Taxonomy and Design Analysis for Distributed Web Caching

Sandra G. Dykes Clinton L. Jeffery Samir Das
Division of Computer Science
University of Texas at San Antonio
{sdykes, jeffery, samir}@cs.utsa.edu

Abstract

Proxy server caches reduce Internet traffic and improve response times. However, limited duplication among requests restricts most proxy server hit rates to under 50%, requiring other methods to expand cache sharing. This paper proposes a taxonomy for distributed network caching based on discovery, dissemination, and delivery methods. We analyze the match between taxonomy categories and Web characteristics, and use the taxonomy to classify current Web caching projects. Next we describe our cooperative Web caching protocol, in which proxy servers locate cache copies by looking in local metadata directories. Local lookup provides fast discovery, and allows multiple criteria be used to select a cache site. We propagate metadata by lazy prefetching, in which returned objects carry metadata for related or popular objects. The protocol is simulated using empirically-derived analytical workloads. Results indicate it substantially reduces server load and connection denials as compared to standard proxy server caching.

1. Introduction

Rapid growth and the bursty nature of Internet HTTP traffic have increased connection refusals and response time variability. Connection refusals occur when a Web server lacks sufficient resources to handle the current load. Large and unpredictable response times occur when network routers and exchange points become congested. These problems are exacerbated by the poor scalability of HTTP's direct client-server connection protocol.

Caching is widely viewed as necessary for improving scalability and reliability of the Web. However, caching does not scale if all users access a single cache group. Organizing users into groups and using a different cache for each group solves the scalability problem, but limits hit rates because it restricts the amount of available sharing. This is exactly the case with Web proxy servers. A proxy server caches Web objects for its clients, and shares cached objects across its user community. Unfortunately, there is not enough request duplication within most communities to achieve high hit rates. Studies show that even with infinite cache size, the maximum hit rates of proxy server caches range from about 30% to 50% [1], [7], [10]. To improve hit rates, the Web needs a scalable method for widespread cache sharing.

In this paper, we discuss guidelines for designing distributed Web caches, introduce a taxonomy for distributed network caches, and propose a Web caching protocol built upon cooperating proxy servers. The taxonomy is useful for comparing caching projects because it separates basic design choices from implementation details. Further, it helps focus on factors that may not be measured in simulation or performance studies. We analyze how alternative approaches in the taxonomy match characteristics of the Web. Although our analysis is qualitative, it serves as a guideline for basic design decisions.

Guided by the taxonomy analysis, we describe a cooperative Web caching protocol designed to reduce both response time and network congestion. Our design organizes proxy servers into a flat mesh, which incurs much lower delivery cost on wide-area networks than do hierarchical organizations [21]. However, flat organizations have the problem of discovering where objects are cached. Our answer is to use local *metadata* directories [21] that store information about data in other caches, and to propagate the metadata using a piggybacking technique we term *lazy prefetching*. When a proxy returns a cached object to another proxy, it piggybacks a list of the most popular files in its cache. After an initial warmup period, proxies know cache locations of many popular objects. This takes advantage of the skewed popularity of Web pages [4], [19] to build a cache mesh with low discovery cost *and* low delivery cost.

Metadata directories have a second advantage in that they separate location discovery from location selection. That is, proxies first obtain a list of cache sites with copies, then select a cache from the list. This separation allows various criteria be used to choose a cache site, such as site load, network proximity, copy time, or security privileges. Any num-

0-7695-0001-3/99 $10.00 (c) 1999 IEEE

ber of criteria are possible, provided the metadata includes the necessary information.

We begin by describing design guidelines for distributed Web caches in Section 2. Section 3 tabulates characteristics of Web server workloads reported in the literature and extracted from our traces. Section 4 describes our taxonomy and Section 5 uses it to classify related research in Web caching. Our cache design is explained in Section 6, simulation results are given in Section 7, and conclusions presented in Section 8.

2. Design guidelines for distributed Web caches

A distributed network cache should improve both individual user performance and global network performance. A cache design is unacceptable if it improves an individual's response time at the expense of increased network congestion and causes longer delays for other users. Unlike caches for memory and disks, cache sites on wide-area networks are not inherently faster than servers because cache sites do not necessarily have faster hardware or larger bandwidths. If caches and servers have similar resources and connection bandwidths, remote network caching actually increases average response time *unless* it offsets discovery overhead and cache miss penalties by reducing delivery time. We can reduce delivery time if we know which cache sites have copies and then select the fastest site. Unfortunately, it is difficult to predict response times because they depend upon current server load and network congestion, as well as upon static factors such as server capacity and network topology. Because Web pages typically consist of several independent objects, we can reduce overall page retrieval time by concurrently retrieving objects from different cache sites. This concept is similar to parallel HTTP, in which browsers send concurrent requests to the same server. However, parallel HTTP perturbs the TCP/IP congestion control mechanism by generating multiple client-server connections that use the same network route [5]. Spreading the concurrent requests across different sites reduces the traffic per route, which reduces burstiness because queuing delays on different routes are independent. To summarize, design goals for a distributed Web cache should include 1) low discovery cost, 2) object dissemination that adapts to rapid shifts in popularity, 3) a method for selecting fast cache sites, and 4) concurrent delivery of page or object components from different cache sites.

