts such a situation.

Consider the situation depicted in Figure 6(b). Let us assume that the ratio of "future" content to "past" content in R_2's buffer is β and hence, it currently has $\left(\frac{\beta\alpha T_b}{1+\beta}\right)$ bytes of prefetched data. Assume that the "missing" content is T_H bytes and that the playout rate is 1 byte/second. If $\alpha > 1$, then as discussed earlier, R_2 can open two simultaneous streams, one from the server and the other from R_3, and terminate its stream from the server after it has downloaded the "missing" content and continue to download from R_3 thereafter. Note that for this stream patching technique to work, R_3 should be over-writing contents in its buffer at a rate less than α; in our model we assume that clients over-write the content in their buffer either at the download rate (α) or at the playout rate. Hence, in this case, R_3 should be over-writing its buffer at the rate of 1 byte/second. If this is the case, then R_2 can download from R_3 at the playout rate of 1 byte/second and download the "missing" content from the server at the rate of $(\alpha - 1)$ bytes/second.

The following constraints, stated from [20], must be satisfied by the size of the "missing" content, T_H bytes, for R_2 to able to stream from R_3's buffer:

1. **Constraint imposed due to α:**

$$(\alpha \times T_b)\left(\frac{\beta}{1+\beta}\right) + T_H \geq \frac{T_H}{\alpha - 1} \tag{1}$$

The above inequality demands that the time taken to playout the prefetched and the "missing" content should be no less than the time taken to download the "missing" content. Note that if $\alpha \geq 2$, then condition (1) is always satisfied. The stream patching technique can be used in the case of a departure even when $1 < \alpha < 2$ if a client has sufficient prefetched content.

2. **Constraint imposed by the size of the buffer:**

$$T_H \leq \frac{\alpha T_b}{1+\beta} \tag{2}$$

The above constraint demands that the size of the missing content, T_H, cannot be greater than the size of the "past" content in R_2'sbuffer.

3 Performance Evaluation

In this section, we compare the performance of the *prefetch-and-relay* based protocol, dPAM [20], and *cache-and-relay* with respect to savings in server bandwidth. We refer to the protocols oStream [6] and OSMOSIS [15] by the generic term *cache-and-relay* because they correspond to the situation where $\alpha = 1$ (hence, a client cannot prefetch any content.)

3.1 Simulation Model

We consider the case of a single CBR media distribution. The playback rate is assumed to be 1 byte/time unit. The client requests are generated according to a Poisson process. The time spent by a client downloading the stream is exponentially distributed with mean 1000 time units (for Figures 7, 8 and 9), or 100 time units (for Figure 10.) The total number of client arrivals during each simulation run is 1500. The parameter β is set to 100,000—so that almost all the content in a client's buffer is "future" content after it has been downloading the stream long enough. For the simulation results presented in Figures 7, 8, 9 and 10, we assume that a client is able to determine whether it should stream from the server or from the buffer of some other client without any delays both in the case of arrivals as well as in the event of a departure; i.e. we do not take delays like propagation delays and the delay involved in searching for a suitable client to download from into consideration. Results in the presence of such delays can be found in [20]. In calculating the server load, we do not consider the small-duration streams that a client opens from the server to obtain the "missing" content.

3.2 Summary of Observations

If the download rate is sufficiently high, $\alpha \geq 2$, the *prefetch-and-relay* scheme of dPAM has an advantage over the *cache-and-relay* scheme in reducing the server bandwidth when the resources available for overlay stream multicast are constrained, for example when the buffer size is small or when the request arrival rate is low. This advantage stems from the fact that a higher download rate enables a client to open two simultaneous connections for a short duration to "catch-up" with the buffer of another client using the technique of stream patching. This advantage is more pronounced for higher client departure rate. If clients depart frequently from the peer-to-peer network, it reduces the caching capacity of the peer-to-peer network, thus patching content from the server becomes more beneficial. As the buffer size and the request arrival rate increase, the advantage of *prefetch-and-relay* over *cache-and-relay* is mitigated and for a given buffer size, at a sufficiently high request arrival rate, *cache-and-relay* matches the performance of *prefetch-and-relay* in terms of server bandwidth even when the download rate is very high.

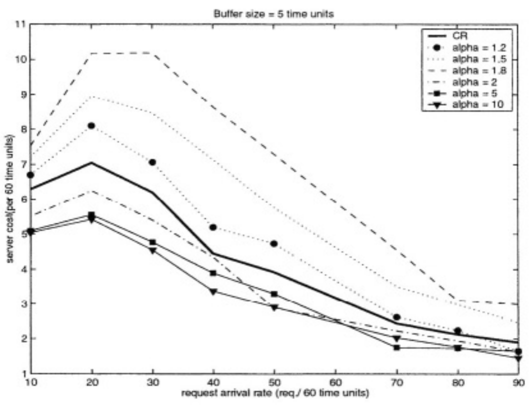


Fig. 7. Mean download time = 1000 time units, buffer size = 5 bytes

3.3 Simulation Results

For a fixed buffer size, we vary the download rate and measure the server load for different client arrival rates. Figures 7, 8 and 9 present the simulation results for different buffer sizes. For a fixed buffer size, *cache-and-relay* (*CR*) is able to match the performance of *prefetch-and-relay,* for all values of α considered in the simulation, once the client arrival rate increases beyond a certain threshold. This threshold depends on the size of the buffer — 80/(60 time units) in Figure 8 and 50/(60 time units) in Figure 9. The reason for this is two-fold. In these simulations, on average, a client downloads the stream for a duration (1000 time units) which is much longer than the time it needs to achieve the desired ratio β between its "future" and "past" content (approximately 5 time units for $\alpha = 5$ and buffer size of 20 bytes when $\beta = 100{,}000$.) Thus, most of the clients have achieved the ratio β by the time the client they were downloading from departs the peer-to-peer network. Since the value of β is set very high ($\beta = 100{,}000$), almost the entire buffer of a client is full of "future" content when the client it was downloading from leaves the overlay network. Hence, in the event of a departure, since condition (2) of Section 2.3 is often violated, clients are not able to take advantage of the higher download rate (by opening two simultaneous streams and doing stream patching.) Thus, on a departure, a client that needs a new source for download, can start downloading from another client only if their buffers (partially) overlap. This situation is similar to *cache-and-relay* ($\alpha = 1$) (refer to the discussion in Section 2) and hence, the stream patching technique employed in the dPAM protocol does not provide any advantage over *cache-and-relay.* If we set β to be small, say 2 or 3, then a client would be able to take advantage of the stream patching technique in the event of a departure.

The second reason for *cache-and-relay* being able to achieve the same server load as *prefetch-and-relay* beyond a certain request arrival rate threshold is as follows. A new arrival can take advantage of higher download rate and stream

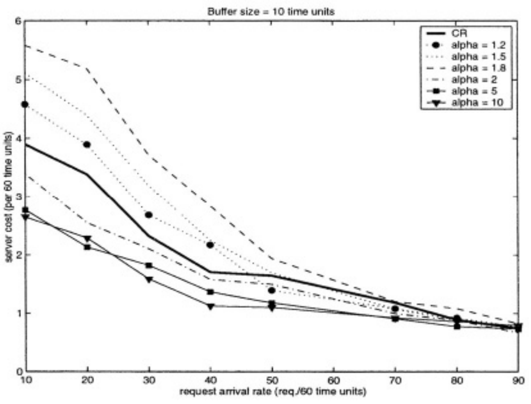


Fig. 8. Mean download time = 1000 time units, buffer size = 10 bytes

patching under dPAM but as the client arrival rate gets higher, a client rarely needs to resort to such capability. Hence, for a sufficiently high client arrival rate, the advantage that *prefetch-and-relay* has over *cache-and-relay,* due to clients being able to patch streams from the server (using a higher total download rate), is mitigated by the presence of a large number of clients in the peer-to-peer network enabling *cache-and-relay* to match the performance of *prefetch-and-relay.*

When $1 < \alpha < 2$, *prefetch-and-relay* leads to a greater server load than *cache-and-relay* for low arrival rates. As α increases, the time taken to fill the buffer at download speed, T_b, decreases. For example, for a buffer size of 5 bytes, for *cache-and-relay* $(\alpha = 1)$, $T_b = 5$; whereas when α=1.8, T_b=2.78. Thus, in the former case, a new arrival can reuse the stream from someone who arrived at most 5 time units earlier whereas in the latter case a new arrival can download from someone who arrived at most 2.78 time units earlier[2]. Hence, in the latter case, more new arrivals have to download from the server. This effect can be mitigated by increasing the buffer size and also for higher arrival rate.

When the download rate at least twice as fast as the playback rate $(\alpha \geq 2)$ *prefetch-and-relay* achieves a much lower server load than *cache-and-relay* even for small buffer sizes and low request arrival rate. When $1 < \alpha < 2$, *prefetch-and-relay* does not have any advantage over *cache-and-relay.* Since in these simulations, we ignore the delay involved in searching for a suitable client to download from, the only advantage that *prefetch-and-relay* has over *cache-and-relay* comes from the fact that it enables a client to "catch-up" with another client by downloading the "missing" data from the server in the event of a departure. But for small α $(1 < \alpha < 2)$, condition (1) of Section 2.3 is often violated. For example, with $\alpha = 1.2$ bytes/second, it will take 5 seconds to

[2] Since $1 < \alpha < 2$, a new arrival cannot take advantage of the stream-patching technique.

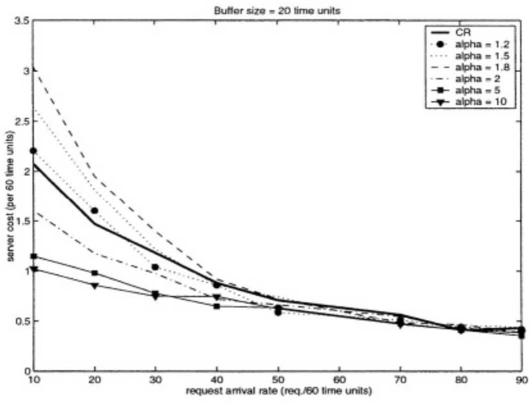


Fig. 9. Mean download time = 1000 time units, buffer size = 20 bytes

acquire a "missing" data of size 1 byte from the server. Hence, when the available buffer size is 5 bytes, a client never opens two simultaneous streams to do stream patching, because if it contains the required 5 bytes of "future" content to satisfy condition (1) of Section 2.3, then its buffer is already full of "future" content and it has no available buffer space to download the "missing" content from the server, which violates condition (2) of Section 2.3. On the other hand, if the client has less than 5 bytes of "future" content then condition (1) of Section 2.3 is violated. Combined with our observation in the preceeding paragraph, it becomes clear why *prefetch-and-relay* has a much higher server load compared to *cache-and-relay* for $1 < \alpha < 2$ when the client arrival rate is low.

The results in Figures 7, 8 and 9 show that as the available buffer at the client increases, the required server bandwidth to support a particular request arrival rate decreases, under both *cache-and-relay* as well as *prefetch-and-relay* (for all values of α.) This observation is in agreement with the results obtained in [6].

The amount of time that clients spend downloading a stream is an important factor in determining server bandwidth requirements. Peer-to-peer asynchronous media content distribution is suited for situations in which the content being distributed is large; so that the end-hosts participating in the peer-to-peer network are available for a long time. In a scenario where end-hosts keep departing after a short interval, the server load can be considerably high due to the fact that a lot of requests may have to start downloading from the server due to the departure of clients they were downloading from. Figure 10 presents the simulation results when the mean time spent by a client downloading the stream $(1/\mu)$ is 100 time units. Compared to the case when $1/\mu = 1000$, the server bandwidth requirement is considerably higher even for very high client arrival rates. Figure 10 illustrates the fact that the *prefetch-and-relay* scheme with $\alpha \geq 2$ performs better than the *cache-and-relay* scheme, in terms of the server bandwidth re-

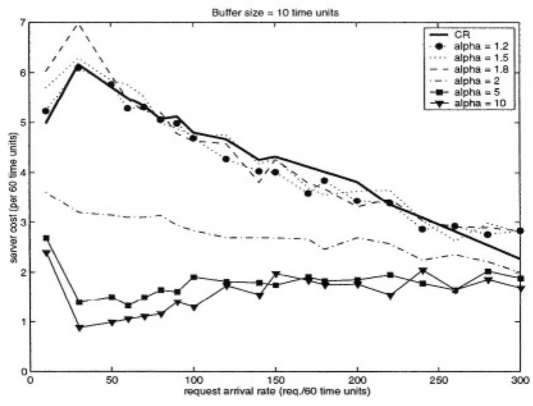


Fig. 10. Mean download time = 100 time units, buffer size = 10 bytes

quirement, even for very high client arrival rates (compare with Figures 7, 8 and 9.) Due to the shorter content download time of the clients, in a significant number of situations, clients are able to take advantage of the higher download rate through stream patching [12] in the event of a departure and hence, the server load is less under the *prefetch-and-relay* scheme compared to the *cache-and-relay* scheme. Figure 10 also shows that if the client request arrival rate keeps on increasing, eventually the *cache-and-relay* scheme will be able to match the performance of the *prefetch-and-relay* scheme.

4 Related Work

Delivery of streams to asynchronous clients has been the focus of many studies, including periodic broadcasting [22,13,11,17] and stream patching/merging techniques [4,10,7,8]. In periodic broadcasting, segments of the media object (with increasing sizes) are periodically broadcasted on dedicated channels, and asynchronous clients join one or more broadcasting channels to download the content. The approach of patching [12] allows asynchronous client requests to "catch up" with an ongoing multicast session by downloading the missing portion through server unicast. In merging techniques [9], clients merge into larger and larger multicast session repeatedly, thus reducing both the server bandwidth and the network link cost. These techniques rely on the availability of a multicast delivery infrastructure at the lower level.

The idea of utilizing client-side caching has been proposed in several previous work [21,19,14,16]. The authors of [6] propose an overlay, multicast strategy, *oStream,* that leverages client-side caching to reduce the server bandwidth as well as the network link cost. Assuming the client arrivals to be Poisson distributed, they derive analytical bounds on the server bandwidth and network link cost. However, this work does not consider the effect of the departure of the end-systems from the overlay network on the efficiency of overlay multicast. *oStream* does not consider the effect of streaming rate—it is a *cache-and-relay* strategy—and hence, does not incorporate patching techniques to reduce server bandwidth when the download rate is high. The main objective of the protocol, *OSMOSIS,* proposed in [15] is to reduce the network link cost. The effect of patching on server load has not been studied.

A different approach to content delivery is the use of periodic broadcasting of encoded content as was done over broadcast disks [1] using IDA [18], and more recently using the Digital Fountain approach which relies on Tornado encoding [3,2]. These techniques enable end-hosts to reconstruct the original content of size n using a subset of any n symbols from a large set of encoded symbols. Reliability and a substantial degree of application-layer flexibility can be achieved using such techniques. But these techniques are not able to efficiently deal with real-time (live or near-live) streaming media content due to the necessity of encoding/decoding rather large stored data segments.

5 Conclusion

Through extensive simulations, we evaluated the performance of the distributed *prefetching* protocol, dPAM [20], proposed for scalable, asynchronous multicast in P2P systems. Our results show that when the download rate is at least twice as fast as the playout rate, a significant reduction in server bandwidth can be achieved, compared to a *cache-and-relay* strategy, when the resources available for overlay multicast are constrained, i.e. small client buffers and low client arrival rate. We evaluated the impact of departure of client nodes on the server bandwidth requirement, and highlighted the fact that the time spent by the clients downloading the stream is a crucial factor affecting the scalability of end-system multicast built upon client-side caching. We also discussed the advantage of prefetching content in improving the playout quality at the client nodes in the presence of various delays. We refer the reader to [20] for an extended analysis of dPAM in the presence of delays as well as its implementation.

References

1. A. Bestavros. AIDA-based Real-time Fault-tolerant Broadcast Disks. In *Proceedings of IEEE RTAS'96,* May 1996.
2. J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *Proceedings of ACM SIGCOMM,* 2002.
3. J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *Proceedings of ACM SIGCOMM,* 1998.
4. S. W. Carter and D. D. E. Long. Improving Video On-demand Server Efficiency through Stream Tapping. In *Proceedings of IEEE International Conference on Computer Communication and Networks (ICCN),* 1997.
5. Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM,* 2001.

6. Y. Cui, B. Li, and K. Nahrstedt. oStream: Asynchronous Streaming Multicast in Application-Layer Overlay Networks. *IEEE Journal on Selected Areas in Communications,* 22(1), January 2004.
7. D. Eager, M. Vernon, and J. Zahorjan. Minimizing Bandwidth Requirements for On-demand Data Delivery. In *Proceedings of Workshop on Multimedia and Information Systems(MIS),* 1998.
8. D. Eager, M. Vernon, and J. Zahorjan. Bandwidth Skimming: A Technique for Cost-efficient Video On-demand. In *Proceedings of ST/SPIE Conference on Multimedia Computing and Networking (MMCN),* 2000.
9. D. Eager, M. Vernon, and J. Zahorjan. Minimizing Bandwidth Requirements for On-Demand Data Delivery. *IEEE Transactions on Knowledge and Data Engineering,* 13(5), 2001.
10. L. Gao and D. Towsley. Supplying Instantaneous Video On-demand Services using Controlled Multicast. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS),* 1999.
11. A. Hu. Video-on-demand Broadcasting Protocols: A Comprehensive Study. In *Proceedings of IEEE INFOCOM,* April 2001.
12. K. Hua and Y. Cai amd S.Sheu. Patching: A Multicast Technique for True On-demand Services. In *Proceedings of ACM Multimedia,* 1998.
13. K. A. Hua and S. Sheu. Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-on-demand Systems. In *Proceedings of ACM SIGCOMM,* September 1997.
14. K. A. Hua, D. A. Tran, and R. Villafane. Caching Multicast Protocol for On-demand Video Delivery. In *Proceedings of ST/SPIE Conference on Multimedia Computing and Networking (MMCN),* 2000.
15. S. Jin and A. Bestavros. OSMOSIS: Scalable Delivery of Real-Time Streaming Media in Ad-Hoc Overlay Networks. In *Proceedings of IEEE ICDCS'03 Workshop on Data Distribution in Real-Time Systems,* 2003.
16. D. Ma and G. Alonso. Distributed Client Caching for Multimedia Data. In *Proceedings of the 3rd International Workshop on Multimedia Information Systems(MIS97),* 1997.
17. A. Mahanti, D. Eager, M. Vernon, and D. Sundaram-Stukel. Scalable On-demand Media Streaming with Packet Loss Recovery. In *Proceedings of ACM SIGCOMM,* 2001.
18. M. O. Rabin. Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance. In *Journal of the Association for Computing Machinery,* volume 36, pages 335–348, April 1997.
19. S. Ramesh, I. Rhee, and K. Guo. Multicast with Cache (mcache): An Adaptive Zero-delay Video-on-demand. In *Proceedings of IEEE INFOCOM,* 2001.
20. A. Sharma, A. Bestavros, and I. Matta. dPAM: A Distributed Prefetching Protocol for Scalable, Asynchronous Multicast in P2P Systems. In *Technical Report BUCS-TR-2004-026,* 2004.
21. S. Sheu, K. Hua, and W. Tavanapong. Chaining: A Generalized Batching Technique for Video On-demand. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS),* 1997.
22. S. Viswanathan and T. Imielinski. Pyramid Broadcasting for Video On Demand Service. In *Proceedings of ST/SPIE Conference on Multimedia Computing and Networking (MMCN),* 1995.

On the Equivalence of Forward and Reverse Query Caching in Peer-to-Peer Overlay Networks

Ali Raza Butt[1], Nipoon Malhotra[1], Sunil Patro[2], and Y. Charlie Hu[1]

[1] Purdue University, West Lafayette IN 47907, USA,
{butta, nmalhot, ychu}@purdue.edu
[2] Microsoft Corporation, One Microsoft Way, Redmond WA 98052, USA,
patro@microsoft.com

Abstract. Peer-to-peer systems such as Gnutella and Kazaa are used by millions of people for sharing music and many other files over the Internet, and they account for a significant portion of the Internet traffic. The traffic in a peer-to-peer overlay network is different from that in WWW in that each peer is both a client and a server. This suggests that one can deploy a forward cache at the Internet gateway of a network to reduce the amount of queries going outside, or a revere cache at the gateway to reduce the amount of queries going inside, which in turn reduces the queries that are forwarded outside. In this paper, we study the effectiveness of forward and reverse caching at the gateway via analysis and experimental measurement. Our study shows that forward caching and reverse caching at the gateway are equally effective at reducing query and query reply traffic across the gateway.

1 Introduction

In the first six years of the Internet explosion, one type of dominating traffic over the Internet had been HTTP traffic generated from widespread accessing of the World Wide Web. Around the year 2000, a new paradigm for Internet applications emerged and has quickly prevailed. Today, the so-called peer-to-peer (p2p) systems and applications such as Gnutella and Kazaa are routinely used by millions of people for sharing music and other files over the Internet, and they account for a significant portion of the Internet traffic. For example, the continuous network traffic measurement at the University of Wisconsin (http://wwwstats.net.wisc.edu) shows that peer-to-peer traffic (Kazaa, Gnutella, and eDonkey) accounts for 25–30% of the total campus traffic (in and out) in August 2002, while at the same time, web-related traffic accounts for about 23% of the total incoming and 10% of the total outgoing traffic.

The traffic generated by a p2p network such as Gnutella falls under two categories: the protocol messages for maintaining the overlay network and for searching data files, and the actual data messages for downloading files. Since the actual data download is performed through direct connections between the source and destination servents via HTTP, such traffic can be cached using off-the-shelf web caching proxies. Thus the more interesting question is how to reduce the protocol messages which are typically propagated in the overlay network via controlled flooding.

Similar to accesses of web content, researchers have found that search messages (queries) in p2p networks such as Gnutella exhibit temporal locality. This suggests that caching can prove to be an effective technique in reducing network bandwidth consumed by these queries and the latency in retrieving query replies. While the locality in web accesses is determined solely by the URLs being accessed, the locality in the queries in p2p networks also takes into account the TTLs and the next hop nodes in the overlay, as these two factors also affect the set of the query replies that will be received.

Several query caching schemes have been developed and have confirmed the effectiveness of query caching [1,2,3]. In particular, a transparent caching scheme has been proposed in [3] where p2p caching proxies are attached to the gateway routers of organizations or ISPs, i.e., similar to how web caching proxies are typically deployed, with the goal of reducing p2p traffic in and out of the gateways. The gateway router is configured to redirect TCP traffic going outside and to well known p2p ports, e.g., 6346 for Gnutella, to the attached p2p caching proxies. We call this *forward caching* as the the cache acts on outgoing queries.

A fundamental difference between the traffic in p2p overlay networks and web traffic is that a peer in a p2p network is both a client and a server. This observation suggests that one can also deploy a *reverse caching* proxy at the gateway of an organization to exploit the locality in queries coming into that organization. Intuitively, compared to a forward proxy, a reverse proxy will see fewer distinct queries since there are fewer peers within an organization than their neighbors outside, and query locality distinguishes queries sent to different forwarders. Consequently, reverse caching is expected to achieve a higher cache hit ratio than forward caching due to fewer capacity misses, and thus is seemingly more effective than forward caching in reducing query traffic across the gateway.

In this paper, we study the relative effectiveness between transparent forward and reverse caching at the gateway focusing on Gnutella networks. We present a detailed analysis that shows reverse caching and forward caching are equally effective assuming the caches store the same number of query hits, disregarding how many queries they are for. We further confirm our analysis with experimental results collected from a testbed running eight Gnutella servents behind the caching proxies.

2 Preliminaries

2.1 The Gnutella Protocol

To set up the context for the analysis of forward and reverse caching, we briefly discuss Gnutella's node joining process, its implications on the topology of the part of Gnutella network inside an organization, and the query request and reply protocols. The details of the Gnutella protocol can be found in [4,5].

Topology of Gnutella networks inside an organization. The joining process of a typical Gnutella servent is as follows. When a servent wants to connect, it first looks up its *host cache* file to find addresses of Gnutella servents to connect to. The servent addresses are removed from the host cache after they are read. If the host cache is empty, it tries to connect to a well known Gnutella *host cache server* (also called PONG server) to receive PONG messages in response to its PING message. After receiving the PONG messages, the servent tries to establish connections with the servents whose addresses are obtained from the PONG messages.

A typical Gnutella servent S, after establishing a pre-specified number of Gnutella connections, periodically sends PING messages to monitor these connections. In response S receives a number of PONG messages[1], which are appended at the end of S's host cache file. In addition, when an existing connection with some servent S_1 is broken down, S_1's address information is saved and eventually will be added to S's host cache when it leaves the Gnutella network.

In summary, during the joining process of a typical Gnutella servent, the neighbors are chosen from the host cache whose content is fairly random. This suggests that it is unlikely servents from the same organization will become neighbors of each other, and consequently query messages will travel across the gateway of the organization.

Query and query replies. In order to locate a file, a servent sends a query request to all its direct neighbors, which in turn forward the query to their neighbors, and the process repeats. Each Gnutella query is identified by a unique tuple of (`muid`, `query string`, `forwarder`, `neighbor`, `ttl`, `minimum speed`) values, where the query is forwarded from the `forwarder` servent to the `neighbor` servent, and the `minimum speed` value specifies the minimum speed that a servent should be able to connect at if replying to the query. When a servent receives a query request, it searches its local files for matches to the query and returns a query reply containing all the matches it finds. Query replies follow the reverse path of query requests to reach the servents that initiated the queries. The servents along the path do not cache the query replies.

To avoid flooding the network, each query contains a TTL field, which is usually initialized to a default value of 7. When a servent receives a query with a positive TTL, it decrements the TTL before forwarding the query to its neighbors. Queries received with TTL equal to 1 are not forwarded.

2.2 Transparent Query Caching

The natural way of performing transparent query caching in p2p overlay networks is similar to transparent web caching. A caching proxy is attached to the gateway router, the router is configured to redirect outgoing TCP traffic to certain designated ports (e.g., 6346 for Gnutella) to the attached proxy, and the proxy hijacks such connections going out of the gateway. The proxy can then

[1] In Gnutella version 0.6, a servent uses its PONG cache to generate PONG messages.

cache query replies from outside the gateway and use them in the future to reply to queries from inside. Since the cached query replies are used to reply to queries going outside the gateway, we call this *forward query caching*.

However, queries can also come inside the gateway along the hijacked outgoing connections at the proxy. This is due to a fundamental difference between caching in p2p and web caching, that is, a peer node inside an organization acts both as a client and a server, while nodes involved in web traffic inside an organization are only clients (i.e., browsers). Therefore, in principle, the caching proxy can also inspect the queries that come from outside the gateway on the hijacked connections and cache query replies received from the nodes inside. The proxy can then use the cached query replies to serve future queries coming from outside along the hijacked connections. We call this *reverse query caching*.

Hijacking incoming connections. The above transparent caching scheme only hijacks outgoing connections from servents inside the router, i.e., Gnutella connections that are initiated by inside servents to port 6346 of outside servents. The incoming connections from servents outside the router to servents inside can also be hijacked by configuring the external interface of the router to redirect incoming traffic to an attached proxy, similar to how outgoing connections are hijacked.

We note that the difference between our definitions of forward and reverse caching are orthogonal to whether the hijacked connections are outgoing or incoming connections in the overlay, since the query traffic going through them is indifferent to who initiated the connections.

2.3 The Caching Algorithm

We summarize the caching algorithm used by the proxy previously described in [3]. First, all the PING/PONG messages initiated or forwarded by the servents, inside or outside, going across the gateway will be forwarded by the caching proxy. The caching proxy will not change the TTL, and thus the reachability of PING/PONG messages remains the same as before. Similarly, HTTP data download messages will be tunneled through unaffected.

Our caching algorithm caches query hits according to the tuple of `(query string, neighbor, ttl)` values of the query they correspond to. It uses two main data structures. First, any time the proxy tunnels a query to outside servents, it records the `muid`, the query string, and the TTL information in a Cache Miss Table (CMT). When a query hit is received from outside, its `muid` is checked against CMT to find the corresponding query string and TTL, which is used to index into the cache table. The cache table (CT) is the data structure for storing cached query hits. For each CT entry CT(i), the algorithm adds a vector field to remember the `muid` and `forwarder` of up to 10 most recent queries for which query hits are replied using that cache entry.

Every time a new query results in a cache hit with CT(i), i.e., with a matching tuple `(query string, neighbor, ttl)`, the `muid` of the new message is compared with those stored in CT(i). If the `muid` matches that of any of the

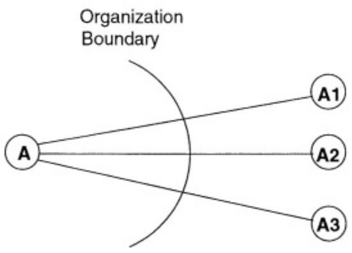


Fig. 1. Servent node *A* and its three neighbors outside the gateway.

10 stored previous queries replied to by the proxy using CT(i), it suggests that the same query reached the proxy a short while ago, and the query is dropped. Otherwise, the proxy is seeing this query message with this `muid` for the first time, and hence the proxy replies from the cache. The `muid` of this new query is then stored in the vector field of the corresponding CT entry.

Speed rewriting. Upon a cache miss, the proxy rewrites the minimum speed field of the query to zero before forwarding it to outside the gateway. As a result, it collects query hits with all possible speeds. For a subsequent query that matches all other parameters with the cached query hits, but specifies a non-zero minimum speed requirement, the proxy can always extract a subset of the cached query hits that satisfy the minimum speed requirement, without forwarding the query out of the gateway again.

3 Analysis

In this section, we analytically compare the effectiveness of forward and reverse query caching at the gateway. Without loss of generality, we focus on a single Gnutella servent inside the gateway, which has *C* neighbors in the p2p overlay, all of which are outside the gateway (Section 2.1). The following comparison analysis between forward and reverse caching also holds true for multiple Gnutella servents inside the gateway if the multiple servents do not share any neighboring nodes outside the gateway. The disjoint neighborhood among multiple servents behind the gateway is expected because of the node join process as discussed in Section 2.1, and is also confirmed to be true in our experiments. Figure 1 shows servent *A* and its 3 neighbors in the p2p network outside the gateway.

We assume all servents in the p2p network initiate queries at about the same frequency. Since queries have a typical TTL of 7, each query is expected to reach a large number of servents in the p2p network. Conversely, compared to the queries generated by each servent, a much larger number of queries will reach and be forwarded by that servent. Thus we can ignore the queries generated by servent *A* in analyzing the query traffic related to it, e.g., in the scenario in Figure 1.

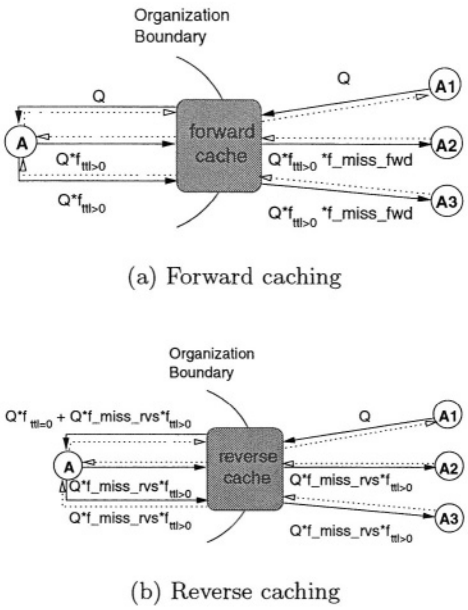


Fig. 2. The query and query hit traffic on each hijacked neighbor connection of servent A triggered by Q queries initiated from servent $A1$, under forward and reverse caching at the gateway, respectively. Starting from $A1$, solid arrows show queries (1) received from outside, (2) sent to inside, (3) received from inside, and (4) sent to outside. Starting from $A2$ or $A3$, dashed arrows show query hits (5) received from outside, (6) sent to inside, (7) received from inside, and (8) sent to outside. $f\_ttl > 0$ denotes the fractions of queries with $ttl > 0$ after decrementing at A. f_miss_fwd and f_miss_rvs denote the cache miss ratio in forward and reverse caching, respectively.

We assume all connections between A and its neighbors are hijacked by the proxy. We analyze the amount of query and query traffic on each hijacked connection triggered by a set of Q queries initiated from or forwarded by a single neighbor – servent $A1$ – in the forward and reverse caching scenarios. The amount of queries sent on each connection are shown in Figure 2(a) and Figure 2(b) for forward and reverse caching, respectively.

The numbers of queries on each connection in forward caching shown in Figure 2(a) are explained as follows. First, the proxy receives all Q queries from servent $A1$. It then passes all of them through to servent A. Out of these queries, $Q \cdot f_{ttl>0}$ (after decrementing the TTL at A) will be forwarded to each of A's remaining $(C - 1)$ neighbors in the p2p network, where $f_{ttl>0}$ is the fraction of queries with $ttl > 0$. These are the queries on which the proxy acts, i.e., by performing cache lookups. Finally, $Q \cdot f_{ttl>0} \cdot f_{miss\_fwd} \cdot (C-1)$ of them will be cache misses and forwarded to the $(C - 1)$ neighbors outside the gateway, where f_{miss_fwd} is the query miss ratio in forward caching.

The numbers of queries on each connection in reverse caching shown in Figure 2(b) are explained as follows. Once again, the proxy receives all Q queries from servent $A1$. First, it forwards the ones with $ttl = 0$ (after decrementing the TTL) directly to A. The reason the proxy does not perform caching on queries with $ttl = 0$ is that such queries will not be forwarded further once reaching the servent inside the gateway, thus caching such queries will not reduce the outgoing query traffic. Furthermore, the proxy can avoid significant consumption of its cache capacity by such queries; around 60-70% queries received by any servent and thus going through the proxy have $ttl = 0$ (after decrementing the TTL). Second, the proxy acts on the incoming queries with $ttl > 0$ by directly replying to them if the corresponding query hits are already cached. It then passes the remaining $Q \cdot f_{ttl>0} \cdot f_{miss\_rvs}$ queries to servent A, where f_{miss_rvs} is the query miss ratio in reverse caching. Servent A then forwards these queries to each of its remaining $(C - 1)$ neighbors in the p2p network.

Table 1 summarizes the number of queries and query hits sent along each of the four directions. Query hits travel in opposite directions to their query counterparts. In forward caching (Figure 2(a)), $Q \cdot f_{ttl>0} \cdot f_{miss\_fwd} \cdot (C-1)$ queries sent to outside the gateway result in $Q \cdot f_{miss\_fwd} \cdot (C - 1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$ query hits, where $C_{QH}(k)$ is the average number of query hits for a query with $ttl = k$. Combined with the query hits from cache hits in the forward cache, the total number of query hits sent by the proxy to servent A is $Q \cdot (C - 1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$. Finally, $Q \cdot L_{QH}$ query hits are generated from servent A's local object store, and sent back to servent $A1$, along with the query hits received by A from the proxy.

The query hits in reverse caching (Figure 2(b)) are also shown in Table 1. Out of the Q queries received from $A1$, $Q \cdot f_{ttl=0} + Q \cdot f_{ttl>0} \cdot f_{miss\_rvs}$ are sent to A by the proxy, as it does not cache query hits for queries with $ttl = 0$. A in turn forwards the $Q \cdot f_{ttl>0} \cdot f_{miss\_rvs} \cdot (C - 1)$ duplicated queries to outside servents, and receives back $Q \cdot f_{miss\_rvs} \cdot (C - 1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$ query hits. Servent A then sends these query hits along with $(Q \cdot f_{ttl=0} + Q \cdot f_{ttl>0} \cdot f_{miss\_rvs}) \cdot L_{QH}$ query hits generated locally to the proxy. Finally, the proxy sends all the query hits forwarded from A along with $Q \cdot f_{hit\_rvs} \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k+1)$ query hits from its cache to $A1$.

The following conclusions can be drawn from Table 1.

- The volumes of the queries sent to outside in forward caching and in reverse caching ((4) in Table 1) would be the same if f_{miss_fwd} and f_{miss_rvs} are equal.
- The volumes of the query hits received from outside in forward caching and in reverse caching ((5) in Table 1) would be the same if f_{miss_fwd} and f_{miss_rvs} are equal.
- The total number of query hits sent back to outside (e.g., to servent $A1$) in forward caching and in reverse caching ((8) in Table 1) are the same. This makes sense as caching is expected to preserve the end user experience in searching data files.

Table 1. Query and query hit traffic going in and out hijacked connections of servent A, triggered by Q queries coming in from one neighbor servent outside the gateway. C is the number of neighbors in the p2p overlay. All neighbors are outside the gateway. L_{QH} is the average query hits per query from servent A's local object store. $C_{QH}(k)$ is the average number of query hits for a query with $ttl = k$. $f_{ttl=k}$ denotes the fractions of all Q queries with $ttl = k$ after decrementing at A. The derivation in (8) for reverse caching uses $C_{QH}(k+1) = (C-1) \cdot C_{QH}(k) + L_{QH}$.

Category	Forward Caching
Queries:	
(1) rcvd from outside	Q
(2) sent to inside	Q
(3) rcvd from inside	$Q \cdot f_{ttl>0} \cdot (C-1)$
(4) sent to outside	$Q \cdot f_{ttl>0} \cdot f_{miss\_fwd} \cdot (C-1)$
Query Hits:	
(5) rcvd from outside	$Q \cdot f_{miss\_fwd} \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$
(6) sent to inside	$Q \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$
(7) recv from inside	$Q \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k) + Q \cdot L_{QH}$
(8) sent to outside	$Q \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k) + Q \cdot L_{QH}$

Category	Reverse Caching
Queries:	
(1) rcvd from outside	Q
(2) sent to inside	$Q \cdot f_{ttl=0} + Q \cdot f_{ttl>0} \cdot f_{miss\_rvs}$
(3) rcvd from inside	$Q \cdot f_{ttl>0} \cdot f_{miss\_rvs} \cdot (C-1)$
(4) sent to outside	$Q \cdot f_{ttl>0} \cdot f_{miss\_rvs} \cdot (C-1)$
Query Hits:	
(5) rcvd from outside	$Q \cdot f_{miss\_rvs} \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$
(6) sent to inside	$Q \cdot f_{miss\_rvs} \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$
(7) recv from inside	$Q \cdot f_{miss\_rvs} \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$
	$+(Q \cdot f_{ttl=0} + Q \cdot f_{ttl>0} \cdot f_{miss\_rvs}) \cdot L_{QH}$
(8) sent to outside	$Q \cdot f_{miss\_rvs} \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k)$
	$+(Q \cdot f_{ttl=0} + Q \cdot f_{ttl>0} \cdot f_{miss\_rvs}) \cdot L_{QH}$
	$+Q \cdot f_{hit\_rvs} \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k+1)$
	$= Q \cdot (C-1) \cdot \sum_{k=1}^{6} f_{ttl=k} \cdot C_{QH}(k) + Q \cdot L_{QH}$

Thus the effectiveness comparison of forward and reverse caching boils down to the relative values of the query cache miss ratios in forward and reverse caching, i.e., f_{miss_fwd} and f_{miss_rvs}.

