

RC 22427 (Log# W0205-027) (05/02/2002)
Computer Science/Mathematics

IBM Research Report

Enhancing Web Performance

A. Iyengar, E. Nahum, A. Shaikh

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598

R. Tewari

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Some reports are available at <http://domino.watson.ibm.com/library/CyberDig.nsf/home>. Copies may be requested from IBM T.J. Watson Research Center, 16-220, P.O. Box 218, Yorktown Heights, NY 10598 or send email to reports@us.ibm.com.

This page intentionally left blank.

ENHANCING WEB PERFORMANCE

Arun Iyengar

IBM T.J. Watson Research Center

aruni@watson.ibm.com

Erich Nahum

IBM T.J. Watson Research Center

nahum@watson.ibm.com

Anees Shaikh

IBM T.J. Watson Research Center

aashaikh@watson.ibm.com

Renu Tewari

IBM Almaden Research Center

tewarir@us.ibm.com

Abstract

This paper provides an overview of techniques for improving Web performance. For improving server performance, multiple Web servers can be used in combination with efficient load balancing techniques. We also discuss how the choice of server architecture affects performance. We examine content distribution networks (CDN's) and the routing techniques that they use. While Web performance can be improved using caching, a key problem with caching is consistency. We present different techniques for achieving varying forms of cache consistency.

Keywords: cache consistency, content distribution networks, Web caching, Web performance, Web servers

Introduction

The World Wide Web has emerged as one of the most significant applications over the past decade. The infrastructure required to support Web traffic is significant, and demands continue to increase at a rapid rate. Highly accessed Web sites may need to serve over a million hits per minute. Additional demands are created by the need to serve dynamic and personalized data.

This paper presents an overview of techniques and components needed to support high volume Web traffic. These include multiple servers at Web sites which can be scaled to accommodate high request rates. Various load balancing techniques have been developed to efficiently route requests to multiple servers. Web sites may also be dispersed or replicated across multiple geographic locations.

Web servers can use several different approaches to handling concurrent requests including processes, threads, event-driven architectures in which a single process is used with non-blocking I/O, and in-kernel servers. Each of these architectural choices has certain advantages and disadvantages. We discuss how these different approaches affect performance.

Over the past few years, a number of content distribution networks (CDN) have been developed to aid Web performance. A CDN is a shared network of servers or caches that deliver content to users on behalf of content providers. The intent of a CDN is to serve content to a client from a CDN server so that response time is decreased over contacting the origin server directly. CDN's also reduce the load on origin servers. This paper examines several issues related to CDN's including their overall architecture and techniques for routing requests. We also provide insight into the performance improvements typically achieved by CDN's.

Caching is a critical technique for improving performance. Caching can take place at several points within the network including clients, servers, and in intermediate proxies between the client and server. A key problem with caching within the Web is maintaining cache consistency. Web objects may have expiration times associated with them indicating when they become obsolete. The problem with expiration times is that it is often not possible to tell in advance when Web data will become obsolete. Expiration times are not sufficient for applications which have strong consistency requirements. Stale cached data and the inability in many cases to cache dynamic and personalized data limits the effectiveness of caching.

The remainder of this paper is organized as follows. Section 1 provides an overview of techniques used for improving performance at a Web site. Section 2 discusses different server architectures. Section 3 presents an overview of content distribution networks. Section 4 discusses Web caching and cache consistency techniques.

1. Improving Performance at a Web Site

Highly accessed Web sites may need to handle peak request rates of over a million hits per minute. Web serving lends itself well to concurrency because transactions from different clients can be handled in parallel. A single Web server can achieve parallelism by multithreading or multitasking between different requests. Additional parallelism and higher throughputs can be achieved by using multiple servers and load balancing requests among the servers.

Figure 1 shows an example of a scalable Web site. Requests are distributed to multiple servers via a load balancer. The Web servers may access one or more databases for creating content. The Web servers would typically contain replicated content so that a request could be directed to any server in the cluster. For storing static files, one way to share them across multiple servers is to use a distributed file system such as AFS or DFS [42]. Copies of files may be cached in one or more servers. This approach works fine if the number of Web servers is not too large and data doesn't change very frequently. For large numbers of servers for which data updates are frequent, distributed file systems can be highly inefficient. Part of the reason for this is the strong consistency model imposed by distributed file systems. Shared file systems require all copies of files to be completely consistent. In order to update a file in one server, all other copies of the file need to be invalidated before the update can take place. These invalidation messages add overhead and latency. At some Web sites, the number of objects updated in temporal proximity to each other can be quite large. During periods of peak updates, the system might fail to perform adequately.

Another method of distributing content which avoids some of the problems of distributed file systems is to propagate updates to servers without requiring the strict consistency guarantees of distributed file systems. Using this approach, updates are propagated to servers without first invalidating all existing copies. This means that at the time an update is made, data may be inconsistent between servers for a little while. For many Web sites, these inconsistencies are not a problem, and the performance benefits from relaxing the consistency requirements can be significant.