3. Characterization of HTTP workload

Cache designs perform well only if they match the workload, so first we examine the pattern of HTTP requests. Table 1 summarizes statistical characteristics of HTTP server workloads reported by a number of researchers, and Table 2 reports the HTTP workloads we observed on two Web servers at the University of Texas at San Antonio: UTSA-CS and UTSA-VIS. Characteristics of the UTSA traces appear similar to traces reported in the literature. Studies uniformly show skewed popularity distributions: most Web requests are for a small fraction of the objects. Arlitt and Williamson [4] report object size is well-modeled by Pareto distributions and the interrequest intervals for individual objects follow exponential distributions. Seltzer's results [19] show object popularity varies by region, and Web invariants apply most strongly to the more popular servers.

3.1. Web page statistics

Two statistics not reported in these earlier studies are the fraction of embedded image requests and the number of embedded images per page. These data are important for evaluating client prefetching or concurrent retrieval algorithms. In the UTSA traces, we found embedded images account for an average of 41% of the transfers and 33% of the returned bytes, with an average of 3.2 images per page.

3.2. HTTP session statistics

We categorize HTTP requests as either *session requests* or *component requests*. Users initiate a session request when they instruct the browser to retrieve a Web page or an independent object. For Web pages, the browser first retrieves the HTML file (the session request), then issues component requests for the embedded objects. Requests for independent objects such as multimedia, postscript, and text-only HTML files generate only a session request; no component requests follow. In the UTSA traces, we observed an average of 59% session requests and 41% component requests. Within the session requests, 13% correspond to multi-object Web pages and 46% to single, independent Web objects.

Distinguishing between session and component requests is important because human-initiated session requests are uncorrelated, while machine-generated component requests are correlated with other requests within the session. Consequently, we expect different arrival time behaviors. For FTP, TELNET, SMTP and NNTP, Paxson and Floyd [18] find Poisson distributions are valid for user session arrivals, but not for machine-generated requests. Machine-generated requests exhibit burstier behavior that is better modeled by self-similar processes. In particular, Paxson and Floyd show FTP session arrivals follow a Poisson distribution, while FTPDATA connections within a session are clustered into bursts for which Poisson modeling fails.

Characteristic	*Reference*	*Detail*
Server load is growing exponentially	Seltzer [19]	Requests and documents stored show exponential growth.
Most requests are from remote clients	Arlitt [4]	Remote clients: $\geq$ 70% of requests and $\geq$ 60% of bytes.
Arrival time distribution is heavy-tailed	Mogul [15]	Appears log-normal
Most transfers are small	Arlitt [4] Almeida [2]	Mean size $<$21 KB Image 13 KB, Text 4 KB, Audio 179 KB, Video 2300 KB
Object size distribution is Pareto	Arlitt [4] Crovella [9]	$0.40 < \alpha < 0.63$ for transferred objects $\alpha = 1.06$ for transferred objects
Most transfers are images or HTML	Arlitt [4] Gwertzman [12] Almeida [2]	Image 36-78% HTML 20-50% Dynamic 0-7% Image 65% HTML 22% Dynamic 9% Image 75% HTML 19% Dynamic 0% Audio 5% Video 0.4%
Object popularity is skewed	Arlitt [4] Bestavros [7]	10% of objects sent satisfy 90% of transfers 5% of bytes sent satisfy 85% of byte traffic
Most transfers are duplicates	Arlitt [4]	$>$97% sent more than once.
Objects have long lifetimes	Bestavros [7]	JPEG 100 days GIF 85 days HTML 50 days
Popularity varies by region	Seltzer [19]	Clients tend to access geographically related servers.
Popularity can change rapidly	Seltzer [19]	Objects and server popularity can change very fast, producing *flash crowds*.

Table 1. HTTP Web servers characteristics summarized from the literature.

Characteristic	*UTSA-CS*			*UTSA-VIS*		
Dates	Apr 97 - Sept 97			May 96 - Aug 96, Dec 96 - Aug 97		
Number of requests	561,292			547,272		
Traffic (bytes transferred)	3.8 GB			2.9 GB		
Remote clients	82.4%			77.9%		
Mean object size	$<$12 KB			$<$11 KB		
Object type	Size	Transfers	Bytes	Size	Transfers	Bytes
HTML	4 KB	44.2%	15.2%	4 KB	41.9%	15.2%
Image	15 KB	47.6%	61.8%	6 KB	53.7%	30.4%
Audio	200 KB	0.4%	7.6%	81 KB	0.1%	1.1%
Video	na	0.0%	0.0%	452 KB	0.9%	40.2%
Application	135 KB	1.0%	12.2%	386 KB	0.3%	10.2%
Dynamic	1 KB	3.6%	0.3%	1 KB	0.1%	0.0%
Other	11 KB	3.2%	2.9%	10 KB	3.0%	2.9%
Object popularity	10% of objects = 69% of transfers			10% of objects = 79% of transfers		
Duplicate transfers	$>$99% sent more than once			$>$99% sent more than once		
Embedded images	37% of transfers 43% of bytes			46% of transfers 24% of bytes		
Request types:						
Session, single file	50%			41%		
Session, multi-file	13%			13%		
Component	37%			46%		
Embedded images per page	2.9			3.5		

Table 2. UTSA Web server characteristics.