Assuming the forward and reverse caches have the same capacity, i.e., they can store the same total number of query hits for however many queries, the cache hit ratios in forward and reverse caching should be similar for the following reasons. First of all, the localities in the queries filtered by the forward cache and the reverse cache are the same. This is because the $Q \cdot f_{ttl>0} \cdot (C-1)$ queries acted on by the forward cache is from duplicating $(C-1)$ times the $Q \cdot f_{ttl>0}$ queries that would be seen by the reverse cache, one for each of A's $(C-1)$ neighbors outside the gateway, and the caching algorithm distinguishes the same query forwarded to different next hop nodes.

Moreover, since the number of query hits to be received by the proxy from sending $Q \cdot f_{ttl>0}$ queries to servent A in reverse caching is similar to the number of query hits to be received by the proxy from sending the duplicated $Q \cdot f_{ttl>0} \cdot (C - 1)$ queries to $A2$ and $A3$ (after decrementing the TTL), assuming the difference – the query hits generated by A locally – is insignificant, the total number of query hits received by the proxy is expected to be similar in forward and reverse caching. Thus if the proxy cache has the same capacity in terms of the number of query hits, and the queries have the same locality, the hit ratios are expected to be comparable.

Finally, consider the case where query hits expire before the cache capacity is reached. Since the cache hits stored in the reverse proxy and in the forward proxy effectively correspond to the same set of queries, i.e., those whose query hits have not expired, the hit ratios in the two caches are again expected to be comparable.

4 Experiments

We implemented both forward and reverse query caching proxies. In the following, we experimentally compare the two proxies in a testbed consisting of eight machines running eight Gnutella servents.

4.1 Experimental Setup

Our testbed consists of a cluster of eight PCs running FreeBSD 4.6, each of which runs a Gnutella servent, configured to allow zero incoming and three outgoing connections. Each servent is passive; it only forwards queries and query hits, but does not initiate any queries. Furthermore, it does not store any files for sharing.

To simplify the setup, instead of using a real router, we configured each servent machine to use the caching proxy machine as the default router. IP forwarding rules are specified on the caching proxy machine such that packets going to port 6346 of any destination will be forwarded to port 6346 of localhost, and all other traffic are forwarded. Thus only outgoing Gnutella connections will be hijacked by the proxy.

We started the experiments with all eight servents at 5:00am EST on May 8, 2004 (after a 30-minute warm-up period), and the experiment lasted for an hour. In all tests, we find the neighboring servents of the eight servents behind the caching proxies to be distinct. The proxy recorded all Gnutella packets going in and out on the hijacked outgoing connections.

4.2 Results

We fixed the total number of query hits the cache can store to be 350000 and compared the caching results under forward and reverse caching. The cache

Table 2. Query and query hit traffic going in and out the cache. The cache stores up to a total of 350000 query hits for different queries.

Category	Forward Caching	Reverse Caching
Time measured (EST)	6:30-7:30am May 8, 2004	5:00-6:00am May 8, 2004
Average # connections/servent	2.70	2.69
Cache Miss Ratio	58.03%	56.07%
Queries:		
rcvd from outside	1503770	1531813
sent to inside	1503770	1220787
with *ttl* > 1	656804	654221
rcvd from inside	1102104	578992
sent to outside	604042	578992
queries dropped at cache	55656	42134
Query Hits:		
rcvd from outside	44356	42814
sent to inside	75938	42814
recv from inside	75938	42814
sent to outside	75938	80175
Average query hit size/query (Bytes)	4693.16	4857.32

replacement policy was LRU. In addition, a 30-minute expiration is imposed for cached query hits. However, in our experiments, the cache capacity was reached first before any query hit expired. The measured statistical results of the two caching proxies are shown in Table 2. Note while the threshold for triggering cache replacement is the total number of cached query hits, the granularity for replacement is a query entry, i.e., all of its query hits.

For the forward cache, 1503770 queries crossed the cache and were forwarded to the servents. Out of these, only 656804 have *ttl* > 1 (or *ttl* > 0 after decrementing) and would be forwarded to the remote servents outside the gateway by the servents inside. The average number of connections per servent was 2.70, averaged over the duration of the experiment. Hence, the total number of queries received by the proxy from inside would be about $656804 * 1.70 = 1116567$. The observed value of 1102104 was within 1.3% of this. Only 58.03% (miss ratio) of these would be sent to outside. The actual number of queries sent outside was slightly lower than this number due to dropped queries by the proxy because of repeated `muid` (Section 2.3).

The number of query hits received from outside was 44356, an additional 31582 were serviced from the cache, for a total of 75938. Notice that the percentage of traffic served from cache, 41.6%, is the saving in bandwidth that the forward cache provides, assuming a constant number of bytes per query hit.

For the reverse cache, 1531813 queries were received from outside of which 654221 have *ttl* > 1. The reverse proxy performed caching on them and the cache miss ratio was 56.07%. The proxy then sent the 366822 misses along with the 877592 queries with *ttl* = 1 to the servents inside, and thus the total number queries sent inside was expected to be 1244414. Due to queries dropped by the cache (becasue of repeated `muid`), the actual number of queries sent inside (1220787) was about 1.9% lower. The average number of connections per servent was 2.69, averaged over the duration of the experiment. Thus the expected number of queries received from the servents inside would be $366822 * 1.69 = 619929$. Again, due to dropped queries by the cache when sending inside, fewer than 366822 queries were sent inside, and the actual number received from inside was 578992.

In terms of query hits, a total of 80175 replies to outside servents were sent out of which 42814 were received from outside, and the rest were served out of the cache. Thus the bandwidth saving for query hits in the reverse cache is 46.6%, assuming a constant number of bytes per query hit.

Comparing forward caching and reverse caching, the cache miss ratios, the numbers of queries sent to outside, and the numbers of query hits received from outside are within 2.0%, 4.1%, 3.5%, respectively. The total numbers of query hits sent outside for forward and reverse caching are within 0.63% of each other. This confirms the analysis that if the cache capacity is in terms of the number of cached query hits, the traffic reduction will be comparable in reverse and forward caching.

5 Related Work

There have been many studies that measured, modeled, or analyzed peer-to-peer file sharing systems such as Gnutella (for example, [6,7]) and Kazaa (for example, [8,9]). Many of these studies also discussed the potential of caching data object files (for example, [8]) or retrieving files from other peers within the same organization [9] in reducing the bandwidth consumption.

Several previous work studied query caching in Gnutella networks. Sripanidkulchai [1] observed that the popularity of query strings follows a Zipf-like distribution, and proposed and evaluated a simple query caching scheme by modifying a Gnutella servent. The caching scheme proposed was fairly simple; it caches query hits solely based on their query strings and ignores TTL values. In [2], Markatos studied one hour of Gnutella traffic traces collected at three servents located in Greece, Norway, and USA, and proposed a query caching scheme by modifying servents to cache query hits according to the query string, the forwarder from which the query is forwarded, and the TTL. In our previous work [3], we proposed transparent forward query caching at the gateway and experimentally showed its effectiveness. This paper builds on top of our previous work and shows that reverse and forward query caching are equally effective in the context of transparent query caching at the gateway.

Several recent work studied other p2p traffic. Leibowitz et al. [10] studied one month of FastTrack-based [11] p2p traffic at a major ISP and found that the majority of p2p files are audio files and the majority of the traffic are due to video and application files. They also reported significant locality in the studied p2p data files. Saroiu et al. [12] studied the breakdowns of Internet traffic going through the gateway of a large organization into web, CDN, and p2p (Gnutella and Kazaa) traffic. They focused on HTTP traffic. In contrast, this paper focuses on the p2p protocol traffic, and compares different transparent caching schemes for query traffic.

6 Conclusions

In this paper, we studied the effectiveness of forward and reverse caching of p2p query traffic at the gateway of an organization or ISP via analysis and experimental measurement. Our study showed that forward caching and reverse caching at the gateway are equally effective in reducing query and query reply traffic across the gateway. Since in a peer-to-peer network the communication are symmetric – query and query hit traffic travel on both incoming and outgoing connections (with respect to the peers inside the gateway), transparent caching will be even more effective if traffic on both type of connections are filtered.

Acknowledgment. This work was supported in part by Cisco Systems through the University Research Program and by Purdue Research Foundation.

References

1. Sripanidkulchai, K.: The popularity of gnutella queries and its implication on scaling. ⟨ http://www-2.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html ⟩ (2001)
2. Markatos, E.P.: Tracing a large-scale peer to peer system: an hour in the life of gnutella. In: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'02). (2002)
3. Patro, S., Hu, Y.C.: Transparent Query Caching in Peer-to-Peer Overlay Networks. In: Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03). (2003)
4. Clip2: The Gnutella protocol specification. ⟨ http://dss.clip2.com/GnutellaProtocol04.pdf ⟩ (2000)
5. Kirk, P.: The Gnutella 0.6 protocol draft. ⟨ http://rfc-gnutella.sourceforge.net/ ⟩ (2003)
6. Adar, E., Huberman, B.: Free riding on gnutella. First Monday **5** (2000)
7. Saroiu, S., Gummadi, P., Gribble, S.: A measurement study of peer-to-peer file sharing systems. In: Proceedings of Multimedia Computing and Networking (MMCN'02). (2002)
8. Saroiu, S., Gummadi, K.P., Dunn, R.J., Gribble, S.D., Levy, H.M.: An analysis of internet content delivery systems. In: Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI'02). (2002)
9. Gummadi, K.P., Dunn, R.J., Saroiu, S., Gribble, S.D., Levy, H.M., Zahorjan, J.: Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP'03). (2003)

10. Leibowitz, N., Bergman, A., Ben-Shaul, R., Shavit, A.: Are file swapping networks cacheable? Characterizing p2p traffic. In: Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW7). (2002)
11. Truelove, K., Chasin, A.: Morpheus out of the underworld. The O'Rielly Network, ⟨ http://www.openp2p.com/pub/a/p2p/2001/07/02/morpheus.html ⟩ (2001)
12. Saroiu, S., Gummadi, P.K., Dunn, R.J., Gribble, S.D., , Levy, H.M.: An analysis of internet content delivery systems. In: Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI'02). (2002)

FatNemo: Building a Resilient Multi-source Multicast Fat-Tree

Stefan Birrer, Dong Lu, Fabián E. Bustamante, Yi Qiao, and Peter Dinda

Northwestern University, Evanston IL 60201, USA,
{sbirrer,donglu,fabianb,yqiao,pdinda}@cs.northwestern.edu

Abstract. This paper proposes the idea of emulating fat-trees in overlays for multi-source multicast applications. Fat-trees are like real trees in that their branches become thicker the closer one gets to the root, thus overcoming the "root bottleneck" of regular trees. We introduce FatNemo, a novel overlay multi-source multicast protocol based on this idea. FatNemo organizes its members into a tree of clusters with cluster sizes increasing closer to the root. It uses bandwidth capacity to decide the highest layer in which a peer can participate, and relies on co-leaders to share the forwarding responsibility and to increase the tree's resilience to path and node failures.
We present the design of FatNemo and show simulation-based experimental results comparing its performance with that of three alternative protocols (Narada, Nice and Nice-PRM). These initial results show that FatNemo not only minimizes the average and standard deviation of response time, but also handles end host failures gracefully with minimum performance penalty.

1 Introduction

High bandwidth multi-source multicast among widely distributed nodes is a critical capability for a wide range of important applications including audio and video conferencing, multi-party games and content distribution. Throughout the last decade, a number of research projects have explored the use of multicast as an efficient and scalable mechanism to support such group communication applications. Multicast decouples the size of the receiver set from the amount of state kept at any single node and potentially avoids redundant communication in the network.

The limited deployment of IP Multicast [16,17], a best effort network layer multicast protocol, has led to considerable interest in alternate approaches that are implemented at the application layer, using only end-systems [14,24,19,32,11,2,10,33,39,31,35]. In an end-system multicast approach participating peers organize themselves into an overlay topology for data delivery. Each edge in this topology corresponds to a unicast path between two end-systems or peers in the underlying Internet. All multicast-related functionality is implemented at the peers instead of at routers, and the goal of the multicast protocol is to construct and maintain an efficient overlay for data transmission.

Among the end-system multicast protocols proposed, tree-based systems have proven to be highly scalable and efficient in terms of physical link stress, state and control overhead, and end-to-end latency. However, normal tree structures have two inherent problems:

Resilience: They are highly dependent on the reliability of non-leaf nodes. Resilience is particularly relevant to the application-layer approach, as trees here are composed of autonomous, unpredictable end systems. The high degree of transiency of the hosts [1] has been pointed out as one of the main challenges for these architectures [4].

Bandwidth limitations: They are likely to be bandwidth constrained [2] as bandwidth availability monotonically decreases as one descends into the tree. The bandwidth limitations of normal tree structures is particularly problematic for multi-source, bandwidth intensive applications. For a set of randomly placed sources in a tree, higher level paths (those closer to the root) will become the bottleneck and tend to dominate response times. Once these links become heavily loaded or overloaded, packets will start to be buffered or dropped.

We have addressed the resilience issue of tree-based systems in previous work [5] through the introduction of *co-leaders* and the reliance on *triggered negative acknowledgements* (NACKs). In this paper, we address the bandwidth limitations of normal tree overlays.

Our approach capitalizes on Leiserson's seminal work on fat-trees [27]. Paraphrasing Leiserson, a fat-tree is like a real tree in that its branches become thicker the closer we get to the root, thus overcoming the "root bottleneck" of a regular tree. Figure 1 shows a schematic example of a binary fat-tree. We propose to organize participant end-systems in a tree that closely resembles a Leiserson fat-tree by dynamically placing higher degree nodes (nodes with higher bandwidth capacity) close to the root and increasing the cluster sizes as one ascends the tree.

This paper makes three main contributions. First, we introduce the use of Leiserson fat-trees for application-layer multi-source multicast, overcoming the inherent bandwidth limitations of normal tree-based overlay structures (Section 2). Second, after reviewing some background material in Section 3, we describe the design and implementation of *FatNemo,* a new application-layer multicast protocol that builds on this idea (Section 4). Lastly, we evaluate the FatNemo design in simulation, illustrating the benefits of a fat tree approach compared to currently popular approaches to application-layer multicast (Sections 5 and 6). We review some related work in Section 7 and present our conclusion and directions of future work in Section 8.

2 Fat-Trees and the Overlay

The communication network of a parallel machine must support global collective communication operations in which all processors participate [26]. These operations have a wide range, including reduction and broadcast trees, and neighbor exchange. All-to-all personalized communication [21], in which each processor sends a unique message to every other processor, is key to many algorithms. To support such operations well, the network should have (1) minimal and scalable diameter, and (2) maximal and scalable

[1] Measurement studies of widely used application-layer/peer-to-peer systems have reported median session times ranging from an hour to a minute [7,20,34,12].

[2] The access link of an end system becomes its bandwidth bottleneck, thus we can model the bandwidth capacity as a property of the end-system.

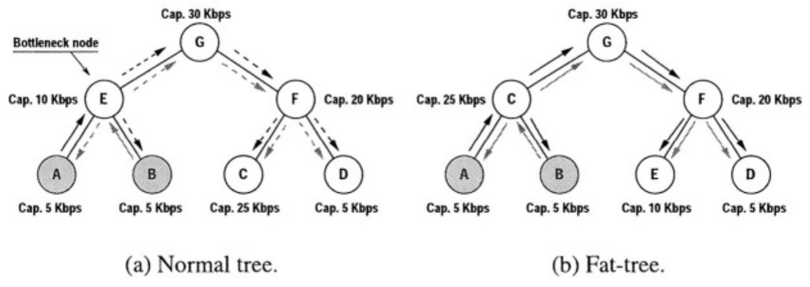


Fig. 1. Two binary trees with nodes *A* and *B* as sources, publishing at 5 Kbps each. On the left, a normal binary tree where node *E* becomes the bottleneck, resulting on a reduced (dash line) outgoing stream quality. Node *E* has to forward the stream *A* to node *B* and node *G,* as well as stream *B* to node *A* and node *G,* thus it needs an outgoing bandwidth capacity of 20 Kbps. However, it has only 10 Kbps available, making it a bottleneck in the tree. On the right, a fat-tree with higher capacity nodes placed higher in the tree.

bisection bandwidth. These goals are very similar to those of a multisource multicast overlay, which can be thought of as providing many-to-many personalized communication, a subset of all-to-all personalized communication. We expect a multisource multicast overlay to have a latency between end-systems that grows only slowly or not at all as the number of end-systems grows. Similarly, we expect the aggregate throughput of the overlay to grow linearly as end-systems are added.

In his seminal work on fat-trees [27], Leiserson introduced a universal routing network for interconnecting the processors of a parallel supercomputer, where communication can be scaled independently of the number of processors. Based on a complete binary tree, a fat-tree consists of a set of processors, located at the leaves, and interconnected by a number of switching nodes (internal to the tree) and edges. Each edge in the tree corresponds to two unidirectional *channels* connecting a parent with each of its children. Channel consist of a bundle of *wires,* and the number of wires in a channel is called its *capacity.* The capacity of the channels of a fat-tree grows as one goes up the tree from the leaves, thus allowing it to overcome the “root bottleneck” of a regular tree. Since their introduction, fat-trees have been successfully applied in massively parallel systems [28,22] as well as in high performance cluster computing [23].

In this paper we propose to organize the end-systems participant in a multicast group, in an overlay tree that closely resembles a Leiserson fat-tree. In common with Leiserson, our goal is to minimize the mean and standard deviation of inter-node communication performance with multiple potential sources.

Emulating fat-trees in an overlay entails a number of challenges such as handling the high level of transiency of end-system populations and addressing their degree of heterogeneity. A straightforward way of approximating a fat-tree is placing those nodes with higher bandwidth capacity[3] closer to the root. Since interior nodes are involved in most inter-node communication paths they strongly influence the overall end-to-end delay and can soon become bottlenecks as one increases the number of sources. Figure 1 shows a schematic example of both a regular binary tree with two 5 Kbps sources *(A* and *B)* and a bottleneck node *(E)* unable to keep up with the publishing rate of the nodes downstream.

Available bandwidth can differ significantly from bandwidth capacity over time, due typically to competing traffic, and any algorithm that attempts to emulate fat-trees in an overlay needs to take account of such dynamism. Also, per-path characteristics must be taken into consideration. Since end-to-end latency is an important factor in the performance of interactive applications, the latency of each link in the overlay, the processing time at each node, and the number of intermediate nodes should be considered carefully. When selecting among possible parents, a closer node may be a better candidate, if it is able to support the load, than an alternative node offering higher available bandwidth. Finally, the mean time to failure of end-systems is significantly lower than for routers, possibly resulting in long interruptions to the data stream as the failure is discovered and the distribution tree repaired.

Although overlay fat-trees can be built above most tree-based multicast systems, in this paper we consider their implementation using Nemo, a high-resilience, latency-optimized overlay multicast protocol [5]. The following section provides some background material on overlay multicast in general and on the operational details of Nemo, before describing the FatNemo design in Section 4.

3 Background

All peer-to-peer or application-layer multicast protocols organize the participating peers into (1) a control topology for group membership related tasks, and (2) a delivery tree for data forwarding. Available protocols can be classified according to the sequence adopted for their construction [1,13]. In a tree-first approach [19,24,32], peers directly construct the data delivery tree by selecting their parents from among known peers. Additional links are later added to define, in combination with the data delivery tree, the control topology. With a mesh-first approach [13,11], peers build a more densely connected graph (mesh) over which (reverse) shortest path spanning trees, rooted at any peer, can be constructed. Protocols adopting an implicit approach [2,10,33,39,35] create only a control topology among the participant peers. Their data delivery topology is implicitly determined by the defined set of packet-forwarding rules.

FatNemo builds on Nemo to emulate a fat-tree; thus, it inherits the latter's high scalability and resilience. In the following paragraphs, we provide a summarized description of Nemo; for more complete details we direct the reader to our previous work [5].

Nemo

Nemo follows the implicit approach to building an overlay for multicasting. The set of communication peers are organized into clusters based on network proximity, where

[3] The maximum outgoing bandwidth that the node is capable of, the capacity of the IP link attaching the node to the network.

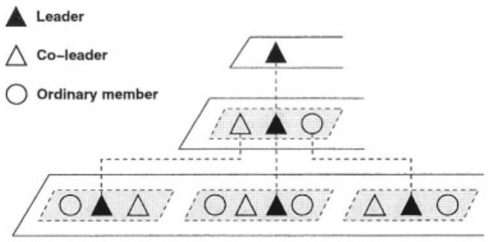


Fig. 2. Nemo's logical organization. The shape illustrates only the role of a peer within a cluster: a leader of a cluster at a given layer can act as leader, co-leader, or an ordinary member at the next higher layer.

every peer is a member of a cluster at the lowest layer. Clusters vary in size between d and $3d - 1$, where d is a constant known as the *degree.* Each of these clusters selects a *leader* that becomes a member of the immediately superior layer. In part to avoid the dependency on a single node, every cluster leader recruits a number of co-leaders to form a supporting crew. The process is repeated at each new layer, with all peers in a layer being grouped into clusters, crew members selected, and leaders promoted to participate in the next higher layer. Hence peers can lead more than one cluster in successive layers of this logical hierarchy. Figure 2 illustrates the logical organization of Nemo.

A new peer joins the multicast group by querying a well-known special end-system, the rendezvous point, for the identifier of the root node. Starting there and in an iterative manner, the incoming peer continues: *(i)* requesting the list of members at the current layer from the cluster's leader, *(ii)* selecting from among them who to contact next based on the result from a given cost function, and *(iii)* moving into the next layer. When the new peer finds the leader with minimal cost at the bottom layer, it joins the associated cluster.

Member peers can leave Nemo in a graceful manner (e.g. user disconnects) or in an ungraceful manner (unannounced, e.g. when the end-system crashes). For graceful departures, since a common member has no responsibilities towards other peers, it can simply leave the group after informing its cluster's leader. On the other hand, a leader must first elect replacement leaders for all clusters it owns before it leaves the session.

To detect unannounced departures, Nemo relies on heartbeats exchanged among the cluster's peers. Unreported members are given a fixed time interval before being considered dead, at which point a repair algorithm is initiated. If the failed peer happens to be a leader, the tree itself must be fixed, the members of the victim's cluster must elect the replacement leader from among themselves.

To deal with dynamic changes in the underlying network, every peer periodically checks the leaders of the next higher layers and switches clusters if another leader has a lower cost (i.e. lower latency) than the current one. Additionally, in a continuous process of refinement, every leader checks its highest owned cluster for better suited leaders and transfers leadership if such a peer exists.

Nemo addresses the resilience issue of tree-based systems through the introduction of co-leaders. Co-leaders improve the resilience of the multicast group by avoiding depen-

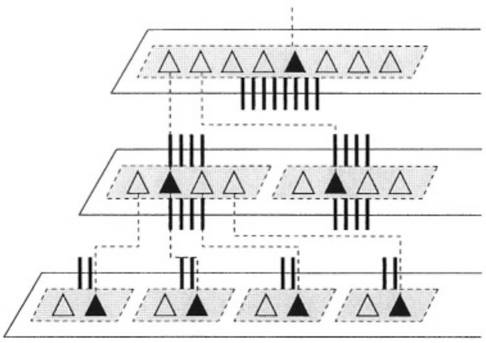


Fig. 3. FatNemo's Topology. The figure illustrates how the tree gets fatter when moving toward the root. This tree has a cluster degree of 2.

dencies on single nodes and providing alternative paths for data forwarding. In addition, crew members share the load from message forwarding, thus improving scalability.

4 FatNemo Design

To build an overlay fat-tree, FatNemo relies on three heuristics: (1) higher bandwidth degree nodes should be placed higher up in the tree, (2) all peers must serve as crew members in order to maximize load balancing, and (3) the size of clusters should increase exponentially as one ascends the tree. The following paragraphs provide the motivations behind each of these heuristics.

The per-node bandwidth constraint is critical for bandwidth-demanding applications and can be stated as the number of full-rate streams a peer is able to support, i.e. its *out-degree*. By organizing peers based on their out-degrees [36,13], we intend to reduce the bandwidth constraints of links higher up the tree. Since the process of estimating available bandwidth is time consuming, peers initially join the tree based on proximity. Once in the tree, every leader checks its highest owned cluster for better suited leaders in terms of bandwidth availability, and transfers leadership if such a peer exists. This process assures that high out-degree peers will gradually ascend to higher layers, thus transforming the tree into a bandwidth optimized fat-tree.

In traditional fat-trees the number of wires increases as one ascends the tree. Considering an overlay unicast connection as a "wire", the number of wires in FatNemo increases together with the crew size as one moves toward the root – the maximum possible number of wires is thus achieved by setting the crew size equal to the cluster size. The size of a cluster at layer i in FatNemo varies between d_i and $2d_i + 2$, and grows exponentially ($d_i = d_0^{i+1}$) as we move up the layers. The 0-th layer contains the leaf nodes and has a degree d_0 of 3 (same as Nice and Nemo). The increased number of wires helps avoid higher level links from becoming the bottleneck of the system, as alternate paths share the load and reduce the forward responsibility of each peer.

Table 1. Cluster and Crew Size as a function of the cluster degree d, for a 20,000 peer population. The variable x is a place holder for the cluster index starting at 0 for the lowest layer.

Protocol	Cluster Size $k(x)$		Crew Size $c(x)$
Nice	$d \ldots 3d-1$	$d=3:3\ldots8$	1
Nemo	$d \ldots 3d-1$	$d=3:3\ldots8$	3
FatNemo	$d^{x+1} \ldots 2d^{x+1}+2$	$d=3, x=1:9\ldots20$	$k(x)$

Beyond increasing the number of wires, large crew sizes also help reduce the depth of a tree (when compared with its constant cluster-sizes equivalent). A smaller depths means a lower total number of end-system hops, and should translate in a reduction on the end-to-end delay.

Figure 3 illustrates how FatNemo constructs a fat-tree. In this simple example $d_0 = 2$, so clusters scale up by a factor of 2 as we ascend the tree. Notice that these links/wires are indeed $(d_{i+1} \ldots 2d_{i+1} + 2)$ to $(d_i \ldots 2d_i + 2)$ relations, as every crew member of the next higher layer will talk to the crew members of the immediately lower layer. For clarity in the graph, this set of links is represented in the graph by d_i lines.

To better understand the positive effect of FatNemo's heuristics, we show an instantiation of them with a population of 20,000 peers from a popular on-line game. [4]

To begin, let's generalize the concept of *out-degree*. The out-degree of a peer, d_{out}, is equivalent to the total forwarding responsibility of a node, and it can be stated as a function of the number of layers L in which the node participates, the cluster size $k(x)$ and the crew size $c(x)$ at layer x (Equation (1)). Table 1 illustrates the parameter values for FatNemo and two alternative protocols.

$$d_{out} = \sum_{x=0}^{L-1} \frac{k(x)-1}{c(x)} \tag{1}$$

Now, to calculate the requirements for the root node, we must first estimate the number of layers for a given protocol which is a function of the number of peers in a cluster[5]. Nice and Nemo have, in expectation, 5.5 nodes per cluster at every layer. Using this value as an approximation for the cluster size, a traditional tree for this population size will be about 7 layers in depth. FatNemo, on the other hand, has a variable expected number of nodes per cluster, and its expected tree depth is 4 layers.

Based on the expected depth of the different trees for this example population, we can now calculate the out-degree requirements on their root nodes. According to the generalized out-degree equation introduced in the previous paragraph (Equation (1)), Nice requires a root out-degree of 31.5, or almost three times more than what is needed from a Nemo's root (10.5) with a crew size of 3. In other words, the root of a traditional

[4] The number corresponds to the active populations of players in *hattrick.org,* an online soccer game.

[5] The average number of peers in a cluster is equal to the mean cluster size, which can be computed as the mean of the low and high cluster boundary.

tree for a 20,000 peer population must support 31.5 times the source rate to fulfill its forwarding responsibility! By emulating a fat-tree in the overlay, FatNemo avoids this "root bottleneck" requiring only a root with an out-degree of 3.7.

5 Evaluation

We analyze the performance of FatNemo through simulation and compare it to that of three other protocols – Narada [14], Nice [2] and Nice-PRM [3]. We evaluate the effectiveness of the alternative protocols in terms of performance improvements to the application and protocol's overhead, as captured by the following metrics:

Response Time: End-to-end delay (including retransmission time) from the source to the receivers, as seen by the application. This includes path latencies along the overlay hops, as well as queueing delay and processing overhead at peers along the path. A lower mean response time indicates a higher system responsiveness, and a smaller standard deviation implies better synchronization among the receivers.

Delivered Packets: Number of packets successfully delivered to all subscribers within a fixed time window. It indirectly measures the protocol's ability to avoid bottlenecks in the delivery tree.

Delivery Ratio: Ratio of subscribers that have received a packet within a fixed time window. Disabled receivers are not accounted for.

Duplicate Packets: Number of duplicate packets per sequence number, for all enabled receivers, reflecting an unnecessary burden on the network. Packets arrived outside of the delivery window are accounted for as duplicates, since the receiver already assumed them as lost.

Control-Related Traffic: Total control traffic in the system, in mega bits per second (Mbps); part of the system's overhead. We measure the total traffic during the observation interval by accounting packets at the router level. A network packet traversing four routers (including the source and destination node) will account as three sent packets, one for every router which has to forward it.

The remainder of this section briefly discusses implementation details of the compared protocols and describes our evaluation setup. Section 6 presents our evaluation results.

5.1 Details on Protocol Implementations

For each of the three alternative protocols, the values for the available parameters were obtained from the corresponding literature [13,2,3].

For Narada [13], we employ the bandwidth-only scheme for constructing the overlay, as this will result in maximum throughput. For Nice [2] and Nice-PRM [3], the cluster degree, k, is set to 3. We use Nice-PRM(3,0.02) with three random peers chosen by each node, and with two percent forwarding probability.

For FatNemo, the cluster degree at the lowest layer is set to three. Cluster degree grows exponentially with every layer, being nine in the second lowest layer, 27 in the third, and so on.

Our implementations of the alternative protocols closely follow the descriptions from the literature, and have been validated by contrasting our results with the published values. However, there are a number of improvements to the common algorithms, such as the use of periodic probabilistic maintenance operations, that while part of FatNemo, were made available to all protocols in our evaluations. The benefits from these algorithms help explain the performance improvements of the different protocols with respect to their original publications [13,2,3].

5.2 Experimental Setup

We performed our evaluations through detailed simulation using SPANS, a locally written, packet-level, event-based simulator. Simulations were run using GridG [29,30] topologies with 5510, 6312 and 8115 nodes, and a multicast group of 256 members. GridG leverages Tiers [18,8] to generate a three-tier hierarchical network structure, before it applies a power law enforcing algorithm that retains the hierarchical structure.

Members were randomly associated with end systems, and a random delay of between 0.1 and 80ms was assigned to every link. The links use drop-tail queues with a buffer capacity of 0.5 sec. We configured GridG to use different bandwidth distributions for different link types [25]. We assume that the core of the Internet has higher bandwidth capacities than the edge, as shown in Fig. 4. In all three scenarios, the bandwidth has a uniform distribution with ranges shown in Fig. 4.

Scenario	Routers	End systems	Links	Client-Stub	Stub-Stub	Transit-Stub	Transit-Transit
Low-B/W	510	5000	11240	400-6000	3000-8000	4000-10000	10000-20000
Medium-B/W	312	6000	12730	800-8000	4000-10000	6000-15000	15000-30000
High-B/W	615	7500	16450	1000-15000	10000-30000	10000-50000	50000-100000

Fig. 4. Three simulation scenarios: Low-, Medium- and High-Bandwidth. Bandwidth is expressed in Kbps.

Each simulation experiment lasted for 500 sec. (simulation time). All peers join the multicast group by contacting the rendezvous point at uniformly distributed, random times within the first 100 sec. of the simulation. A warm-up time of 200 sec. is omitted from the figures. Publishers join the network and start publishing at the beginning of the simulation. Starting at 200 sec. and lasting for about 300 sec., each simulation has aphase with membership changes. We exercise each protocol with and without host failures during this phase. Failure rates are set based on those obtained from a published report of field failures for networked systems [37]. Nodes fail independently at a time sampled from an exponential distribution (with *mean time to failure* (MTTF) equal to 60min.) and rejoin shortly after (time sampled from an exponential distribution with *mean time to repair* (MTTR) equal to 10 min.). The two means were chosen asymmetrically to allow, on average, 6/7 of all members to be up during this phase. The failure event sequence was generated a priori based on the above distribution and used for all protocols and all runs.

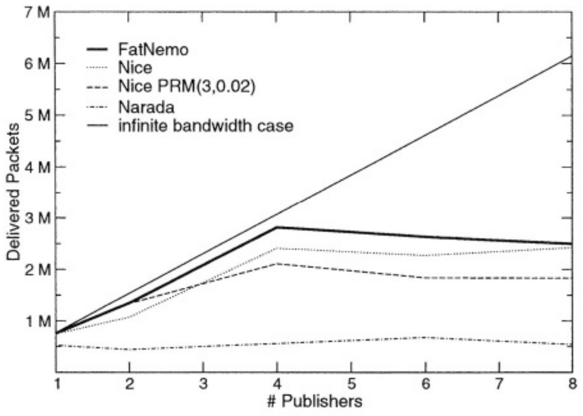


Fig. 5. Delivered packets (256 end hosts, Low-Bandwidth scenario).

In all experiments, we model multi-source multicast streams to a group. Each source sends constant bit rate (CBR) traffic of 1000 Byte payload at a rate of 10 packets per second. The buffer size was set to 16 packets, which corresponds to the usage of a 1.6-second buffer, a realistic scenario for applications such as video conferencing.

6 Experimental Results

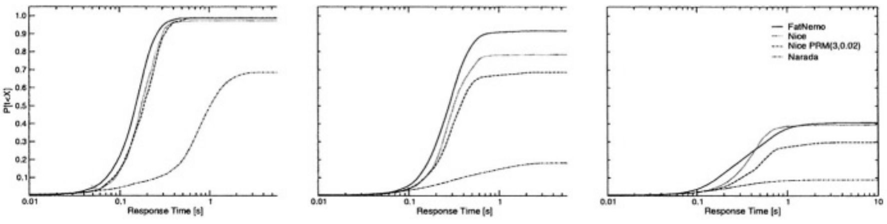
This section presents early evaluation results of FatNemo and compares them with those of three alternative protocols. The reported results are from five runs per protocol obtained with the different GridG topologies and the Low-Bandwidth scenario. Similar results were obtained with the Mid- and High-Bandwidth scenarios.

Figure 5 shows the average number of delivered packets of all runs with no host failures. As we increase the number of publishers, the protocol's data delivery topology collapses. This happens first for Narada, which is unable to handle the full publishing rate from one publisher. Nice and Nice PRM handle an increasing number of publishers better; however, they deliver substantially fewer packets when compared with FatNemo. FatNemo is best at avoiding bottlenecks in the delivery tree, delivering the most packets when the network is overloaded (as seen with 8 publishers).

The performance of a multi-source multicast system can be measured in terms of mean and standard deviation of the response time. Table 2 shows these two metrics for the evaluated protocols with one publisher. FatNemo outperforms Nice, Nice PRM and Narada in terms of mean and standard deviation of response time. With an increased number of publishers the relative number of delivered packets for Nice, Nice PRM and Narada decreases compared to FatNemo, which makes it impossible to fairly compare the response time for more than one publisher based only on one number. The problem stems from that fact that, when lowering its delivery ratio a protocol will drop those

Table 2. Response Time (1 Publisher, 256 end hosts, Low-Bandwidth scenario).

Protocol	Mean	Std
FatNemo	0.158	0.073
Nice	0.183	0.082
Nice-PRM(3,0.02)	0.195	0.086
Narada	0.770	0.464



(a) 1 Publisher (b) 4 Publisher (c) 8 Publisher

Fig. 6. Response Time CDF with 1,4 and 8 publishers (256 end hosts, Low-Bandwidth scenario).

packets with high response time more likely than others. Thus, comparing response times across protocols with significantly different delivery ratios under stress will give less resilient protocols an unfair advantage.

Figure 6.(a) shows the Cumulative Distribution Function (CDF) of the response time per packet for one publisher. The y-**axis** is normalized to the infinite bandwidth case, i.e. when all receivers receive all possible packets. FatNemo, Nice and Nice PRM perform well, but FatNemo's flatter tree results in an improved response time. Narada is only able to deliver a fraction of all possible packets, and only then with a substantially high delay. With increasing number of publishers, the protocols start running into bottlenecks. Despite the harder conditions, FatNemo is able to outperforms the alternative protocols in terms of packet delivery times as illustrated in Fig. 6.(b) and Fig. 6.(c).

Table 3 shows the delivery ratio using one publisher with and without end-systems failures. We see that FatNemo performs as well as Nice PRM under a low-failure rate scenario with only a drop of 2.1% in delivery ratio. Nice has a slightly lower delivery

Table 3. Delivery Ratio (1 Publisher, 256 end hosts, Low-Bandwidth scenario).

Protocol	No Failures	With Failures
FatNemo	0.987	0.966
Nice	0.973	0.956
Nice-PRM(3,0.02)	0.989	0.970
Narada	0.685	0.648

Table 4. Overhead (1 Publisher, 256 end hosts, Low-Bandwidth scenario).

Protocol	Duplicate packets	Control Traffic [Mbps]
FatNemo	0.367	17.99
Nice	0.000	9.211
Nice-PRM(3,0.02)	5.168	11.48
Narada	0.006	157.3

ratio, while Narada suffers already from a collapsed delivery tree with only about 70% delivery ratio. In general, the delivery ratio will decrease as the number of publishers increases, as the protocol's data delivery topology slowly collapses.