1.1. Load Balancing

The load balancer in Figure 1 distributes requests among the servers. One method of load balancing requests to servers is via DNS servers. DNS servers provide clients with the IP address of one of the site's content delivery nodes. When a request is made to a Web site such as `http://www.research.ibm.com/compsci/`, "www.research.ibm.com" must be translated to an IP address, and DNS servers perform this translation. A name associated with a Web site can map to multiple IP addresses, each associated with a different

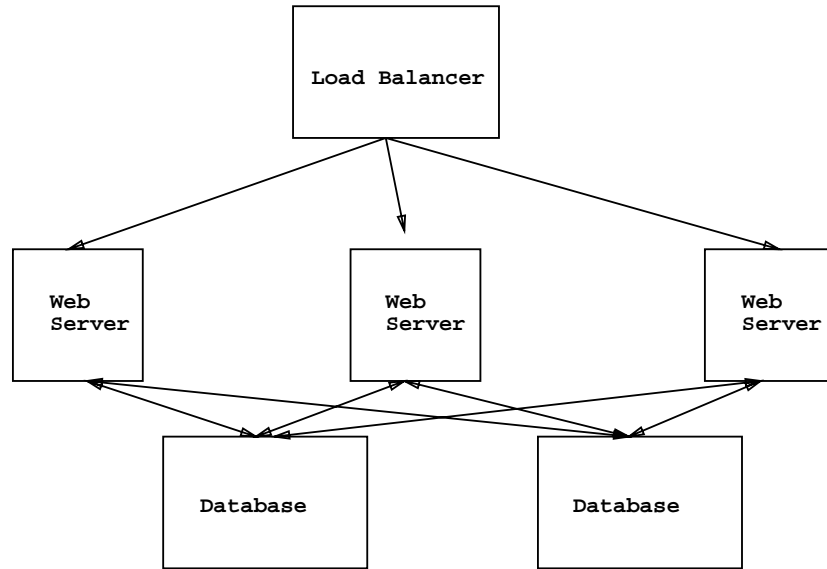


Figure 1. Architecture of a scalable Web site. Requests are directed from the load balancer to one of several Web servers. The Web servers may access one or more databases for creating content.

Web server. DNS servers can select one of these servers using a policy such as round robin [10].

There are other approaches which can be used for DNS load balances which offer some advantages over simple round robin [13]. The DNS server can use information about the number of requests per unit time sent to a Web site as well as geographic information. The Internet2 Distributed Storage Infrastructure Project proposed a DNS that implements address resolution based on network proximity information, such as round-trip delays [8].

One of the problems with load balancing using DNS is that name-to-IP mappings resulting from a DNS lookup may be cached anywhere along the path between a client and a server. This can cause load imbalance because client requests can then bypass the DNS server entirely and go directly to a server [19]. Name-to-IP address mappings have time-to-live attributes (TTL) associated with them which indicate when they are no longer valid. Using small TTL values can limit load imbalances due to caching. The problem with this approach is that it can increase response times [59]. Another problem with this approach is that not all entities caching name-to-IP address mappings obey TTL's which are too short.

Adaptive TTL algorithms have been proposed in which the DNS assigns different TTL values for different clients [12]. A request coming from a client

with a high request rate would typically receive a name-to-IP address mapping with a shorter lifetime than that assigned to a client with a low request rate. This prevents a proxy with many clients from directing requests to the same server for too long a period of time.

Another approach to load balancing is using a connection router in front of several back-end servers. Connection routers hide the IP addresses of the back-end servers. That way, IP addresses of individual servers won't be cached, eliminating the problem experienced with DNS load balancing. Connection routing can be used in combination with DNS routing for handling large numbers of requests. A DNS server can route requests to multiple connection routers. The DNS server provides coarse grained load balancing, while the connection routers provide finer grained load balancing. Connection routers also simplify the management of a Web site because back-end servers can be added and removed transparently.

IBM's Network Dispatcher [32] is one example of a connection router which hides the IP address of back-end servers. Network Dispatcher uses Weighted Round Robin for load balancing requests. Using this algorithm, servers are assigned weights. All servers with the same weight receive a new connection before any server with a lesser weight receives a new connection. Servers with higher weights get more connections than those with lower weights, and servers with equal weights get an equal distribution of new connections.

With Network Dispatcher, requests from the back-end servers go directly back to the client. This reduces overhead at the connection router. By contrast, some connection routers function as proxies between the client and server in which all responses from servers go through the connection router to clients.

Network Dispatcher has special features for handling client affinity to selected servers. These features are useful for handling requests encrypted using the Secure Sockets Layer protocol (SSL). When an SSL connection is made, a session key must be negotiated and exchanged. Session keys are expensive to generate. Therefore, they have a lifetime, typically 100 seconds, for which they exist after the initial connection is made. Subsequent SSL requests within the key lifetime reuse the key.

Network dispatcher recognizes SSL requests by the port number (443). It allows certain ports to be designated as "sticky". Network Dispatcher keeps records of old connections on such ports for a designated affinity life span (e.g. 100 seconds for SSL). If a request for a new connection from the same client on the same port arrives before the affinity life span for the previous connection expires, the new connection is sent to the same server that the old connection utilized.

Using this approach, SSL requests from the same client will go to the same server for the lifetime of a session key, obviating the need to negotiate new session keys for each SSL request. This can cause some load imbalance, par-

ticularly since the client address seen by Network Dispatcher may actually be a proxy representing several clients and not just the client corresponding to the SSL request. However, the reduction in overhead due to reduced session key generation is usually worth the load imbalance created. This is particularly true for sites which make gratuitous use of SSL. For example, some sites will encrypt all of the image files associated with an HTML page and not just the HTML page itself.

Connection routing is often done at layer 4 of the OSI model in which the connection router does not know the contents of the request. Another approach is to perform routing at layer 7. In layer 7 routing, also known as content-based routing, the router examines requests and makes its routing decisions based on the contents of requests [55]. This allows more sophisticated routing techniques. For example, dynamic requests could be sent to one set of servers, while static requests could be sent to another set. Different quality of service policies could be assigned to different URL's in which the content-based router sends the request to an appropriate server based on the quality of service corresponding to the requested URL. Content-based routing allows the servers at a Web site to be asymmetrical. For example, information could be distributed at a Web site so that frequently requested objects are stored on many or all servers, while infrequently requested objects are only stored on a few servers. This reduces the storage overhead of replicating all information on all servers. The content-based router can then use information on how objects are distributed to make correct routing decisions.