Service	*Categories*	*Details*
Discovery	**Fixed cache**	Client always sends request to the same cache.
	Group query	Client queries a group of cache sites.
	Manual	Clients maintain individual member addresses.
	Auto	Clients maintain a group address.
	Directory lookup	Client looks up object in a metadata directory.
	Centralized	One centralized directory for each server.
	Distributed	Metadata directory is distributed across nodes; (distributed structure may be hierarchical, mesh, etc.)
Dissemination	**Client-initiated**	Object cached as a result of client request (*pull-caching*)
	Server-initiated	Server decides what, where and when to cache (*push-caching*)
Delivery	**Direct**	Object always returned directly to the client.
	Indirect	Object may pass through several sites before reaching client.

Table 3. A taxonomy for distributed network caches.

4. Taxonomy of distributed network caches

Distributed network caches provide three services: *discovery, dissemination* and *delivery* of cache objects. Discovery refers to how clients locate a cached object. Dissemination is the process of selecting and storing objects in the caches; that is, deciding which objects are cached, where they are cached, and when they are cached. Delivery defines how objects make their way from the server or cache site to the client. Our taxonomy is given in Table 3 and explained in the sections below.

4.1. Discovery

In *fixed cache discovery*, a client sends all cache requests to a single, pre-configured location. Servers are stateless; it is the client who maintains knowledge of the cache location. Stateless server caching has two advantages: no caching information is lost in the event of server failure, and the design requires no changes to existing Web servers. Fixed cache discovery is used in proxy server caching, where browsers are configured to send all requests to their site's proxy server.

In *group query discovery*, clients locate cached copies by querying members of a cache group. This permits servers to be stateless. Clients are responsible for knowing where to send their queries, requiring some method for maintaining group membership information and addresses. Two methods have been proposed: *manual configuration* and *automatic configuration*. With manual configuration, clients must be informed when the group membership changes. Typically the clients' systems administrators must manually modify their configuration files. This is the approach in the Harvest [8] and Squid [20] caching software, which run on many national cache systems throughout the world [16]. Zhang and coauthors point out manual configuration is cumbersome, error-prone and does not scale. Instead, they propose using IP multicast to automatically configure cache groups [22]. Queries are sent to a multicast address, so clients need not be reconfigured every time a member joins or leaves the group. The drawback to group query is that competing factors constrain group size. If the cache group is too large, the client floods the network with query packets. IP multicasting will not relieve this flooding until true multicast routers are in wide-spread use, and even then, replies to the query may cause congestion and overloading. On the other hand, Web caching cannot achieve high hit rates unless cache groups are large because there is not enough overlap in client requests within smaller groups. Cache hierarchies attack the size problem by organizing groups into a cache tree, which increases the sharing exponentially with number of levels. Although hierarchies improve aggregate hit rate, the response time suffers because each subsequent cache level miss spawns another remote network transfer of the object. Thus distributed cache hierarchies cannot achieve both high hit rates and low overall response time.

With *directory discovery*, the client locates cached copies by looking in a metadata directory. Along with copy locations, directories may store timestamps or version numbers for cache consistency, access control data for security, and site performance figures for use in resource selection. Retrieving metadata involves the same discovery, dissemination and delivery issues as in retrieving the data objects. However, metadata is almost always much smaller than the corresponding data, which lowers propagation and prefetching costs.

Centralized directories store metadata for a server's ob-

jects together on one network node, with the obvious advantage that clients immediately know which directory to contact. Unfortunately, centralization does not scale and ultimately leads to congestion and bottlenecks. Server-initiated dissemination projects typically use stateful servers and centralized directories. *Distributed directories* spread metadata for a server's objects across the network.

4.2. Dissemination

Client-initiated dissemination ("pull-caching") is a client-driven technology in which clients determine what, when and where to cache. Servers may be stateless because they do not need to know client request patterns; the client-driven nature automatically matches object distribution to request patterns. The problem of cache consistency arises if servers are stateless, but this not associated *per se* with client-initiated dissemination. The major advantage of client-initiated dissemination is that it automatically adapts to rapidly changing request patterns. This is critical when hot spots or flash crowds occur.