The overhead of a protocol can be measured in terms of duplicate packets. We show this metric in the second column of Table 4. Despite its high delivery ratio, FatNemo incurs, in average, only 0.367 duplicate packets per sequence number, while Nice-PRM suffers from 5.168 duplicate packets per sequence number generated by its probabilistic forwarding algorithm. Nice and Narada feature almost no duplicates, but at a high cost in term of delivery ratio as shown in Table 3. FatNemo's control related traffic is higher than for Nice and Nice-PRM, a result of its larger cluster cardinality higher in the tree. The control traffic is accounted for at router level, thus the choice of a peer's neighbors in FatNemo also adds additional overhead, as it opts not for the closest, but for the peer with highest bandwidth.

7 Related Work

Numerous protocols have been proposed to address the demand for live streaming applications. One of the first end-system multicast protocol was Narada [13], a multisource multicast system designed for small to medium sized multicast groups. Peers in Narada are organized into a mesh with fixed out-degree, with every peer monitoring all others to detect end-system failures and network partitions. The per-source multicast tree is built on top of this mesh from the reverse shortest path between each recipient and the source. Since the tree construction algorithm does not account for cross traffic, a powerful link is likely to be used by many multicast links, limiting the efficiency of the multicast system. FatNemo uses crew members to share the forwarding load, thus relaxing the burden on a single high bandwidth path. Overcast [24] organizes dedicated servers in a single-source, bandwidth optimized, multicast tree. In contrast, FatNemo is an end-system overlay that constructs a global optimized fat-tree for multisource multicasting. Banerjee et al. [2] introduce Nice and demonstrate the effectiveness of overlay multicast across large scale networks. The authors also present the first look at the robustness of alternative overlay multicast protocols under group membership changes. FatNemo adopts the same implicit approach, and its design draws on a number of ideas from Nice such as its hierarchical control topology. FatNemo introduces co-leaders to improve the resilience of the overlay and adopts a periodic probabilistic approach to reduce/avoid the cost of membership operations.

A new set of projects have started to address the resilience of overlay multicast protocols [3,9,35,5,31,38]. ZigZag [35], a single-source protocol, explores the idea of splitting the control and data delivery task between two peers in each level, making both responsible for repairs under failures. With PRM [3] Banerjee et al. propose the use of probabilistic forwarding and NACK-based retransmission to improve resilience. In order to reduce the time-to-repair, Yang and Fei [38] argue for proactively, ahead of failures, selecting parent replacements. CoopNet [31] improves resilience by building several disjoint trees on a centralized organization protocol and employing Multiple Description Coding (MDC) for data redundancy. Nemo [5] and FatNemo build redundancy into the overlay through co-leaders; different from the previously described protocols, they make all crew members share forwarding responsibilities while all cluster members are in charge of repair operations. These simple measures enable an uninterrupted service to downstream peers especially during recovery intervals. We are currently exploring the use of data redundancy using forward error correction (FEC) encoding [6].

Aiming at bulk data distribution, protocols such as Splitstream [9] , Bittorrent [15] and Bullet [25] have proposed simultaneous data streaming over different paths to better share the forwarding load and increased downloading capacity. The methods differ in how they locate alternate streaming peers. In comparison, FatNemo exploits alternate paths for resilience and load balancing.

8 Conclusions and Further Work

In this paper we introduced the parallel architecture concept of fat trees to overlay multicast protocols. We have described FatNemo, a novel scalable peer-to-peer multicast protocol that incorporates this idea to build data delivery topologies with minimized mean and standard deviation of the response time. Simulation results show that FatNemo can achieve significantly higher delivery ratios than alternative protocols (an increase of up to 360% under high load), while reducing the mean (by up to 80%) and standard deviation (by up to 84%) of the response time in the non-overloaded case. Under a heavy load and a realistic host failure rate, the resulting protocol is able to attain high delivery ratios with negligible cost in terms of control-related traffic. We are currently validating our findings through wide-area experimentation.

Acknowledgments. We are grateful to Janine M. Casler, Kevin Livingston and Ananth Sundararaj for their helpful comments on early drafts of this paper.

References

1. S. Banerjee and B. Bhattacharjee. A comparative study of application layer multicast protocols, 2002. Submitted for review.
2. S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM,* August 2002.
3. S. Banerjee, S. Lee, B. Bhattacharjee, and A. Srinivasan. Resilient multicast using overlays. In *Proc. of ACM SIGMETRICS,* June 2003.

4. M. Bawa, H. Deshpande, and H. Garcia-Molina. Transience of peers & streaming media. In *Proc. of HotNets-I,* October 2002.
5. S. Birrer and F. E. Bustamante. Nemo - resilient peer-to-peer multicast without the cost. Tech. Report NWU-CS-04-36, Northwestern U., April 2004.
6. R. E. Blahut. *Theory and Practice ofError Control Codes.* Addison Wesley, 1994.
7. F. E. Bustamante and Y. Qiao. Friendships that last: Peer lifespan and its role in P2P protocols. In *Proc. of IWCW,* October 2003.
8. K. L. Calvert, M. B. Doar, and E. W. Zegura. Modeling internet topology. *IEEE Communications Magazine,* 35(6):160–163, June 1997.
9. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Proc. of the 19th ACM SOSP,* October 2003.
10. M. Castro, A. Rowstron, A.-M. Kermarrec, and P. Druschel. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication,* 20(8), 2002.
11. Y. Chawathe. *Scattercast: an architecturefor Internet broadcast distribution as an infrastructure service.* Ph.D. Thesis, U. of California, Berkeley, CA, Fall 2000.
12. Y.-H. Chu, A. Ganjam, T. S. E. Ng, S. G. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early experience with an Internet broadcast system based on overlay multicast. In *Proc. of USENIX ATC,* June 2004.
13. Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communication,* 20(8), October 2002.
14. Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proc. ofACM SIGMETRICS,* June 2000.
15. B. Cohen. BitTorrent. bitconjurer.org/BitTorrent/, 2001. File distribution.
16. S. E. Deering. Multicast routing in internetworks and extended LANs. In *Proc. ofACM SIGCOMM,* August 1988.
17. C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network,* 14(1), January/February 2000.
18. M. B. Doar. A better model for generating test networks. In *Proc. of Globecom,* November 1996.
19. P. Francis. Yoid: Extending the Internet multicast architecture. http://www.aciri.org/yoid, April 2000.
20. K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling and analysis of a peer-to-peer file-sharing workload. In *Proc. of ACM SOSP,* December 2003.
21. S. Hinrichs, C. Kosak, D. O'Hallaron, T. Strieker, and R. Take. An architecture for optimimal all-to-all personalized communication. In *Proceedings ofthe 6th ACM Symposium on Parallel Algorithms and Architectures (SPAA),* pages 310–319, 1994.
22. M. Homewood and M. McLaren. Meiko CS-2 interconnect elan – elite design. In *IEEE Hot Interconnects Symposium,* August 1993.
23. InfiniBand Trade Association. Infiniband architecture specification (1.0.a). www.infinibandta.com, June 2001.
24. J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole Jr. Overcast: Reliable multicasting with and overlay network. In *Proc. ofthe 4th USENIX OSDI,* October 2000.
25. D. Kostić, A. R. adn Jeannie Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. of the 19th ACM SOSP,* October 2003.
26. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

27. C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers,* 34(10): 892–901, October 1985.
28. C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing,* 33(2):145–158, 1996.
29. D. Lu and P. A. Dinda. GridG: Generating realistic computational grids. *ACM Sigmetrics Performance Evaluation Review,* 30(4):33–41, March 2003.
30. D. Lu and P. A. Dinda. Synthesizing realistic computational grids. In *Proc. of SC2003,* November 2003.
31. V. N. Padmanabhan, H. J. Wang, and P. A. Chou. Resilient peer-to-peer streaming. In *Proc. of IEEE ICNP,* 2003.
32. D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proc. of USENIX USITS,* March 2001.
33. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proc. ofNGC,* November 2001.
34. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of USENIXATC,* December 2004.
35. D. A. Tran, K. A. Hua, and T. Do. ZIGZAG: An efficient peer-to-peer scheme for media streaming. In Proc. *ofIEEE INFOCOM,* April 2003.
36. Z. Wang and J. Crowcroft. Bandwidth-delay based routing algorithms. In *Proc. ofIEEE GlobeCom,* November 1995.
37. J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT system field failure data analysis. In *Proc. ofPRDC,* December 1999.
38. M. Yang and Z. Fei. A proactive approach to reconstructing overlay multicast trees. In *Proc. ofIEEE INFOCOM,* March 2004.
39. S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. ofNOSSDAV,* June 2001.

A Real-Time Selection Method of an Acceptable Transcoding Path in a MPEG21 DIA for Mobile Terminals*

Sungmi Chon[1] and Younghwan Lim[2]

[1] Soongsil Computer Science Institute,
Seoul, Korea
smchon@hananet.net

[2] School of Media, Soongsil University,
Seoul, Korea
yhlim@computing.ssu.ac.kr

Abstract. In order to access the multimedia contents in a server via mobile devices, the contents should be transcoded and streamed into the mobile device according to the system environment and user preferences. In this paper, we propose a method of finding a sequence of multiple unit transcoders called a transcoding path that can be utilized in MPEG21 digital item adaptation for resource adaptation engine based on the theory of a Context Free Grammar.

1 Introduction

Digital item adaptation (DIA) is one of main parts in MPEG21. The goal of the DIA is to achieve interoperable transparent access to multimedia contents by shielding users from network and terminal installation, management and implementation issues. As shown in Fig.1, the combination of resource adaptation and descriptor adaptation produces newly adapted Digital Item. DIA tools store all the necessary data of descriptor information[1]. There could be a variety of Digital Item that need to be managed in the DIA framework. Depending on the characteristics of digital item, the architecture or implementation method of Resource Adaptation Engine(RAE) may be different. Current research works for RAE are focused on the transcoding of single medium transcoder such as format transcoder, size transcoder, frame rate transcoders, MPEG2 into H.263 transcoder, or 3D stereoscope video into 2D video transcoder[2] etc. Also they provide a fixed single transcoder for each case of transcoding need.

One critical problem of these approaches is that a finite set of fixed transcoders cannot meet all the transcoding requirements because adaptation requirements at terminal side are changing always. Our proposed idea is that the RAE has a set of unit transcoder whose functions are limited to one transcoding function such frame rate transcoding, or size transcoding etc.

* This work was supported by the Soongsil University Research Fund

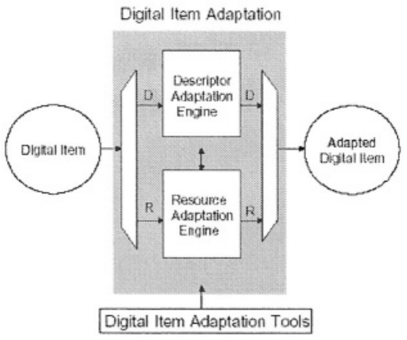

Digital Item Adaptation
Descriptor Adaptation Engine
Resource Adaptation Engine
D
D
R
R
Digital Item
Adapted Digital Item
Digital Item Adaptation Tools

Fig. 1. Illustration of DIA

In addition, the RAE has a transcoding path generation mechanism which finds an appropriate sequence of necessary unit transcoders by analyzing the resource of input Digital Item (called source contents) and the adaptation requirement at the terminal side. The separation of the set of transcoders and adaptation requirements provides a complete functionality of transcoding and flexibility of implementation. The transcoding operation in proposed RAE consists of two steps, transcoding path generation step and transcoding path application step. Since most multimedia Digital Item consist of multiple streams such text, image, audio, video etc, a series of multiple transcoders should be applied in order to transcode the source data into the destination data which satisfies the QoS of destination. The problem of generating the transcoding path is that, for a given multimedia presentation, different transcoding paths should be regenerated depending on the terminal or network environment. That means that a fixed transcoding path cannot be applied to a multimedia presentation. Instead, whenever the terminal and the network bandwidth get determined, a proper transcoding path should be regenerated dynamically.

In this paper, we propose a method of generating an acceptable transcoding path that has minimum delay time for a given Digital Item and adaptation requirement. For generating all the possible transcoding paths, we may apply a classical brute force search method. Then the method may never end because of the cyclic process, resulting in endless creation of transcoding paths. Or the method may require too long processing time to find all the possible transcoding paths. Therefore our idea is to incorporate the knowledge of information about the stored multimedia data, the set of transcoders on both server and client side, the characteristics of the terminal and the limitation of the network in the process of generating a proper transcoding path. As the means of incorporating the knowledge, we propose a searching algorithm based on the theory of a Context Free Grammar. When using CFG, the key problem to solve is how to find proper production rules for a given condition. We, also suggest an algorithm for automatic generation of CFG production rules and to find one acceptable transcoding path. Finally we describe the experimental results.

2 Proposed Architecture of RAE in a DIA

2.1 Related Works

As the mentioned above, the research for RAE are focused on the transcoding of single medium transcoder. And there are many research works about the adaptation with multimedia contents, especially MPEG series. Their concerns are on metadata for mobility characteristics[3], description tools[4] and metadata driven adaptation[5]. They provide a fixed single transcoder for each case of transcoding need. With these methods, if the QoS of source and the QoS of destination is changed, the same transcoder that was used for the adaptation at the last time cannot be used again. And in the previous studies, they suggested the frame rate, color depth and resolution in linking order of transcoders[6]. They considered network bandwidth and requiring bandwidth from a mobile host about multimedia content in mobile environment. In another study, they suggested the bit rate based transcoding with frame rate and resolution transcoding concurrently [7]. They considered network environment and capacity of server and client for multimedia service in real-time. But both of them did not consider the user preference about content occurring in real-time. And there was not clear explanation about a criterion to decide linking sequence between transcoders.

A critical problem of these researches is that a given transcoder cannot satisfy all the adaptation requirements. That is, whenever the adaptation requirement of source and destination is changed, the same transcoder that was used at the last time cannot be used because of changing the end-to-end QoS. For example, 3D/2D video conversion in MPEG21 DIA is used only in case of 3D/2D video conversion.

2.2 A Proposed Architecture of Resource Adaptation Engine

Our proposed idea is that the RAE has a set of unit transcoder whose functions are limited to one transcoding function such frame rate transcoding, or size transcoding etc. In addition, the RAE has a transcoding path generation mechanism which finds an appropriate sequence of necessary unit transcoders by analyzing the resource of input Digital Item (called source contents) and the adaptation requirement at the terminal side. The separation of the set of transcoders and adaptation requirements provides a complete functionality of transcoding and flexibility of implementation. Main component of the proposed resource adaptation engine are transcoders and transcoder manager as shown in Fig. 2. The transcoder is the set of unit transcoders in the sense that their transcoding capability is limited to one specific function. The manipulation of transcoders such as addition, deletion or upgrade of a unit transcoder can occur independently from the characteristics of the Digital Item and adaptation requirements. In transcoders, we assume that there is an information table for unit transcoders such their functionality description, performance in time, throughput in bps per second etc.

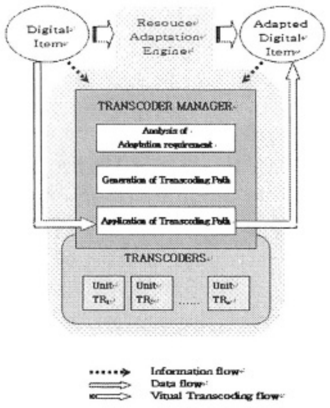


Fig. 2. A Proposed Architecture of Resource Adaptation Engine

The transcoder information may be used to calculate the efficiency in the process of selecting an efficient transcoding path among possible paths. The transcoder manager performs the three functions in sequence, analysis of the adaptation requirement, generation of a transcoding path, and application of transcoding to the resource in the Digital Item.
In this paper, our main focus is on the generation of an acceptable transcoding path that has minimum delay time in the proposed model architecture.

2.3 Problems and Our Idea

There was a previous research about connection between transcoders called transcoding path generation algorithm based on QoS transition diagram[8]. In the study, they used brute-force method trying to link all transcoders that could accept file format of content as an input format in real-time. And the cycles (the connected transcoders previously were linked again) occurred. Therefore the processes to check and to remove cycles were needed. The method generated too many nodes and took too long time to create the sequence between transcoders.

Our idea is to incorporate the knowledge of information about the stored multimedia data, the set of transcoders on both server and client side, the characteristics of the terminal and the limitation of the network in the process of generating a proper transcoding path in a procedure for finding an acceptable transcoding path for a given multimedia presentation. As the means of incorporating the knowledge, we propose a searching algorithm based on the theory of a Context Free Grammar. When a transcoding path is generated, search rules and situational knowledge are included in the production rules of the CFG, resulting in creation of only small number of nodes. When we are trying to use CFG in the searching algorithm, the key problem to solve is how to find proper production rules of the CFG for a given condition. We, also suggest an algorithm for automatic generation of CFG production rules. Our main idea is that the production rules which are dependent on the set of unit transcoders can be generated before starting the transcoding path finding process. That means we may use sufficient time to generate well defined and optimized production rules related with the transcoders in non-real time mode. The remaining work to do in real time is to generate the set of start rules which are dependents on the terminals and network environment. Therefore we are able to have the algorithm, we propose, generate transcoding paths in real time by limiting the number of rules to be regenerated every time when the network environment and terminals get changed.

3 Method of Generating a CFG an Acceptable Transcoding Path

For further discussion, some definitions and notations are to be described here. *Presentation QoS* is the information of multimedia content such as frame rate, color, size, format etc. For the simplicity, we restrict our discussion on the presentation QoS to four components- frame rate, color depth, resolution and compression format - and denote it as qos(value of frame rate, value of color depth, value of size, value of format). *Source QoS* is defined as the presentation QoS of original source content, denoted by qos_{src}. *Destination QoS* is defined as the presentation QoS of a mobile host or user preference of content, and network bandwidth denoted by qos_{dest}. *Presentation control program* is the text file that has the destination QoS for adaptation requirement. A *transcoder* is a resource that converts certain presentation QoS of content into another presentation QoS of it which has the different presentation QoS from the source data, denoted by *tr*. Every transcoder has the type and data formats for input and output. Here the tr type represents the function of the transcoder. We will denote a tr as following.

$$_{\text{input\_format}}\text{tr-type}_{\text{output\_format}}$$

For example, a MPEG-1 encoder is a transcoder which converts the YUV data into MPEG-1 data. Then the encoder can be denoted as following.

$$_{\text{YUV}}\text{Encoder}_{\text{MPE1}}$$

For the simplicity, we will use a short notation for transcoder types as, encoder(ec)/decoder(dc), frame rate transcoder(frt), color transcoder(ct), size transcoder(st), format transcoder(ft). For example, $_{\text{mpeg-2}}\text{frt}_{\text{mpeg-2}}$ refers to transcoder converting the frame rate of incoming MPEG-2 stream. Usually a server or a mobile host has several transcoders. The set of transcoders will be denoted by *TR*.

3.1 The Necessity and the Usage of a CFG

Although there are the transcoders with identical type in the server or mobile host, the transcoders have different input/output format. Then a transcoder is connected when the file format of content is identical with input format of a transcoder. For example, while there are two transcoders which type is color transcoder, one has the input file format as MPEG-2 and the other has as MPEG-4. If we want to transcode color of MPEG-2 contents, the needed transcoder must have transcoder's type as color and its input file format as MPEG-2.

So, because there are many transcoders with various input/output file format about one type of transcoder, a tool to generate the sequence between transcoders according to the situation automatically is needed. CFG(Context Free Grammar) is used for that. In Other word, when playing a multimedia stream that has different presentation QoS between the source and destination, we want to use a context free grammar, G_{tp} in order to produce a set of transcoding paths between the source and the destination. The grammar, G_{tp} is defined as like G_{tp}=(S, V_N, V_T, P) where S is the start symbol, V_T is the set of terminal symbols representing the transcoders available on the server or mobile host, V_N is the set of nonterminal symbols representing an intermediate symbol used when producing the transcoding path, and P is the set of rules. The key problem is to find the set of production rules P including start production rule automatically.

When we were trying to build the transcoding paths, we found that the two factors had influence on building production rules. The one is the set of transcoders both the server and the client have, and the other is the presentation control program. The main point we consider is that the former factor is relatively static but the latter is dynamic in a sense of time dependency. That is, the set of transcoders on the server and mobile hosts is fixed for a relatively long time period but the kind of mobile hosts, networks and user preferences on which the presentation gets played may be variable. Our idea is that the production rules related with the static factor can be generated in non-real time, denoted by P_{static}. That means we have plenty of time to spend to build and optimize the transcoder-related production rules. The only thing to do in real time is to build the production rules related with the dynamic factor. Those factors can be incorporated in the start production rules, denoted by P_{start}. We derive the connection of only terminals that mean the sequence between physical transcoders from start production rule of completed G_{tp}. The result of derivation is transcoding path.

3.2 Method for Generation of V_T, V_N, and P_{static} in Non Real-Time Mode

Terminals, nonterminals and production rules excluding start production rule is generated in non-real time. We represent each physical transcoder by means of each terminal and various combinations between transcoders by means of nonterminials and production rules about them.

When transcoders are registered into the system, terminal symbols, nonterminal symbols, and static production rules can be created automatically. Lets assume that there is three transcoders, say TR={ ${}_{mpeg\text{-}2}ft_{mpeg\text{-}4}$, ${}_{mpeg\text{-}2}dc_{yuv}$, ${}_{yuv}ec_{mpeg\text{-}4}$ } in the system. Then it is obvious that the set of terminal symbols is to be the same as TR.

But the generating of nonterminal symbols is not trivial. Looking into the TR, one way of transcoding the MPEG2 data into MPEG-4 data is to use the transcoder ${}_{mpeg\text{-}2}ft_{mpeg\text{-}4}$. The other way is the combination of two transcoders, ${}_{mpeg\text{-}2}dc_{yuv}$ and ${}_{yuv}ec_{mpeg\text{-}4}$. Therefore there is more than one way of the same transcoding process. Then we need a nonterminal $<{}_{mpeg\text{-}2}FT_{mpeg\text{-}4}>$ that transcodes the format of multimedia content from MPGE-2 to MPEG-4 can function as ${}_{mpeg\text{-}2}ft_{mpeg\text{-}4}$ or ${}_{mpeg\text{-}2}dc_{yuv}$, ${}_{yuv}ec_{mpeg\text{-}4}$. The production rule of the nonterminal for generated transcoding path is denoted by following.

$$<{}_{mpeg\text{-}2}FT_{mpeg\text{-}4}> ::= {}_{mpeg\text{-}2}ft_{mpeg\text{-}4} \mid {}_{mpeg\text{-}2}dc_{yuv}\ {}_{yuv}ec_{mpeg\text{-}4}$$

Although if there does not exist ${}_{mpeg\text{-}2}ft_{mpeg\text{-}4}$ in a server or mobile host, $<{}_{mpeg\text{-}2}FT_{mpeg\text{-}4}>$ guides for transcoding from MPGE-2 to MPEG-4 connecting the transcoders in the order of ${}_{mpeg\text{-}2}dc_{yuv}$ ${}_{yuv}ec_{mpeg\text{-}4}$, too. That is why nonterminals and their production rules are needed.

Our idea of generating nonterminal symbol is that, for each transcoder in TR, check if there is another way of doing the same transcoding process. For example, consider the transcoder ${}_{mpeg\text{-}2}ft_{mpeg\text{-}4}$, and assume all data format is one of {mpeg-2, yuv, mpeg-4}. Then we form the set of nonterminal candidates $\{<{}_{mpeg\text{-}2}FT_{yuv}>, <{}_{mpeg\text{-}2}FT_{mpeg\text{-}4}>, <{}_{yuv}FT_{mpeg\text{-}2}>, <{}_{yuv}FT_{mpeg\text{-}4}>, <{}_{mpeg\text{-}4}FT_{mpeg\text{-}2}>, <{}_{mpeg\text{-}4}FT_{yuv}>\}$ which are all the possible combinations of data formats {mpeg-2, yuv, mpeg-4}. Then for each candidate, we check if there can be a transcoding path using the transcoder set TR. By applying algorithm in non-real time mode, we found that the candidate $<{}_{mpeg\text{-}2}FT_{mpeg\text{-}4}>$ has the transcoding paths ${}_{mpeg\text{-}2}ft_{mpeg\text{-}4}$ and ${}_{mpeg\text{-}2}dc_{yuv}$, ${}_{yuv}ec_{mpeg\text{-}4}$, but other nonterminal candidates have nothing. So we can form the production rule $<{}_{mpeg\text{-}2}FT_{mpeg\text{-}4}>::={}_{mpeg\text{-}2}ft_{mpeg\text{-}4} \mid {}_{mpeg\text{-}2}dc_{yuv}\ {}_{yuv}ec_{mpeg\text{-}4}$.

3.3 Generation of P_{static} in Real-Time Mode Phase

Start production rule is created comparing QoS of the contents for network environment and user preference occurring in real-time with QoS of the contents in the server. The start rules play the role of global guide at the top level in a search space for the transcoding path. Therefore, first, we check if there is a different component of QoS between source and destination in real-time. For example, the source QoS of content is as qos_{src}(30fps, 24bit color, QCIF, MPEG-2) and the destination QoS as qos_{dest}(5fps, 24bit color, CIF, MPEG-4). Then we have six options to complete transcoding process as follows;

<1> FrT-ST-FT <2> FrT-FT-ST <3> ST-FrT-FT
<4> ST-FT-FrT <5> FT-FrT-ST <6> FT-ST-FrT

In addition, the constraint of network related QoS for the network bandwidth, denoted by $QoS_{net\_related}$ should be considered. For example about above example, when the data transfer speed in 3G mobile network is 384Kbps with a walk or the outdoor[9], the content of qos(30fps, 24bit color, QCIF, MPEG-4) can pass the network but qos(30fps, 24bit color, QCIF, MPEG-2) can not. Therefore we should decide which intermediate qos among the following intermediate qos can be passed the network bandwidth as following,

<1>qos(5fps, 24bit color, QCIF, MPEG-2) <2>qos(30fps, 24bit color, QCIF, MPEG-4)
<3>qos(5fps, 24bit color, QCIF, MPEG-4) <4>qos(30fps, 24bit color, CIF, MPEG-4)
<5>qos(5fps, 24bit color, CIF, MPEG-4)

According to the network related QoS, the place in which content will be transcoded can be decided whether on the server side or on the client side. Then the needed transcoders are recognized between the source QoS and the network related QoS, and between the network related QoS and the destination QoS.

The needed transcoders are listed in the permutation and assigned same format for input and output format for connection between the former and latter transcoders. The result of that is start production rule. From the above example, in case that we decide the network related QoS to be $qos_{net\_related}$(5fps, 24bit color, QCIF, MPEG-4) from <3>, the needed transcoders for the source are frame rate transcoder and format transcoder, and the needed transcoder for the destination is size transcoder. Then before going into the network, the transcoding order, FrT-FT or FT-FrT on the server can be applied and one size transcoding process, ST should be done on the client. This process can be described in a start production rules as following

$$S ::= <_{mpeg\text{-}2}FrT_{\ mpeg\text{-}2}><_{mpeg\text{-}2}FT_{mpeg\text{-}4}\ >_{mpeg\text{-}4}mnet_{mpeg\text{-}4}<_{mpeg\text{-}4}ST_{\ mpeg\text{-}4}>$$
$$|<_{mpeg\text{-}2}FT_{mpeg\text{-}4}><_{mpeg\text{-}4}FrT_{mpeg\text{-}4}>_{mpeg\text{-}4}mnet_{mpeg\text{-}4}<_{mpeg\text{-}4}ST_{mpeg\text{-}4}>$$

where mnet is represented with a mobile network and is a terminal going into null. Before deriving transcoding paths by G_{tp}, an optimization process of the rule such as elimination of useless symbols or non-reachable symbols from S can be applied as like compiler theory.

4 Method of Selecting an Acceptable Transcoding Path

Transcoding path is defined as a sequence of transcoders, denoted by tp. In case there are many tps, they are denoted by TP.
Algorithm for generation of an acceptable transcoding path is as following.

```
<Algorithm> Acceptable Transcoding Path Selection Algorithm
    INPUT    Digital Item, Presentation Control Program
    OUTPUT   Acceptable Transcoding Path
    STEP     1) Generation of Transcoding Path using CFG
             2) Selection of an Acceptable Transcoding Path
```

4.1 Generation of Transcoding Path Using CFG Phase

The created G_{tp} is derived from start production rule using production rules with compiler technique. The result of derivation, the connection of only terminals, Is a transcoding path. While the key efforts are to build the CFG, method for creating a transcoding path using G_{tp} is relatively simple.
Starting the start symbol, it is to generate all the possible strings of terminal symbols by applying a leftmost derivation method in the compiler theory. The set of derived strings are, in fact, the transcoding paths we try to find.

```
<Algorithm> Method for Derivation of CFG
    INPUT    $G_{tp}$
    OUTPUT   Transcoding Path
    STEP     Derivation for $G_{tp}$
```

As S continues the leftmost derivation, eventually all non-terminals become terminals. This is one of the transcoding paths. As an example, assume that qos_{src}(30fps, 24 bit color, QCIF, MPEG-2), qos_{dest}(5fps, 24 bit color, CIF, MPEG-4) and suppose the $G_{tp} = (S, V_{N\ src} \cup V_{N\ dest}, V_{T\ src} \cup V_{T\ dest}, P_{src} \cup P_{dest})$ as shown in Appendix[1] is provided by any means, the steps of leftmost deriving from S is like as follow.

Step 1) Start from S $::=<_{mpeg\text{-}2}FT_{src\ \ mpeg\text{-}4}>_{mpeg\text{-}4}mnet_{mpeg\text{-}4}<_{mpeg\text{-}4}FrT_{dest\ \ mpeg\text{-}4}><_{mpeg\text{-}4}ST_{dest\ mpeg\text{-}4}><_{mpeg\text{-}4}DC_{dest\ yuv}>$ and perform the leftmost derivation.

Step 2) The second rule of start rules $S ::= <{}_{mpeg\text{-}2}FT_{src\ mpeg\text{-}4}><{}_{mpeg\text{-}4}FrT_{src\ mpeg\text{-}4}>{}_{mpeg\text{-}4}mnet_{mpeg\text{-}4}<{}_{mpeg\text{-}4}ST_{dest\ mpeg\text{-}4}><{}_{mpeg\text{-}4}DC_{dest\ yuv}>$ can derive.

In the similar way, all non-terminals becomes terminals resulting in, ${}_{mpeg\text{-}2}dc_{src\ yuv}\ {}_{yuv}ec_{src\ mpeg\text{-}4}\ {}_{mpeg\text{-}4}frt_{src\ mpeg\text{-}4}\ {}_{mpeg\text{-}4}mnet_{mpeg\text{-}4}\ {}_{mpeg\text{-}4}st_{dest\ mpeg\text{-}4}\ {}_{mpeg\text{-}4}dc_{dest\ yuv}$. This result is one of the produced transcoding paths.

4.2 Selection of an Acceptable Transcoding Path Phase

The derived transcoding paths using G_{tp} may be valid in the sense of transcoding from the source QoS of content into the destination QoS of it. But the time require to do the transcoding process may vary depending the input workload of transcoder and the throughput of the individual transcoder.
The delay for a transcoder is expressed in the following way.

$$\text{Delay} = \text{Workload}_{\text{transcoder name}} / \text{Throughput}_{\text{transcoder name}}$$

where, $\text{Workload}_{\text{transcoder name}}$ is the amount of data for the transcoder to transcode, and $\text{Throughput}_{\text{transcoder name}}$ is the amount of data for the transcoder to be able to process in a unit time period. The former is recognized from information of each contents. For example, when content with MPEG-2 are transcoded, the workload of transcoder is 15Mbit per second. The latter is recognized from a supplier/vender for the transcoder.
Then the end-to-end delay is a sum of all transcoder delays on one transcoding path below the equation.

$$\text{End-to-End Delay} = \sum_{i=1}^{n} \text{Delay}_i$$

When the end-to-end delay of each transcoding path from the result of G_{tp} derivation is calculated using the equation above, an acceptable transcoding path with minimum delay can be decided.

5 Experiments

The purpose of experiment is as follows.

First, this experiment is to check the number of nodes to be produced by the number of transcoders used in creating a transcoding path.
Second, this experiment is to check the success rate of nodes to be created by the number of transcoders used in producing a transcoding path.

When the QoS of source to be used in the experiment is 30fps, 24bit color, QCIF, MPEG-2 file and it is to be played in 5fps, 24 bit color, CIF, decoding of MPEG-4 through a mobile network in the destination, four experiments were performed to create a transcoding path by executing two algorithms-previous QoS transition diagram based method and proposed CFG based method- of an available transcoder set 1 and an available transcoder set 2 as shown in Appendix[2]. The number of transcoding paths created through the four experiments is shown in Table 1.

Table 1. Result of Experiments

	QoS transition diagram based algorithm		proposed CFG based algorithm	
TR Set	Set 1	Set 2	Set 1	Set 2
Number of Generated Node	(Experiment 1) 50	(Experiment 2) 120	(Experiment 3) 14	(Experiment 4) 32
Success Node Rate	(16/50)*100 = 32.0%	(35/120)*100 = 29.17%	(14/14) *100 = 100%	(32/32) *100 = 100%
Number of tp	4	12	4	12

The following were concluded as per results in regards to each of available transcoder set 1 and 2.

First, the number of nodes generated was approximately four times higher in the CFG based algorithm than the QoS transition diagram based algorithm. This was because the start production rule was arranged in the former so that input/output format were determined only for transcoders found necessary after having analyzed presentation control program. Furthermore, it also because on the other hand the latter examines for generation of circulation after making connections whenever available resources with input format that equal preceding transcoder output format to prevent connections when transcoders are found to be unnecessary. Second, success rate of the generated node was approximately 3 to 3.5 times higher in the CFG based algorithm than the QoS transition diagram based algorithm for the reasons identical to that of difference in number of generated nodes. Third, types and number of generated transcoding paths were identical in both CFG based and QoS transition diagram based algorithms. This is because both algorithms connect transcoders in the form of permutation in regards to required transcoders and also because the nonterminals used in the former are created by the latter. Fourth, node success rate in the CFG based algorithm is 100%. This is caused by absence of circulations occurring in available resource collections 1 and 2. If circulation paths do not occur, failure nodes do not occur. Its reason is that nonterminals with all potential data formats as input/output format are produced beforehand and only the necessary and available transcoders are formed with nonterminals and terminals in the start production rule.

6 Conclusion

Multimedia content may be transcoded for real-time in mobile environment. For that, we proposed an acceptable trandcoding path selection method that found minimal delay path analyzing many kind of the information in real-time. CFG were used to create the path automatically. That method was implemented with TransCore. And comparing with previous method, this decreased the creation time for the transcoding path that has the minimal process time to transcode contents. That was why we used situational information and distinguished real-time work from non-real time work, while the previous method that used brute force and all the job are executed in real-time.