The key problem with content-based routing is that the overhead which is incurred can be high [60]. In order to examine the contents of a request, the router must terminate the connection with the client. In a straightforward implementation of content-based routing, the router acts as a proxy between the client and server, and all data exchanged between the client and server go through the router. Better performance is achieved by using a TCP handoff protocol in which the client connection is transferred from the router to a back-end server; this can be done in a manner which is transparent to the client.

A number of client-based techniques have been proposed for load balancing. A few years ago, Netscape implemented a scheme for doing load balancing at the Netscape Web site (before they were purchased by AOL) in which the Netscape browser was configured to pick the appropriate server [49]. When a user accessed the Web site `www.netscape.com`, the browser would randomly pick a number i between 1 and the number of servers and direct the request to `wwwi.netscape.com`.

Another client-based technique is to use the client's DNS [23, 58]. When a client wishes to access a URL, it issues a query to its DNS to get the IP address of the site. The Web site's DNS returns a list of IP addresses of the servers instead of a single IP address. The client DNS selects an appropriate

server for the client. An alternative strategy is for the client to obtain the list of IP addresses from its DNS and do the selection itself. An advantage to the client making the selection itself is that the client can collect information about the performance of different servers at the site and make an intelligent choice based on this. The disadvantages of client-based techniques is that the Web site loses control over how requests are routed, and such techniques generally require modifications to the client (or at least the client's DNS server).

1.2. Serving Dynamic Web Content

Web servers satisfy two types of requests, static and dynamic. *Static requests* are for files that exist at the time a request is made. *Dynamic requests* are for content that has to be generated by a server program executed at request time. A key difference between satisfying static and dynamic requests is the processing overhead. The overhead of serving static pages is relatively low. A Web server running on a uniprocessor can typically serve several hundred static requests per second. Of course, this number is dependent on the data being served; for large files, the throughput is lower.

The overhead for satisfying a dynamic request may be orders of magnitude more than the overhead for satisfying a static request. Dynamic requests often involve extensive back-end processing. Many Web sites make use of databases, and a dynamic request may invoke several database accesses. These database accesses can consume significant CPU cycles. The back-end software for creating dynamic pages may be complex. While the functionality performed by such software may not appear to be compute-intensive, such middleware systems are often not designed efficiently; commercial products for generating dynamic data can be highly inefficient.

One source of overhead in accessing databases is connecting to the database. Many database systems require a client to first establish a connection with a database before performing a transaction in which the client typically provides authentication information. Establishing a connection is often quite expensive. A naive implementation of a Web site would establish a new connection for each database access. This approach could overload the database with relatively low traffic levels.

A significantly more efficient approach is to maintain one or more long-running processes with open connections to the database. Accesses to the database are then made with one of these long-running processes. That way, multiple accesses to the database can be made over a single connection.

Another source of overhead is the interface for invoking server programs in order to generate dynamic data. The traditional method for invoking server programs for Web requests is via the Common Gateway Interface (CGI). CGI forks off a new process to handle each dynamic request; this incurs significant

overhead. There are a number of faster interfaces available for invoking server programs [34]. These faster interfaces use one of two approaches. The first approach is for the Web server to provide an interface to allow a program for generating dynamic data to be invoked as part of the Web server process itself. IBM's GO Web server API (GWAPI) is an example of such an interface. The second approach is to establish long-running processes to which a Web server passes requests. While this approach incurs some interprocess communication overhead, the overhead is considerably less than that incurred by CGI. FastCGI is an example of the second approach [53].

In order to reduce the overhead for generating dynamic data, it is often feasible to generate data corresponding to a dynamic object once, store the object in a cache, and subsequently serve requests to the object from cache instead of invoking the server program again [33]. Using this approach, dynamic data can be served at about the same rate as static data.

However, there are types of dynamic data that cannot be precomputed and served from a cache. For instance, dynamic requests that cause a side effect at the server such as a database update cannot be satisfied merely by returning a cached page. As an example, consider a Web site that allows clients to purchase items using credit cards. At the point at which a client commits to buying something, that information has to be recorded at the Web site; the request cannot be solely serviced from a cache.

Personalized Web pages can also negatively affect the cacheability of dynamic pages. A personalized Web page contains content specific to a client, such as the client's name. Such a Web page could not be used for another client. Therefore, caching the page is of limited utility since only a single client can use it. Each client would need a different version of the page.

One method which can reduce the overhead for generating dynamic pages and enable caching of some parts of personalized pages is to define these pages as being composed of multiple fragments [15]. In this approach, a complex Web page is constructed from several simpler fragments. A fragment may recursively embed other fragments. This is efficient because the overhead for assembling a Web page from simpler fragments is usually minor compared to the overhead for constructing the page from scratch, which can be quite high.

The fragment-based approach also makes it easier to design Web sites. Common information that needs to be included on multiple Web pages can be created as a fragment. In order to change the information on all pages, only the fragment needs to be changed.

In order to use fragments to allow partial caching of personalized pages, the personalized information on a Web page is encapsulated by one or more fragments that are not cacheable, but the other fragments in the page are. When serving a request, a cache composes pages from its constituent fragments, many of which are locally available. Only personalized fragments have

to be created by the server. As personalized fragments typically constitute a small fraction of the entire page, generating only them would require lower overhead than generating all of the fragments in the page.

Generating Web pages from fragments provides other benefits as well. Fragments can be constructed to represent entities that have similar lifetimes. When a particular fragment changes but the rest of the Web page stays the same, only the fragment needs to be invalidated or updated in the cache, not the entire page. Fragments can also reduce the amount of cache space taken up by multiple pages with common content. Suppose that a particular fragment is contained in 2000 popular Web pages which should be cached. Using the conventional approach, the cache would contain a separate version of the fragment for each page resulting in as many as 2000 copies. By contrast, if the fragment-based method of page composition is used, only a single copy of the fragment needs to be maintained.