Server-initiated dissemination ("push-caching") is a server-driven technology in which servers choose what, when and where to cache [7],[11]. Cache sites may, however, have the authority to refuse or remove objects. Replication is a server-initiated dissemination method that restricts the authority of the cache sites to refuse or remove objects. On the Internet, replication is commonly associated with "mirror sites" that duplicate an entire Web site. In general, server-initiated dissemination operates on a finer scale by storing individual objects at remote cache sites. Server-initiated dissemination uses stateful servers because servers need historical data to make dissemination decisions. This server-centric approach provides strong consistency and assures objects with long-term demand are not prematurely replaced by short-term, "hot" items. Conversely, server dissemination does not cope well with rapid or localized changes in request patterns [11]. A second disadvantage lies in resource and security concerns that arise because cache sites store unsolicited objects instead of only storing objects requested by local users. This raises serious concerns on the Internet, where sharing occurs across different companies and governments. In short, server-initiated dissemination works best for objects with long-term or static request distributions, or for objects where consistency is more important than response time.

4.3. Delivery

Direct delivery always returns the object directly from the "hit site" to the client, where hit site refers to the cache or server at which the object is found. This method assures the lowest latency for delivery.

In *indirect delivery*, the object may travel through intervening cache layers on its path from hit site to client. Each layer adds a store-and-forward network transfer, which generates longer and more variable response times, and increases the number of packets on the network. Performance penalties are greatest for large multimedia objects. Typically, indirect delivery follows the reverse request path. Intervening cache layers may save the object as it passes through; combining delivery with dissemination.

The HTTP protocol has no provision for returning an object other than via the request connection. This has repercussions for multi-level cache protocols that may send the request through several cache levels. Consider the following scenario: 1) X requests an HTTP object from C_0, 2) C_0 misses and requests the object from C_1, and 3) C_1 hits. Using HTTP, C_1 cannot return the object directly to X because C_1 does not know X made the request. Even if C_1 did know, HTTP cannot deliver the reply through a new connection. Consequently, multi-level Web caches must either use indirect delivery or use a protocol other than HTTP.

4.4. Analysis: Which methods match the Web?

Our taxonomy analysis argues the best distributed caching approaches for Internet HTTP traffic are 1) directory-based discovery using a distributed directory, 2) client-initiated dissemination, and 3) direct delivery.

Directory discovery methods are best suited because Web caching achieves high hit rates and low response time only if caches are distributed, cache sharing is widespread, and discovery overhead is low. Fixed cache and flat group query methods do not allow scalable, widespread discovery, while multi-level groups suffer from the latency of multiple store-and-forward transfers. Distributed directories offer scalable discovery, provided the metadata is propagated efficiently. Further, the smaller size of metadata records opens up possibilities for prefetching and piggybacked propagation.

Client-initiated dissemination is preferred because it best matches the Web's rapid popularity shifts and flash crowds. Direct delivery is preferred because it produces lower latencies, fewer TCP/IP connections, and less network traffic. Multi-level Web caches use indirect delivery because they are constrained by HTTP semantics. This argues distributed Internet caches should be organized into a relatively flat structure for delivery purposes.

5. Classification of Web caching research

Combining choices for discovery, dissemination and delivery yields twenty taxonomy classes. In Table 4, we classify some Web caching projects according to the taxonomy.[1]

[1] We assume the use of proxy server caching, therefore a protocol is classified as direct delivery if objects are returned directly to the proxy server.

Discovery	*Dissemination*	*Delivery*	*Projects*	*Ref.*
Fixed cache	Client-initiated	Direct	Proxy server caching	[3]
Group query, Manual	Client-initiated	Indirect	Harvest/Squid hierarchical caches NLANR, other national caches	[8] [16]
Group query, Automatic	Client-initiated	Indirect	Adaptive Web Caching Cooperating WWW cache servers	[22] [14]
Directory lookup, Centralized	Server-initiated	Direct	Geographic Push-caching Demand-based Document Dissemination	[11] [6]
Directory lookup, Distributed	Server-initiated	Direct	Metadata hierarchy	[21]
Directory lookup, Distributed	Client-initiated	Direct	Server-Directed Proxy Sharing	[13]

Table 4. Classification of some distributed Web caching projects.

Manual group query projects using Harvest and Squid caches organize proxy servers into hierarchical groups. When a cache enters or leaves the system, its siblings and children must be manually reconfigured. The National Laboratory for Applied Network Research (NLANR) manages a hierarchical Internet cache system built upon Squid caches. NLANR supplies the top level cache group, proxy servers constitute the intermediate levels, and users make up the bottom level. Cache misses in NLANR are expensive because each miss adds an HTTP connection and object transfer, which are costly operations on wide-area networks.