References

1. Harald K.: Distributed Multimedia Database Technologies Supported MPEG-7 and MPEG-21, CRC Press, (2004) 116.
2. Kim, M., B., Nam, J., H., Baek, W.,H., Son, J.,W., and Hong J., W. : The Adaptation of 3D stereoscopic video in MPEG-21 DIA, Signal Processing Image Communication, (2003) . 685-607
3. Zafer S., and Anthony V. : Mobility Characteristic for Multimedia Service Adaptation, Signal Processing:Image Communication, (2003) 699-719
4. Gabriel P., Anderas H., Jorg H., Hermann H., Harald Kosch, Christian T., Sylvain D., and Myriam A.: Bitstream Syntax Description: a tool for multimedia Resource Adaptation within MPEG-21, Signal Processing:Image Communication, (2003) 721-747
5. Laszlo B., Hermann H., Harald Kosch, Mulugeta L., and Stefan P. : Metadata Driven Adaptation in the ADMITS Project Signal Processing:Image Communication, (2003). 749-766
6. Lee S.,J., Lee H., S., Park S., Y., Lee S., W., and Jeong G., D. : Bandwidth Control scheme using Proxy-based Transcoding over Mobile Multimedia Network, Association of Information Science of Korea Proceeding, Vol. 29 No.02, (2002) 293 ~304
7. Kim J., W., Kim Y., H., Paik J., H., Choi B., H., and Jeong H., G.: Design and Implementation of Video Transcoding System for the Real-time Multimedia Service, IPIU, Vol.15 No.01, (2003) 177 ~182
8. Chon S., M., and Lim Y., H.: Algorithm Using a QoS Transition Diagram for Generating Playable Transcoding Paths of the Multimedia Presentation with Different QoS between the Source and the Destination, Association of Multimedia of Korea, Vol. 6, (2003) 208-215
9. http://www.uangel.com/eng/imt2000/comparison.html

Appendix

[1] Example of G_{tp}

	Source	Destination
S	$S ::= <{}_{\text{mpeg-2}}FT_{\text{src mpeg-4}}> {}_{\text{mpeg-4}}mnet_{\text{mpeg-4}}$ $<{}_{\text{mpeg-4}}FrT_{\text{dest mpeg-4}}><{}_{\text{mpeg-4}}ST_{\text{dest mpeg-4}}><{}_{\text{mpeg-4}}DC_{\text{dest yuv}}>$ $\mid <{}_{\text{mpeg-2}}FT_{\text{src mpeg-4}}><{}_{\text{mpeg-4}}FrT_{\text{src mpeg-4}}> {}_{\text{mpeg-4}}mnet_{\text{mpeg-4}}$ $<{}_{\text{mpeg-4}}ST_{\text{dest mpeg-4}}><{}_{\text{mpeg-4}}DC_{\text{dest yuv}}>$	
V_T	$V_{T\ \text{src}}$ $=\{ {}_{\text{mpeg-2}}frt_{\text{src mpeg-2}}, {}_{\text{mpeg-4}}frt_{\text{src mpeg-4}},$ ${}_{\text{mpeg-2}}dc_{\text{src yuv}}, {}_{\text{yuv}}ec_{\text{src mpeg-4}} \}$	$V_{T\ \text{dest}}$ $=\{ {}_{\text{mpeg-4}}frt_{\text{dest mpeg-4}}, {}_{\text{mpeg-4}}ct_{\text{dest mpeg-4}},$ ${}_{\text{mpeg-4}}st_{\text{dest mpeg-4}}, {}_{\text{mpeg-4}}dc_{\text{dest yuv}},$ ${}_{\text{yuv}}ec_{\text{dest mpeg-4}} \}$
V_N	$V_{N\ \text{src}}$ $=\{ <{}_{\text{mpeg-2}}FT_{\text{src mpeg-4}}>, <{}_{\text{mpeg-4}}FrT_{\text{src mpeg-4}}>,$ $<{}_{\text{mpeg-2}}DC_{\text{src yuv}}>, <{}_{\text{yuv}}EC_{\text{src mpeg-4}}> \}$	$V_{N\ \text{dest}}$ $=\{ <{}_{\text{mpeg-4}}FrT_{\text{dest mpeg-4}}>, <{}_{\text{mpeg-4}}ST_{\text{dest mpeg-4}}>,$ $<{}_{\text{mpeg-4}}DC_{\text{dest yuv}}> \}$

P	P_{src} $\langle {}_{\text{mpeg-2}}FT_{\text{src mpeg-4}}\rangle$ $::=\langle {}_{\text{mpeg-2}}DC_{\text{src yuv}}\rangle\langle {}_{\text{yuv}}EC_{\text{src mpeg-4}}\rangle$ $\langle {}_{\text{mpeg-4}}FrT_{\text{src mpeg-4}}\rangle ::= {}_{\text{mpeg-4}}frt_{\text{src mpeg-4}}$ $\langle {}_{\text{mpeg-2}}DC_{\text{src yuv}}\rangle ::= {}_{\text{mpeg-2}}dc_{\text{src yuv}}$ $\langle {}_{\text{yuv}}EC_{\text{src mpeg-4}}\rangle ::= {}_{\text{yuv}}ec_{\text{src mpeg-4}}$	P_{dest} $\langle {}_{\text{mpeg-4}}FrT_{\text{dest mpeg-4}}\rangle ::= {}_{\text{mpeg-4}}frt_{\text{dest mpeg-4}}$ $\langle {}_{\text{mpeg-4}}ST_{\text{dest mpeg-4}}\rangle ::= {}_{\text{mpeg-4}}st_{\text{dest mpeg-4}}$ $\langle {}_{\text{mpeg-4}}DC_{\text{dest yuv}}\rangle ::= {}_{\text{mpeg-4}}dc_{\text{dest yuv}}$

[2] Type of Available Transcoders for the Experiments

	available transcoders Set 1	available transcoders Set 2
Source $V_{T\ \text{src}}$	$\{{}_{\text{mpeg-2}}frt_{\text{src mpeg-2}},\ {}_{\text{mpeg-4}}frt_{\text{src mpeg-4}}$ $\ {}_{\text{mpeg-2}}dc_{\text{src yuv}},,\ \ {}_{\text{yuv}}ec_{\text{src mpeg-4}}\}$	$\{{}_{\text{mpeg-2}}frt_{\text{src mpeg-2}},\ {}_{\text{mpeg-4}}frt_{\text{src mpeg-4}},$ ${}_{\text{mpeg-2}}st_{\text{src mpeg-2}},\ {}_{\text{mpeg-4}}st_{\text{src mpeg-4}},$ ${}_{\text{mpeg-2}}ft_{\text{src mpeg-4}}\}$
Destination $V_{T\ \text{dest}}$	$\{{}_{\text{mpeg-4}}frt_{\text{dest mpeg-4}},\ {}_{\text{mpeg-4}}ct_{\text{dest mpeg-4}},\ {}_{\text{mpeg-4}}st_{\text{dest mpeg-4}},\ {}_{\text{mpeg-4}}dc_{\text{dest yuv}},\ \ {}_{\text{yuv}}ec_{\text{dest mpeg-4}}\}$	

Design and Analysis of a Variable Bit Rate Caching Algorithm for Continuous Media Data

Ligang Dong[1] and Bharadwaj Veeravalli[2]

[1] College of Information and Electronic Engineering
Zhejiang Gongshang University
149 Jiaogong Road, Hangzhou, Zhejiang, P.R.China, 310035
donglg@mail.hzic.edu.cn

[2] Department of Electrical and Computer Engineering
The National University of Singapore,
10, Kent Ridge Crescent, Singapore 119260
elebv@nus.edu.sg

Abstract. We designed a novel algorithm, referred to as *variable bit rate caching* (VBRC) algorithm for continuous media data. Comparing with previous caching algorithms, VBRC can allocate the cache space and I/O bandwidth in a dynamic and just-in-time way. Using the simulation, it is testified that VBRC can improve the byte-hit-ratio of the cache and reduce the probability of "switching" operations, which is harmful to the performance of the original server.

1 Introduction

The research of data caching is always a fertile ground. Some studies in the recent literature on a variety of caching problems include, memory page caching [11], web caching [1,2], and multimedia caching [6,18]. Two categories of the caching are memory caching and disk caching [17]. Memory caching refers to caching objects in the main memory whereas, disk caching refers to caching objects in the hard disk. Bandwidth of a memory cache is rarely a bottleneck [18]. On the contrary, disk-caching policies have to consider constraints imposed by the disk bandwidth as well as the disk space.

Typically, there are two kinds of multimedia documents [3]: continuous media (CM) data and non-continuous media (non-CM) data. Firstly, CM data (e.g., movies) can have an order of magnitude much larger than the atomic objects (e.g., text and image object). Hence, caching an entire stream object leads to a poor performance [18]. Secondly, we also need to guarantee the playback continuity of CM data.

In the past research, some strategies for caching CM data have been proposed, e.g., prefix-caching [12,15], work-ahead smoothing [16,14], prefetching [4]. Besides, there are two classes of caching strategies/algorithms for CM data. One class is the block-level algorithm, e.g. BASIC [13]. BASIC selects to replace a block that would not be accessed for the longest period of time. The other class is the interval-level algorithm, e.g., DISTANCE [13], interval caching (IC) [5], resource-based caching (RBC) [18], and generalized interval caching (GIC) [6]. As far as the design of strategies are concerned, DISTANCE is observed to be similar to IC [6]. RBC is a disk-caching algorithm while DISTANCE, IC, and GIC are memory-caching algorithms. IC is suitable for streams accessing long objects, and GIC extends IC so that both long and short CM data can be managed. In an interval-level algorithm, the basic caching entity is an *interval,* which is the amount of data that is in between two adjacent streams, so that the latter stream always can read the date cached by the former stream(s). In comparison with the interval-level caching algorithm, the block-level caching algorithm has much higher operational overhead. Also, it is difficult to guarantee a continuous playback at the client end for the block-level caching algorithm. Thus, we will consider designing efficient interval-level caching algorithms in this paper.

1.1 Motivation

We found the following drawbacks in past interval caching algorithms.

Firstly, for any of the above algorithms, if a stream that is reading from the cache finds that there is no *required data* or *sufficient retrieval bandwidth* in cache before its retrieval finishes, this stream has to retrieve the corresponding data from the original server (or disk storage in terms of memory caching). We refer to this simply as *switching back* to the server (either the disk storage in memory caching or the original server in disk caching). In order to avoid any discontinuity while retrieving the required data owing to shortage of resources (i.e., server I/O bandwidth and network bandwidth), the required amount of bandwidth should be reserved in advance. Furthermore, the switching operation not only causes additional bandwidth consumption, but also incurs additional processing overheads, and hence, it is undesirable. Thus, in this paper, one of our objectives is to design an interval-level caching algorithm that minimizes the number of switches to the server. Lee et al. [10] also notices the disadvantage of switching operations. However, the studies in [10] are only based on IC (i.e., interval caching) used in the memory caching. In fact, the shortage of bandwidth in disk caching may also result in the switching operation.

Secondly, previous interval-level algorithms assumed that the streams that retrieve the data of the same object have the identical retrieval bandwidth. However, this assumption is not always true. There are two reasons. (1). Different streams probably have different retrieval bandwidth. (2). The retrieval bandwidth of one stream probably varies with time. In an interval-level caching algorithm, if two streams that form an interval have different retrieval bandwidths, then the cache space requirement of this interval and the cache I/O bandwidth requirement of these two streams will be *variable.* Past interval-level caching algorithms allocate the resource (including bandwidth and space) only once for every interval or every stream, and therefore they cannot handle the case of variable retrieval bandwidth.

Finally, in the case of disk caching algorithm RBC, the allocation of the cache I/O bandwidth for a stream is a reservation mode, which is often less efficient. As the cache I/O bandwidth is crucial, if we can allocate the bandwidth *just-in- time* instead of by a reservation, then the caching performance can be considerably improved.

The above-mentioned issues considerably motivate us to propose novel strategies for caching CM data that demand a variable bit rate retrieval. This paper is organized as follows. Firstly, in Section 2, we introduce the system architecture and the performance metric. Then we describe the proposed strategies in Section 3. Next, in Section 4, we introduce the outline of the VBRC algorithm. Furthermore, in Section 5, we show the result of rigorous simulation and provide a detailed discussion. Finally, in Section 6, we highlight the key contributions.

2 System Model

2.1 System Architecture

The disk-caching strategies discussed in this paper are applicable to an architecture formed by "the server - the disk cache - the clients" configuration whereas, the memory-caching strategies discussed in this paper are applicable to an architecture formed by "the server - the memory cache - the clients" configuration. In memory caching, "the server" actually means "the local disk storage". The connection between the server and the cache, or between the cache and the client is via a network. In this paper, for both the disk caching and the memory caching schemes, we only consider a single cache case (system with a single cache storage) as attempted by other researchers [13,5,18,6].

2.2 Performance Metrics

In this paper, the design of caching strategies fundamentally aims to reduce the traffic overload on the network, through which the requested objects are transferred, and the server. In other words, the saving on the network bandwidth or the server I/O bandwidth is treated as the main *benefit* of caching. Therefore, we evaluate the caching strategies in terms of their ability to improve **byte hit ratio.** Byte hit ratio equals the ratio of the total number of bytes accessed from the cache to the total number of bytes accessed.

3 Caching Strategies

In this section, we present three basic strategies that will be used in our caching algorithm (introduced in Section 4). Because of the limited space in this paper, we put the detail of these strategies in [8] and [9]. Here we give only very concise introduction.

(1). Caching strategy for the variable retrieval bandwidth
As the cache bandwidth requirement of one stream and the cache space requirement of one interval are time-varying, we periodically check the availability of the bandwidth and space in the cache . If the total requirement for

bandwidth (or space) of intervals is greater than the cache I/O bandwidth (or space) capacity, some amount of bandwidth (or space) will be reclaimed from some streams using a *cache bandwidth (or space) reclaim strategy.* Besides, when the size of the retrieved data of a following stream probably exceeds a preceding stream, .we handle the repositioning of streams within an interval using an *exchange strategy.*

(2). Caching strategy under the non-switch constraint
Through satisfying the non-switch constraint, the switching operation is eliminated or reduced, and therefore bandwidth of servers and networks is saved to a significant extent.

(3). Allocation strategy of the cache I/O bandwidth
We found that the bandwidth of the disk cache is really required by a stream in *only* two cases: 1) when a stream begins to write the data into the cache, 2) when a stream begins to read the data from the cache. We will not allocate (reserve) bandwidth for any other case.

4 Variable Bit Rate Caching Algorithm

Using the three strategies introduced before, we design a new caching algorithm - variable bit rate caching (VBRC) algorithm, which can be used in disk caching or memory caching. Without any loss of generality, we describe VBRC in the context of disk caching. When we ignore the bandwidth constraint, VBRC is directly applicable for the memory caching.

we introduce the entire process of the VBRC algorithm in Fig. 1. This algorithm includes three phases, which are carried out once in every service cycle D. D is chosen in such a way that it is more than the time overhead of carrying out three phases. From our experiments, to be presented later, the overheads are observed to be very small.

4.1 Remarks

In the original GIC proposed in [7], non-switch constraint was not considered, and hence the original GIC is referred as *GIC without non-switch constraint.* If the non-switch constraint is imposed on the original GIC, then the resulting caching algorithm is referred as *GIC with non-switch constraint.* Obviously, the memory caching version of VBRC is identical with GIC with non-switch constraint when both of them are adopted in the case of constant retrieval bandwidth.

In VBRC, an interval size g is adopted to measure the “importance” of intervals for the allocation of bandwidth and space. In fact, the interval length (h) can also be adopted instead of the interval size. The performance of VBRC will not be considerably affected.

Besides, both GIC and RBC do not analyze the bad effect of the switching operation and not try to reduce the number of the switching operation, also these two algorithm cannot directly used in the case of variable retrieval bandwidth. In comparison, strategies are proposed in VBRC to handle these two issues.

Phase 1: For every unfinished stream. We carry out one of the following three cases appropriately.
Case 1. For a stream that is reading from the server, if the block to be read is available in the cache, then the stream is switched from the server to the cache. Meanwhile, the available bandwidth A_b decreases by an amount *bw,* which is current retrieval bandwidth of the stream.
Case 2. For a stream that is reading from the cache, if the stream is not the former stream of an interval, the read block by the stream is swapped out of the cache.
Case 3. For a writing stream, the writing data is cached.
If the stream will exceed the preceding stream during *D,* then the exchange strategy for handling the exceeding stream is carried out.

Phase 2: Check the available bandwidth A_b and available space A_s
1. If $A_b < 0$, then bandwidth reclaim strategy for the cache bandwidth is carried out to obtain bandwidth from streams which form unimportant cached intervals. 2. If $A_s < 0$, then a space reclaim strategy for the cache space is carried out to obtain space from unimportant cached intervals.

Phase 3: For the arriving requests. For every request m for accessing an object i, a stream S_m is formed. Assume that the retrieval bandwidth of S_m is *bw*. We carry out one of the following three cases appropriately.
Case 1. If the first block to be read has been cached, a new interval is formed (refer to Fig. 2 for an example).

- If the available bandwidth A_b is not enough (e.g., $bw > A_b$), an allocation strategy for bandwidth is carried out to obtain bandwidth from streams which form unimportant cached intervals.
- If the available bandwidth A_b is enough, then the stream begins to read from the cache.

Case 2. If the first block has not been cached and there are not any preceding streams for the same object i, then the stream starts to read from the server.

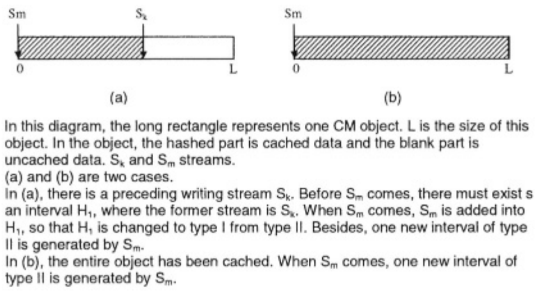
- A new interval H_n is formed by a single stream S_m. Simultaneously, S_m becomes a writing stream.
- Check the space and bandwidth requirement of H_n. If A_s is not enough (i.e., $R_{sn} > A_s$), then the allocation strategy for space is carried out to obtain space from unimportant cached intervals.

- If available bandwidth and space are not enough, then H_n is deleted.

Case 3. If the first block has not been cached and there is one preceding stream for the same object i, e.g., stream S_k, then the stream starts to read from the server.

- A new interval H_n is formed by stream S_m and S_k. Simultaneously, if S_k is reading from the server, then S_k becomes a writing stream.
- Check the space and bandwidth requirement of H_n. If A_s is not enough (i.e., $R_{sn} > A_s$), then the allocation strategy for space is carried out to obtain space from unimportant cached intervals.
- If available bandwidth and space are not enough, then H_n is deleted.

Fig. 1. VBRC algorithm



In this diagram, the long rectangle represents one CM object. L is the size of this object. In the object, the hashed part is cached data and the blank part is uncached data. S_k and S_m streams.
(a) and (b) are two cases.
In (a), there is a preceding writing stream S_k. Before S_m comes, there must exist s an interval H_1, where the former stream is S_k. When S_m comes, S_m is added into H_1, so that H_1 is changed to type I from type II. Besides, one new interval of type II is generated by S_m.
In (b), the entire object has been cached. When S_m comes, one new interval of type II is generated by S_m.

Fig. 2. An example illustrating the form of a new interval

5 Performance Evaluation

In this section, we conduct rigorous simulation experiments to testify all our strategies presented in Section 4. The performance measure in our study is the byte hit ratio under several influencing factors such as, disk cache size *B,* disk bandwidth *BW,* arrival rate λ, and percentage of requests for large objects *P,* respectively. Table 1 shows the system parameters that are used in our simulation experiments. These parameters and their values used in our experiments below are typical to real-life situations, and the similar values are considered in the past literature [18,7]. We adopt 70 – 20 access skew (i.e., about 70% of the accesses are restricted to 20% of the objects) of Zipf distribution as in [18].

We show only part results of the simulation because of limited space in this paper.

Table 1. System parameters in the CM caching

System Parameters	Symbol	Parameter Values
Number of Different Objects	M	100
Disk Cache Storage Size	B	$1G - 80GB$
Disk Cache Bandwidth Size	BW	$10 - 90MB/s$
Memory Cache Storage Size	B	$100 - 1000MB$
Skew Factor of Access Prob. Distribution	θ	0.06
Number of Requests	N	1500
Arrive Rate of Requests	λ	$0.1 - 1.0/s$
Service Cycle	D	$1sec$
Playback Rate	r	uniform in $0.25 - 0.75MB/s$
Variable Scope of Retrieval Bandwidth	V	$0 - 20\%$
Percentage of Large Objects	P	$0 - 100\%$

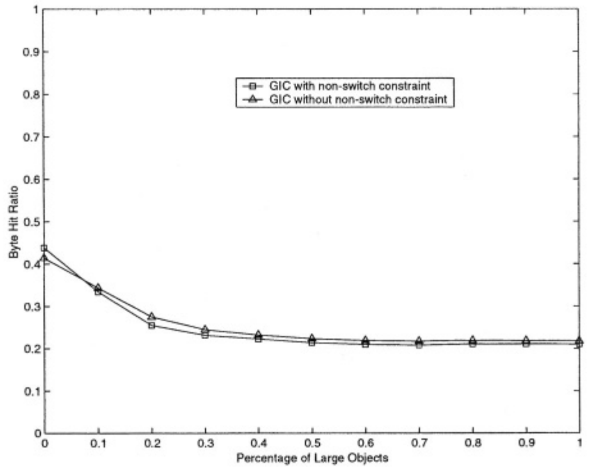


Fig. 3. Performance comparison between GIC with and without the non-switch constraint ($B = 500MB, \lambda = 0.25/s$)

5.1 Effect on Performance Due to Non-switch Constraint

In this section, the non-switch constraint is imposed on GIC, so that the influence on the performance due to the non-switch constraint can be observed. We compare the performance of GIC with non-switch constraint and GIC without non-switch constraint. The result shows GIC without non-switch constraint exhibits a little better performance than GIC with non-switch constraint with respect to byte hit ratio (e.g., Fig. 3). However, note that here, only the performance of cache is shown. The performance improvement of GIC without non-switch constraint is achieved at the cost of poor utilization of bandwidth resource in the server. GIC with the non-switch constraint has gotten rid of all the switching operation while GIC without the non-switch constraint has to reserve bandwidth for the streams reading from the cache.

On the one hand, if we reserve bandwidth of the server for every stream that is reading data from the cache, then the total amount of bytes read from the server is obviously reduced. Note that, the amount of bytes read from the cache is equivalent to a resource wastage in the server. To understand this, assume A_1 is the amount of data read from cache, which is also the amount of resource wasted in the server, by reserving the server bandwidth for stream reading from cache. On the other hand, when we adopt a non-switch constraint to eliminate the switching operation, the byte hit ratio decreases. We denote A_2 as the difference in the number of bytes that are read from the cache without non-switch constraint and with non-switch constraint. Thus, we compare these two cases by observing the ratio A_2/A_1 and we found the average value of A_2/A_1

is less than 0.1 (refer to Table 2). It means that to satisfy the non-switching constraint can bring benefits for the entire system (including the server and the cache).

Table 2. Effect of Non-switch constraint(totally 1500 requests)

$B(MB)$	100	200	300	400	500	600	700	800	900	1000	
A_2/A_1	0.06	0.03	0.05	0.04	0.04	0.04	0.03	0.50	0.34	0.26	
λ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	
A_2/A_1	0.01	0.03	0.03	0.04	0.07	0.24	0.25	0.26	0.26	0.27	
P	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
A_2/A_1	0.01	0.02	0.03	0.05	0.02	0.02	0.02	0.02	0.04	0.04	0.04

5.2 Performance Comparison of RBC and VBRC

In the case of memory caching, it should be clear at this stage that VBRC is actually GIC with non-switch constraint. Hence we do not need to compare the performance of VBRC and GIC. Thus, in this section, we compare the performance of RBC and VBRC in the case of constant retrieval bandwidth (since RBC cannot be used in the case of variable retrieval bandwidth).

Because of limited space, we only use Fig. 4 as an example.

In comparison with RBC, our VBRC algorithm adopts the "just-in-time" allocation method of bandwidth, and the allocation of space is among intervals

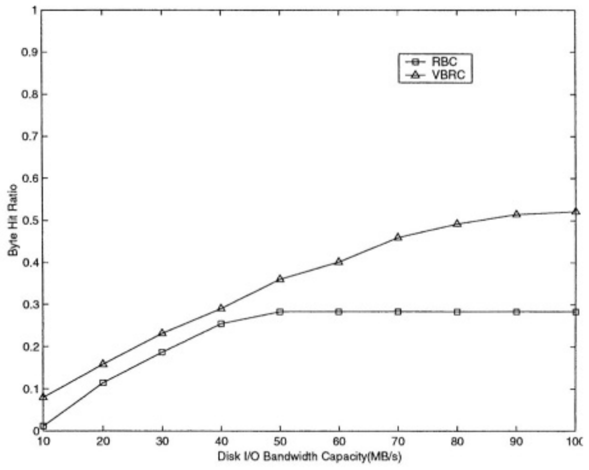


Fig. 4. Performance comparison between RBC and VBRC ($B = 5000MB, \lambda = 0.25/s, P = 80\%$)

and not the runs. We observe that VBRC has much better performance than RBC. Besides, there are no switching operations in VBRC (as we use non-switch constraint), while the switching operation exists in RBC (refer to Table 3).

Table 3. Number of the switching operation in RBC (totally 1500 requests)

$B(GB)$	1	5	10	20	30	40	50	60	70	80	
	0	59	64	56	56	56	56	56	56	56	
$BW(MB/s)$	10	20	30	40	50	60	70	80	90	100	
	44	59	74	24	0	0	0	0	0	0	
λ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	
	11	50	84	116	107	115	54	109	88	40	
P	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
	0	68	75	109	54	59	78	112	59	50	100

6 Conclusions

In this paper, we have proposed a novel algorithm VBRC, which is used in caching CM data in the disk and the main-memory storage media. The design of VBRC stems from two reasons. The first reason is due to the fact that current interval-level caching algorithms cannot handle the case of variable retrieval bandwidth (or variable bit rate). Variable retrieval bandwidth demand is most commonly encountered in multimedia applications wherein a client interacts during presentation of the CM data. Our VBRC algorithm checks the space and bandwidth requirements in every service cycle, and handles the exceeding streams in the case of variable retrieval bandwidths. The second reason is that there exist switching operations in all the so far proposed caching algorithms in the literature and these switching operations result in low utilization of bandwidths in servers. Besides, in VBRC, the bandwidth resource is allocated in *just-in-time* manner instead of carrying out any advanced reservation. The performance of VBRC in terms of byte hit ratio under several influencing parameters such as, cache space size, cache I/O bandwidth, request arrival rates, and percentage of large objects are studied in our simulation experiments. VBRC is conclusively testified to perform better than the popular disk caching algorithm RBC [18].

References

1. C.Aggarwal, J.L.Wolf, and Yu, P.S., "Caching on the World Wide Web," *IEEE Transactions on Knowledge and Data Engineering,*Vol.11 (1):94 -107, 1999.

2. G.Barish and K.Obraczke, "World Wide Web caching: trends and techniques," *IEEE Communications Magazine,* Vol.38(5):178-184, May 2000.
3. K.S.Candan, B.Prahakaran, and V.S. Subrahamanian, "Retrieval schedules based on resource availability and flexible presentation specifications," *Multimedia System,* No.6, pp.232-250,1998.
4. R.Cucchiara, M.Piccardi, and A.Prati, "Temporal analysis of cache prefetching strategies for multimedia application," *Proc. IPCCC 2001,* April 4-6, 2001 .
5. A.Dan and D.Sitaram, "buffer management policy for an on-demand video server," *Technical Report RC 19347,* IBM Research Report, 1994.
6. A.Dan and D.Sitaram, "A generalized interval caching policy for mixed interactive and long video environments," *Proc. IS&T SPIE Multimedia Computing and Networking Conference,* San Jose, CA, January 1996.
7. A.Dan and D.Sitaram "Multimedia caching strategies for heterogeneous application and server environment," *Multimedia Tools and Applications,* Vol.4(2):279-312, 1997.
8. Ligang Dong, "Design, Analysis, and Experimental Verification of Continuous Media Retrieval And Caching Strategies For Network-Based Multimedia Services," PhD thesis, Department of Electrical & Computer Engineering, National University of Singapore, 2002.
9. Ligang Dong and Bharadwaj Veeravalli, "Design and Analysis of Variable Bit Rate Caching Strategies for Continuous Media Data," *Proc. IEEE International Conference on Multimedia & Expo (ICME),* June 2004.
10. K.Lee, J.B.Kwon, and H.Y.Yeom, "Exploiting Caching for Realtime Multimedia Systems," in *Proc. of sixth IEEE International Conference on Multimedia Computing and Systems,* Florence, Italy, 1999.
11. A.Leff, J.Wolf, and P.S.Yu, "Efficient LRU-based Buffering in a LAN Remote Caching Architecture," *IEEE Trans. Parallel and Distributed Systems,* Vol.7,No.2,pp.191-206, Feb.1996.
12. H.Lim and D.H.C.Du, "Protocol considerations for video prefix-caching proxy in wide area networks,"" *Electronics letters,* Vol.37, No.6, March 2001.
13. B.Ozden, R.Rastogi, and A.Silberschatz, "Buffer replacement algorithms for multimedia databases," *Multimedia information storage and management,* S.Chung ed.,pp. 163-180. Kluwer Academic Publishers, Boston, MA, 1996.
14. Y.Wang, Z.L.Zhang, D.Du and D.Su, "A network conscious approach to end-to-end video delivery over wide area networks using proxy servers," *Proc. IEEE INFOCOM,* April 1998.
15. S.C.Park, Y.W.Park, and Y.E.Son "A proxy server management scheme for continuous media objects based on object partitioning," *Proc. Eighth International Conference on Parallel and Distributed Systems (ICPADS 2001),* 26-29 June 2001.
16. S.Sen, J.Rexfordz, and D.Towsley "Proxy prefix caching for multimedia streams," *Proc. INFOCOM 1999.*
17. D.Sitaram and A.Dan, "Multimedia servers : design, environments, and applications," *San Francisco, Calif. : Morgan Kaufman Pub,* 1999.
18. R.Tewari, H.M.Vin, A.Dan, and D.Sitaram, "Resource-based Caching for Web Servers," *Proc. of SPIE/ACM Conf. on Multimedia Computing and Networking,* San Jose, 1998.

A Configuration Tool for Caching Dynamic Pages

Ikram Chabbouh and Mesaac Makpangou

INRIA - Projet Regal - B.P. 105 - 78153 Le Chesnay Cedex - France
{ikram.chabbouh,mesaac.makpango}@inria.fr

Abstract. The efficacy of a fragment-based caching system fundamentally depends on the fragments' definition and the bringing into play of mechanisms that improve reuse and guarantee the consistency of the cache content (notably "purification" and invalidation mechanisms). Existing caching systems assume that the administrator provides the required configuration data manually, which is likely to be a heavy, time-consuming task and one that is prone to human error. This paper proposes a tool that helps the administrator to cope with these issues, by automating the systematic tasks and proposing a default fragmentation with the prerequisite reuse and invalidation directives, that may either be augmented or overwritten if necessary.

1 Introduction

Web pages can be classified into two broad categories: static pages which do not change each time they are visited, and dynamic pages, whose text and appearance may change depending on some factors (such as text input by the user). Static pages are stored on the web server in HTML format fully prepared for display in the browser, while dynamic pages are written in specific programming languages (PERL, PHP, ASP, etc.). When a dynamic page is requested by the user, the Web server runs the programming language interpreter or compiled code which produces the content of the document based on the programmed logic and parameters passed. The response is then sent back to the browser "on-the-fly". Web sites increasingly rely on dynamic content generation tools to provide visitors with dynamic, interactive and personalized content. However, dynamic content generation comes at a cost. High performance Web servers can typically deliver up to several hundred static files per second on a uniprocessor. In contrast, the rate at which dynamic pages are delivered is often several orders of magnitude slower; it is not uncommon for a program to consume over a second of CPU time in order to generate a single dynamic page [1].
Caching is currently the primary mechanism used to reduce latency as well as bandwidth consumption in order to deliver Web content. The caching of the dynamic content is motivated by the observation that dynamic Web pages are not completely dynamic: often, the dynamic and personalized contents are embedded in relatively static Web pages [2].

Existing tools, that cache dynamic pages, can either cache the entire page ([3],[4]) or a fragmented[1] version of it ([1],[5]). It should be mentioned that caching at page level provides very limited reusability potential.
The efficiency of a fragment-based caching system fundamentally depends on the fragments' definition and the bringing into play of mechanisms that improve reuse and guarantee the consistency of the cache content. These in particular concern "purification" and invalidation mechanisms.
To cope with these issues, existing caching systems generally assume that the administrator provides the required configuration data manually. It seems natural to turn to the administrator for such tasks since he is supposed to be the person who knows most about such application details. Nevertheless, this is likely to be a heavy, time consuming task and one that is prone to human error.
To address the problem, we propose a tool that helps an administrator to configure his site easily by calculating default parameters which can be used by subsequent caching modules to generate a default fragmentation with the prerequisite reuse and invalidation mechanisms. These parameters can be tuned or overwritten in accordance with the application's actual needs. There are basically two distinct modules in our system: a program analyzer and a configuration module. The first module analyzes the programs and generates a high-level description of them, while the second uses this description to generate default settings which can either be augmented or overwritten by the administrator.

The rest of the paper is organized as follows. Section 2 provides the background to this work. Section 3 explains the underlying principles of our system then details its two modules (the analyzer and the configuration modules) and illustrates their functioning with a concrete example. Finally, Section 5 raises some open questions and delineates the directions of our future work.

2 Background

Recent work on dynamic content caching, has proven the advantages of fragment-based schemes ([6],[5]). However, a fragment-based solution is not effective unless adequate solutions are found to the three key issues: fragmentation of dynamically generated pages; reuse of HTTP responses for equivalent requests and invalidation of the cached content.

The first issue concerns identifying the "interesting" fragments (those that are likely to be reused by some other dynamic pages). Once identified, the fragments are reified and stored separately by the server or by the caches. Henceforth, each dynamic page will only contain the references of its fragments rather than the fragments themselves. Thus, the fragmentation should lead to savings in storage, CPU time and bandwidth consumption (see section 3).
Most of the existing studies assume that web pages are fragmented manually ([1], [6], [5], [7]). To the best of our knowledge, only one research study ([8]), has delved deeper into the automation of fragment detection. It formally defines a *candidate fragment* as a fragment that is shared among "M" already

[1] a fragment can be defined as a part, or all, of a rendered HTML page which can be cached

existing fragments (where "M"> 1), and that have different lifetime characteristics from those of its encompassing candidate fragment. To detect and flag candidate fragments from pages, the study proceeds in three steps. First, a data structure (called *Augmented Fragment Tree*) representing the dynamic web pages is constructed. Second, the system applies the fragment detection algorithm to augmented fragment trees to detect the candidate fragments. In the third step, the system collects statistics about the fragments (such as size, access rates etc.). These statistics are meant to help the administrator to decide whether to activate the fragmentation. This approach has several disadvantages. First and foremost, the granularity of the fragmentation cannot really be tuned accordingly within the different pages of a Web site. Actually, the administrator can only act on: the minimum fragment size, the sharing factor[2] and the minimum matching factor[3]. While these parameters certainly affect the resulting fragments, they do not allow the administrator to specify particular fragments. For instance, a keyword-based search may return several independent results which might well be considered as fragments, however, one can hardly determine the appropriate parameters that will result in their generation. The second point concerns the perception of the concept of fragment itself. For file fetches on the Web the network is responsible for a significant portion of the response time, but for dynamically generated web pages, the response time is not only increased by the propagation time on the network, but also and particularly by the page generation time on the server. The fragments are expected to reduce the generation time on the server by manipulating (invalidating, generating, etc.) a fragment rather than the complete page. However, in the case of this study (as in many others), the fragment's definition does not take account of the application implementation in that when a fragment is invalidated, there are no means of asking for the particular fragment and instead, the entire page is regenerated.

The second issue concerns the reuse of HTTP responses. The idea is to know when it is possible to reuse the result of a previous request. In practice, it consists in trying to define a unique "cache-wide" key per document. If the key matches a number of distinct versions of a document, then the cache may deliver the wrong content, and if several keys match the same document then the effectiveness of the cache is diminished due to the unjustified cache misses and the duplicated stored content. We call *purification* the operation that computes this key. The primary component of a document key is simply the corresponding URL, however, in many cases the URL is either not sufficient or contains too much information to be the key. One of the rare studies that has dealt with the problem ([9]), proposes augmenting the URL with HTTP request header data or stripping data from the URL as appropriate. The data in question still has to be identified manually. A more generic study ([10]), distinguishes three kinds of locality in dynamic Web content: requests may be identical, equivalent or partially equivalent. According to this study, identical requests have identical URLs which result in the generation of the same content. Equivalent requests

[2] The sharing factor indicates the minimum number of pages that should share a segment in order for it to be declared a fragment

[3] The minimum matching factor specifies the minimum overlap between the segments to be considered as a shared fragment

have syntactically different URLs but result in the generation of identical content. Partially equivalent requests are also syntactically different but result in generation of content which can be used as a temporary place-holder. As in the previous study these equivalence classes are defined manually.

The third issue concerns consistency management. Since dynamic content changes more frequently and less predictably than static content, the basic TTL (Time To Live) mechanism is likely to be inappropriate in such a context as it may either result in a significant number of unnecessary invalidations or in stale content being sent to the clients. On the other hand, dynamic content is often based on underlying data stored in databases, thus, events external to the Web application can trigger changes in the Web content. The major hurdle in maintaining consistency is establishing mappings between underlying relational data and cached Web data [9]. With such mappings, changes in the underlying data can be translated into invalidations of the affected cache content.
A great number of studies rely on the basic TTL as an invalidation mechanism ([11],[7]). More sophisticated approaches propose using the page dependencies. Yet, the majority of these assume that the administrator manually defines the page (or the fragment) dependencies([3],[1]).The automation of page dependencies detection has, however, also been investigated in [12]. This work proposes splitting the mapping process into two steps: (i) the mapping between web pages and queries that are used to generate these pages and (ii) the mapping between the queries and the data changes that affect these queries. While the idea of automating the process of dependencies' calculation is very apposite, this implementation spawns some redundant processing insofar as the mapping between the pages and the queries used to generate them is processed for each request, neglecting the fact that several calls (with different parameters) to the same script will always generate the same queries to the database. In addition, these dependencies are computed at the page level and not the fragment level, therefore when an invalidation is sent, the whole page is considered to be stale and is regenerated although, in fact, only a fragment needed to be.

As we can see from the above range of studies, the current tendencies for tackling the cache of dynamic content issue are either to put the burden of any configuration task on the administrator (or someone else) or to try to relieve him of this task by automating some steps and literally annihilate his role in the decision-making process. These two trends are somewhat extreme, as the administrator should have the possibility to configure the caching system without dealing with what can be automated. A suitable alternative should offer a reasonable trade-off between the automation and human intervention.

3 Description of the System

Taking into account the previous considerations, we propose a tool that lightens the administrator's burden, by automating the "systematic" tasks (such as analyzing and extracting the information from the Web pages), and proposing a default fragmentation with the prerequisite reuse and invalidation directives. Nonetheless, these default actions and directives can either be augmented or overwritten by the administrator.

According to a recent study [13], a Web page written in a scripting language (see figure 1) typically consists of a number of code blocks, where each code block performs some work (such as retrieving or formatting content etc.) and produces an HTML fragment as output. A write to out statement, which follows each code block, places the resulting HTML fragment in a buffer.

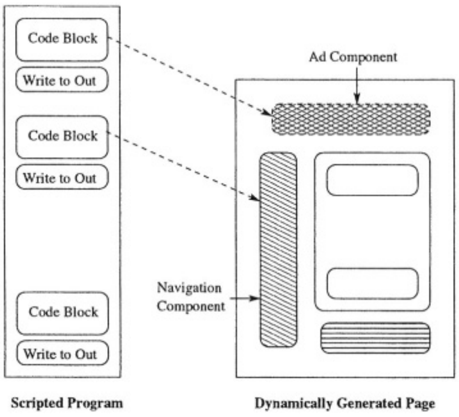


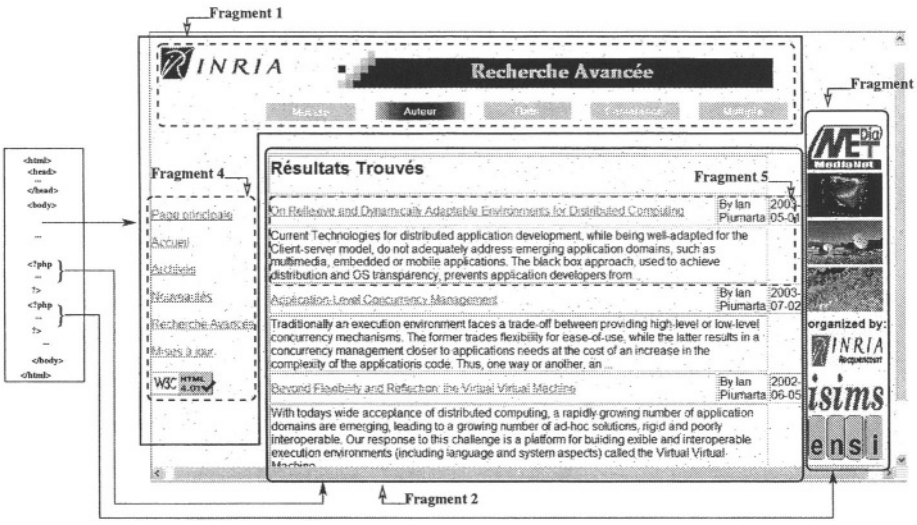
Fig. 1. Dynamic scripting Process

We agree with the definition except that we emphasize the existence of a, greater or smaller, static part.