A key problem with caching dynamic content is maintaining consistent caches. It is advantageous for the cache to provide a mechanism, such as an API, allowing the server to explicitly invalidate or update cached objects so that they don't become obsolete. Web objects may be assigned expiration times that indicate when they should be considered obsolete. Such expiration times are generally not sufficient for allowing dynamic data to be cached properly because it is often not possible to predict accurately when a dynamic page will change.

2. Server Performance Issues

A central component of the response time seen by Web users is, of course, the performance of the origin server that provides the content. There is great interest, then, understanding the performance of Web servers: How quickly can they respond to requests? How well do they scale with load? Are they capable of operating under overload, i.e., can they maintain some level of service even when the requested load far outstrips the capacity of the server?

A Web server is an unusual piece of software in that it must communicate with potentially thousands of remote clients simultaneously. The server thus must be able to deal with a large degree of *concurrency*. A server cannot simply respond to each client in a non-preemptive, first-come first-serve manner, for several reasons. Clients are typically located far away over the wide-area Internet, and thus connection lifetimes can last many seconds or even minutes. Particularly with HTTP 1.1, a client connection may be open but idle for some time before a new request is submitted. Thus a server can have many concurrent connections open, and should be able to do work for one connection when another is quiescent. Another reason is that a client may request a file which is not resident in memory. While the server CPU waits for the disk to retrieve the

file, it can work on responding to another client. For these and other reasons, a server must be able to multiplex the work it has to do through some form of concurrency.

A fundamental factor which affects the performance of a Web server is the *architectural model* that it uses to implement that concurrency. Generally, Web servers can be implemented using one of four architectures: processes, threads, event-driven, and in-kernel. Each approach has its advantages and disadvantages which we go into more detail below. A central issue in this decision of which model to use is what sort of performance optimizations are available under that model. Another is how well that model *scales* with the workload, i.e., how efficiently it can handle growing numbers of clients.

2.1. Process-Based Servers

Processes are perhaps the most common form of providing concurrency. The original NCSA server and the widely-known Apache server [2] use processes as the mechanism to handle large numbers of connections. In this model, a process is created for each new request, which can block when necessary, for example waiting for data to become available on a socket or for file I/O to be available from the disk. The server handles concurrency by creating multiple processes.

Processes have two main advantages. First, they are consistent with a programmers' way of thinking, allowing the developer to proceed in a step-by-step fashion without worrying about managing concurrency. Second, they provide isolation and protection between different clients. If one process hangs or crashes, the other processes should be unaffected.

The main drawback to processes is performance. Processes are relatively heavyweight abstractions in most operating systems, and thus creating them, deleting them, and switching context between them is expensive. Apache, for example, tries to ameliorate these costs by pre-forking a number of processes and only destroys them if the load falls below a certain threshold. However, the costs are still significant, as each process requires memory to be allocated to them. As the number of processes grow, large amounts of memory are used which puts pressure on the virtual memory system, which could use the memory for other purposes, such as caching frequently-accessed data. In addition, sharing information, such as a cached file, across processes can be difficult.

2.2. Thread-Based Servers

Threads are the next most common form of concurrency. Servers that use threads include JAWS [31] and Sun's Java Web Server [64]. Threads are similar to processes but are considered lighter-weight. Unlike processes, threads share the same address space and typically only provide a separate stack for

each thread. Thus, creation costs and context-switching costs are usually much lower than for processes. In addition, sharing between threads is much easier. Threads also maintain the abstraction of an isolated environment much like processes, although the analogy is not exact since programmers must worry more about issues like synchronization and locking to protect shared data structures.

Threads have several disadvantages as well. Since the address space is shared, threads are not protected from one another the way processes are. Thus, a poorly programmed thread can crash the whole server. Threads also require proper operating system support, otherwise when a thread blocks on something like a file I/O, the whole address space will be stopped.

2.3. Event-Driven Servers

The third form of concurrency is known as the *event-driven* architecture. Servers that use this method include Flash [56] and Zeus [72]. With this architecture, a single process is used with *non-blocking I/O*. Non-blocking I/O is a way of doing asynchronous reads and writes on a socket or file descriptor. For example, instead of a process reading a file descriptor and blocking until data is available, an event-driven server will return immediately if there is no data. In turn, the O.S. will let the server process know when a socket or file descriptor is ready for reading or writing through a *notification mechanism*. This notification mechanism can be an active one such as a signal handler, or a passive one requiring the process to ask the O.S. such as the `select()` system call. Through these mechanisms the server process will essentially respond to events and is typically guaranteed to never block.

Event-driven servers have several advantages. First, they are very fast. Zeus is frequently used by hardware vendors to generate high Web server numbers with the SPECWeb99 benchmark [61]. Sharing is inherent, since there is only one process, and no locking or synchronization is needed. There are no context-switch costs or extra memory consumption that are the case with threads or processes. Maximizing concurrency is thus much easier than with the previous approaches.

Event-driven servers have downsides as well. Like threads, a failure can halt the whole server. Event-driven servers can tax operating system resource limits, such as the number of open file descriptors. Different operating systems have varying levels of support for asynchronous I/O, so a fully event-driven server may not be possible on a particular platform. Finally, event-driven servers require a different way of thinking from the programmer, who must understand and account for the ways in which multiple requests can be in varying stages of progress simultaneously. In this approach, the degree of

concurrency is fully exposed to the developer, with all the attendant advantages and disadvantages.