The automatic group query project of Zhang, Floyd, and Jacobson [22] partitions users into multicast groups for discovery and dissemination. IP multicast allows for automatic group configuration: caches join and leave groups without having to reconfigure other group members. Within a cache group, the request is multicast to all members. If no member has the object, the request is forwarded via a cache in the overlapping group closest to the server. Delivery follows the reverse route, with the object multicast to every group along the way. Acceptable performance will likely require hardware IP multicast to avoid flooding the network with queries. Because group members need to determine which overlapping group is closest to the server, this project requires some integration between caches and routers. Malpani, Lorch and Berger [14] propose a different multicast discovery design that uses a single group. If the group misses, the protocol resorts to contacting the server. Here the problem is scalability: a single group design does not scale to the Internet, but smaller groups suffer from the same lack of request overlap that limits proxy server hit rates.

Push-caching projects of Gwertzman and Seltzer [11] and Bestavros and Cunha [6] use stateful servers who decide what, where, and when to cache. Clients send requests directly to the Web server, which redirects the client to a "nearby" cache. Servers decide which cache is nearest the client by comparing geographic or network topology (hop count) information. However, Crovella and Carter [9] show geographic distance and hop count both poorly predict the latency of Internet traffic. Consequently, these projects face difficulties in responding to flash crowds and in selecting cache sites.

Most similar to our design is the metadata caching project of Tewari, et.al [21]. As with our design, cache location information is stored and propagated separately from data objects. However, they store cache metadata in a distributed hierarchy and propagate the metadata by moving it through the hierarchy. Conversely, our design stores metadata in local directories and propagates it by lazy prefetching during object transfer.

6. Server-directed proxy sharing (SDP)

6.1. SDP components and features

SDP implements Internet-wide cache sharing through cooperating proxy server caches. Proxies find cached copies by looking in local metadata directories, and propagate the metadata by piggybacking it onto data transfers. In concert with directory-based discovery, SDP uses client-initiated dissemination and direct delivery. Figure 1 shows SDP's components, and Table 5 highlights its major features.

SDP employs four information structures: *Proxy Tables*, *Proxy Lists*, *Cache Site Directories* and *Popular Lists*. Each server maintains a Proxy Table containing information about copies of its objects, including cache IP addresses. From

Features of Server-Directed Proxy Sharing
Cooperating proxy servers share cached objects.
Proxy servers are organized into a flat mesh.
Proxies find cached copies by looking in local metadata directories.
Metadata is propagated by lazy prefetching.
Cache discovery is orthogonal to cache selection.
Proxy servers cache only objects requested by local clients.

Table 5. Features of the SDP cache design.

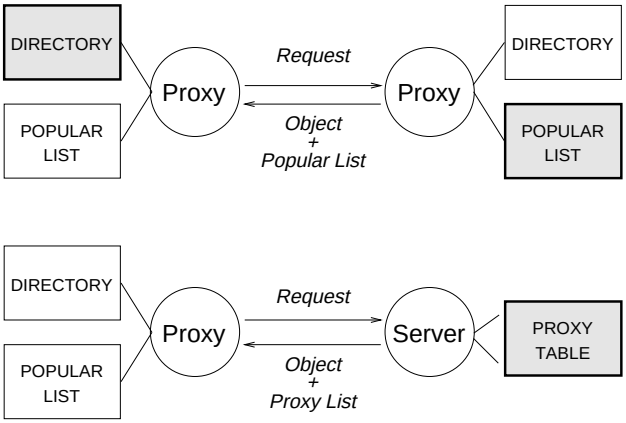

Figure 1. Proxy to proxy request (top), and proxy to server request (bottom).

```
proc SDP_GET (object)
    if (object in local data cache) then return object
    else if (object is listed in Directory) then
        select best cache site
        request object from selected site
        if (ok) then
            store Popular List in Directory
            return object
        end if
    end if

    /* Object is not in Directory or request failed */
    request object from primary server
    if (Proxy List was attached to reply) then
        store Proxy List in Directory
    if (Proxy List and no object) then SDP_GET(object)
    return object
end
```

Figure 2. SDP client proxy

the Proxy Table, the server compiles Proxy Lists of individual object metadata. Each proxy server maintains a Popular List and a Cache Site Directory. A proxy's Popular List points to the most popular objects in its local cache, while its Cache Site Directory contains metadata for objects at other cache sites. Metadata is used for both discovery and site selection. Criteria for selecting sites can change, so metadata records have variable length and use tags to indicate information type. For example, metadata can include copy timestamp, security data and site performance information such as average load, network bandwidth, or latency history. Metadata is propagated by lazy prefetching: 1) when a server returns an object, it appends a Proxy List for related objects such as embedded images, and 2) when a proxy returns an object copy, it appends its Popular List. After receiving objects from a few caches, a proxy knows where many other globally popular objects are cached.

6.2. SDP protocol

Users send requests to their local proxy server. If the proxy has the object cached, it returns a copy; otherwise it retrieves the object as explained below. To better follow the discussion, refer to pseudocode in Figures 2 and 3.

1. Client proxy looks in its local Cache Site Directory.
If the directory lists one or more cache sites for the object, the client proxy selects a site. If directory lookup fails, the proxy contacts the original server. Information in Cache Site Directories is not meant to be comprehensive; its function is to provide fast hints about copy locations, with small miss penalties and low propagation overhead.