Any solution that aims to address the key issues of caching dynamic content must, one way or another, collect various types of information relative to the execution of the programs that generate the dynamic pages.
Our approach consists in analyzing statically these programs rather than their outputs. This approach has the considerable advantage of operating once and off-line on the "parent" of the generated pages rather than on each instance of the execution. Such a static analysis paves the way for the automation of the information extraction process as the static code contains the exact set of variables that are actually used, the exact operations made on them and the exact set of database accesses (unlike the generated dynamic pages in which the code has already been executed and the HTML generated).

This modus operandi helps to address the three main issues discussed in section 2.

Firstly, the fragmentation is meant to reduce three parameters: bandwidth consumption, CPU time and storage space. The saving in storage space is achieved by reusing fragments between different pages (as a shared fragment is only stored once), while CPU time and bandwidth consumption can only be reduced when the server is aware of the fragment entity. Actually, CPU time is saved if instead of generating the whole page, the server could only generate the missing and the stale fragments of the documents already cached. The same is



Résultats Trouvés

On Reflexive and Dynamically Adaptable Environments for Distributed Computing	By Ian Piumarta	2003-05-01
Current Technologies for distributed application development, while being well-adapted for the Client-server model, do not adequately address emerging application domains, such as multimedia, embedded or mobile applications. The black box approach, used to achieve distribution and OS transparency, prevents application developers from ...		
Application-Level Concurrency Management	By Ian Piumarta	2003-07-02
Traditionally an execution environment faces a trade-off between providing high-level or low-level concurrency mechanisms. The former trades flexibility for ease-of-use, while the latter results in a concurrency management closer to applications needs at the cost of an increase in the complexity of the applications code. Thus, one way or another, an ...		
Beyond Flexibility and Reflection: the Virtual Virtual Machine	By Ian Piumarta	2002-06-05
With todays wide acceptance of distributed computing, a rapidly growing number of application domains are emerging, leading to a growing number of ad-hoc solutions, rigid and poorly interoperable. Our response to this challenge is a platform for building exible and interoperable execution environments (including language and system aspects) called the Virtual Virtual Machine		

Fig. 2. Example of PHP Application

true for bandwidth consumption, as this parameter can only be reduced if missing and stale fragments of a cached document are sent rather than the entire document. Then, to gain efficiency, it should be possible to fetch the fragments separately. However, this is seldom the case with existing systems.

In the remainder of the paper we will refer to the concrete example of a PHP site on which we made our experiments to illustrate the various problems and solutions. The page illustrated in figure 2 corresponds to a keyword-based search of scientific papers. This figure also schematizes the different kinds of fragments that can be distinguished. Fragment 1, represents the static part of the page. Fragment 2 and Fragment 3 are generated by two different PHP scripts. Each part or grouping of these fragments can be individually considered as a fragment. The problem is then to find the fragments which exhibit potential benefits and thus are cost-effective as cache units. As explained previously, most benefits can be obtained when the server is able to handle the fragments, which is neither the case for this site nor for the majority of sites.

Our system offers the tag of automatically detected fragments as a default, and proposes augmenting the site's capabilities, with minor changes, so as to handle individual fragments. This option is not imposed and must, on the contrary, be explicitly selected by the administrator in order to become active. In both cases the administrator can afterward augment or overwrite the fragmentation mechanism (see section 3.2).
Detecting fragments may simply boil down to detecting the code's independent scripts. Nonetheless, there is one main reason why the delimitation of the scripts may have to differ from the delimitation of the fragments: the Granularity of the fragmentation. The problem particularly arises in keyword-based searches as, in some cases, individual parts of the response may be appropriately considered as

fragments (for instance, in figure 2, Fragment5 may be preferable as a fragment to Fragment2). In such cases, the administrator should have the possibility of tuning the fragmentation accordingly. The detection of the fragments is easier when statically parsing the programs in as much as the instructions that generate the fragments are easier to detect than the rendered HTML fragments themselves (see figure 5). Moreover, the static identification of fragments is done once and is reflected on all the responses generated by the same program. The tag of fragments is done by inserting the appropriate markup instructions into the static part of the page, so that every query to the page generates tagged fragments.

The idea behind the option that makes it possible to handle separate fragments, is to create wrappers of the pages that will keep their logic and the organization while splitting up the independent parts so that they can be called separately. Figure 3 illustrates the splitting up process applied to the PHP page that generated the result in figure 2. In practice, this consists in extracting independent program portions and replacing their occurrence by an include instruction.

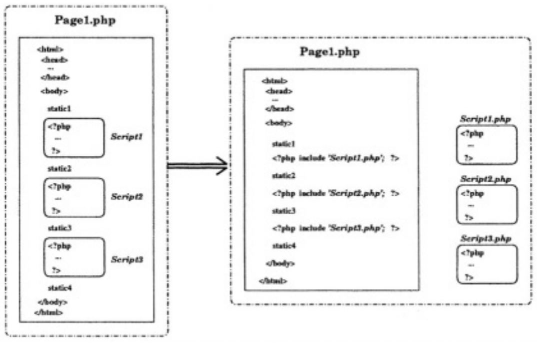


Fig. 3. The splitting-up process

The extracted elements can afterward be called separately. An element can either correspond to a single script or a sequence of scripts. In the best case scenario, each independent block code (of figure 1) corresponds to a single script. This processing results in two major benefits. On the one hand, it leads to the separation of the static content from the dynamic content (which fosters the reuse of the stored cache entries). On the other hand, this makes it possible to ask for a single fragment so that only stale and missing fragments need to be regenerated.

As concerns the second issue (i.e. the reuse of HTTP responses), the static analysis of the code is far more dependable than other techniques. Actually, to define an ad hoc key for the document it is necessary to know all the parameters that affect this particular document. Some studies assume that the URL is a sufficient identifier, but this is obviously not the case with dynamic pages. On the one hand, the URL parameters are not all implicitly meaningful for the processing of the page. On the other hand, the page tends to be dependent on several other variables. To find out which variables a page actually depends on, it is logical to parse the program code and gather the set of meaningful parameters. This echoes one of the cache content invalidation requisites, as to invalidate a page or a fragment, one must know the dependencies that characterize it. The static analysis of the code not only makes it possible to determine the dependencies of a page but also provides the means to carry out the same operation per fragment.

Our system consists of two main modules: the program analyzer which traverses the Web site's directory structure, analyzes the programs code and outputs a high level description of it. A configuration module that utilizes this data to furnish the administrator with a set of default settings as well as providing him with informational data and appropriate tools that allow him to adjust some configuration parameters if he deems it necessary to do so.

3.1 Program Analyzer

The program analyzer statically parses the code of the programs that generate dynamic pages. It goes without saying that to do so the program must handle the scripting language in which the program has been written. This is done by parsers that are intended to abstract the content away from the grammar of the language. Figure 4, depicts the structure of the program analyzer.

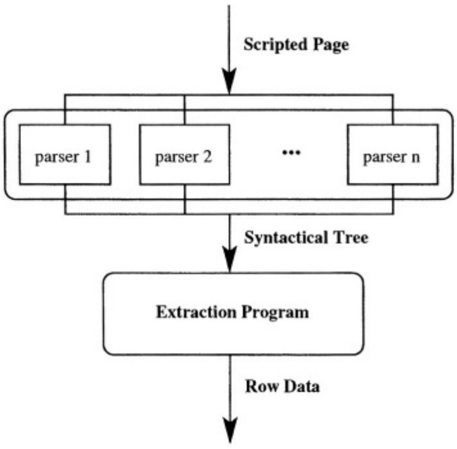
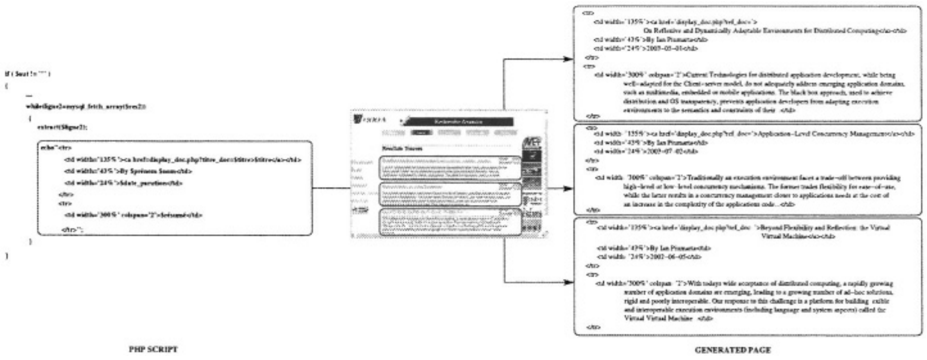


Fig. 4. Structure of the Program Analyzer

The program analyzer takes the scripted pages as input and produces a syntactical tree as output. This syntactical tree describes the operations of the



```
if ( $aut != "" )
{
    ...
    while($gne2=mysql_fetch_array($res2))
     {
      extract($lgne2);
      echo"<tr>
            <td width='135%'><a href=display_doc.php?titre_doc=$titre>$titre</a></td>
            <td width='43%'>By $prénom $nom</td>
            <td width='24%'>$date_parution</td>
          </tr>
          <tr>
            <td width='300%' colspan='2'>$resumé</td>
          </tr>";
     }
}
```

```
<tr>
   <td width='135%'><a href='display_doc.php?ref_doc='>
               On Reflective and Dynamically Adaptable Environments for Distributed Computing</a></td>
   <td width='43%'>By Ian Piumarta</td>
   <td width='24%'>2003-05-01</td>
</tr>
<tr>
   <td width='300%' colspan='2'>Current Technologies for distributed application development, while being
          well-adapted for the Client-server model, do not adequately address emerging application domains,
          such as multimedia, embedded or mobile applications. The black box approach, used to achieve
          distribution and OS transparency, prevents application developers from adapting execution
          environments to the semantics and constraints of their </td>
</tr>
<tr>
   <td width='135%'><a href='display_doc.php?ref_doc='>Application-Level Concurrency Management</a></td>
   <td width='43%'>By Ian Piumarta</td>
   <td width='24%'>2003-07-02</td>
</tr>
<tr>
   <td width='300%' colspan='2'>Traditionally an execution environment faces a trade-off between providing
          high-level or low-level concurrency mechanisms. The former trades flexibility for ease-of-use,
          while the latter results in a concurrency management closer to applications needs at the cost of
          an increase in the complexity of the applications code...</td>
</tr>
<tr>
   <td width='135%'><a href='display_doc.php?ref_doc='>Beyond Flexibility and Reflection: the Virtual
                                 Virtual Machine</a></td>
   <td width='43%'>By Ian Piumarta</td>
   <td width='24%'>2002-06-05</td>
</tr>
<tr>
   <td width='300%' colspan='2'>With todays wide acceptance of distributed computing, a rapidly growing
          number of application domains are emerging, leading to a growing number of ad-hoc solutions,
          rigid and poorly interoperable. Our response to this challenge is a platform for building exible
          and interoperable execution environments (including language and system aspects) called the
          Virtual Virtual Machine </td>
</tr>
```

Fig. 5. Static vs Dynamic Analysis

code freed from the specifics of scripting languages as, almost all the scripting languages express the same set of operations in more or less the same way. This tree is then passed as a parameter to the extraction program that travels through it recording the relevant data for the subsequent processing. A document roughly corresponds to a chained list of scripts descriptions. Each node of the chained list contains at least the following information:

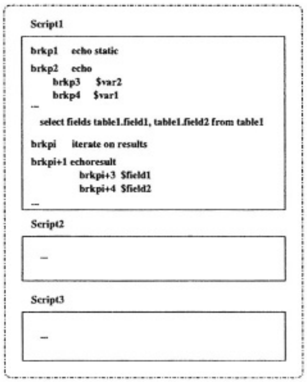
✓ `Script id` : A string identifying the script.
✓ `Beginning offset` : The offset of the script's opening tag.
✓ `End offset` : The offset of the script's closing tag.
✓ `Used Variables` : The set of the variables actually used by the script (these include the GET or the POST arguments, the cookies used etc.).
✓ `Database Accesses` : The details of database accesses, these includes the operation type, the tables and fields that are concerned, etc.
✓ `Functions` : The functions used by the script and their parameters.

3.2 Configuration Module

This module is organized into two functional submodules. The first submodule generates the default configuration file. In the current version, the configuration file concerns fragmentation, purification and invalidation tasks.
The program automating the fragmentation process must be able to detect the beginning and the end of the fragments, in order to tag or extract the right buffers and replace them by the corresponding include instructions.
The program, automating the purification, requires knowledge of the set of parameters affecting the result in order to compute a unique cache-wide key per document. The parameters that may affect the generated pages are:

✓ the arguments that are passed into the request (via the GET or the POST methods),
✓ the fields that appear in the header of the HTTP request (Cookies, User_Agent etc.)
✓ and other variables (such as the client IP).

Finally, the automation of the invalidation process requires knowledge of the pages' dependencies (which consists in the accessed tables and fields in the database). The first module analyzes the output of the program analyzer in order to generate the corresponding configuration file that will keep track of the previous parameters. The second module provides the administrator with intelligible data and appropriate tools that help him to visualize the page's characteristics and to tune the actual default parameters if he considers it necessary. Regarding fragmentation, as the definition of fragments fundamentally depends on the HTML code produced, the administrator needs to know about the output instructions scattered throughout the code in order to adapt the granularity. Of course, he is not supposed to know exactly what is printed (though technically he could), he only needs to know about the breakpoints that can be considered in the code with a quick insight into the semantics.
Figure 6 shows an example of possible breakpoints in the search page. The administrator may then specify (in a predefined language) a set of breakpoints that will delimit the fragments.

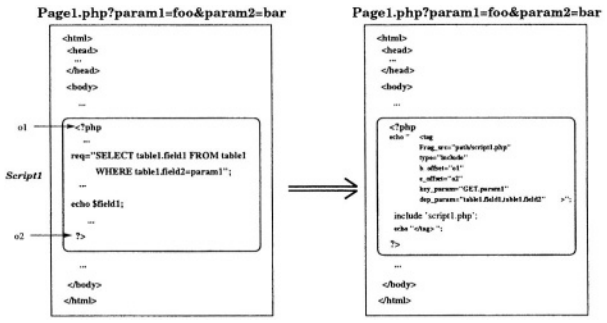


```
Script1
brkp1    echo static
brkp2    echo
      brkp3    $var2
      brkp4    $var1
...
  select fields table1.field1, table1.field2 from table1
brkpi    iterate on results
brkpi+1 echoresult
             brkpi+3  $field1
             brkpi+4  $field2
...
Script2
  ...
Script3
  ...
```

Fig. 6. Informational data

The specified fragments are presumed to be shared among a set of pages but these are not assumed to be separately fetchable, so the main benefit of such fragments is to reduce storage space.
The administrator can also modify the purification parameters (for instance he may deem it unimportant to change the ad banners according to the clients' IP addresses), or the invalidation parameters (as he may consider, for example, that a change in the statistics tables should not invalidate the page). If any modification is specified, the configuration files are updated.

The generated configuration files are used by subsequent caching modules. For instance the fragmentation module makes the appropriate alterations to the Web site directories (in accordance with the configuration files). Figure 7 illustrates the alterations that are made when the default parameters are set and the fragment handling option is activated. It should be noticed that such a result is normally interpreted by "normal" clients as the inserted tags will simply be ignored.


Page1.php?param1=foo&param2=bar
o1
Script1
o2
Page1.php?param1=foo&param2=bar

```
<html>
 <head>
  ...
 </head>
 <body>
  ...
 <?php
  ...
 req="SELECT table1.field1 FROM table1
      WHERE table1.field2=param1";
  ...
 echo $field1;
  ...
 ?>
  ...
 </body>
</html>
```

```
<html>
 <head>
  ...
 </head>
 <body>
  ...
 <?php
 echo "    <tag
           Frag_src="path/script1.php"
           type="include"
           b_offset="o1"
           e_offset="o2"
           key_param="GET.param1"
           dep_param="table1.field1,table1.field2"        >";
  include 'script1.php';
  echo "</tag> ";
 ?>
  ...
 </body>
</html>
```

Fig. 7. Example of page alteration

4 Evaluation

The system we propose can be evaluated according to various criteria. First, we can debate functional aspects such as ease of deployment and the system's automation level. We can also try to quantify the benefits of the approach (such as the reuse rate of the fragments and bandwidth savings), and evaluate the impact of human intervention on the observed gains. Regarding the first point, we ran our program on the site previously referred to, keeping the default calculated parameters while activating the fragment's handling option. We observed that changes were applied in a very reasonable time (although this criterion is not really important as the changes are made off-line). Regarding the level of automation, it should be noticed that the system can be fully automated as it generates the complete set of default parameters that are to be used by the subsequent caching modules, without any human intervention. Nonetheless, human intervention is expected to improve the results as it adds further comprehension of the application's particular case. Concerning the quantification of the gains, at this stage of implementation, it was difficult to quantify the benefits of the approach. We have already developed the system described in this paper, but to have significant metrics, we need to develop the whole cache relying on the latter. We are now working on the development of the complete cache that fully takes advantage of the information returned by the system. Meanwhile, in order to get round the necessity of having a real cache and to give an idea of the benefits, we considered the set of dynamic pages of the site and we evaluated the difference between the mean time to generate the whole page and the mean time to generate a random set of the page scripts (assuming that the other fragments were already cached). It goes without saying that the response time for a single fragment is lower than the response time for all the page. The difference mainly depends on the the number of scripts in the page and the number and type of operations they perform. The main characteristic of the site considered is that the pages contain few scripts and that the majority of these perform heavy accesses to the database. Yet, the execution time of a script still remains lower than that of the whole page. On this site, the execution time of a script is between 10 and 70% faster than the execution of the whole page. When generating a random number of the page's actual scripts, we observed a mean amelioration of 20% of the response time. Finally, we have ascertained that in the worst case scenario, if all the scripts of the pages have to be regenerated each time the page is requested, the response time is not worse than that of the basic site.

The noticeable thing about this result, is that the site is written in PHP, which is a powerful server-side scripting language, and so improving the performances of a PHP site was particularly challenging.

5 Discussions and Perspectives

In this paper we proposed a tool that offers a reasonable trade-off between the automation of the caching configuration tasks and administrator intervention. It should be stressed that the tool can be fully automated even though performance is likely to benefit from human intervention. Our approach is based on the static analysis of the Web site programs so that the analysis is made once and off-line. Our tool offers the possibility of augmenting the site's capabilities so as to manipulate fragments separately. One might consider the modification of the site repository as a drawback. However, this is a minor intrusion as the architecture of the site and the layout of the components remain the same and that only a few instructions are automatically inserted into the code.
We tested our tool in a roundabout way and we achieved an appreciable improvement in the page's response time. We expect to obtain significant benefits when coupling this tool with the adequate cache. Therefore, we are now working on a stronger validation of our approach by developing the cache system that interprets the meta-data inserted into the response and which implements the appropriate processing so as to fully benefit from the potential of this automation. We are also working on the detection of more apposite breakpoints inside the scripts, and finally we are considering the possibility of integrating new parsers into our system in order to handle the maximum number of scripting languages.

References

1. Arun Iyengar Jim Challenger and Paul Dantzig. A scalable system for consistently caching dynamic web data. *Proceedings of the IEEE INFOCOM'99, New York,* 1999.
2. Z. Bar-Yossef and S. Rajagopalan. Data mining and its applications. *Proceedings of WWW-2002,* May 2002.
3. Huican Zhu and Tao Yang. Class-based cache management for dynamic web content. Technical report, University of California Santa Barbara, 2001.

4. Ben Smith Vegard Holmedahl and Tao Yang. Cooperative caching of dynamic content on a distributed web server. Technical report, University of California Santa Barabara, 1998.
5. Anindya Datta Kaushik Dutta Helen Thomas Debra VanderMeer Suresha and Kirthi Ramamritham. Proxy-based acceleration of dynamically generated content on the worl wide web: An approach and implementation. *ACM SIGMOD 2002,* june 2002.
6. J. Challenger A. lyengar K. Witting C. Ferstat and P. Reed. Publishing system for efficiently creating dynamic web content. *Proceedings of the IEEE INFOCOM 2000,* May 2000.
7. Zhigang Hua Chun Yuan and Zheng Zhang. Proxy+ : Simple proxy augmentation for dynamic content processing. Technical report, Microsoft Research Asia, 2003.
8. Lakshmir Ramaswamy Arun Iyengar Ling Liu and Fred Douglis. Techniques for efficient fragment detection in web pages. *Proceedings of the 12th International Conference on Information and Knowledge Management, CIKM 2003,* November 2003.
9. Xiang Liu Jordan Parker Zheng Zeng Jesse Anton, Lawrance Jacobs and Tie Zhong. Web caching for database applications with oracle web cache. *proceedings of the ACM SIGMOD 2002 International Conference on Management of Data,* june 2002.
10. Tao Yang Ben Smith, Anurag Acharya and Huican Zhu. Exploiting result equivalence in cacing dynamic web content. Technical report, Department of Computer Science, University of California, 2000.
11. *Oracle9i data cache.* http://www.oracle.com/ip/deploy/ias/ caching/index.html?database-caching.html.
12. Qiong Luo Wang-Pin Hsiung k Selçun, Candan When-Syan Li and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. *ACM SIGMOD 2001,* May 2001.
13. Kaushik Dutta Krithi Ramamritham Helen Thomas Anindya Datta and Debra VanderMeer. Dynamic content acceleration: A caching solution to enable scalable dynamic web page generation. *Proceeding of the fifteenth ACM Symposium on Operating Systems Principles (SIGOPS),* May 2001.

Towards Informed Web Content Delivery

Leeann Bent[1], Michael Rabinovich[2], Geoffrey M. Voelker[1], and Zhen Xiao[2]

[1] University of California, San Diego
9500 Gillman Dr. MS-0114
La Jolla, CA 92093-0114 USA
{lbent,voelker}@cs.ucsd.edu

[2] AT&T Labs-Research
180 Park Ave.
Florham Park, NJ 07932 USA
{misha,xiao}@research.att.com

Abstract. A wide range of techniques have been proposed, implemented, and even standardized for improving the performance of Web content delivery. However, previous work has found that many Web sites either do not take advantage of such techniques or unknowingly inhibit their use. In this paper, we present the design of a tool called Cassandra that addresses these problems. Web site developers can use Cassandra to achieve three goals: (i) to identify protocol correctness and conformance problems; (ii) to identify content delivery performance problems; and (iii) to evaluate the potential benefits of using content delivery optimizations. Cassandra combines performance and behavioral data, together with an extensible simulation architecture, to identify content delivery problems and predict optimization benefits. We describe the architecture of Cassandra and demonstrate its use to evaluate the potential benefits of a CDN on a large Web server farm.

1 Introduction

A wide range of techniques have been proposed, implemented, and standardized for improving the performance of Web content delivery, from compression to caching to content distribution networks (CDNs). However, previous work has found that many Web sites either do not take advantage of such techniques, or unknowingly inhibit their use. For example, Web sites do not fully exploit the potential of persistent connections [6], and indiscriminate use of cookies can unnecessarily and unknowingly limit the benefits of content delivery optimizations like caching [5]. This gap between potential and practice is due to a number of challenging factors. First, HTTP/1.1 is a complex protocol [22] that requires considerable expertise to fully take advantage of its features. Even achieving protocol compliance is challenging (as evidenced by intermittent success [17]), much less optimizing its use. For instance, HTTP/1.1 provides a number of advanced features for cache coherence so that sites can maximize the effectiveness of downstream caching mechanisms. However, for Web site developers to take full advantage of cache coherence, they must understand the complex interactions of over half a dozen HTTP headers in terms of their standardized semantics, legacy interactions, as well as practical conventions [25]. Second, the use and content of contemporary Web sites have complex requirements that, used naively, can interact poorly with content delivery optimizations. For example, many sites use cookies to personalize, restrict, or track access, but, again, indiscriminate use can severely inhibit downstream caching optimizations. Finally, measuring Web site workloads and evaluating site performance requires considerable time and effort, and evaluating the potential effectiveness of content delivery optimizations rarely extends beyond the realm of research. Top-tier Web sites like Google and Amazon.com may have the resources to aggressively improve their content delivery, but even medium-sized and small Web sites can benefit significantly from a number of improvements [5].

We propose that this gap between potential and practice can be bridged with a general tool for analyzing and predicting Web site performance and behavior. By encapsulating protocol and optimization complexities within a tool, a much broader user base of Web site developers and maintainers can more easily quantify the effect of their content delivery decisions and evaluate potential performance optimizations.

In this paper, we present the design and use of such a tool called Cassandra. Web site developers can use Cassandra to achieve three goals. First, Cassandra can identify protocol correctness and conformance issues, such as properly setting cache control and coherence headers. Second, it can identify content delivery performance problems, such as the indiscriminate use of cookies. Third, it can evaluate the potential benefits of using content delivery optimizations for a particular site, such as tuning cache control headers, using cookies more efficiently, using CDNs, enabling persistent connections, etc. Based on such evaluations, a site developer can decide whether pursuing a particular optimization is worth the effort and expense before committing to it.

To achieve these goals, Cassandra's design combines an extensible architecture of *analysis modules* with site-specific performance and behavioral data. The analysis modules model protocol behavior, such as content cacheability, and simulate performance optimizations, such as compression and CDNs. To apply these analyses to a particular site, Cassandra can use a combination of raw packet traces, Web server logs, and active probing to gather workload data as input.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the architecture of Cassandra and outlines its analysis modules. Section 4 describes the prototype implementation of one analysis module, the analysis of CDN benefits. Section 5 presents a case study that applies the CDN analysis module in Cassandra to the top 100 Web sites in a large server farm. Section 6 concludes this paper and outlines directions for future work.

2 Related Work

Most of the previous studies that analyze Web sites were conducted on a small scale (sometimes just a single, highly popular site) [13,17,20,21,24]. Some of them (e.g., [17,20,24]) only consider access patterns to root pages. Moreover, they typically only focus on a specific aspect of the Web site. For example, the work in [17] focuses primarily on protocol compliance, while the work in [20] focuses on persistent and parallel connection usage. While these studies may yield valuable insights on certain Web site design issues, their results are not as comprehensive as those obtainable a tool like Cassandra. The benefits of using a CDN for content delivery were previously studied in various contexts. Jung et al. investigate the ability of a CDN to protect a Web site from a flash crowd by analyzing known flash events that occurred on two Web sites [14]. Krishnamurthy et al. conducted a comparative performance study on the download time of pre-selected pages through various CDNs [19]. In contrast, the focus of Cassandra is to evaluate the general effects of the CDN on an entire Web site.

There are several existing tools that measure Web server performance by generating various HTTP workloads. httperf is a commonly used tool that provides a flexible interface for constructing various benchmarks [23]. It supports the HTTP/1.1 protocol and can be easily extended to new workload generators and performance measurements. Similar to httperf, Web-Polygraph is a tool for constructing Web benchmarks [28]. Web-Polygraph includes standard workloads generated from real Web content and supports HTTP and SSL content. SURGE (Scalable URL Reference Generator) generates references that match empirical measurements of various request distributions (e.g., request size, relative file popularity, temporal locality) [4]. Banga et al. proposed a method for Web traffic generation that can create heavy, concurrent traffic through a number of client processes. It can generate bursty traffic with peak loads that exceed the capacity of the server [3]. Although such tools are helpful in measuring server performance and protocol compliance, they use synthetic workloads and thus are not specific to a given Web site. In contrast, Cassandra produces site-specific recommendations and a comprehensive set of "what-if" analyses, which greatly facilitate the understanding and improvement of Web site performance.

There are also numerous commercial tools and services for monitoring and testing the performance of Web sites. These generally fall into two categories. The first set of tools are designed to monitor the current state or performance of a Web site. They answer questions such as "Is my Web site up?" or "How long is an average transaction on my web taking from South America?". Some examples of these are Keynote's Web Site Perspective [15] and AlertSite's Server/Website Monitoring [1]. The second set are tools designed to stress test the performance of a Web site, answering questions such as "How should I provision my site for search transactions?". Some examples include Keynote's Performance tune, Empirix's e-Test suite [10], and AlertSite's Web Load Testing Services. These tools focus on measuring server and transaction performance. While useful for analyzing existing Web site performance, these tools do not easily facilitate "what-if" analysis like Cassandra.

Finally, there are free tools for testing the performance and cacheability of Web pages, e.g `www.web-caching.com` [7] or `www.websiteoptimization.com` [29]. While cachability is only one aspect of the Cassandra tool, Cassandra also provides a means for detecting pervasive problems throughout a Web site rather than on individual pages.

3 The Cassandra Architecture

The high-level goal of the Cassandra tool is to make it easier for Web site developers and maintainers to both quantify the effect of their content delivery decisions, as well as to evaluate potential optimizations for improving content delivery performance. We envision Cassandra helping Web sites achieve these goals in three ways.

Protocol Compliance. HTTP is a complex protocol with complicated semantics, particularly with respect to caching and coherence. Due to the nature in which the HTTP protocol has evolved over time, taking full advantage of protocol features is a difficult task. In addition to a detailed understanding of standardized protocol semantics, it also requires understanding how protocol features interact with legacy implementations of caches and browsers as well as practical conventions for dealing with ambiguous aspects of the protocol [25]. Cassandra could be used to evaluate protocol compliance and report anomalous behavior, such as inconsistencies among the cache-control headers.

Performance Debugging. Web sites can make content delivery decisions that can unknowingly have a profound impact on site performance. For example, when many Web sites use cookies, they often implement cookie use by requiring all requests to the site to use a cookie [5]. Using cookies in this fashion is often unnecessary, yet severely limits the cacheability of a site's content. Cassandra can identify such behavior, estimate the performance impact of a site's use of cookies (e.g., additional server load and bandwidth consumed), and identify content that may not require cookies (e.g., embedded images versus container pages).

Optimization Evaluation. A wide range of technologies have been developed over time to improve Web content delivery, such as compression, prefetching, and CDNs. However, most of these technologies are only selectively deployed, if at all. A significant hurdle for adopting a technology is evaluating to what extent it would benefit a particular Web site. Cassandra could be a valuable tool for Web site operators to address this problem by facilitating the evaluation of various technologies on a particular Web site. Based on the results of such an evaluation, Web site operators can make an informed decision regarding the optimal content delivery methods for their Web site.

We have implemented an initial prototype of Cassandra, as well as an analysis module for evaluating the benefits of using CDNs for Web sites. In the rest of this section, we discuss the Cassandra architecture and outline two other analysis modules that we plan to implement. Then, in Sections 4 and 5, we describe in detail the CDN analysis module that we have implemented and evaluated as an example demonstration of using Cassandra.

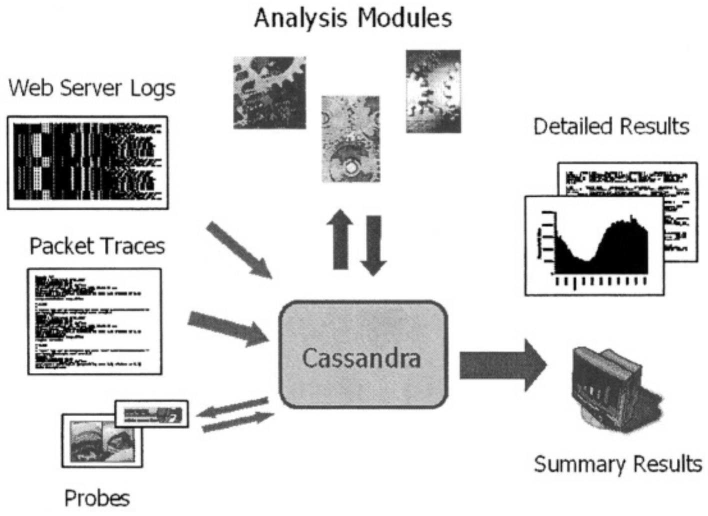


Fig. 1. Cassandra Architecture.

3.1 Architecture

To achieve the above goals, Cassandra's design combines an extensible architecture of *analysis modules* with site-specific performance and behavioral data. As illustrated in Figure 1, Cassandra provides a framework for incorporating workload data as input, evaluating that data according to specified analyses, and providing support for producing user feedback in terms of tabular and graphical results of analyses as output. In our usage model, Web site maintainers gather the workload data, invoke Cassandra to perform the desired analysis, and interpret the evaluation results to decide whether they should make content delivery changes to their site.

The ability of Cassandra to perform its analyses depends on the site workload data available to it. Cassandra can use any combination of raw packet traces (such as from the Gigascope network appliance [9] or `tcpdump` [27]) , Web server logs, and active probing for workload analysis. Packet traces provide the most detailed workload data, since they include headers, payload, and timing information. If packet traces are unavailable, Cassandra can use Web server logs to obtain higher level workload information, such as identifying popular content or content that affects Web site performance most significantly. After identifying important content, Cassandra can combine active probing or short `tcpdump` traces with server logs to obtain header and performance data. This detailed data drives its performance analyses.

Cassandra uses an extensible architecture of *analysis modules* to support a large range of Web site analyses. The analysis modules model protocol behavior and simulate performance optimizations. For example, a coherence module can model the cache coherency policies and mechanisms of HTTP, and a CDN module can model the potential benefits of using a CDN for a Web site. Cassandra provides a framework in which analysis modules can be developed and used independently, as well as in combination. For example, Cassandra implements a Web cache simulator, and this simulator can be used in combination with other analyses such as those that modify the cacheability of site content.

Our goal is for Cassandra to be able to support a wide range of Web site analyses and content delivery optimizations, from caching to prefetching to persistent connections. We are developing modules for a wide range of common analyses, such as caching, cache coherence, compression, and CDN usage. The Cassandra architecture also makes it straightforward for other researchers to extend Cassandra with their own analyses.

In addition to analyzing individual Web sites, Cassandra supports the simultaneous analysis of multiple Web sites using workloads containing interleaved accesses to those sites, and the reporting of results on a per-site basis. Such support is useful for situations where a hosting service provider wants to analyze performance across a number of its Web site customers. With this support, Cassandra avoids the need to split the trace into individual Web site accesses and simplifies the per-site analysis. We use this feature for our case study in Sections 4 and 5.

3.2 Cacheability Analysis Module

A cacheability analysis module can evaluate the use of a Web site's cache-coherence mechanisms and policies. Content cacheability is important because it increases the effectiveness of downstream proxies and browser caches. Our previous work in [5] motivates a number of problems that this analysis module can identify. For example, in that workload study only a small fraction of responses (9%) used cache-controlling headers, thereby requiring downstream caches to use heuristics and historical practices to provide coherency for the objects in the responses. Explicit and pervasive use of such headers can greatly assist caches in managing cached content, further reducing server load and bandwidth. The cacheability analysis module can identify content that could benefit from using cache-controlling headers, model the effect of using those header on a site's workload, and predict the caching benefits of doing so. As another example, sites often use cookies indiscriminately on their content. Such indiscriminate use severely limits the benefits of caching, and is often unnecessary. For instance, when using cookies to track user accesses, it is typically sufficient to track accesses to container pages and infer accesses to embedded objects. The cacheability analysis tool can measure cookie use, flag excessive usage, identify content that may not require cookies (e.g., embedded objects), and predict the caching benefits of using cookies in a more informed approach.

The cacheability analysis module can perform other analyses as well. It can check that the cache control headers are internally consistent (e.g., `Date`, `Age`, `Cache-Control` and `Expires` headers). And the module can predict the effect on overall workload of tuning time-to-live (TTL) values. For example, repeated `If-Modified-Since` requests to an object might indicate that the site is setting the object's TTL value too low, and Cassandra can estimate the resulting impact on server load and bandwidth consumption of such settings.

Cassandra can use object headers, obtained by either probing objects on a Web site or by looking at packet traces, to obtain cacheability information about the objects on a Web site. Packet traces would allow both request and response object headers to be examined; probing would only allow response object headers to be examined. Such headers include `Pragma` headers, `Cache-Control` headers, `Cookie` headers, `Age`, `Expires`, and `Last-Modified`. The cacheability analysis module would use these headers, combined with object access information, to identify and report specific problems with a Web site's content cacheability or delivery. Object cacheability would be determined according to the HTTP/1.1 specification [12] as well as known prevalent caching policies used in practice.

3.3 Compression Analysis Module

A compression analysis module for Cassandra can simulate the benefits of using compression on some or all of the content of a Web site. Content compression is useful because it can reduce the bandwidth demands of a Web site, as well as reduce the download latency of objects. For example, Google compresses the results of search queries to reduce the number of packets required to return search results [8]. The goal of this module is to estimate the benefits of compression (bandwidth and latency reduction) as well as the costs of using the optimization (e.g., additional server load).

Since compressing all content on a Web site might be prohibitively expensive, we plan to model compression of only a subset of objects on a site. These objects can be chosen according to their popularity and contribution to bandwidth consumption as determined from server logs or packet traces. To estimate compressibility, we intend to use Cassandra's site prober to actually download objects or `tcpdump` to reconstruct these objects from network traces. After obtaining the objects, we will apply a common compression utility to them. Once objects are obtained and compressed, we have a measure of object size in bytes, both with and without compression. We can compute bandwidth reduction for a particular site or group of sites by replaying access logs with both compressed and uncompressed objects.

In addition, we can also consider compression in conjunction with CDN usage. Most Web access are image downloads, which are already in a compressed format, thus undermining the effectiveness of compression. However, while images are responsible for most downloads, they are also the object most amenable to CDN caching. For a site using a CDN, most of the residual (non-image) content is compressible. Further, static content can be stored in compressed form and does not incur computation overhead, while compression of dynamic content might have lower relative overhead because it already requires more computation to be generated. Cassandra can test whether objects are dynamic by the presence or absence of the `Last-Modified` header together with the response code. This heuristic can be used to determine whether it would be beneficial to compress the object, and how to do so. We can study the effect of using a CDN (both ideal and realistic) with compression, similar to the way we study the effect of using a CDN with cacheability improvements. Cassandra would compute the benefit of applying compression with a CDN by compressing those objects as identified above, and using those compressed objects in the CDN simulation as in Section 4.