2.4. In-Kernel Servers

The fourth and final form of server architectures is the *in-kernel* approach. Servers that use this method include AFPA [36] and Tux [66]. All of the previous architectures place the Web server software in user space; in this approach the HTTP server is in kernel space, tightly integrated with the host TCP/IP stack.

The in-kernel architecture has the advantages that it is extremely fast, since potentially expensive transitions to user space are completely avoided. Similarly, no data needs to be copied across the user-kernel boundary, another costly operation.

The disadvantages for in-kernel approaches are several. First, it is less robust to programming errors; a server fault can crash the whole machine, not just the server! Development is much harder, since kernel programming is more difficult and much less portable than programming user-space applications. Kernel internals of Linux, FreeBSD, and Windows vary considerably, making deployment across platforms more work. The socket and thread APIs, on the other hand, are relatively stable and portable across operating systems.

Dynamic content poses an even greater challenge for in-kernel servers, since an arbitrary program may be invoked in response to a request for dynamic content. A full-featured in-kernel web server would need to have a PHP engine or Java runtime interpreter loaded in with the kernel! The way current in-kernel servers deal with this issue is to restrict their activities to the static content component of Web serving, and pass dynamic content requests to a complete server in user space, such as Apache. For example, many entries in the SPECWeb99 site [61] that use the Linux operating system use this hybrid approach, with Tux serving static content in the kernel and Apache handling dynamic requests in user space.

2.5. Server Performance Comparison

Since we are concerned with performance, it is thus interesting to see how well the different server architectures perform. To evaluate them, we took an experimental testbed setup and evaluate the performance using a synthetic workload generator [51] to saturate the servers with requests for a range of web documents. The clients were eight 500 MHz PC's running FreeBSD, and the server was a 400 MHz PC running Linux 2.4.16. Each client had a 100 mbps Ethernet connected to a gigabit switch, and the server was connected to the switch using Gigabit Ethernet. Three servers were evaluated as representatives

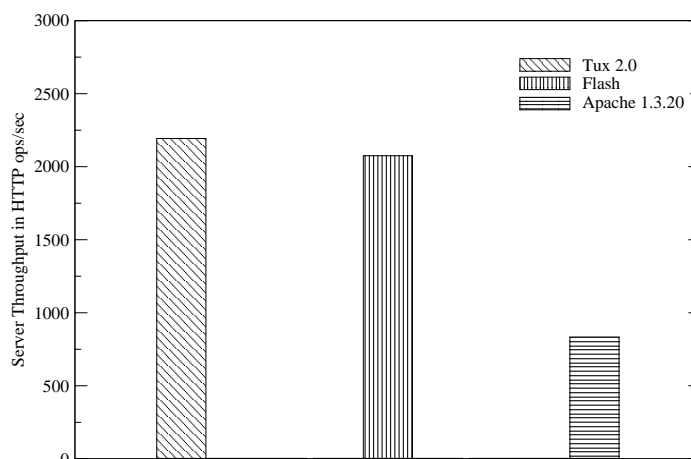


Figure 2. Server Throughput

of their architecture: Apache as a process-based server, Flash as an event-driven server, and Tux as an in-kernel server.

Figure 2 shows the server throughput in HTTP operations/sec of the three servers. As can be seen, Tux, the in-kernel server, is the fastest at 2193 ops/sec. However, Flash is only 10 percent slower at 2075 ops/sec, despite being implemented in user space. Apache, on the other hand, is significantly slower at 875 ops/sec. Figure 3 shows the server response time for the three servers. Again, Tux is the fastest, at 3 msec, Flash second at 5 msec, and Apache slowest at 10 msec.

Since multiple examples of each type of server architecture exist, there is clearly no consensus for what is the best model. Instead, it may be that different approaches are better suited for different scenarios. For example, the in-kernel approach may be most appropriate for dedicated server appliances, or as CDN nodes, whereas a back-end dynamic content server will rely on the full generality of a process-based server like Apache. Still, web site operators should be aware of how the choice of architecture will affect Web server performance.

3. CDNs: Improved Web Performance through Distribution

End-to-end Web performance is influenced by numerous factors such as client and server network connectivity, network loss and delay, server load, HTTP protocol version, and name resolution delays. The content-serving architecture has a significant impact on some of these factors, as well factors

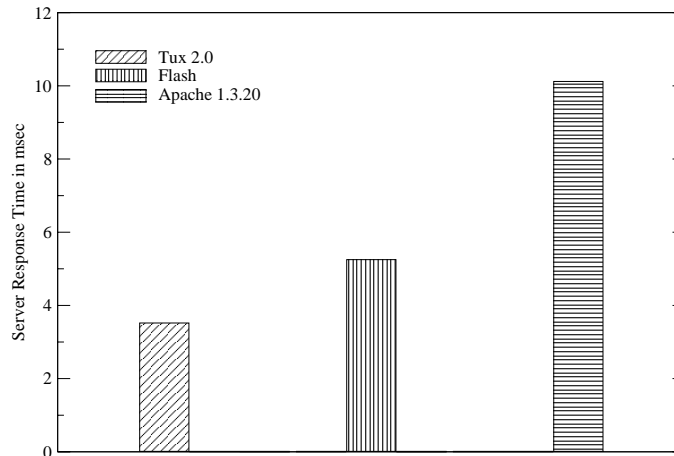


Figure 3. Server Response Time

not related to performance such as cost, reliability, and ease of management. In a traditional content-serving architecture all clients request content from a single location, as shown in Figure 4. In this architecture, scalability and performance are improved by adding servers, without the ability to address poor performance due to problems in the network. Moreover, this approach can be expensive since the site must be overprovisioned to handle unexpected surges in demand.