2. Client proxy selects a site.
Here we face a critical issue: how does the client proxy select a site? One benefit of SDP is that it separates discovery from site selection, allowing us to choose sites based upon any criteria. The server is included in the list of candidate sites because it may be the best choice. Our current criteria is a fast response time, as predicted by static, statistical and dynamic estimators. Static estimators include site bandwidth and whether or not the client proxy and site share regional or backbone networks. Statistical estimators include the site's average available bandwidth, server load, and past performance for this client-site pair. Dynamic estimators are run-time measurement of current conditions. and include ICMP echoes ("pings") for estimating latency and ICMP packet pairs for estimating available bandwidth.

3. Server returns object and/or Proxy List
or Cache site returns object and Popular List.
When a server returns an object, it appends a Proxy List for related objects and updates its Proxy Table. If a server is too busy and the object is large, the server instead returns a Proxy List for the object. When a cache site returns an object, it appends its Popular List. If a cache site is too busy or has removed the object, it returns an error and the client proxy sends the server an ordinary HTTP request. Proxy Lists and Popular Lists need not be comprehensive: large lists are culled using static criteria mentioned above. Following successful requests, the client proxy stores the piggybacked metadata in its Cache Site Directory, and returns the object to the user.

6.3. Concurrent retrieval of embedded objects

If the requested object is a Web page, the client proxy retrieves the embedded images concurrently from different cache sites (see Figure 3). The client proxy selects the best

```
proc SDP_GET_EMBEDDED (Web page)
    select set of best cache sites
    for each (embedded object)
        send request to next site
        mark site as "Unavailable"
    end

    /* Receive objects as they come in and return them */
    while (more objects)
        if (object received) then
            mark page site "Available"
            store Popular List in Directory
            return object
        else if (proxy timeout) then
            request object from next available site
        else if (proxy error) then
            request object from primary server
        else if (server timeout or error) then
            return error
    end
end
```

Figure 3. Concurrent image retrieval.

site, and sends it the first image request. Without waiting for a response, the client selects the next best site and continues until all images have been requested or all sites have outstanding requests. That is, no site is sent more than one request at a time. To guard against excessive delay, the client sets reply timers and sends the request to another site if a timer expires. The request to the slower sites is cancelled after the image is received. This technique can also be used for other strongly related objects. For example, large independent objects might be split up and retrieved concurrently.

6.4. Cache consistency issues

Standard proxy server caching does not guarantee cache consistency: a proxy cache may return stale objects to its local users. Object timestamps, expiration headers, and proxy server directives are used to limit stale returns. SDP provides the same safeguards by including the timestamps and expiration dates in the metadata, thereby ensuring cache consistency is no worse for shared objects than for objects returned from the local proxy cache.

7. Analytical simulation

We simulated SDP and normal proxy caching using empirically-derived analytical workloads. Unlike trace-driven workloads, statistical workloads can be easily varied, which allows for a systematic investigation of performance. Consequently, we can model the system over a wide range of parameter space. While traces provide a more exact model, each trace represents only a single point in parameter space. The ability to vary parameters is especially important when traffic characteristics vary between sites or over time. Paxson [17] provides a clear discussion of these and other advantages in a study comparing analytical and empirical models of wide-area TCP/IP traffic. Table 6 lists the workload distributions, parameters and probabilities we used in the simulation. These were gathered from server traces cited in Table 1 and from our own UTSA traces.

Workload Variable	*Distribution and Probability*	
Interarrival times		
Session	Exp(a)	$a = varied$
Component	Log_{10}-normal(a, b)	$a = -0.7$
$mean = 221ms$		$b = 0.3$
Connection duration (sec)	Log_{10}-normal(a, b)	$a = -0.75$
$mean = 289ms$		$b = 0.65$
File size (KB)	Pareto(a)	
HTML	$a = 4$	$P = 0.430$
Image	$a = 11$	$P = 0.506$
Audio	$a = 140$	$P = 0.003$
Video	$a = 452$	$P = 0.004$
Application	$a = 260$	$P = 0.007$
Dynamic	$a = 1$	$P = 0.019$
Other	$a = 11$	$P = 0.031$
Page request (multi-obj)	Fixed	$P_{page} = 0.13$
Embedded objects per page	Normal (a, b)	$a = 3.2, b = 1$

Table 6. Simulation workload.