Compression can also be used to decrease client download times. It is a little more difficult to compute download latency and we do not lay out a complete blueprint here. However, we can make a preliminary estimate of download latency improvements for individual objects by comparing the download time of the original objects with objects that are approximately the same size as compressed objects.

Thus, Cassandra will be able to analyze realistic benefits of compression for a Web site, taking into account pre-compression of static objects, and considering compression in combination with CDN caching. Since the CPU overhead of compression depends on the exact platform used, Cassandra can be used in initial analyses to ascertain whether compression would be worthwhile to consider. In addition, some CDNs offer compression services, and Cassandra could help Web site administrators weigh the benefits promised by these services.

4 Analysis of CDN Benefits

Content delivery networks (CDNs) deploy caches throughout the Internet to move content closer to the client and decrease client access time. CDNs are also used to decrease hit rate and bandwidth consumption at the Web site. Without Cassandra, Web site operators must decide whether to subscribe to a CDN based on faith or intuition. Cassandra provides the means to make an informed decision.

4.1 CDN Analysis Module

Cassandra currently implements a model of a full-time revalidation CDN, which assumes that all user requests are routed through CDN caches, and uses a standard time-to-live (TTL) approach to decide the validity of cached responses. For expired responses, Cassandra's CDN module simulates `If-Modified-Since` requests. The CDN analysis module in Cassandra simulates the benefits of using a CDN through trace-driven simulation. Our current implementation uses a packet-level trace gathered using the Gigascope appliance [9]. The trace includes the first packet of every request and response and contains all HTTP header information. In the future, we forsee using a less rich Web access logs as input, augmenting them with Web site probing or short `tcpdump` traces as discussed in Section 3. We assume that CDN caches have unlimited cache capacity given plentiful disk resources.

To map clients to CDN caches, we group clients into clusters using a network-aware clustering tool [18]. Cassandra assigns clients to CDN caches at the granularity of clusters: all clients in the same cluster forward requests through the same cache. Client clusters are randomly assigned to a cache when the cluster's first client generates its first request, and that cache is used throughout the simulation. We do not measure latency effects in this study, hence a different cache assignment will not change our results in any significant way.

The Cassandra CDN analysis module generates estimates of the lower and upper bounds of the potential benefits of using a CDN on a workload. It generates the upper bound by assuming ideal content cacheability: it assumes unlimited lifetimes for cached objects and allows CDN caches to serve all subsequent requests to an object. It generates the lower bound by modeling actual content cacheability and consistency: it simulates the effects of existing cookie and cache-controlling headers in the trace, and considers the trace response sizes in simulating bandwidth consumption. In practice, CDNs will fall between these bounds. The upper bound is idealistic because it assumes that all content is cacheable and never changes. The lower bound is overly conservative in terms of bandwidth consumption because a shared CDN cache can convert some requests (e.g., those with cookies) from unconditional downloads to `If-Modified-Since` requests.

Web sites can benefit from these results in two ways. First, the lower bounds indicate the potential benefits of using a CDN on its current workload. Second, the upper bounds indicate to what extent the site could benefit from improving the cacheability of its content (e.g., by making more informed, targeted use of cookies), and how such changes increase the benefits of using a CDN. Based upon these results, sites can decide whether it is worthwhile to pursue the use of a CDN, or to improve the cacheability of its content.

Due to the highly extensible nature of Cassandra, one can easily add modules for other flavors of CDNs, such as server-driven invalidation CDNs, content push CDNs, or overflow CDNs (which is engaged by a Web site dynamically at peak loads). A discussion of the trade-offs involved among different flavors of CDNs can be found in [11].

4.2 Validation

Eventually, we will validate the simulation models used in the Cassandra tool with real implementations. Our approach is to do this validation on a per-analysis module basis. Here, we explain how we could validate one optimization: CDN caching. We show the validation architecture in Figure 2. The validation setup contains a workload generator driven by requests from the original trace, the server stub, which supplies responses of the proper size and with proper HTTP headers taken from the trace, and a real CDN cache in the middle. For the current CDN analysis module we are only interested in validating hit rate and byte rate for CDNs. These two metrics are straightforward to compute based on proxy cache statistics.

The workload generator, or simulated clients, could be any tool which allows replay of the request stream such as Medusa [16], httpperf [23], or wget [30]. The server stub would mirror either the actual response content from the Web site or mimic responses of the appropriate size. These objects would be served with the

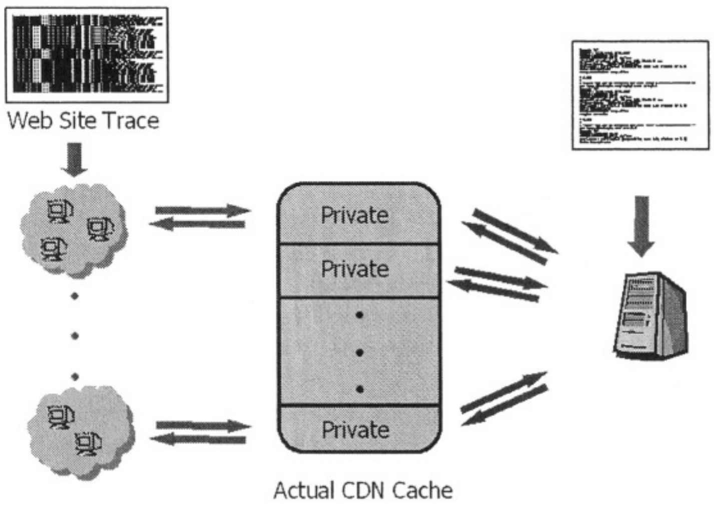


Fig. 2. Validation of Cassandra recommended improvements.

appropriate headers taken from the trace, modified as necessary to change the cacheability properties of the objects. The server stub would serve these objects using either a standard Web server implementation, such as the Apache HTTP Server [2], modified as necessary, or a custom Web server stub implementation. Finally, the CDN cache would be a real proxy cache implementation, such as the Squid Proxy Cache [26]. For our validation, both the server stub and the CDN cache will operate on separate machines connected by a LAN. The workload generator will also be connected via LAN to the CDN cache, however, depending on the workload requirements, more than one machine may be necessary.

The main challenge in validating a CDN simulation is to model accurately a large number of CDN caches. Clearly, it is impractical to replicate the entire CDN network. Our trick to do this is as follows. Our stub origin server inserts a `Cache-Control:Private` header in every response it sends to the sole CDN cache we are using. This header allows our cache to store and use the responses for repeated requests from the same client, but prevents sharing cached responses across clients. In effect, it compartmentalizes the cache into partitions serving each client separately. Further, we assign each client in the trace to a CDN cache in our target CDN configuration, so we end up with client groups corresponding to each CDN cache. Finally, we associate a "pseudo-client" with each of these client groups. When sending a request to the cache, our workload generator replaces the real client information in the request headers with the pseudo-client associated with the client group to which the real client belongs. Thus, the cache will use cached responses to service all requests from the same pseudo-client. In other words, the cache partitions are done for pseudo-clients, not real ones. Since each pseudo-client aggregates all requests that would have gone to the corresponding CDN cache in the target CDN, this results in exactly the same caching behavior assuming our validation cache has enough cache space.

5 Results of CDN Analysis

To provide a concrete example of using Cassandra, we now use it to evaluate the potential benefits of an optimization on Web sites. In this case study, we use a simulator of CDN caches as the optimization and apply it to a workload of HTTP requests and responses to and from a Web server farm in a large commercial hosting service of a major ISP. We simulate the use of 20 CDN cache nodes based on a known CDN provider's configuration. We focus on the top 100 Web sites in the following analyses because these sites are the ones most impacted by the use of a CDN. This trace workload consists of over 17 million request/response pairs to 3,000 commercial Web sites over the course of 21 hours in July, 2003 [5].

For the experiments in this paper, we make the following modifications to the trace. Our CDN simulations require cacheability information, which is not present in all responses, specifically `304 Not Modified` responses (other responses with no cacheability information are negligible [5]). When a `304 Not Modified` response is the first response to a request for an object, our simulation must discard it, since this response contains no meaningful information for our CDN cache (remember that we are simulating `If-Modified-Since` requests for objects once they are in-cache). Subsequent responses for this same object may contain cacheability information, which we use. This reduces the range of requests to 2.04 million requests for the most popular site and less than 100 requests for the least popular site. The request rates to the 100 most popular Web sites change only slightly.

5.1 Peak Request Rate

We start by examining the impact on the peak request rates of the Web sites when using a CDN. Reducing the peak request load on origin servers is one of the key benefits of using a CDN. Figures 3 and 4 show the peak request rate of the 100 most popular Web sites with and without using a CDN. Figure 3 shows peak request rates using the actual content cacheability characteristics of the requests and responses in the trace. Figure 4 shows peak request rates assuming content is ideally cacheable: cached objects have unlimited lifetimes, and CDN caches can serve all subsequent requests to an object. Note that Figure 4 repeats the optimistic CDN simulation from [5], but `304 Not Modified` request/response pairs are removed when they are the first access to the object. Figure 3 shows new results from a more advanced simulator that accurately models content cacheability and consistency in detail.

We compute the peak request rate for each Web site at the granularity of 10 seconds across the entire trace. Each bar corresponds to a Web site and the Web sites are shown in order of decreasing request popularity. The entire bar shows the peak request rate for that Web site across our trace without using a CDN. The dark part of each bar corresponds to the peak request rate of that Web site when a CDN is used. The white part of each bar shows the portion of

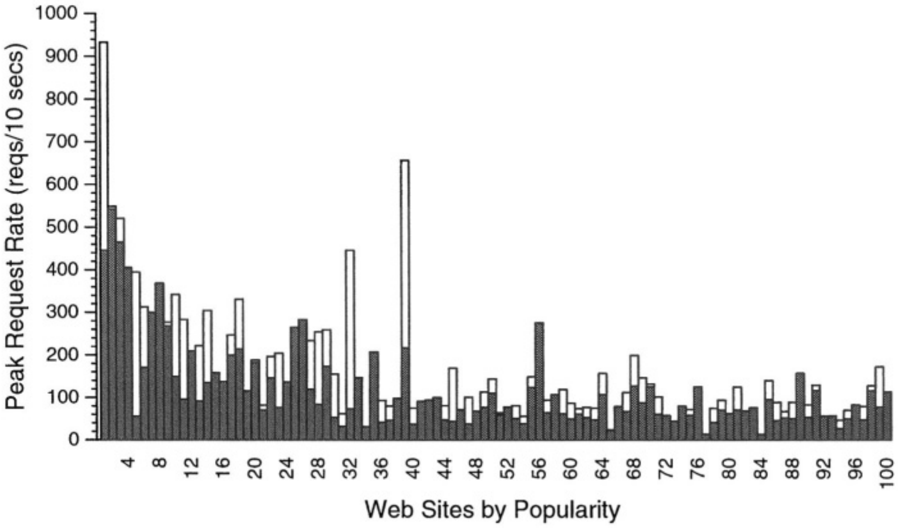


Fig. 3. Comparison of peak request rate with a CDN (dark part of each bar) and the peak request rate without a CDN (total height of each bar). Models content cacheability and consistency.

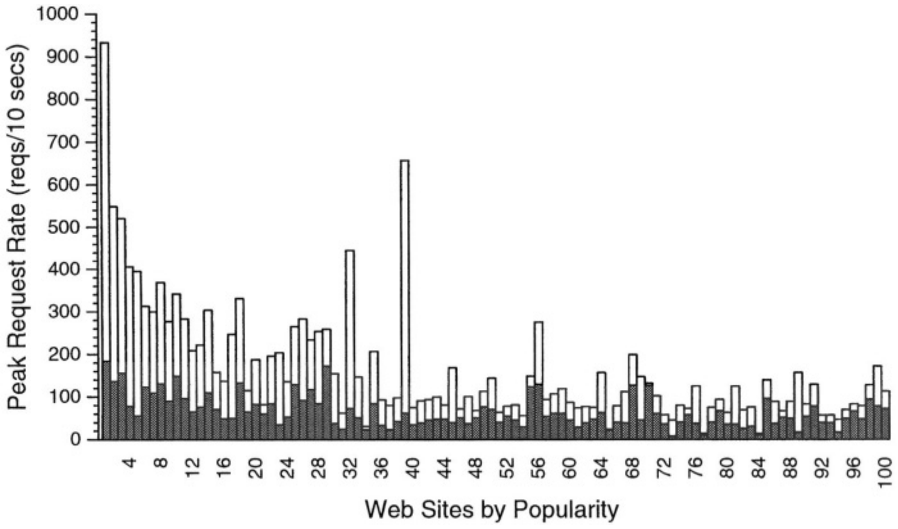


Fig. 4. Comparison of peak request rate with a CDN (dark part of each bar) and the peak request rate without a CDN (total height of each bar). Assumes ideal cacheability. Similar to [5].

the peak request rate handled by the CDN. It shows directly the benefit of using the CDN for that Web site in terms of the reduction in peak request rate.

Both the maintainers of the Web server farm and individual Web sites can use results such as these to determine the potential benefits of using a content distribution network to serve their content. The Web server farm can determine the effect of using a CDN for all sites it serves. For the 100 most popular sites shown in Figure 3, a CDN would decrease the peak request rate across all sites by a factor of 1.4. If the sites improved the cacheability of their content, in the ideal case a CDN would decrease the peak request rate across all sites by a factor of 2.5, a 79% improvement.

Apart from the Web server farm as a whole, individual sites can use these results to make independent decisions about whether to use a CDN to help deliver their content. Since some sites benefit significantly from using a CDN while others do not, different sites will arrive at different conclusions. For example, Figure 3 shows that the most popular site has its peak request rate more than halved (reduced by 52%) when using a CDN, but the second most popular site achieves little benefit (reduced by 1.1%).

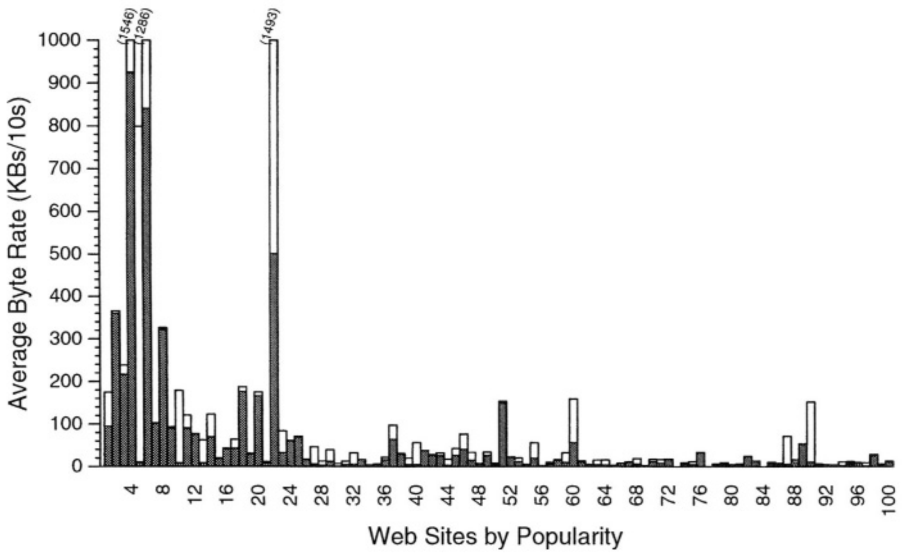


Fig. 5. Comparison of average byte rate with a CDN (the dark part of each bar) and the average byte rate without a CDN (total height of each bar). Models content cacheability and consistency.

Similarly, each site can compare its more realistic performance with its ideal performance to determine whether (1) it is worthwhile to improve the cacheability of its content, e.g., by making more informed use of cookies, and (2) it can gain additional benefit from using a CDN as a result. For example, by comparing

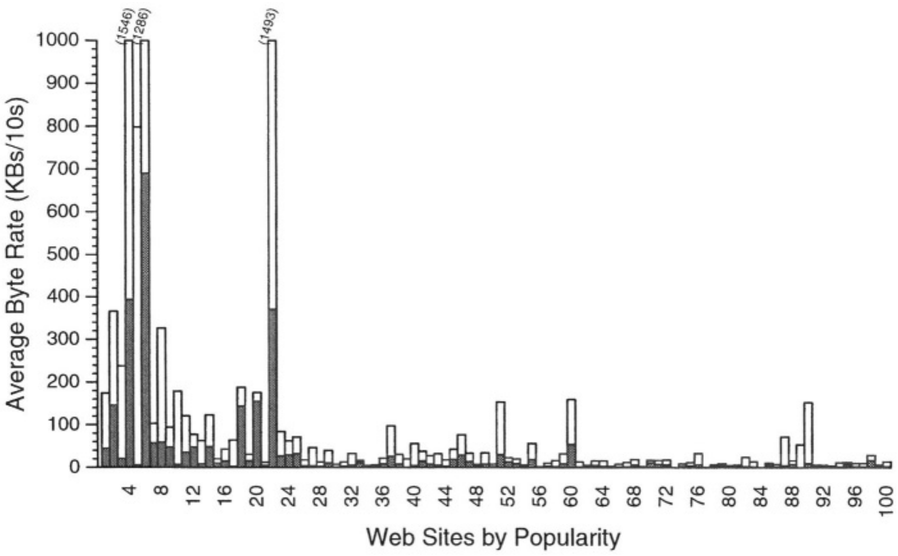


Fig. 6. Comparison of average byte rate with a CDN (the dark part of each bar) and the average byte rate without a CDN (total height of each bar). Assumes ideal cacheability. Similar to [5].

Figures 3 and 4 we see that the second most popular site significantly benefits from improving the cacheability of its content for delivery by a CDN. With current cacheability, a CDN decreases the site's peak request rate by only 1.1%, but by improving the cacheability of its content a CDN could decrease the site's peak request rate by 75% in the ideal case (an improvement of 67%). Of course, the ideal case is an upper bound that may be difficult to achieve completely in practice. Nevertheless, a site can use the difference between the realistic and ideal cases to determine whether it is worth the effort to examine and consider the cacheability of its content. In the case of the second most popular site, it appears worthwhile, while for several other sites (such as 70th and 97th most popular sites) it does not seem to be the case.

5.2 Average Byte Rate

CDNs also reduce the bandwidth requirements for Web sites, in addition to reducing server load by alleviating peak request rates. Reducing bandwidth can directly reduce costs for both individual Web sites and for Web server farms. To evaluate the impact of CDNs on bandwidth requirements, we study the impact of our simulated CDN on the average byte rate of the Web sites in our trace. Figures 5 and 6 show the average byte rate for the 100 most popular Web sites in our trace with and without a CDN. Figure 5 shows the average byte rate using the actual content cacheability characteristics of the requests and responses in the trace, and Figure 6 assumes content is ideally cacheable. Note that Figure 6 repeats the optimistic CDN simulation from [5] where `304 Not Modified` request/response pairs are removed when they are the first access to the object. Figure 5 shows new results from a more advanced simulator that accurately models content cacheability and consistency in detail. We compute the average byte rate for a Web site by totaling the header and content lengths of all transactions to the site in the trace, then dividing by the trace duration. Each bar corresponds to a Web site. The height shows the average byte rate for the Web site without a CDN. The dark part shows the average byte rate at the Web site when using a CDN, while the white part shows the average byte rate handled by the CDN.

As above, the Web server farm and individual sites can use these results to evaluate the potential bandwidth improvements of using a CDN to deliver content. For the sites shown in Figure 5, a CDN would decrease total server farm bandwidth by a factor of 1.8. These results model the content cacheability and consistency found in the original trace, and represent a lower bound on the benefits of a CDN. In contrast, for the results shown in Figure 6 a CDN would decrease bandwidth by a factor of 3.3. These results model ideal content cacheability, and represent an upper bound. From both graphs, we see that using a CDN for the server farm can significantly reduce bandwidth given its current workload, and that improving the content cacheability of the sites has the potential to make a CDN even more effective for the server farm.

Individual sites can also use these results to determine the impact of using a CDN for their content. For example, as with the peak request rate metric, the second most popular site achieves little benefit in terms of bandwidth on its current workload (Figure 5), but can potentially gain considerable bandwidth improvements by improving the cacheability of its content (Figure 6).

6 Conclusion and Future Work

This paper describes Cassandra, a tool for analyzing the benefits of various performance optimizations to a Web site. The main advantages of Cassandra include extensibility, so that the tool can add new optimization technologies as they become available, the ability to obtain detailed workload characteristics of a Web site, and a user interface to simplify the “what-if” analysis. As a preliminary demonstration of the tool utility, we applied it to analyze the benefits from content delivery networks that a large server farm and individual Web sites hosted on that farm can expect. Web site operators currently have to either rely on intuition in deciding on whether or not to subscribe to CDN services, or to resort to ad-hoc performance analysis, which require considerable expertise and time. Cassandra will make this kind of analysis more straightforward.

Our future work includes extending the analysis module library to incorporate more optimization techniques outlined in Section 3, most notably compression. We also plan to demonstrate the use of Cassandra on a real Web site and on different kinds of workloads. Our ultimate goal is to make Cassandra a valuable tool for Web site operators to deliver Web content in an informed manner.

Acknowledgments. We want to thank Oliver Spatscheck for many helpful discussions and suggestions during this project. We would also like to thank the anonymous reviewers for comments on an early draft of this paper. Bent was supported in part by AT&T sponsorship of the UCSD Center for Networked Systems, and Voelker was supported in part by AFOSR MURI Contract F49620-02-1-0233.

References

1. AlertSite. http://www.alertsite.com/.
2. The Apache HTTP server. http://httpd.apache.org/.
3. G. Banga and P. Druschel. Measuring the capacity of a Web server. In *Proc. of the USENIX Symp. on Internet Technologies and Systems,* 1997.
4. P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proc. of ACM SIGMETRICS,* 1998.
5. L. Bent, M. Rabinovich, G. M. Voelker, and Z. Xiao. Characterization of a large Web site population with implications for content delivery. In *Proc. of the 13th International World Wide Web Conference,* May 2004.
6. L. Bent and G. M. Voelker. Whole page performance. In *Proc. of the Seventh International Workshop on Web Content Caching and Distribution,* Aug. 2002.
7. Cacheability tools. http://www.web-caching.com/tools.html.
8. Y.-C. Cheng, U. Hoelzle, N. Cardwell, S. Savage, and G. M. Voelker. Monkey see, monkey do: A tool for TCP tracing and replaying. In *Proc. of the USENIX Annual Technical Conference,* June 2004.
9. C. Cranor, T. Johnson, and O. Spatscheck. Gigascope: a stream database for network applications. In *Proc. of ACM SIGMOD,* June 2003.
10. Empirix. http://www.empirix.com/.
11. Z. Fei. A novel approach to managing consistency in content distribution networks. In *Proc. of Web Caching and Content Distribution Workshop,* 2001.
12. R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1 RFC 2616, 1998.
13. A. K. Iyengar, M. S. Squillante, and L. Zhang. Analysis and characterization of large-scale Web server access patterns and performance. *World Wide Web,* 2(1-2):85–100, June 1999.
14. Y. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and Web sites. In *Proc. of the 11th International World Wide Web Conference,* May 2002.
15. Keynote. http://www.keynote.com/.
16. M. Koletsou and G. Voelker. The medusa proxy: A tool for exploring user-perceived Web performance. In *Proc. of the 6th International Web Caching Workshop and Content Delivery Workshop,* June 2001.
17. B. Krishnamurthy and M. Arlitt. PRO-COW: Protocol compliance on the Web: A longitudinal study. In *Proc. of the 3rd USENIX Symp. on Internet Technologies and Systems,* pages 109–122, 2001.
18. B. Krishnamurthy and J. Wang. On network-aware clustering of Web clients. In *Proc. of ACM SIGCOMM,* Aug. 2000.
19. B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proc. of the First ACM SIGCOMM Internet Measurement Workshop,* pages 169–182, Nov. 2001.

20. B. Krishnamurthy and C. E. Wills. Analyzing factors that influence end-to-end Web performance. *Computer Networks,* 33(1–6):17–32, 2000.
21. S. Manley and M. Seltzer. Web facts and fantasy. In *Proc. of the USENIX Symp. on Internet Technologies and Systems,* pages 125–133, Dec. 1997.
22. J. Mogul. Clarifying the fundamentals of http. In *Proc. of the 11th International World Wide Web Conference,* pages 444–457, May 2002.
23. D. Mosberger and T. Jin. httperf – a tool for measuring Web server performance. In *Proc. of Workshop on Internet Server Performance,* 1998.
24. V. N. Padmanabhan and L. Qiu. The content and access dynamics of a busy Web site: Findings and implications. In *Proc. of ACM SIGCOMM,* Aug. 2000.
25. M. Rabinovich and O. Spatscheck. *Web Caching and Replication.* Addison Wesley, 2002.
26. The Squid Web proxy cache. http://www.squid-cache.org.
27. tcpdump. http://www.tcpdump.org/.
28. Web-Polygraph. http://www.web-polygraph.org/.
29. Websiteoptimization.com.
    http://www.websiteoptimization.com/services/analyze/.
30. Wget. http://www.gnu.org/software/wget/wget.html.

Unveiling the Performance Impact of Lossless Compression to Web Page Content Delivery

Jun-Li Yuan[1,2] and Chi-Hung Chi[1]

[1] School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
{yuanjl, chich}@comp.nus.edu.sg
[2] Institute for Infocomm Research (I^2R)
21 Heng Mui Keng Terrace, Singapore 119613
junli@i2r.a-star.edu.sg

Abstract. User's perceived latency for web content retrieval is always a big concern to web users and content delivery and distribution network service providers. To improve the performance of web content retrieval, researchers are actively looking into mechanisms which accelerate the downloading process of retrieval objects and pages as the performance of caching-based mechanisms is limited. Compression is one important mechanism among these mechanisms. In this paper, we report our comprehensive study on the effect and implication of compression in web content delivery. Our results show that pre-compression almost always performs better than real-time compression because it reduces not only the retrieval latency of COs, but also the DT times of EOs in a page.

1 Introduction

User's perceived latency for web content retrieval is always a big concern to web users and content delivery and distribution network service providers. To improve the performance of web content retrieval, caching [1] and prefetching [2] have been introduced. However, the performance of these caching-based mechanisms is limited due to the characteristics of web traffic and the cachability of web objects [3] and [4]. To overcome the limitation, researchers are actively looking into mechanisms which accelerate the downloading process of retrieval objects and pages. Examples of such mechanisms include persistent connection [5], bundling [6], delta encoding [7], and compression [8] etc. These mechanisms are believed to have good potential because they cover a wider range of objects and pages.

In this paper, we would like to investigate the effect and implication of compression in web content delivery. Here the term "compression" means a mechanism which applies a lossless compression algorithm on textual web objects. The support for such compression has been included in HTTP 1.1 [5] and most common web browsers [8]. So, in current web system, it is possible for web data to be compressed and decompressed with no user interaction at the end point.

Compression is reported to have good potential in increasing virtual network bandwidth, reducing network traffic and workload on web servers, and reducing download time of web pages [8], [9], . While it is instinctive to understand that it is always going to be faster to transfer a smaller file than a larger one, there are some issues regarding page retrieval latency worth of studying.

Typically, a web page is made up of multiple objects, among which one is called page Container Object (CO) and others called Embedded Objects (EO). CO usually is in the form of an HTML file while EOs are mostly images (and a few scripts). An HTML file consists of only ASCII text so it is highly compressible. But images used in web pages are usually pre-compressed and it is difficult to compress them further. So, the CO is often the only object in a page that is suitable for compression. Since CO only occupies part of the total page size, how effective would it be to just apply compression on the CO?

When talking about whole page latency, we need to taken into consideration the dependency between CO and EOs of a page. Because EOs are defined in CO, so the retrieval processes of EOs would depend on the retrieval process of CO. When compression is applied to CO and thus affects CO's latency, EOs' retrieval latency would be affected as well and this would in turn affect whole page latency.

There are basically two compression mechanisms in current web system, namely Pre-compression and Real-time compression. While people mostly concentrate on the complexity of file management, the real-time feature, and the coverage of static and dynamic objects of these two compression mechanisms, their performance on whole page latency is not well studied. A detailed study on the difference of the performance of these two compression mechanisms could be useful in helping people to have insight view of them.

2 Related Work

The potential benefits of web compression has been largely ignored. Although the idea of web compression emerged as early as when HTTP/1.0 [10] was defined, it did not seem to receive any attention until HTTP/1.1 was out. According to Port80's October 2003 Survey (Port80 Software is a compression vendor), only 3.7% of the top 1000 leading corporations' websites realize the benefits of HTTP compression [11].

There is very limited studies on web compression in research literature. Nielsen et al. reports on the benefits of compression in HTTP [12]. They observed over 60% gain in downloading time in low-bandwidth environment by using the zlib compression library [13] to pre-compress HTML files. Mogul et al [7] studied the potential benefits of delta encoding and data compression for HTTP. They reported about 35% reduction in transferred size and about 20% reduction in retrieval time when gzip compression is used. Other studies by [8] and [9] report that compression could achieve up to above 90% reduction in file size and above 60% reduction in downloading time.

Most of the above studies only used very small number of example documents and they did not investigate real-time compression (although modem compression is similar to real-time compression in some aspects). Most importantly, none of them studied compression's effect on whole page latency.

3 Concepts Related to Compression in Web Access

In current web system, the basic unit of browsing is page. A web page often consists of one Container Object (CO) and multiple Embedded Objects (EO). Usually, the CO of a web page is a basic HTML file and EOs are mostly inlined images. The CO of a page is always the first object returned from the server when the page is requested by client. Web data is transferred through network in a streaming way, chunk by chunk. When one chunk of web data arrives at client, the content would be parsed by the browser in order to present it to the user. In case of CO's data, the parse of one chunk of data might also trigger requests for EOs which are defined in that chunk. That is, the retrieval of EOs highly depends on the retrieval of CO. The presentation of a web page would not be considered complete until all objects belonging to the page have been retrieved. The whole page latency is thus determined by the number of objects in the page and definition dependency between CO and EOs.

From the above, we see that there are different latency components for CO and EO. The retrieval latency for CO is relatively simple. It mainly comes from the retrieval process. But for EOs, the retrieval latency is more complicated. Besides the latency coming from the retrieval process, EOs also have Definition Time (DT) in their whole retrieval latency. The latency component DT of an EO is the time spent from the page request being sent out to the chunk containing the definition of the EO arriving at client. Our study reveals that this latency component plays an important role in object latency. However, they are unnoticed in previous studies.

Web compression is usually achieved by applying a lossless compression algorithm on textual web objects (which usually are the COs). There are basically two ways to apply compression on web objects. The first way is to compress objects beforehand and store the compressed copies on web server to serve future requests. This mechanism is often called Pre-compression. The other way of compression is to compress each chunk of object data on the fly during the actual transmission of the data chunk sequence of the object. This mechanism is referred to as Real-time compression in our study. Considering the streaming nature of web content delivery and dependency between CO and EOs, these two compression mechanisms could have different effect on object latency and whole page latency. In this paper, we present a detailed chunk-level study on these factors to reveal the effect of these compression mechanisms on web content delivery.

4 Methodology

In our study, we performed retrieval for a large number of web pages in real web environment and obtained detailed chunk-level logs for all compression mechanisms, including the normal situation "No Compression".

We first get page URLs from a NLANR trace [14] dated 5th August 2003. Then we replicate those page content on our web server. We make a pre-compressed version of each page and put it on the same server. For real-time compression, we use a reverse proxy to perform the task. We use the zlib compression library [13] to build real-time compression capability into a Squid system [15], version 2.4.STABLE3 to be used as the reverse proxy. Page requests are generated automatically by a web client program pavuk [16]. All requests are forced to pass through a remote proxy to emulate the real web environment. Detailed chunk-level logs are recorded by the instrumented web client program and forward and reverse proxies.

Due to time and space limitation, we collected logs for a little more than 33,000 pages which have about 486,300 objects. The total logs size is around 17 GBytes. The logs are processed and fed into our simulators to get results for this study.

5 Results and Analysis

5.1 Studies on Some Properties About Web Object Transfer

Figure 1 plots the distribution of objects with respect to their size for those objects which are potential candidates for compression. We see that a large percentage of such objects have sizes between 8 KBytes to 128 KBytes, with an average size of 35.5 KBytes. This gives good potential of applying compression on them as we know that compression is usually more effective for bigger files.

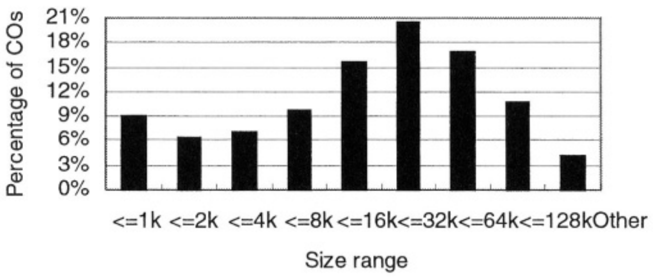


Fig. 1. Distribution of CO's body size

Figure 2 through Figure 4 give some chunk-level properties of web data transfer for compressible objects.

First, Figure 2 shows the number of chunks in an object transfer. We see that while about 12% of objects have only one chunk in their transfer, the majority of objects are made up of multiple chunks. On average, an object has about 6.7 chunks in its transfer.

Figure 3 shows the distribution of chunk sizes. It shows that the majority (65%) of chunks have sizes between 1 KBytes and 2 KBytes. However, there are chunks with bigger sizes, which put the average chunk size at 5.3 KBytes. We also note that there are a high percentage of chunks with sizes larger than 10 KBytes. This could be due to the fact that a large percentage of objects in our test set are quite big (see Figure 1) and that the buffer size in our proxies is as big as 64 KBytes.

Compression could have influence on both chunk sizes and the number of chunks in an object transfer. Pre-compression reduces object size before the object is requested, so the object data could be delivered by a smaller number of chunks. As for real-time compression, it performs compression in real-time by compressing every chunk in the object transfer. Thus, we would expect that real-time compression would reduce the size of every chunk instead of the number of chunks in the chunk transfer sequence. Since the transfer time of every chunk contributes towards the object retrieval latency, compression would affect object retrieval latency through the influence on chunks.

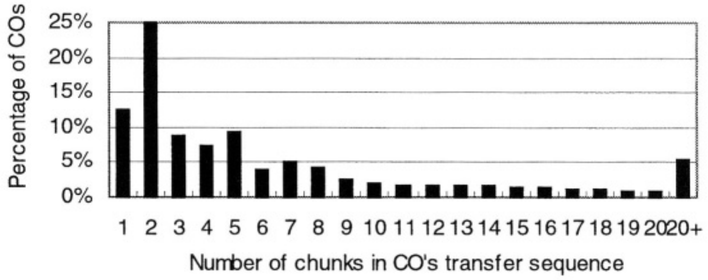


Fig. 2. Distribution of CO w.r.t. number of chunks

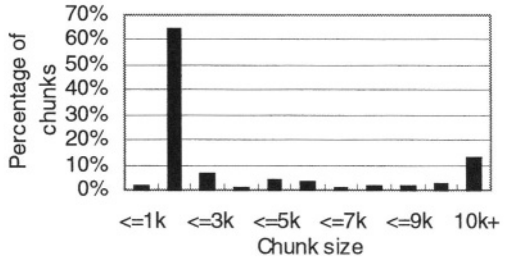


Fig. 3. Distribution of chunk size

Since chunk size would be affected by compression, we would like to see the transfer time for chunks with different sizes. Figure 4 plots the transfer time of chunks with respect to their sizes. Because all chunks were sent out from the same server and delivered along the same path to the same client in our experiments, so there is comparability among latencies of different chunks.

From Figure 4, we see that the distribution of the latencies for chunks with different sizes is quite random. Chunk size does seem to have much influence on chunk latency. The latency for smaller chunks is often comparable to that of much bigger chunks. This could be due to the random nature of network and server workload. This observation is important because it indicates that reducing chunk size might not help much in reducing object retrieval latency. We could further deduce that reducing the number of chunks might be more effective than reducing the size of chunks in terms of reducing object retrieval latency.

There are two extreme phenomena in Figure 4 worth of mentioning. One is that the latency for very large chunks (those with size greater than 30k) is indeed much bigger than that of smaller chunks. This suggests that reducing size for these chunks may still be helpful in reducing their latency. The other phenomenon is that, the latency for the "<=1k" group is even bigger than that of "<=2k" to "<=4k" groups. Further study reveals that this could be due to the TCP slow-start effect as "<=1k" chunks tend to be the first a few chunks in the chunk transfer sequence.

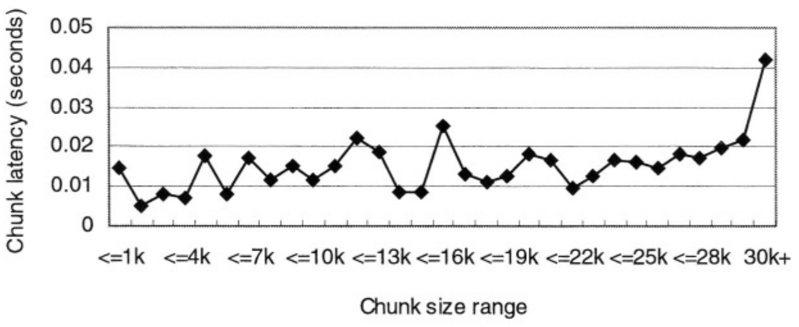


Fig. 4. Average latencies for delivering chunks with different sizes

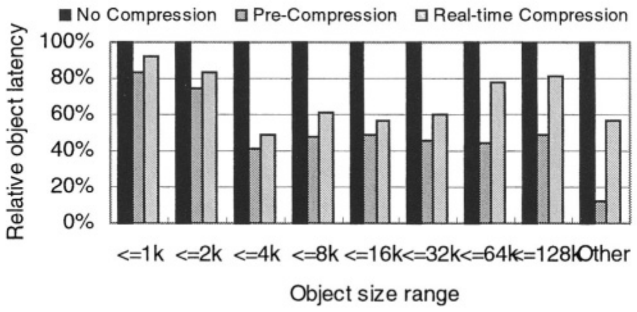


Fig. 5. Effect of different compression mechanisms on object latency

5.2 Studies on Effect of Compression on Single Object at Chunk Level

In this section, we would like to study the effect of compression on object latency at chunk level.