One way to address poor performance due to network congestion, or flash crowds at servers, is to distribute content to servers or caches located closer to the edges of the network, as shown in Figure 5. Such a distributed network of servers comprises a content distribution network (CDN). A CDN is simply a network of servers or caches that deliver content to users on behalf of content providers. The intent of a CDN is to serve content to a client from a CDN server such that the response-time performance is improved over contacting the origin server directly. CDN servers are typically shared, delivering content belonging to multiple Web sites though all servers may not be used for all sites.

CDNs have several advantages over traditional centralized content-serving architectures, including [67]:

- improving client-perceived response time by bringing content closer to the network edge, and thus closer to end-users
- off-loading work from origin servers by serving larger objects, such as images and multimedia, from multiple CDN servers

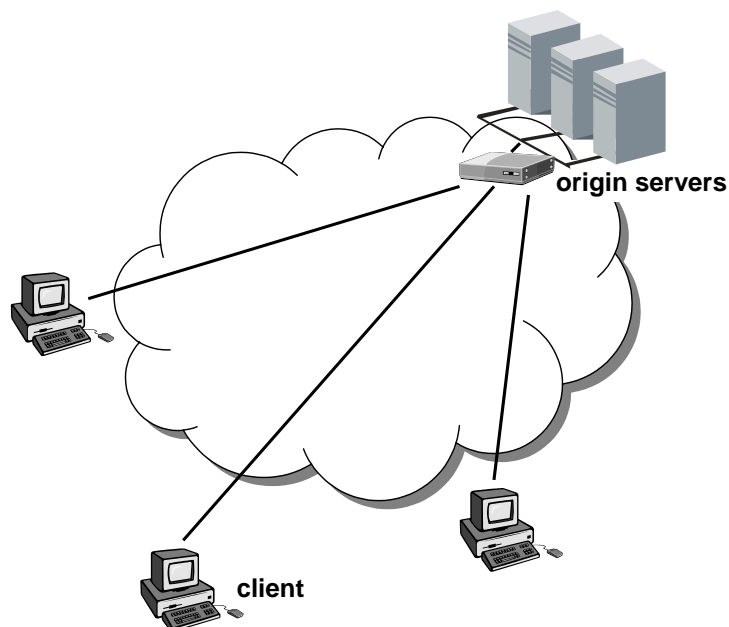


Figure 4. Traditional centralized content-serving architecture

- reducing content provider costs by reducing the need to invest in more powerful servers or more bandwidth as user population increases
- improving site availability by replicating content in many distributed locations

CDN servers may be configured in tree-like hierarchies [71] or clusters of cooperating proxies that employ content-based routing to exchange data [28]. Commercial CDNs also vary significantly in their size and service offerings. CDN deployments range from a few tens of servers (or server clusters), to over ten thousand servers placed in hundreds of ISP networks. A large footprint allows a CDN service provider (CDSP) to reach the majority of clients with very low latency and path length.

Content providers use CDNs primarily for serving static content like images or large stored multimedia objects (e.g., movie trailers and audio clips). A recent study of CDN-served content found that 96% of the objects served were images [41]. However, the remaining few objects accounted for 40–60% of the bytes served, indicating a small number of very large objects. Increasingly, CDSPs offer services to deliver streaming media and dynamic data such as localized content or targeted advertising.

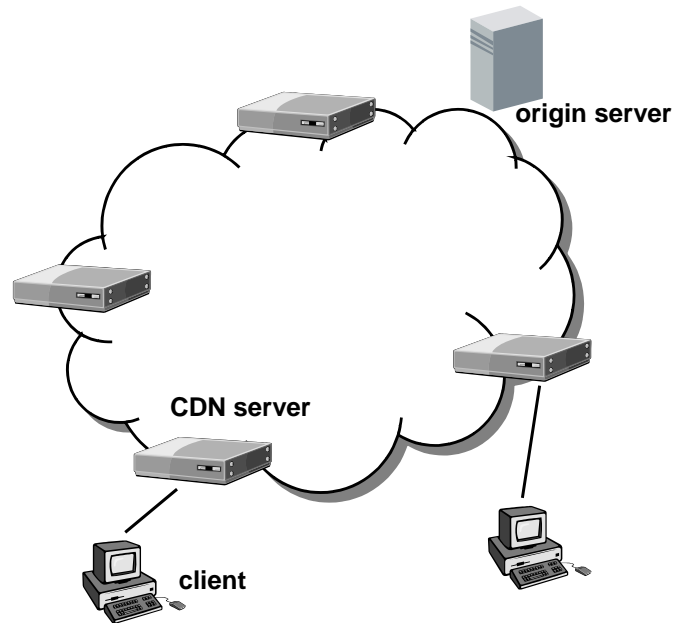


Figure 5. Distributed CDN architecture

3.1. CDN Architectural Elements

As illustrated in Figure 6, CDNs have three key architectural elements in addition to the CDN servers themselves: a distribution system, an accounting/billing system, and a request-routing system [18]. The distribution system is responsible for moving content from origin servers into CDN servers and ensuring data consistency. Section 4.4 describes some techniques used to maintain consistency in CDNs. The accounting/billing system collects logs of client accesses and keeps tracks CDN server usage for use primarily in administrative tasks. Finally, the request-routing system is responsible for directing client requests to appropriate CDN servers. It may also interact with the distribution system to keep an up-to-date view of which content resides on which CDN servers.

The request-routing system operates as shown in Figure 7. Clients access content from the CDN servers by first contacting a request router (step 1). The request router makes a server selection decision and returns a server assignment to the client (step 2). Finally, the client retrieves content from the specified CDN server (step 3).