7.1. Interarrival times (T_a) and service times (T_s)

We use two interarrival time distributions: Poisson for session requests and log-normal for component requests generated by the session. Other research suggests overall HTTP interarrival times are not Poisson, but these analyses did not distinguish between session requests and component requests. For FTP, Paxson and Floyd show session arrivals do appear to be Poisson, while FTPDATA connections spawned by the session are better approximated by a heavy tailed distribution such as log-normal or log-logistic [18]. For normal proxy caching, we assume the client uses parallel HTTP; that is, all inline image requests are sent concurrently rather than sequentially. We model the average interarrival time of a group of parallel inline requests as a heavy-tailed log-normal distribution with respect to the end of the corresponding HTML transfer, and randomly distribute arrivals of the requests around this mean. Because these arrival times of inline images reflect the time for the HTML file to reach the client and an inline request to reach the server, we used parameters that approximate the 207 ms measurement of Mogul for round-trip (RT) pings of 536 byte packets [15]. The service time distribution is based on our interpretation of Mogul's graph of connection durations from

the DEC server traces [15]. For well-connected servers, the connection duration will likely depend upon network bandwidth of the proxy, therefore the duration distribution should be similar for most servers.

7.2. Simulation metrics

For offered load we compute the *request intensity* (T_s/T_a), which normalizes request rate with respect to service rate. Here T_a reflects the request rate generated by client proxies. In normal proxy caching, the server receives all these requests. In SDP many of the embedded object requests are never seen at the server because they are sent to cache sites. Hence the request rate at the SDP server is lower than the request rate generated by the client proxies. Larger request intensities indicate busier servers. For normal proxy caching, the server becomes overloaded and starts refusing connections at $T_s/T_a = 1$. We compared SDP to normal proxy caching by computing server load (arrivals/sec and MB/sec) and percent refused connections as a function of request intensity.

7.3. Simulation results

Simulation results predict that SDP significantly reduces server load when compared to normal proxy caching (see Figures 4 and 5). Consequently, SDP allows the server to handle more requests, thus reducing the number of denied connections (see Figure 6). A particularly critical region lies in the range $0.5 \leq T_s/T_a \leq 1.5$, where servers become saturated and begin to refuse connections. Within this region, SDP reduces the server's traffic load (MB/s) between 41% to 63%, and reduces the request load (arrivals/s) between 26% to 38%. Past the critical region, the SDP advantage levels off to approximately a 35%-40% reduction in traffic load and a 10%-15% reduction in request load. More importantly, SDP does not begin connection refusals until well after normal proxy caching, and continues to refuse a smaller number as traffic intensity increases.

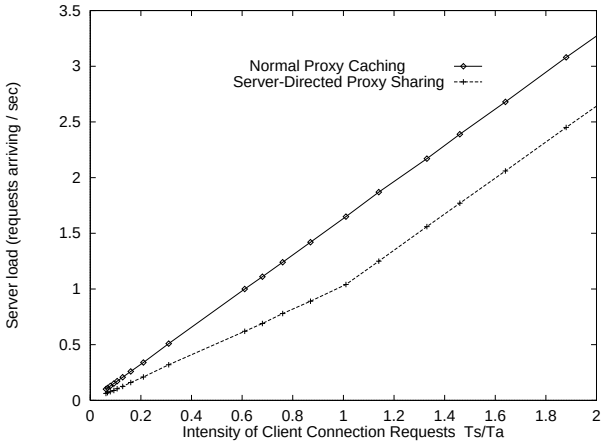

Figure 4. Simulation: Server request load.

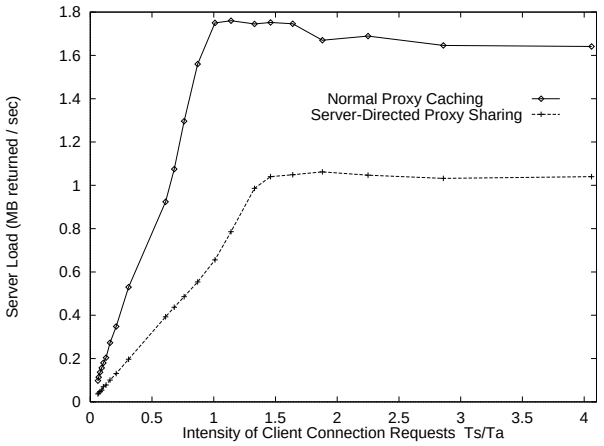

Figure 5. Simulation: Server byte load.

8. Conclusions

The SDP protocol is designed to match known characteristics of the Web, the most important being limited request duplications within user groups, skewed popularity distributions, rapid popularity shifts, fluctuating network congestion, and the structure of Web pages. By organizing cache sites as a flat mesh and propagating metadata with lazy prefetching, SDP combines low delivery cost with low discovery cost. Lazy prefetching takes advantage of skewed popularities, and propagates metadata with little added network traffic. This method is unique to our Web cache design. A second important feature of SDP is the separation of discovery from cache site selection, allowing site selection to be based upon multiple and configurable criteria. Lastly, we believe the Internet presents special problems for network caching because it has no central controlling authority. Our design takes into account this autonomous administrative nature. In push-caching, the local caches store unsolicited objects that may not be needed by local users, raising security and resource-sharing issues. Hierarchical cache designs require dedicated caches supported by government funding or a "pay-for-services" plan; neither is a particularly attractive option. In contrast, SDP requires only that local proxy servers share information they previously cached for their local users. Thus SDP matches both technical and political characteristics of the Web.