Figure 5 shows relative object latencies with respect to object size for different compression mechanisms. Here, the normal situation "No Compression" is used as the reference. From this graph, we see that both pre-compression and real-time compression have improvements on object latency and the improvement is quite big. For pre-compression, the improvements ranges from 16.4% to 88.1%, with an average of 57.2%. For real-time compression, the performance gain is from 8.1% to 51.1% and the average gain is 32.3%. The result shows that pre-compression always gives higher improvement than real-time compression does. This could be due to the reason we deduced earlier that pre-compression reduces the number of chunks of an object transfer which is more effective than reducing the size of chunks attained by real-time compression. We will further look into this reason in the later part of this section.

It is a little surprising to see that real-time compression also has big improvement on object latency since it tends to reduce the size of every chunk instead of reducing the number of chunks. Further study shows that real-time compression also reduces the number of chunks in some situation. This is due to a special phenomenon found in real-time compression. We call this special phenomenon "delay-and-merge" effect and we will discuss it further in the later part of this section.

As object size increases, the performance of pre-compression generally gets better. This could be because of these two reason: first, the compression ratio is normally higher for bigger files; second, when object is small, other latency components (such as connection time) and the TCP slow-start effect are relatively more significant, which makes the effect of compression marginal.

The situation for real-time compression is more complicated. As object size increases, the performance of real-time compression first gets better and then lower, and for the last object size range "Other" (i.e. >128 KBytes), it gets better again. The reasons are related to the "delay-and-merge" effect and we also put the explanations at the later part of this section.

Now, we investigate the reason for compression's effect on object latency at chunk-level by examining how different compression mechanisms affect chunk sizes and the number of chunks in an object transfer.

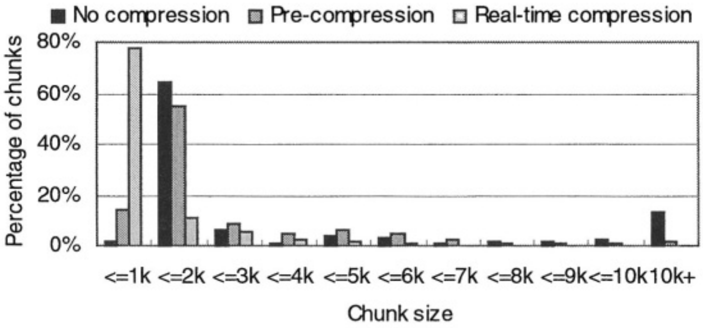


Fig. 6. Distribution of chunks w.r.t. chunk sizes sent out from server

Figure 6 shows the distribution of chunk sizes under different compression mechanisms. As expected, we see that real-time compression shifts the curve to the left significantly. As many as 78% of chunks are compressed to sizes smaller than 1 KBytes by real-time compression. However, this shifting is ineffective as the latency for 1-KByte chunks is similar to or even higher than that for bigger chunks according to Figure 4. On the other hand, we note that real-time compression shifts the 10k+ chunks to smaller chunks. The percentage of chunks belonging to 10k+ group under real-time compression is significantly lower than that of other mechanisms. As we learnt in Figure 4, the latency for very large chunks (30k+) is much bigger than that of smaller chunks. So, to compress such chunks would be helpful in reducing the chunk latency.

For pre-compression, the curve is also shifted to the left a little. The reason could be that, after being pre-compressed, more objects become smaller objects and they could be delivered by smaller number of chunks.

Figure 7 plots the number of chunks with respect to object size for different compression mechanisms. We see the number of chunks for pre-compression is smaller than normal situation and the difference between pre-compression and normal situation becomes bigger as object size increases. This is actually instinctive to understand because pre-compressed objects are smaller than the original ones so they could be delivered by lesser number of chunks, and, the compression ratio is usually higher for bigger objects, so the difference between pre-compression and normal situation becomes bigger. With much smaller number of chunks to transfer, it is easy to understand why pre-compression could improve object latency so significantly (see Figure 5).

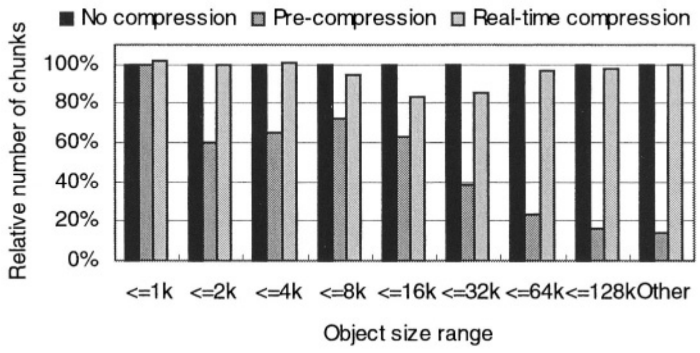


Fig. 7. Number of chunks w.r.t. object size under different compression mechanisms

It is surprising to see that real-time compression also reduces the number of chunks in some situation since real-time compression is believed to reduce the size of every chunk instead of reducing the number of chunks. Further study revealed a special phenomenon behind.

In our experimental system, the real-time compression is performed by a reverse proxy. The reverse proxy receives chunks from the web server next to it and compresses each chunk before sending them out. During the time when the reverse proxy is busy compressing current chunk, the rest of chunks would continuously arrive. Since the reverse proxy is busy, those incoming chunks would be buffered in its buffer and merged into one. Therefore, the size of the following chunk becomes bigger. We name this phenomenon the "delay-and-merge" effect. When object size is big and it has a large number of chunks in its transfer, this effect would accumulate, which would make chunks become bigger and bigger.

Figure 8 tries to show this effect by plotting chunk size with respect to the chunk sequence number. We see that chunk sizes in real-time compression is generally bigger than that of no compression and pre-compression, and the difference usually gets bigger for chunks with bigger chunk sequence number.

We also note that chunk size becomes bigger for all mechanisms as chunk number increases. This could be also due to the TCP slow-start effect. With successful transmission of more chunks, the transfer rate gets higher so that a bigger amount of data could be transferred in one chunk.

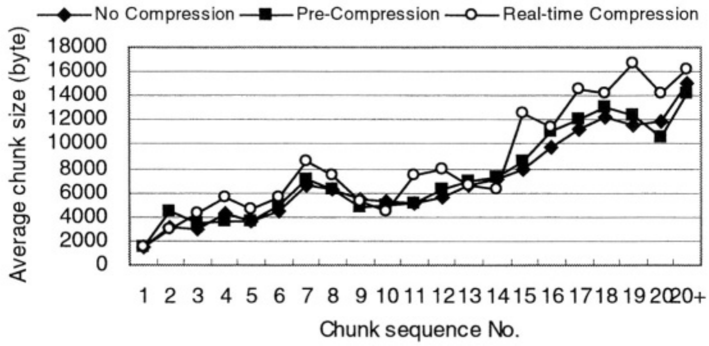


Fig. 8. Average size of chunks w.r.t. chunk sequence number under different compression mechanisms

The "delay-and-merge" effect has a "warming-up" stage and "mature" stage. During the "warming-up" stage, chunk size would become bigger and bigger as reverse proxy takes more and more time to compress each bigger chunks. However, because the buffer size in reverse proxy is fixed (64 KBytes in our experimental system), this effect will "mature" when the chunk size grows close to the buffer size. In "mature" stage, chunk size would stop growing no matter how many more chunks are still in the transfer sequence.

As chunks would grow bigger due to the "delay-and-merge" effect in real-time compression, the number of chunks for a given object would become smaller than the normal situation. This could explain the result of real-time compression in Figure 7.

For small objects, real-time compression does not seem to reduce the number of chunks. This is because the number of chunks is too small for the "delay-and-merge" to "warm-up". As object size increases, the "delay-and-merge" effect start to "warm-up" so the number of chunks becomes smaller. However, this trend stops when object size is big enough. This is because the "delay-and-merge" effect has matured.

With this knowledge, we could give explanation on the performance of real-time compression shown in Figure 5. Because real-time compression also reduces the number of chunks for an object due to the "delay-and-merge" effect, it is understandable that it also improves object latency. As object size grows from 1 KBytes to 4 KBytes, the performance of real-time compression gets better. This could be because that the "delay-and-merge" effect is in the "warming-up" stage. For objects with sizes between 8 KBytes to 128 KBytes, the "delay-and-merge" effect would get mature so that we see that the performance of real-time compression stops getting better. However, for very big objects with size greater than 128 KBytes, the performance of real-time compression becomes better again. Further study reveals the following reason: for the objects in this group, the chunk size could be very big due to the "delay-and-merge" effect, and very big chunks could be effectively compressed to smaller chunks by real-time compression (see Figure 6). With refer to Figure 4, the latency for very large chunks (those with size greater than 30k) is much bigger than that of smaller chunks. So, reducing the size of very big chunks would result in reduction in transfer time. Therefore, the performance of real-time compression for this group gets better again.

Considering that most COs consist of 6.7 chunks (see Figure 2) and most chunks have sizes between 1 KBytes and 2 KBytes (see Figure 3), it would be more effective to reduce the number of chunks than to reduce the size of chunks. This explains why pre-compression always gives higher improvement on object latency than real-time compression does.

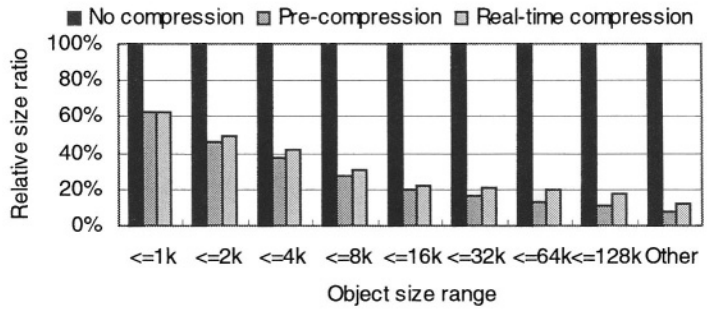


Fig. 9. Distribution of compression ratio of objects

In addition, the compression ratio of pre-compression is also slightly better than real-time compression. Figure 9 shows the compression ratio of different compression mechanisms. We see that pre-compression yields compression ratio about 5.2% better than real-time compression does on average. This could be because pre-compression can see the whole object data while real-time compression can only see one chunk of the data. Generally, a program which can see the entire input file could compress the file more effectively than the program which sees only part of the input file does.

We also note that the compression ratio of pre-compression becomes higher as object size increases. This is because compression is more effective for big files. While it is easy to understand this, it is not so straightforward to understand the case for real-time compression since it often does not see the entire input file. The reason why real-time compression also generates higher compression ratio with the increasing object size is a little "tricky": When object size is large, the chunks in its transfer sequence tends to be large due to the "delay-and-merge" effect, compression on a single chunk would also be effective when chunk size is large.

Overall, the compression ratio of both of the compression mechanisms is very high. On average, object size can be reduced 87.6% by pre-compression and 82.4% by real-time compression. This further explains the high improvement on object latency by these two compression mechanisms in Figure 5.

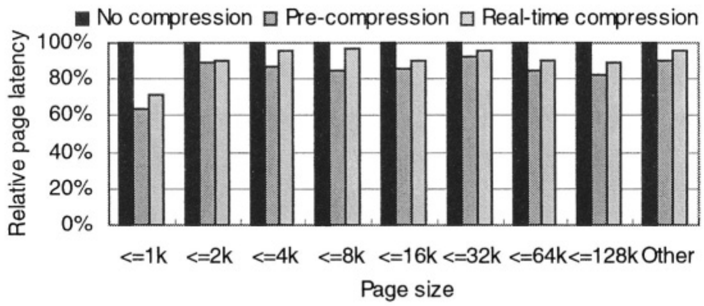


Fig. 10. Compression's effect on whole page latency

5.3 Effect of Compression on Whole Page Latency

Because the basic unit of browsing is page in current web system, whole page latency is more meaningful to clients than object latency. In this section, we would like to investigate the effect of compression on whole page latency.

As explained earlier, page latency is determined by more complicated factors. So, the improvement on single object latency achieved by compression may not be translated into the improvement on page latency directly.

Figure 10 plots relative page latency with respect to page sizes. Comparing it with Figure 5, we see that the improvement on whole page latency achieved by compression is significantly much lower than it does on object latency. The average performance gain on whole page latency is about 12.2% by pre-compression and 7.4% by real-time compression, as compared to the 57.2% and 32.3% gain on object latency by pre-compression and real-time compression respectively.

Although the fact that compressible objects in a page (mainly COs) only occupy part of the total page size could be partially the reason, the big difference between compression's performance on object latency and page latency may also indicate that the dependency among objects in a page plays an important role in determining page latency.

6 Conclusion

This paper presents our detailed chunk-level study on web compression. Results help us to have in-depth understanding on the behavior of compression and its effect on page latency.

Our results show that reducing the number of chunks is more effective in improving object latency than reducing the size of every chunk. So, pre-compression almost always performs better than real-time compression.

Although compression can generate high improvement on retrieval latency for COs, this improvement is not translated into improvement on page latency directly. The improvement on page latency is much smaller.

Compression improves page latency from two aspects: (1) reducing CO's retrieval latency, (2) reducing the DT times of EOs through the dependency between CO and EOs.

Acknowledgment. We thank NLANR for giving us access to their traces.

References

1. Wessels, D., Web Caching, O'Reilly Publishing, 2001.
2. Padmanabhan, V.N., Mogul, J.C., "Using Predictive Prefetching to Improve World Wide Web Latency," ACM Computer Review, July 1996.
3. T.M. Kroeger, D.D.E. Long, and J.C. Mogul Exploring the bounds of Web latency reduction from caching and prefetching Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems, Monterey, CA, Dec. 1997.
4. Jun-Li Yuan, Chi-Hung Chi, Web Caching Performance: How Much Is Lost Unwarily?, In the Proceedings of the Second International Human.Society@Internet Conference, p.23-33, June 18 - 20, 2003, Seoul, Korea.
5. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T., "Hypertext Transfer Protocol -- HTTP/1.1," IETF RFC2616, June 1999.
6. Wills, C.E., Mikhailov, M., Shang, H., "N for the Price of 1: Bundling Web Objects for More Efficient Content Delivery," Proceedings of the 10th International World Wide Web Conference, May 2001.
7. Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy Potential Benefits of Delta-encoding and Data Compression for HTTP In Proceedings of ACM SIGCOMM, pages 181--194, September 1997. An extended and corrected version appears as Research Report 97/4a, Digital Equipment Corporation Western Research Laboratory, December, 1997.
8. HTTP Compression Speeds up the Web, http://www.R25.com/internet/software/servers/http/compression/
9. The Effect of HTML Compression on a LAN and a PPP Modem Line, http://www.w3.org/Protocols/HTTP/Performance/Compression/LAN.html http://www.w3.org/Protocols/HTTP/Performance/Compression/PPP.html
10. T. Berners-Lee, R. Fielding, H. Frystyk, Hypertext Transfer Protocol -- HTTP/1.0, IETF RFC1945, May 1996.
11. Port80's survey on compression: http://www.port80software.com/surveys/top1000compression/
12. Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. InProc. SIGCOMM'97. Cannes, France, September, 1997.
13. zlib home page: http://www.gzip.org/zlib/
14. IRCACHE Proxy Traces, http://ircache.nlanr.net.
15. Squid Web Proxy Cache, http://www.squid-cache.org/
16. pavuk, http://www.idata.sk/~ondrej/pavuk/index.html

An Empirical Study of a Segment-Based Streaming Proxy in an Enterprise Environment

Sumit Roy[1], Bo Shen[1], Songqing Chen[2], and Xiaodong Zhang[3]

[1] Streaming Media Systems Group
Hewlett-Packard Laboratories
Palo Alto, CA 94304
[2] Department of Computer Science
George Mason University
Fairfax, VA 22030
[3] Department of Computer Science
College of William and Mary
Williamsburg, VA 23187
zhang@cs.wm.edu

Abstract. Streaming media workloads have a number of desirable properties that make them good candidates for caching via proxy systems. The content does not get modified, and access patterns exhibit some locality of reference. However, media files tend to be much larger in size than traditional web pages, and users tend to view video clips only partially. Hence, segment-based strategies have been proposed to deliver large streaming media objects via a web-proxy. In this work, we evaluate the performance of such a segment-based streaming proxy using extensive trace driven simulation. We use representative workloads from enterprise media server logs, and evaluate the caching system performance regarding different cache sizes, different segment sizes, and different prefetching methods. Our results show that cost-effective caching requires only about 8 - 16 % of the total unique object size as proxy storage. Secondly, a segment size of around 200 Kbytes provides a good trade-off between cache object granularity and transaction overhead. Finally, a lazier prefetching schedule that provides a half-segment look-ahead has a significant performance gain when compared to a more aggressive one-segment look-ahead scheme.

1 Introduction

Recent years have seen abundant applications of streaming media objects on the global Internet, such as remote education, tele-medical treatment, entertainments, etc.. The increased amount of streaming data requires effective and efficient streaming delivery strategies. In an IP based environment, a continuous streaming session (often with a duration of minutes or hours, compared to milliseconds or seconds for traditional web pages) keeps consuming network bandwidth and disk bandwidth on the hosting server. Multiple concurrent streaming sessions can easily exhaust the available network bandwidth and overload the media content server. Placing multimedia objects closer to clients is an effective solution that would relieve the network bottleneck and distributed the load across multiple streaming media servers. Hence, a lot of research has been done on proxy based streaming delivery approaches, because streaming media objects are generally in-variant, which makes them feasible for proxy caching without worrying about cache consistency [1,2].

Streaming media object are usually large in size, of the order of multiple MBytes, rather than kBytes, as is more common with web-pages. While caching the complete media object would certainly work, it would require a considerable storage space. Moreover, unless this is augmented by smart read-through techniques, the startup-latency for a client that experiences a cache miss increases dramatically with the clip length. Studies of client access patterns to streaming media objects also show that most clips are only viewed partially [3,4]. Hence, partial caching approaches are more effective at delivering quality streaming media to clients. In partial caching, the media object is divided or *segmented* into multiple parts. The proxy deals with individual segments for prefetching, replacement, and cache management issues. This has a number of advantages. First, in case of a cache miss, the startup-latency seen by a client is now dependent on the segment size, rather than the clip length. Second, clips that are viewed only partially do not have to be transferred in their entirety across the network. This reduces the load on the backbone infrastructure. Finally, cache utilization becomes more efficient, since the segments are smaller and can be treated as fixed size units. However, there are some challenges to designing a segment-based proxy cache. If a media client encounters a miss for a segment during play-back, there could be an interruption in smooth media delivery. This problem can be solved by providing large buffers at the client, the streaming server, or by using prefetching techniques in the cache. There are various trade-offs to be considered in deciding the segment size. A large segment size increases startup-latency on a miss, decreases the number of hand-shakes required with the back-end server, but might lead to inefficient cache utilization. Finally, it is interesting to evaluate the optimal cache size, based on an understanding of typical workloads.

In this paper, we evaluate the performance of a segment-based streaming proxy using extensive trace driven simulation. We use representative workloads from enterprise media server logs, and evaluate the system performance with regard to different cache sizes, different segment sizes, and different prefetching methods. Our results show that cost-effective caching requires only about 8-16 % of the total unique object size as proxy storage. Secondly, a segment size of around 200 Kbytes provides a good trade-off between cache object granularity and transaction overhead. Finally, a lazier prefetching schedule that provides a half-segment look-ahead has a significant performance gain when compared to a more aggressive one-segment look-ahead scheme.

The rest of this paper is organized as follows. We review related work in Section 2. We evaluate the system performance through extensive experiments in Section 3. We conclude the paper in Section 4.

2 Related Work

The research on proxy caching of streaming media content has received much attention lately. Middleman [2] studied clusters of proxies for streaming media delivery. However, they ignored an important feature of streaming media accesses: It is found that continuous media objects such as video or music clips are often partially accessed. Based on this observation, *partial* caching approaches have been proposed to reduce the cache space requirement. The basic strategy is to cache segments of objects that are divided in the viewing time domain. Typical examples include prefix caching [5], uniform segmentation [6], exponential segmentation [7] and adaptive and lazy segmentation [8]. Prefix caching always caches the prefix of the objects to minimize the startup latency. The optimal prefix length can be calculated according to [9]. Its protocol consideration, as well as partial sequence caching, were studied in [10].

In uniform segmentation, objects are cached in uniform-size segments, while in exponential segmentation, the segment size doubles along the viewing direction. Considering limited resources available from a single cache, the Rcache [1] considers the usage of multiple proxies, focusing on the memory and disk utilization. Adaptive and lazy segmentation partitions objects when necessary by considering the user access pattern. These strategies are considered as partial caching strategies, in which only partial data of a media object are cached (Though, very popular objects can be fully cached). These strategies focus on protocol design or benefit analysis based on artificial workloads, emphasizing different performance aspects. Recently, authors in [11] proposed a flexible and scalable proxy testbed to support a wide and extensible set of advanced proxy streaming services. We proposed a segment-based streaming proxy design model in [12]. Compared with the previous work, in this study, we focus on evaluating different performance factors, particularly those related to the cache configurations, thus providing guidance on how to use segment-based streaming proxy in a cost-effective fashion.

The partial caching strategy can be extended to the quality domain. Layered caching techniques [13,14] have demonstrated efficient usage of cache space by considering different QoS characteristics of client devices or connectivity. A comparison with multiple version caching is studied in [15] while a model of layered-encoded object distribution is studied in [16]. In [17], the proposed approach attempts to select groups of consecutive frames by the selective caching algorithm, while in [18], the algorithm may select groups of non-consecutive frames for caching in the proxy. A different idea is proposed in video staging [19], in which a portion of bits from the video frames whose size is larger than a predetermined threshold is cut off and prefetched to the proxy a priori to reduce the bandwidth on the server proxy channel. Recently, a fine grained, network aware and media adaptive rate control scheme is used in caching of scalable streaming content [20]. Most of partial caching schemes in quality domain require layered encoded objects or additional support from the proxy or client, and little work has been done to evaluate them against different performance factors. The work presented in this paper does not have this limitation.

3 Performance Evaluation

We evaluate the performance of a streaming proxy that uses uniform segmentation of the content, in the time domain. We first describe the evaluation methodology and the workloads used. We then show results of our simulation using different cache sizes, different segment sizes, and different prefetching methods.

3.1 Evaluation Methodology

We test the performance of the proxy by means of a trace driven cache simulator. We first convert a streaming media server log into a segmented trace, using the media object access time, the media offset and duration of the access, the segment length, and the encoding rate to construct a segment access schedule. This somewhat simplifies the actual segment access pattern, since real media files (like Quicktime or MP4) include regions with meta information, and they interleave audio and video data.

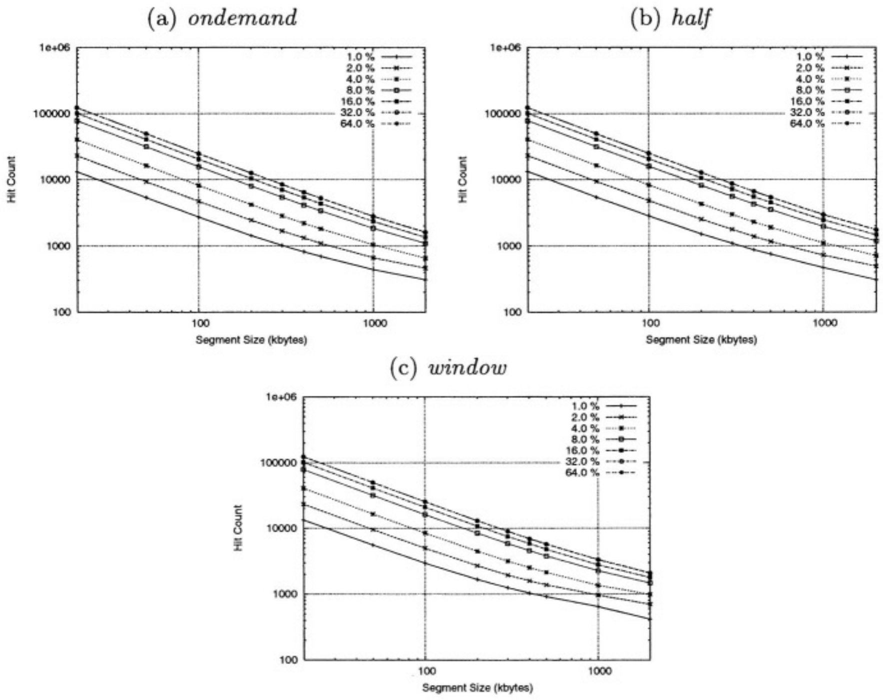


Fig. 1. Cache Hit Counts for LIGHT, different Cache capacities

This synthesized trace is run through a cache simulator, for cache size varying from 1 % to 64 % of the total unique object size. For each cache size, we evaluate

Table 1. Summary of Media Server traces used for evaluation

Workload Name	Num of Requests	Num of Objects	Num of Skewed Accesses (%)	Unique Object Size (GB)	Accessed Data Size (GB)	Accessed Object Size (GB)
LIGHT	655	83	89 (14 %)	2.08	3.232	19.854
MEDIUM	1086	85	376 (35 %)	2.02	3.909	13.314
HEAVY	1840	91	1156 (63 %)	2.44	11.676	34.686

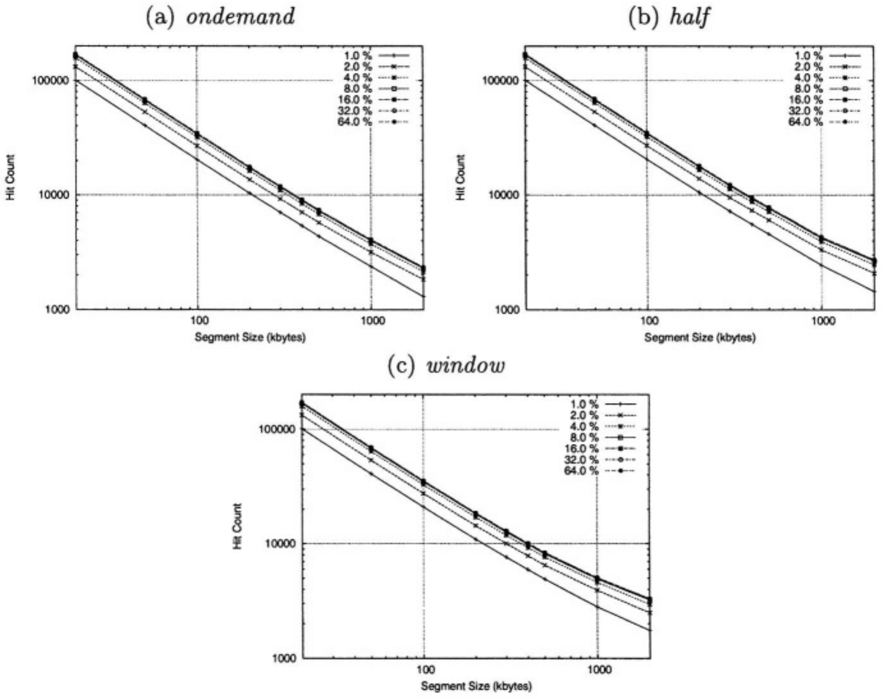


Fig. 2. Cache Hit Counts for MEDIUM, different Cache capacities

a range of segment sizes, from 20 KBytes to 2000 KBytes. We test three different methods for scheduling segment downloads from the original content server in these experiments. The *ondemand* method is entirely demand based, there is not prefetch. The *window* based prefetching uses a single segment look-ahead window. The access of one segment causes an immediate prefetch of the next sequential segment. This is an aggressive scheme and we expect it to perform well if a client accesses a media object in its entirety. On the other hand, clients that terminate early will reduce the effectiveness of prefetching, since the later segments will not be accessed. Finally, we also test the *half* method, where the next segment is prefetched after half of the current segment has been played back. Prefetching is beneficial for directly reducing client play-back jitter, measurement of this quantity is beyond the scope of this simulation base work.

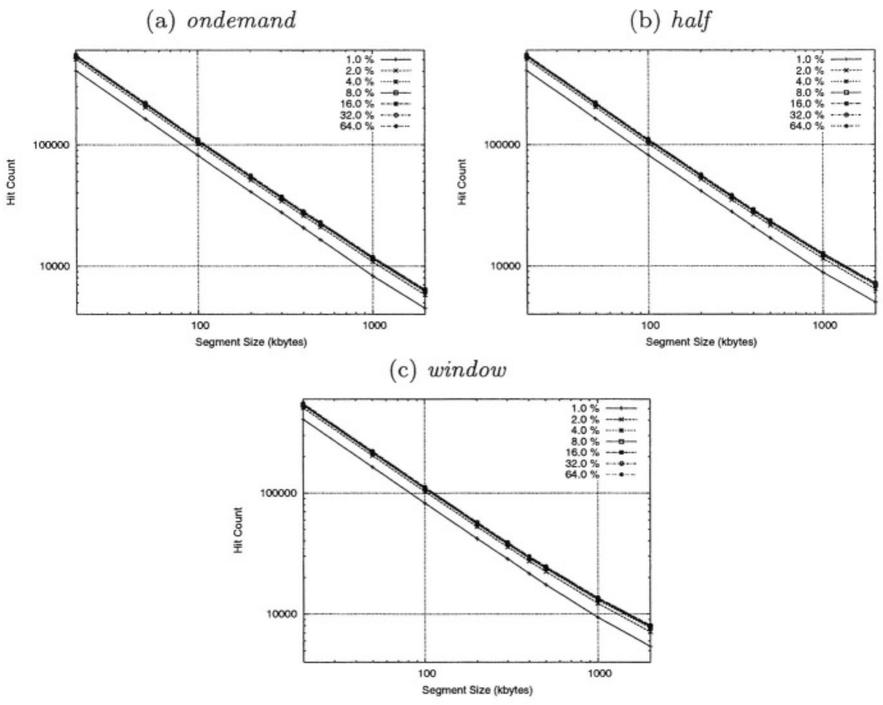


Fig. 3. Cache Hit Counts for HEAVY, different Cache capacities

3.2 Workload Selection

Streaming Media server log studies have shown that there can be considerably skew in terms of object popularity [4,3]. We selected three different 12 hour traces from a multi-month enterprise media server log. The characteristics of these trace, called LIGHT, MEDIUM, and HEAVY, based on the skew, are detailed in Table 1.

Unique object size is used as reference for cache capacity. Accessed object size shows the total of accessed objects (full size). Accessed data shows the the portion of it that is actually accessed by the client. The average access duration for these three workloads are 337, 247 and 311 seconds, respectively.

3.3 Simulation Results

The goal of the simulation was to determine the behavior of the proxy with respect to cache size, segment size, and segment scheduling method. We show the condensed results in the next few sub-sections.

Cache Size. The proxy cache size is an important design consideration. For every run, we assume a cold-start cache. Intuitively, a large cache performs bet-

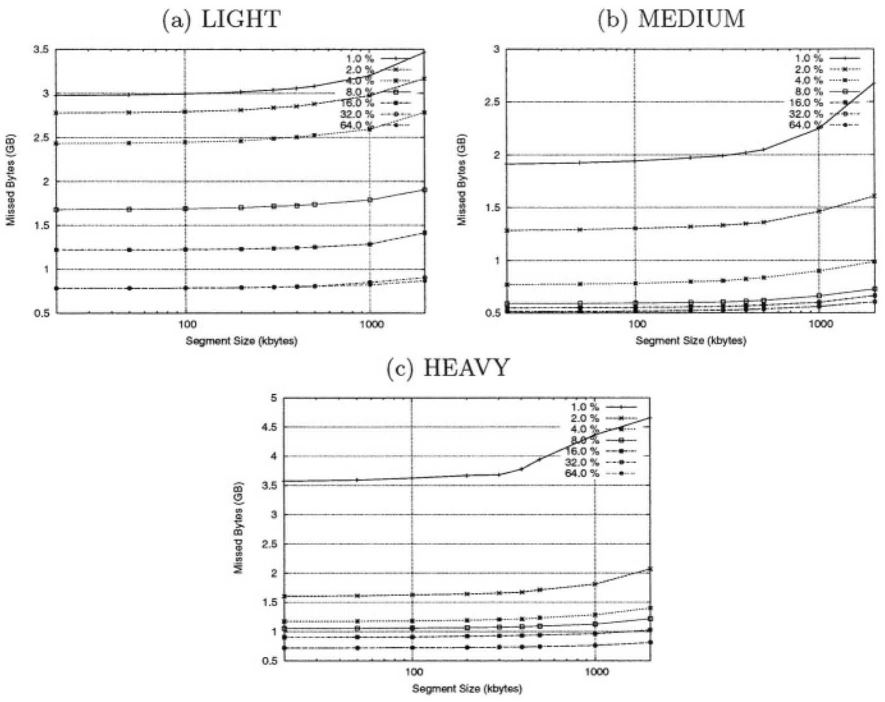


Fig. 4. Missed Bytes of *ondemand* method, different Cache capacities

ter, since it can potentially eliminate misses induced due to limited capacity. A cost-effective cache would be sized such that the short term working set completely fits in the cache. Thus, all misses would be cold-start misses, and segment evictions would be for content that is no longer required. Our simulator uses the standard LRU algorithm for replacing cached segments.

Figures 1, 2, and 3 show the Segment Hit Count for the LIGHT, MEDIUM, and HEAVY traces. The Cache capacity is shown as a percentage of the total *unique* object size. It is seen that the Hit Count approximately doubles when the cache size is doubled, up to about a cache size of 8 % for LIGHT, 4 % for MEDIUM, and 2 % for HEAVY. This behavior is consistent across a large range of segment sizes, and is independent of the segment scheduling method. Hence, a cache size of only 8 % of the total unique object size is sufficient for providing good hit rates in the proxy. Note that this translates into a real size of approximately 512 MBytes. This suggests that streaming proxies could be implemented with solid state storage devices like FlashMernory or battery backed DRAM.

Segment Granularity. The evaluation of an optimal segment sizes is very important. Smaller segment sizes use the cache more efficiently, and provide

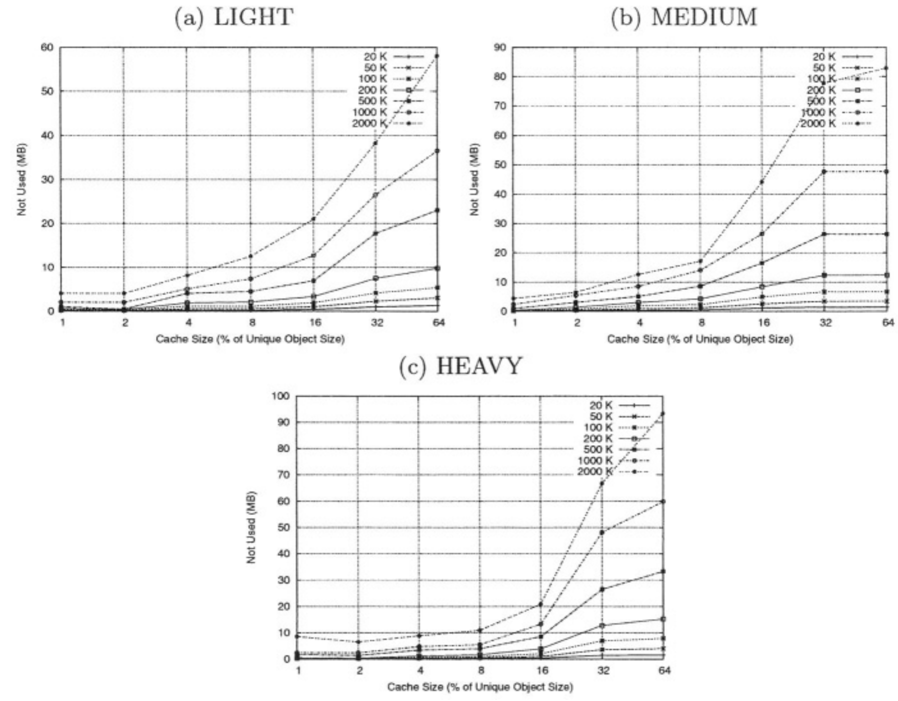


Fig. 5. Data prefetched but never used, *window* method, different Segment sizes

lower startup-latencies to a client when there is a cache miss. However, very small segment sizes can cause excessive transactions with the media content server.

Figure 4 shows the number of Bytes transferred due to cache misses for the different workloads. Beyond a segment size of 100 - 200 KBytes, the curves start sloping up, which means that extra data is transferred due to the larger segment granularity. This effect is even more pronounced when we look at an additional metric for methods that include prefetching. When a segment schedule includes prefetches, it is possible that a client terminates the session, without ever accessing the contents of the prefetched segment. The amount of data transferred and not used is shown in Figures 5 and 6 for the *window* and *half* method respectively.

For the previously determined optimal cache size of approximately 8 %, it is seen that the amount of useless data increases rapidly for segment sizes greater than approximately 200 KBytes, independent of the access pattern. Hence, the optimal segment size is expected to be around 100 - 200 Kbytes.