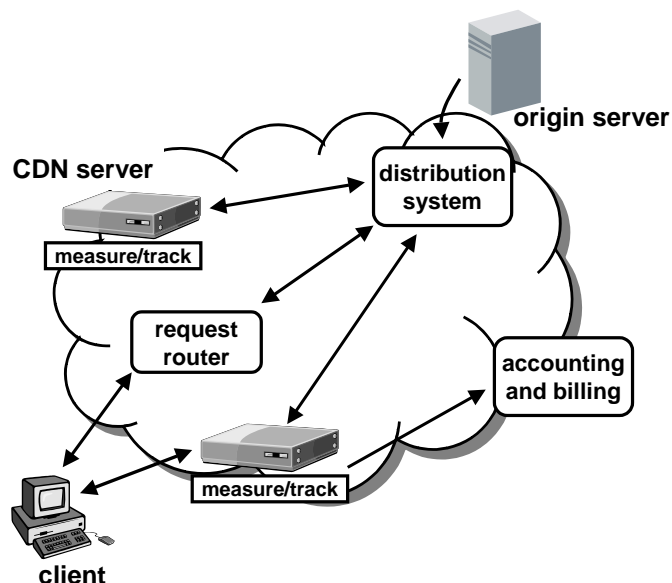


Figure 6. CDN architectural elements

3.2. CDN Request-Routing

Clearly, the request-routing system has a direct impact on the performance of the CDN. A poor server selection decision can defeat the purpose of the CDN, namely to improve client response time over accessing the origin server. Thus, CDNs typically rely on a combination of static and dynamic information when choosing the best server. Several criteria are used in the request-routing decision, including the content being requested, CDN server and network conditions, and client proximity to the candidate servers.

The most obvious request routing strategy is to direct the client to a CDN server that hosts the content being requested. This is complicated, however, if the request router does not know the content being requested, for example if request-routing is done in the context of name resolution. In this case the request contains only a server name (e.g., `www.service.com`) as opposed to the full HTTP URL.

For good performance the client should be directed to a relatively unloaded CDN server. This requires that the request router actively monitor the state of CDN servers. If each CDN location consists of a cluster of servers and local load-balancer, it may be possible to query a server-side agent for server load information, as shown in Figure 8. After the client makes its request, the

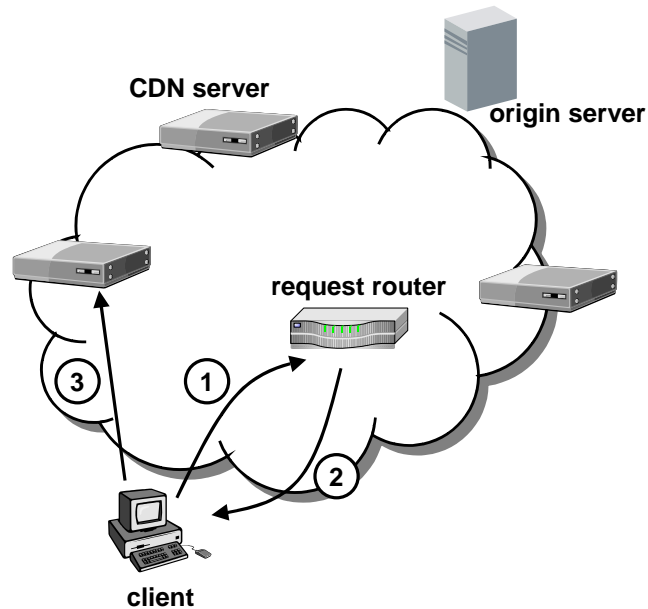


Figure 7. CDN request-routing

request router consults an agent at each CDN site load-balancer (step 2), and returns an appropriate answer back to the client.

As Web response time is heavily influenced by network conditions, it is important to choose a CDN server to which the client has good connectivity. Upon receiving a client request, the request router can ask candidate CDN servers to measure network latency to the client using ICMP echo (i.e., ping) and report the measured values. The request router then responds to the client request with the CDN server reporting the lowest delay. Since these measurements are done on-line, this technique has the advantage of adapting the request-routing decision to the most current network network. On the other hand, it introduces additional latency for the client as the request router waits for responses from the CDN servers.

A common strategy used in CDN request-routing is to choose a server “near” the client, where proximity is defined in terms of network topology, geographic distance, or network latency. Examples of proximity metrics include autonomous system (AS) hops or network hops. These metrics are relatively static compared with server load or network performance, and are also easier to measure.

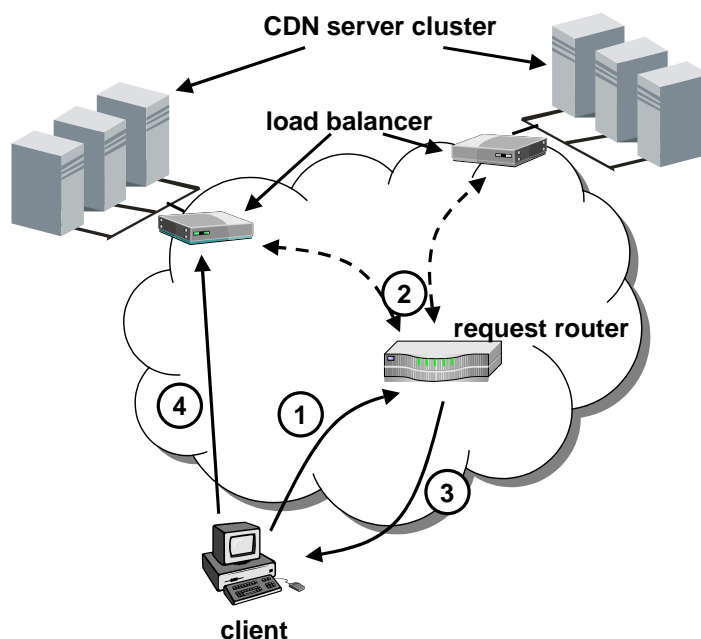


Figure 8. Interaction between request router and CDN servers

Note that it is unlikely that any one of these metrics will be suitable in all cases. Most request routers use a combination of proximity and network or server load to make server selection decisions. For example, client proximity metrics can be used to assign a client to a “default” CDN server, which provides good performance most of the time. The selection can be temporarily changed if load monitoring indicates that the default server is overloaded.