SDP's combination of distributed directory-based discovery, client-initiated dissemination, and direct delivery might be adapted to form a potent replication strategy for other distributed replication systems. However, the details of the SDP protocol exploit specific characteristics found

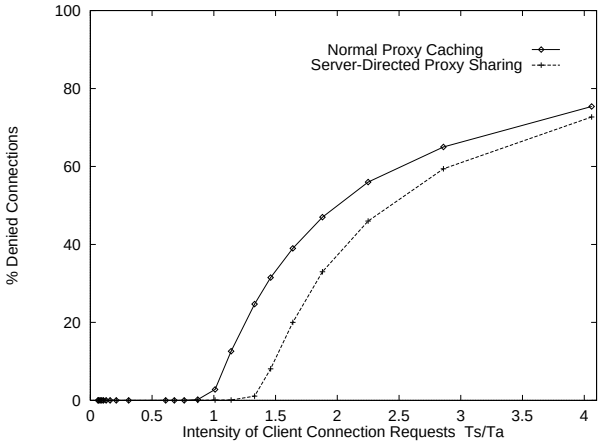

Figure 6. Simulation: Denied connections.

in typical Web behavior. Other distributed caching mechanisms can utilize ideas from SDP to the extent that they have similar properties: long periods with many reads between writes, relatively large caches and skewed object popularities, and collections of related objects that are replicated together. Implementation details that exploit such properties are best tuned individually for different application services.

Our simulation shows SDP substantially reduces both server load and the number of connection refusals as compared to normal proxy caching, and is especially effective in the range where servers begin to experience overloading. Although our simulation did not compute response time, it is a primary consideration in the SDP design. Currently we are experimenting with site selection techniques to speed up response time and lower network congestion, and are planning a prototype implementation.

Acknowledgments: This work is supported in part by the National Science Foundation under grant CDA-9633299, and by a fellowship from the NASA/Texas Space Grant Consortium. The authors wish to thank the anonymous reviewers for their detailed and helpful suggestions.

References

[1] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. In *Proc. of the 4th Int'l. World-Wide Web Conf.*, pages 119–133, Boston, MA, December 1995.

[2] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proc. of the IEEE Conf. on Parallel and Distributed Systems (PDIS'96)*, Miami Beach, FL, December 1996.

[3] Apache HTTP Server Project. http://www.apache.org/.

[4] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Proc. of ACM SIGMETRICS*, Philadelphia, PA, April 1996.

[5] H. Balakrishnam, V. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy internet server: Analysis and improvements. In *Proc. of IEEE INFOCOM*, San Francisco, CA, March 1998.

[6] A. Bestavros. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *Proc. of the Seventh IEEE Symposium on Parallel and Distributed Processing (SPDP'95)*, San Antonio, TX, October 1995.

[7] A. Bestavros. WWW traffic reduction and load balancing through server-based caching. *IEEE Concurrency*, pages 56–67, January/March 1997.

[8] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proc. of the 1996 USENIX Technical Conf.*, San Diego, CA, January 1996.

[9] M. E. Crovella and R. L. Carter. Dynamic server selection in the internet. In *Proc. of the Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95)*, pages 158–162, New York, August 1995. IEEE Communication Society.

[10] S. Glassman. A caching relay for the world wide web. In *Proc. of the First Int'l. World Wide Web Conf.*, pages 69–76, 1994. http://www1.cern.ch/PapersWWW94/steveg.ps/.

[11] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proc. of the 1995 Workshop on Hot Operating Systems (HotOS-V)*, pages 51–55, 1995.

[12] J. Gwertzman and M. Seltzer. World-wide web cache consistency. In *Proc. of the 1996 Usenix Technical Conf.*, San Diego, CA, January 1996.

[13] C. Jeffery, S. Das, and G. Bernal. Proxy sharing proxy servers. In *Proc. of the IEEE etaCOM Conf.*, Portland, Oregon, May 1996.

[14] R. Malpani, J. Lorch, and D. Berger. Making world wide web caching servers cooperate. In *Proc. of the Fourth Int'l. World-Wide-Web Conf.*, Boston, Massachusetts, Dec. 1995.

[15] J. Mogul. Network behavior of a busy web server and its clients. Technical Report Research Report 95/5, Digital Equipment Corporation, October 1995.

[16] National Laboratory for Applied Network Research (NLANR). Global caching hierarchy. http://www.nlanr.net/.

[17] V. Paxson. Empirically-derived analytical models of wide-area TCP connections: Extended report. *IEEE/ACM Transactions on Networking*, 2(4), August 1994.

[18] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[19] M. Seltzer. Issues and challenges facing the world wide web. http://www.eecs.harvard.edu/ ~margo/.

[20] Squid Internet Object Cache. http://squid.nlanr.net/.

[21] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report TR98-04, University of Texas at Austin, 1998.

[22] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proc. of the 2nd Web Cache Workshop*, Boulder, Colorado, June 1997.