Prefetching Effectiveness. The prefetching methods *window* and *half* primarily affect client perceived jitter, whose evaluation is beyond the scope of this

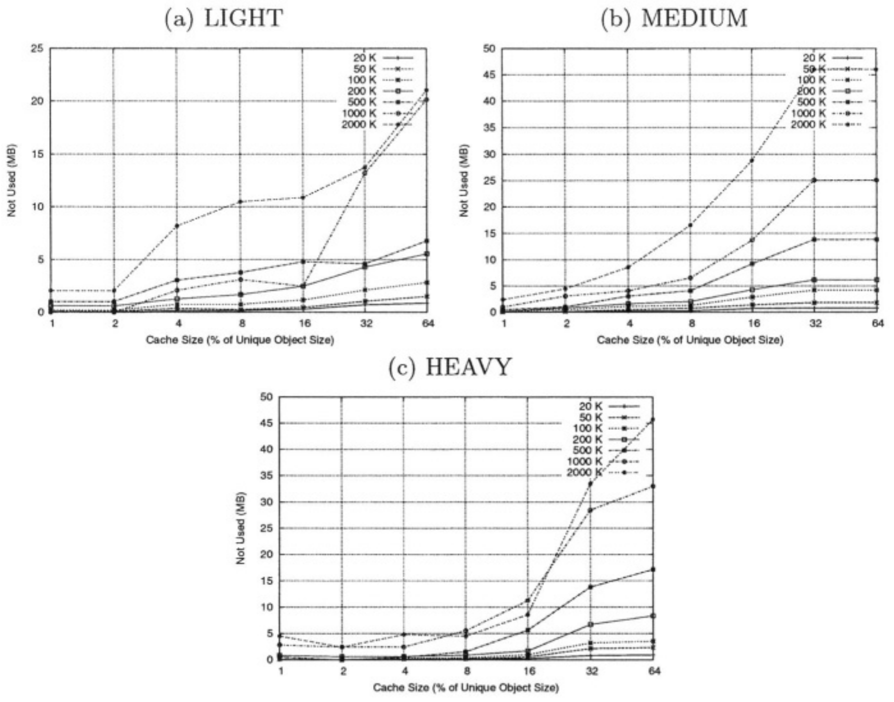


Fig. 6. Data prefetched but never used, *half* method, different Segment sizes

paper. However, overly aggressive prefetching can lead to the following problem: A prefetched segment might be evicted from the cache due to capacity misses, thus causing a completely unnecessary data transfer from the media content server. Similarly, a client may terminate its session, before accessing the prefetched segment. The amount of data transferred needlessly this way is shown in Figures 7 and 8 for *window* and *half,* respectively.

The *half* method causes less than 50 % of the wasted data transfer compared to the *window* method for the optimal cache size and segment size determined previously. Since the schedule is less aggressive, when a client terminates its session early, fewer segments are prefetched. This is particularly visible for the LIGHT workload, where there is a more uniform access distribution to the objects. Hence a lazier prefetch method, for example *half* provides a much better performance when measured at the cache. Note that the benefit is independent of the segment size and workload. The amount of data never used, shown previously in Figures 5 and 6, also agree with this assessment.

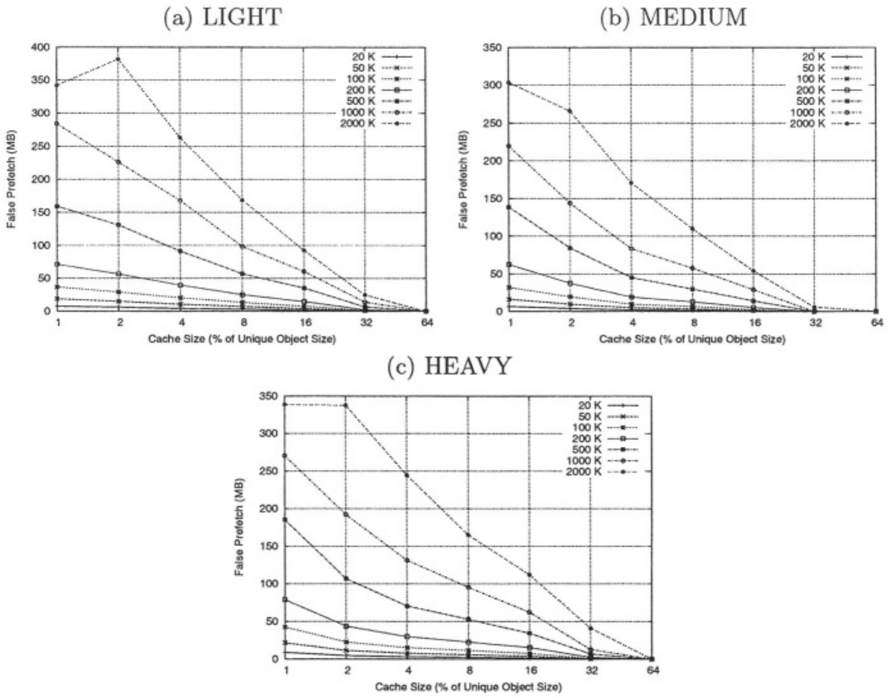


Fig. 7. Data prefetched but evicted before use, *window* method, different Segment sizes

4 Conclusion

Streaming media workloads are amenable to proxy caching via segmentation. In this paper we evaluate the performance of a segment-based streaming proxy using a trace driven simulation. We use three different traces, which have light, medium, and heavy skew in terms of object popularity. We first determine the optimum cache size by looking at the hit counts for different sizes. Our definition of optimality is based on the marginal performance gain due to an increase in size. Based on this measure, our results show that cost-effective caching requires only about 8 - 16 % of the total unique object size as proxy storage. This is an interesting result, since it suggests that streaming proxies could be implemented with solid state data stores for the enterprise workloads that were considered for this paper.

Next we evaluate the performance of the system using different segment sizes. In this case, we suggest that an increase in segment size should not cause an undue increase in total bytes that are transferred on cache misses. At the same time, we propose that unused data due to prefetches should be kept minimal. For the optimal cache size determined earlier, a segment size of around 200 Kbytes provides a good trade-off.

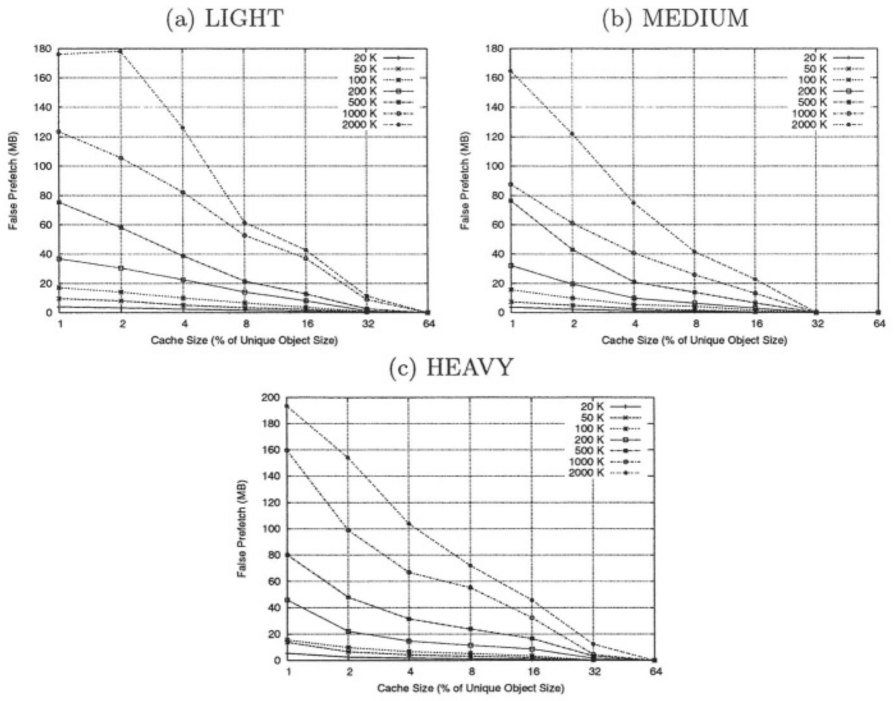


Fig. 8. Data prefetched but evicted before use, *half* method, different Segment sizes

Finally, we look at two different prefetch schemes, and evaluate them based on the amount of *unnecessary* data transferred due to prefetches. This metric is particularly relevant to the interaction between the proxy cache, and the original media content server. The less aggressive prefetching schedule issues a request after serving *half* of a segment. It is seen that this method causes less than 50 % of needless data transfers when compared to the more aggressive, *window* scheme.

References

1. Chae, Y., Guo, K., Buddhikot, M., Suri, S., Zegura, E.: Silo, rainbow, and caching token: Schemes for scalable fault tolerant stream caching. In: IEEE Journal on Selected Areas in Communications. (2002)
2. Acharya, S., Smith, B.: Middleman: A video caching proxy server. In: Proc. of ACM NOSSDAV, Chapel Hill, NC (2000)
3. Cherkasova, L., Gupta, M.: Characterizing locality, evolution, and life span of accesses in enterprise media server workloads. In: Proc. of ACM NOSSDAV, Miami, FL (2002)
4. Chesire, M., Wolman, A., Voelker, G., Levy, H.: Measurement and analysis of a streaming media workload. In: Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems, San Francisco, CA (2001)

5. Sen, S., Rexford, J., Towsley, D.: Proxy prefix caching for multimedia streams. In: Proc. of IEEE INFOCOM, New York City, NY (1999)
6. Rejaie, R., Handley, M., Yu, H., Estrin, D.: Proxy caching mechanism for multimedia playback streams in the internet. In: Proc. of International Web Caching Workshop, San Diego, CA (1999)
7. Wu, K., Yu, P.S., Wolf, J.: Segment-based proxy caching of multimedia streams. In: Proc. of WWW, Hongkong, China (2001)
8. Chen, S., Shen, B., Wee, S., Zhang, X.: Adaptive and Lazy Segmentation Based Proxy Caching for Streaming Media Delivery. In: Proc. of ACM NOSSDAV, Monterey, CA (2003)
9. Wang, B., Sen, S., Adler, M., Towsley, D.: Proxy-based distribution of streaming video over unicast/multicast connections. In: Proc. of IEEE INFOCOM, New York City, NY (2002)
10. Gruber, S., Rexford, J., Basso, A.: Protocol considertations for a prefix-caching for multimedia streams. In: Computer Network. Volume 33(1-6). (2000) 657–668
11. Zhang, X., Bradshaw, M., Guo, Y., Wang, B., Kurose, J., Shenoy, P., Towsley, D.: Amps: A flexible, scalable proxy testbed for implementing streaming services. Technical report, Department of Computer Science, University of Massachusetts (2004)
12. Chen, S., Shen, B., Wee, S., Zhang, X.: Designs of high quality streaming proxy systems. In: Proc. of IEEE INFOCOM, Hong Kong, China (2004)
13. Rejaie, R., H. Yu, M.H., Estrin, D.: Multimedia proxy caching mechanism for quality adaptive streaming applications in the internet. In: Proc. of IEEE INFOCOM, Tel-Aviv, Israel (2000)
14. Rejaie, R., Handely, M., Estrin, D.: Quality adaptation for congestion controlled video playback over the internet. In: Proc. of ACM SIGCOMM, Cambridge, MA (1999)
15. Kim, T., Ammar, M.H.: A comparison of layering and stream replication video multicast schemes. In: Proc. of ACM NOSSDAV 2001, Port Jefferson, NY (2001)
16. Kangasharju, J., Hartanto, F., Reisslein, M., Ross, K.W.: Distributing layered encoded video through caches. In: Proc. of IEEE INFOCOM, Anchorage, AK (2001)
17. Ma, W., Du, H.: Reducing bandwidth requirement for dilivering video over wide area networks with proxy server. In: Proc. of IEEE ICME. Volume 2. (2000) 991–994
18. Miao, Z., Ortega, A.: Scalable proxy caching of video under storage constraints. In: IEEE Journal on Selected Areas in Communications. (2002)
19. Zhang, Z., Wang, Y., Du, D., Su, D.: Video staging: A proxy-server based approach to end-to-end video delivery over wide-area networks. In: IEEE Transactions on Networking. Volume 8. (2000) 429–442
20. Liu, J., Chu, X., Xu, J.: Proxy cache management for fine-grained scalable video streaming. In: Proc. of IEEE INFOCOM, Hong Kong, China (2004)

Bottlenecks and Their Performance Implications in E-commerce Systems*

Qi Zhang[1], Alma Riska[2], Erik Riedel[2], and Evgenia Smirni[1]

[1] College of William and Mary, Williamsburg, VA 23187
{qizhang,esmirni}@cs.wm.edu
[2] Seagate Research, Pittsburgh, PA 15222
{Alma.Riska,Erik.Riedel}@seagate.com

Abstract. We present a detailed workload characterization of a multi-tiered system that hosts an e-commerce site. Using the TPC-W workload and via experimental measurements, we illustrate how workload characteristics affect system behavior and operation, focusing on the statistical properties of dynamic page generation. This analysis allows to identify bottlenecks and the system conditions under which there is degradation in performance. Consistent with the literature, we find that the distribution of the dynamic page generation is heavy-tailed, which is caused by the interaction of the database server with the storage system. Furthermore, by examining the queuing behavior at the database server, we present experimental evidence of the existence of statistical correlation in the distribution of dynamic page generation times, especially under high load conditions. We couple this observation with the existence (and switching) of bottlenecks in the system.

Keywords: TPC-W, bottleneck identification, workload characterization, query time distribution, autocorrelation.

1 Introduction

The ubiquity of web browsers as the standard interface for IT applications such as online news, shopping, and banking, makes web usage a daily practice. Contemporary web servers provide content that is mostly dynamically generated. Understanding the resource requirements of dynamic requests is not straight-forward. It is possible that a request will cause a substantial portion of the database to be accessed even when just a few kilobytes of text are eventually sent back to the client. Identifying the bottlenecks and the exact conditions under which they occur can lead to better system provisioning, resource allocation, and quality of service. In this paper, we focus on the analysis of the necessary time to generate dynamic objects in an e-commerce environment, focusing on the performance behavior of a Business-to-Consumer (B2C) site. We build a three-tier architecture and use the popular TPC-W workload (the Transaction Processing Council's e-commerce benchmark) to drive the web server system. We conduct measurements in

* This work was partially supported by the National Science Foundation under grants CCR-0098278, and ACI-0090221.

all tiers of the system aiming at providing detailed statistical information of the workload behavior, and at identifying bottleneck switching across the tiers. Our experiments show that the lower tiers of the system, i.e, the database server and the storage system, become the bottleneck while the resources at the front-end remain underutilized. The system capacity, directly related with the existence of bottlenecks in the system, depends not only on the load, but also on the characteristics of the system workload. Some workloads utilize system resources much faster than others, for the same number of users in the system. These workloads are more I/O bound than others. Furthermore, we observe the existence of long-range dependence in dynamic page generation times. Long-range dependence in dynamic page generation times has been previously reported in the literature [16] and has been attributed to the bursty nature of the arrival process. Here, we decouple the existence of long-range dependence and correlation from the arrival process and we strictly focus on the service process. More specifically, we find that the observed burstiness in the query time distribution is mainly due to the existence (and switching) of bottlenecks.

In this paper, we dwell on the performance implications of the statistical characteristics in the dynamic page generation of e-commerce workloads aiming to aid system modeling and design. First, knowing the distribution of the response time only as expressed by the mean of the empirical distribution and its higher moments is not enough since queuing behavior does change drastically if burstiness is present [4]. Second, from the systems perspective, observing the presence of correlation in the service process may help in identifying bottlenecks and changes in the work done in the system, which can in turn lead to the design of more effective resource allocation policies at the various tiers. Our experimental results consistently point at the existence of statistical correlation in dynamic page generation.

This paper is organized as follows. Section 2 summarizes related work. Section 3 gives an overview of the experimental environment and the tracing mechanisms we used across the various tiers. In Section 4, we present performance results that can be used for capacity planning, focusing on response time analysis for bottleneck identification. Finally, Section 5 summarizes our contributions.

2 Related Work

Understanding system workload is crucial for robust designs of web servers. Contemporary systems, including today's e-commerce servers, support objects that are mainly dynamically generated, and in contrast to static requests, they need an array of resources: CPU, memory, and I/O, resulting often in *significantly* longer service times than static requests due to the dominating cost of database activity. Multi-tiered architectures that separate the database server from the web and application servers are traditionally employed in such systems to improve performance, but introduces more points where bottlenecks may occur [2]. Furthermore, e-commerce systems have stricter requirements for quick response time, high security transactions, and persistent and reliable storage [17,5,8,11] making the need of detailed workload knowledge even more pressing. Several recently published studies indicate that different e-commerce servers do share similar characteristics: arrivals are best characterized as bursty with high degrees of self-similarity, the most significant portion of requests are for dynamically generated objects, and the popularity distribution is Zipf-like [17,10,16,3].

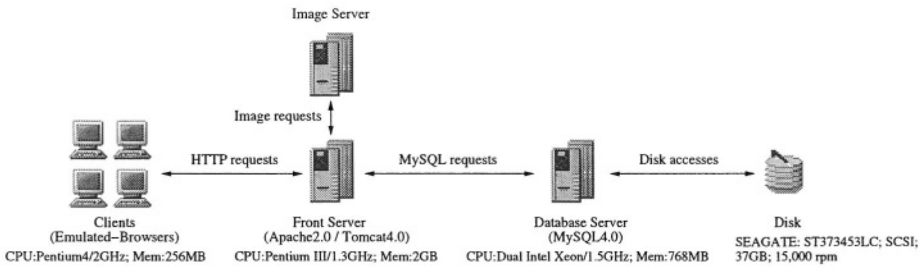


Fig. 1. The e-commerce site experimental set-up driven by the TPC-W workload.

Web server data on actual e-commerce sites are impossible to be obtained as studies on such systems are subject to non-disclosure agreements. Consequently, one can only resort to synthetic workload generators to study such systems, with the most prominent being the TPC-W benchmark [15]. Studies based on the TPC-W and its variants focus on bottleneck identification [6,1]. In these studies, there is a consensus that mostly always the CPU of the database server is the bottleneck while for the TPC-C workload the the time spent waiting for database locks dominates the transaction execution time [9]. Vallamsetty et al. [16], using traces from actual sites that depict autocorrelation in the arrival process, show that the response time distribution of dynamic object generation is heavy-tailed and that the service process also is long-range dependent.

In contrast to all of the above works, we dissect the stochastic characteristics of the response time distribution at the back-end server. In an e-commerce system driven by TPC-W workloads, the arrival process is *not* self-similar or correlated. As such, we isolate the performance implications of the service process at the database server. We further illustrate that autocorrelation in the response time distribution can be an outcome of the nature of the service process at the database server, which we attribute to the database memory and storage system interaction.

3 Experimental Environment

We conduct measurements in an e-commerce site, implemented according to the TPC-W benchmark [15], which simulates the operation of an on-line bookstore. A high-level overview of the experimental set-up and specifics of the software/hardware used are illustrated in Figure 1.

The load in the system is measured as the number of active clients, i.e., emulated browsers (EBs). As depicted in Figure 1, we generate the load from four clients machines. In the TPC-W implementation that we use [13], all requests received by the web server are treated as dynamic requests and forwarded as queries to the database server. The database contains all the data necessary to run the e-commerce site in ten MySQL tables. Although both the number of customers and the number of items for sale determine the database size, we found that the number of items is critical for performance. Therefore, we present results on four databases that are distinguished by the number of available items for sale (see Table 1).

TPC-W defines 14 different Web interactions which are coarsely classified as either browsing or ordering. According to the weight of each type of activity in a given traffic

Table 1. Sizes of the different databases

Number of Items	10K	100K	500K	1M
ITEM Table Size	5.1 MB	51 MB	256 MB	510 MB
Database Size	1.5 GB	1.5 GB	1.9 GB	2.1 GB

mix, TPC-W defines 3 types of traffic mixes: *browsing mix, shopping mix* and *ordering mix* in the order of increasing portion of ordering interactions [15]. Because we focus on understanding the system behavior under a variety of workloads, we introduce two additional traffic mixes that stress the system further by increasing the I/O traffic:

- *modified browsing mix,* where the percentage of new product searches accounts for 90% of all browsing requests while in the browsing mix new product searches are only 11% of requests, and

- *modified ordering,* where we raise the portion of administration interactions of the ordering mix from originally 0.11% to 10%.

Each experiment runs for 60 minutes, of which we discard the first 20 minutes in order to mask out warm-up effects.

4 Workload Analysis

In a multi-tiered system, the load of each tier has an impact on the user-perceived performance. The personalized nature of the requests sent to e-commerce servers makes the second tier, i.e., the database server, often the bottleneck [9]. Our experiments indicate that this bottleneck is triggered either by excessive *load,* as reflected by the number of simultaneous requests in the system, or by excessive *work,* as reflected by the amount of required system resources to service the requests.

4.1 Capacity Planning

Initially, we illustrate how load (see Figure 2) and work (see Figure 3) affect user perceived performance and consequently capacity planning. Figure 2 illustrates system throughput (measured in interactions per second) at the first tier, the front-end CPU utilization, and the database CPU and memory utilizations (i.e., the second tier) as a function of the number of emulated browsers in the system (i.e., system load). We do not report on the front-end memory utilization as it is always low in our experiments. Results are presented for one workload type, the browsing mix, and four database sizes (see Section 3).

Figure 3 illustrates the effect of different workloads. In this case, we experiment with only one database, i.e., the one with 500K items, and report on the system throughput, front-end CPU utilization, and database CPU and memory utilization for the ordering, shopping, and browsing workload mixes. Figure 3 indicates that workload affects system performance and resources availability. Observe that different workloads determine different levels of the sustainable system load, Finally, Figures 2 and 3 show that it is the database server that becomes the bottleneck independently of the load or the work in the system. In the following subsection, we focus on the database server performance and the characteristics of dynamic page generation.

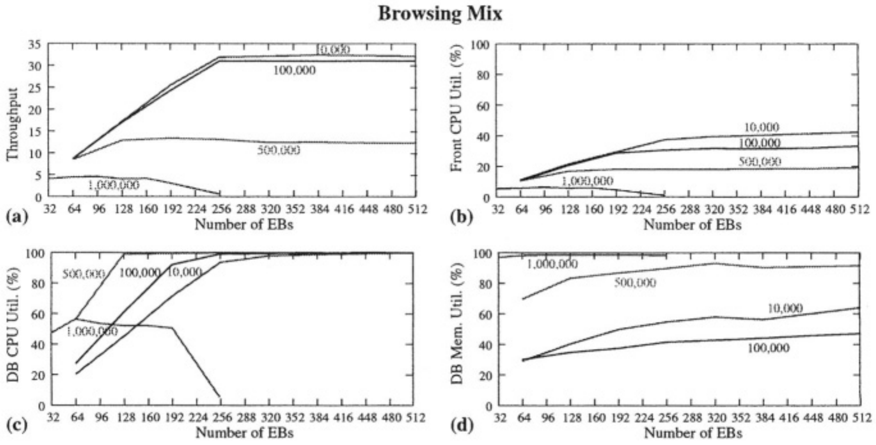


Fig. 2. (a) Throughput, (b) front-end CPU utilization, (c) database CPU utilization, and (d) database memory utilization for the browsing mix and database sizes of 10,000, 100,000, 500,000, and 1,000,000 items.

4.2 Database Performance

Figure 4 illustrates query time distributions, as a metric of the database server performance. Since the database server is the bottleneck, this metric directly relates to the user perceived performance. We focus on the query time distribution for various database sizes, loads, and workloads. For each database size, which essentially determines the level of system resources availability, higher load degrades database server performance. For the two large databases, where the system resources are limited, the impact of higher load is more apparent, resulting in distributions with considerably longer tails (see Figure 4(c)(d)). For various workloads (see the right column of Figure 4), the query time distribution for work-intensive workloads, such as the modified browsing mix, is quite different from that of less work-intensive workloads, such as the shopping or the ordering mix. We identify a workload as work-intensive when it requires more CPU, memory, and I/O to generate the response to a dynamic request.

As a piece of evidence that can assist us toward a more comprehensive understanding of the system behavior, we also plot the autocorrelation function (ACF) of the query times. The ACF is the autocorrelation between observations in a sample as a function of the lag, i.e., the distance between the observations. The **lag-k** autocorrelation value for $(k > 0)$, is computed as

$$\mathrm{corr}[X_0, X_k] = \frac{E[(X_0 - E[X])(X_k - E[X])]}{\mathrm{Var}[X]}$$

where X_0 and X_k denote two observations in the sample that are k events, i.e., lags, apart and $E[X]$ is the sample mean. Figure 5 presents the autocorrelation function of query response times for various databases, loads, and workloads.

Note that the arrival process in the database is independent with measured ACF as low as 0 and as high as 0.00005 for up to lag 1000, indicating practically no autocorrelation in

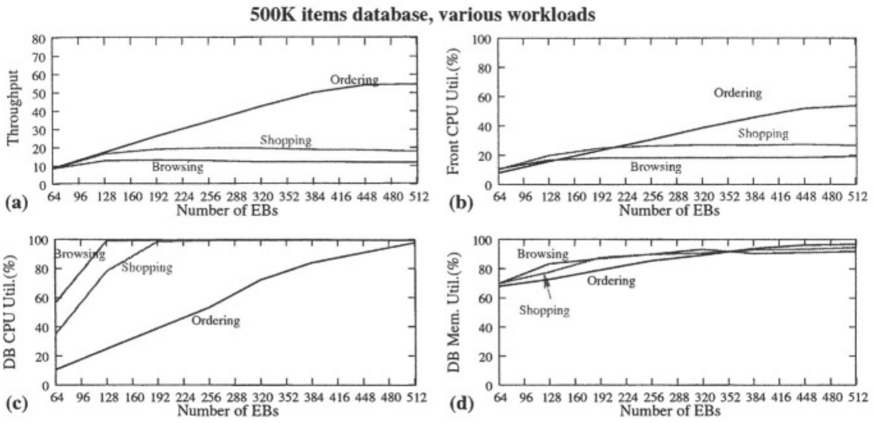


Fig. 3. (a) Throughput, (b) front-end CPU utilization, (c) database CPU utilization, and (d) database memory utilization as a function of system load for the database with 500,000 items and various workload mixes.

the arrival process.[1] Therefore, any autocorrelation that is measured in the query response time process is not due to the arrival process but depends on the scheduling discipline and the stochastic characteristics of the service process. The left column of graphs in Figure 5 shows the ACF of query times of two different load levels for each of the four database sizes. As load increases, the dependence nature of the response times increases. The graphs of the right column in Figure 5 represent the ACF of query response times for medium load but various workloads. Notice the dependence of ACF characteristics to the workload characteristics for the same system settings i.e., hardware, database size, and number of EBs. In the next subsection, we present an explanation of this behavior based on simple queuing systems and on resource consumption during the duration of the experiment.

4.3 Discussion and System Implications

Key to the behavior of the query response time is the interaction of the database CPU and data retrieval either from the database server memory or the disk. The ACF of the query response time sheds some light into this interaction. Recall that since there is no correlation in the arrival process at the database server, the observed ACF in the query response time is attributed only to the service process characteristics. The service process, of course, is complex as it involves two servers: the database server which operates under a time-sharing discipline and the disk which schedules requests depending on their physical location on the disk [14]. The closest theoretical scheduling schemes to those used by the database and the disk are processor sharing (PS) for the database server and first-come-first-serve (FCFS) for the disk. In reality, the operation of the system straddles between the two types of service, but looking at the two extremes it may help us understand system behavior.

[1] In the TPC-W workload specification, think and inter-request times at the EBs are exponentially distributed, resulting in an independent stream of arrivals to the database.

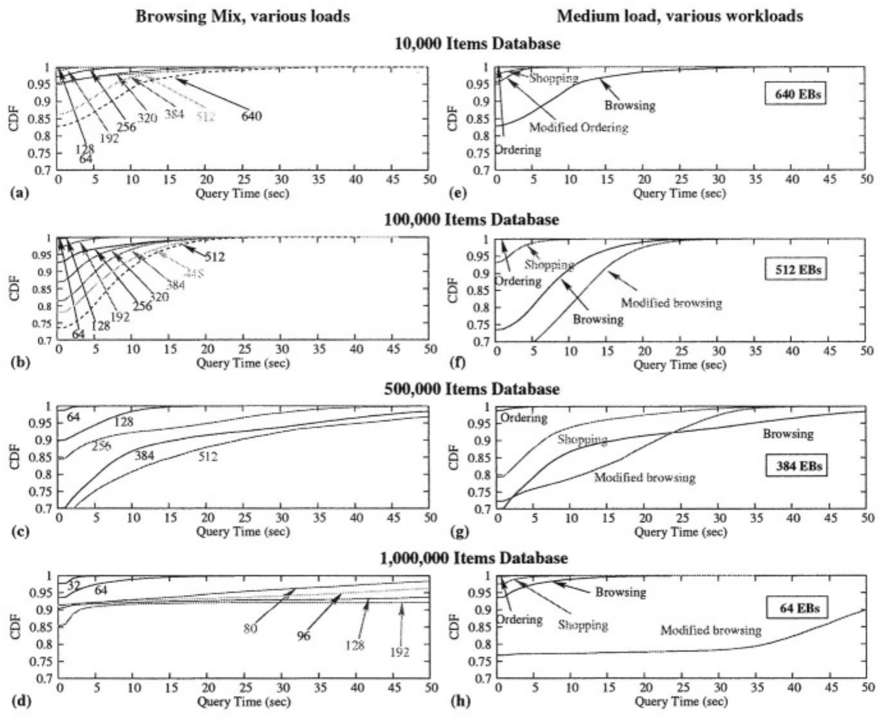


Fig. 4. Query time distribution (CDF) for the browsing mix under various loads (left column) and under medium load for various workloads (right column).

Figure 6 gives examples of the expected ACF in the response time as a function of the system load (as expressed by the system utilization) and the scheduling policy for a simple queue. All results are obtained via simulation of a 10 million sample space. Figure 6(b) illustrates the ACF for a system that accepts Markovian arrivals (i.e., there is no autocorrelation in the arrival process) and uses the FCFS service discipline. Two different experiments are presented. One where the service process is a hyperexponential distribution (captures variability but not correlation [7]) and one where the service process is a Markovian Arrival Process - MAP (captures variability and correlation [12] and depicted in Figure 6(a)). To isolate the effect of autocorrelation in the service process, we select the first and second moments of the hyperexponential and MAP processes to be the same. Figure 6(b) shows that the higher the load, the higher the ACF of the response time. The presence of autocorrelation in the service process also further increases the response time ACF. Figure 6(c) illustrates the same results but for the PS discipline. Again, the presence of autocorrelation in the service process is reflected in the response time process but it is not as high as the FCFS case. In this experiment, high loads tend to sustain correlation at higher lags, indicating the presence of long-range dependence.

The ACF of the workloads that we presented in the previous subsection lie somewhere in-between the ACF curves reported in Figure 6. To better understand the behavior of the service process, we take a close look at the database behavior using the modified browsing

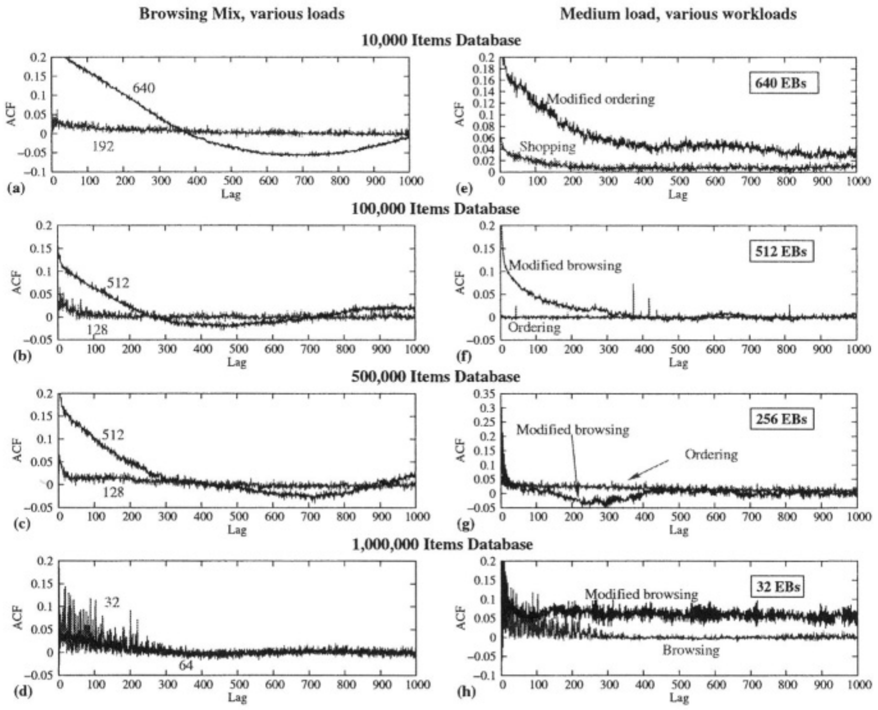


Fig. 5. ACF of query times at the database server.

mix (see Figure 7). The number of EBs is set to 32, which results in the significant arrival rate at the database disk (see Figure 7(a)). Note that in the same experiment but with the original browsing mix the arrival rate to the disk is at most 50. Figure 7(b) reports on the disk access pattern as a function of time. The entire disk is mapped on the y-axis which also marks the physical layout of each database table on the disk. The figure clearly shows that I/O accesses are bursty. Figure 7(c) reports on the memory utilization of the database server and shows that memory is periodically freed because of memory pressure. Intuitively, if there is memory pressure at the database, a memory miss suggests that another memory miss will soon occur, or if the disk is accessed then more accesses to the disk are to follow, which suggests that the service process is correlated. Such bulks of disk I/O operations result in heavier tail of the query-time distribution (see Figures 4(h)) and higher autocorrelation (see Figure 5), which is attributed to the slower service rates at the disk and the FCFS-like scheduling discipline. On the other hand, the burstiness of the disk accesses indicates that the system does not consistently suffer from memory misses. During the less I/O intensive period, CPU/memory performance, which employs PS scheduling discipline, dominates the response time.

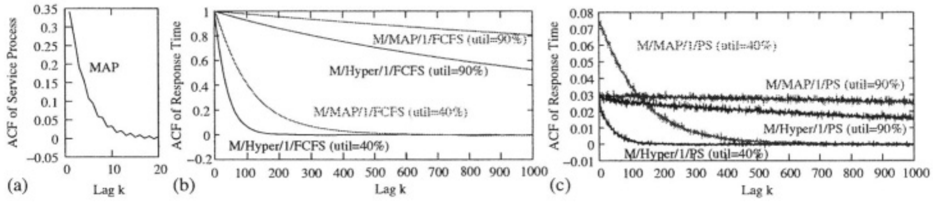


Fig. 6. (a) ACF of the service time distribution, (b) ACF of the response time distribution under FCFS, and (c) ACF of the response time distribution under PS.

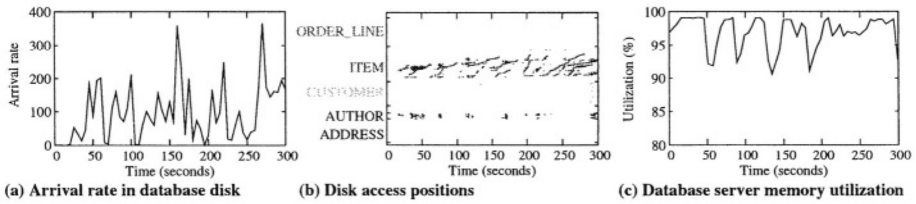


Fig. 7. Disk access pattern for the 1,000,000 items database. The system is under 90% new searches mix.

5 Conclusions

We used measurements in a 3-tier e-commerce system, built according to the TPC-W specifications, to evaluated how workload impacts system performance. We conducted our measurements in systems with different levels of resources, which we determine by the database size. By measuring resource utilization across all tiers, we identified that the time spent at the database server (including the disk) is the one that dominates user-perceived performance. We, further, analyzed the statistical characteristics of the query time distributions, and observe the presence of autocorrelation despite the fact that in our experiments there is no autocorrelation in the arrival stream of queries at the database. We showed that the workloads which utilize mostly the database server memory and less the disk, can sustain more load, while the workloads that are more I/O oriented sustain less load as a result of a slower service process and FCFS-like service discipline at the disk. In the future, we plan to use the results that we presented in this paper to construct approximate analytic models of multi-tier systems, in order to understand their behavior for a wider range of workloads and system resources.

References

1. C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic content benchmarks. In *5th IEEE Workshop on Workload Characterization(WWC-5),* Nov. 2002.
2. M. Andreolini, M. Colajanni, and R. Morselli. Performance study of dispatching algorithms in multi-tier web architectures. *ACM SIGMETRICS Performance Evaluation Review,* 30(2):10–20, Sept. 2002.

3. M. Arlitt, D. Drishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology,* 1(1):44–69, 2001.
4. A. Erramilli, O. Narayan, and W. Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Trans. Netw.,* 4(2):209–223, 1996.
5. G. M. C. Gama, W. M. Jr., and M. L. B. Carvalho. Resource placement in distributed e-commerce servers. In *The Evolving Global Communications Network (GLOBECOM 2001),* San Antonio, Texas, Nov. 2001.
6. D. Garcia and J. Garcia. Tpc-w e-commerce benchmark evaluation. *IEEE Computer,* pages 42–48, Feb. 2003.
7. L. Kleinrock. *Queueing Systems, Volume I: Theory.* Wiley, 1975.
8. Z. Liu, M. Squillante, and J. Wolf. On maximizing service-level-agreement profits. In *Third ACM Conference on Electronic Commerce,* pages 213–223, Tampa, Florida, Oct. 2001.
9. D. McWherter, B. Schroeder, N. Ailamaki, and M. Harchol-Balter. Priority mechanisms for oltp and transactional web applications. In *20th International Conference on Data Engineering (ICDE),* Boston, MA, Apr. 2004.
10. D. Menasce, V. Almeida, R. Riedi, F. Pelegrinelli, R. Fonseca, and W. Meira. In search of invariants for e-business workloads. In *SecondACM Conference on Electronic Commerce,* Minneapolis, MN, Oct. 2000.
11. D. Menasce, D. barbara, and R. Dodge. Preserving qos of e-commerce sites through self-tuning: A performance model approach. In *EC'01,* pages 224–234, Tampa, Florida, Oct. 2001.
12. M. F. Neuts. *Structured Stochastic Matrices of M/G/1-type and their Applications.* Marcel Dekker, New York, NY, 1989.
13. PHARM Project. *Java TPC-W Implementation Distribution.* Department of Electrical and Computer Engineering and Computer Sciences Department, University of Wisconsin-Madison.
14. A. Riska and E. Riedel. It's not fair - evaluating efficient disk scheduling. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis, and Similation of Computer Telecommunication Systems, (MASCOTS),* pages 288–295, Orlando, FL, Oct. 2003.
15. Transaction processing performance council. `http://www.tpc.org/`.
16. U. Vallamsetty, K. Kant, and P. Mohapatra. Characterization of e-commerce traffic. In *WECWIS2002,* Newport Beach, California, 2002.
17. Q. Wang, D. Makaroff, H. K. Edwards, and R. Thompson. Workload characterization for an e-commerce web site. In *CASCON 2003,* pages 313–327, Toronto, Ontario, Canada, Oct. 2003.

Author Index