Request-routing techniques fall into three main categories: transport-layer mechanisms, application-layer redirection, and DNS-based approaches [6]. Transport-layer request routers use information in the transport-layer headers to determine which CDN server should serve the client. For example, the request router can examine the client IP address and port number in a TCP SYN packet and forward the packet to an appropriate CDN server. The target CDN server establishes the TCP connection and proceeds to serve the requested content. Forward traffic (including TCP acknowledgements) from the client to the target server continues to be sent to the request router and forwarded to the CDN server. The bulk of traffic (i.e., the requested content) will travel on the direct path from the CDN server to the client.

Application-layer request-routing has access to much more information about the content being requested. For example, the request-router can use HTTP

headers like the URL, HTTP cookies, and Language. A simple implementation of an application-layer request router is a Web server that receives client requests and returns an HTTP redirect (e.g., return code 302) to the client indicating the appropriate CDN server. The flexibility afforded by this approach comes at the expense of added latency and overhead, however, since it requires TCP connection establishment and HTTP header parsing.

With request-routing based on the Domain Name System (DNS), clients are directed to the nearest CDN server during the name resolution phase of Web access. Typically, the authoritative DNS server for the domain or subdomain is controlled by the CDSP. In this scheme, a specialized DNS server receives name resolution requests, determines the location of the client and returns the address of a nearby CDN server or a referral to another nameserver. The answer may only be cached at the client-side for a short time so that the request router can adapt quickly to changes in network or server load. This is achieved by setting the associated time-to-live (TTL) field in the answer to a very small value (e.g., 20 seconds).

DNS-based request routing may be implemented with either full- or partial-site content delivery [41]. In full-site delivery, the content provider delegates authority for its domain to the CDSP or modifies its own DNS servers to return a referral (CNAME record) to the CDSPs DNS servers. In this way, all requests for `www.company.com`, for example, are resolved to a CDN server which then delivers all of the content. With partial-site delivery, the content provider modifies its content so that links to specific objects have hostnames in a domain for which the CDSP is authoritative. For example, links to `http://www.company.com/image.gif` are changed to `http://cdsp.net/company.com/image.gif`. In this way, the client retrieves the base HTML page from the origin server but retrieves embedded images from CDN servers to improve performance.

The appeal of DNS-based server selection lies in both its simplicity – it requires no change to existing protocols, and its generality – it works across any IP-based application regardless of the transport-layer protocol being used. This has led to adoption of DNS-based request routing as the *de facto* standard method by many CDSPs and equipment vendors. Using the DNS for request-routing does have some fundamental drawbacks, however, some of which have been recently studied and evaluated [59, 45, 6].

3.3. CDN Performance Studies

Several research studies have recently tried to quantify the extent to which CDNs are able to improve response-time performance. An early study by Johnson *et al.* focused on the quality of the request-routing decision [35]. The study compared two CDSPs that use DNS-based request-routing. The methodology

was to measure the response time to download a single object from the CDN server assigned by the request router and the time to download it from all other CDN servers that could be identified. The findings suggested that the server selection did not always choose the best CDN server, but it was effective in avoiding poorly performing servers, and certainly better than choosing a CDN server randomly. The scope of the study was limited, however, since only three client locations were considered, performance was compared for downloading only one small object, and there was no comparison with downloading from the origin server.

A study done in the context of developing the request mirroring Medusa Web proxy, evaluated the performance of one CDN (Akamai) by downloading the same objects from CDN servers and origin servers [37]. The study was done only for a single-user workload, but showed significant performance improvement for those objects that were served by the CDN, when compared with the origin server.

More recently, Krishnamurthy *et al.* studied the performance of a number of commercial CDNs from the vantage point of approximately 20 clients [41]. The authors conclude that CDN servers generally offer much better performance than origin servers, though the gains were dependent on the level of caching and the HTTP protocol options. There were also significant differences in download times from different CDNs. The study finds that, for some CDNs, DNS-based request routing significantly hampers performance due to multiple name lookups.

4. Cache Consistency

Caching has proven to be an effective and practical solution for improving the scalability and performance of Web servers. Static Web page caching has been applied both at browsers at the client, or at intermediaries that include isolated proxy caches or multiple caches or servers within a CDN network. As with caching in any system, maintaining cache consistency is one of the main issues that a Web caching architecture needs to address. As more of the data on the Web is dynamically assembled, personalized, and constantly changing, the challenges of efficient consistency management become more pronounced. To prevent stale information from being transmitted to clients, an intermediary cache must ensure that the locally cached data is consistent with that stored on servers. The exact cache consistency mechanism and the degree of consistency employed by an intermediary depends on the nature of the cached data; not all types of data need the same level of consistency guarantees. Consider the following example.

Example 1 Online auctions: *Consider a Web server that offers online auctions over the Internet. For each item being sold, the server maintains in-*

formation such as its latest bid price (which changes every few minutes) as well as other information such as photographs and reviews for the item (all of which change less frequently). Consider an intermediary that caches this information. Clearly, the bid price returned by the intermediary cache should always be consistent with that at the server. In contrast, reviews of items need not always be up-to-date, since a user may be willing to receive slightly stale information.

The above example shows that an intermediary cache will need to provide different degrees of consistency for different types of data. The degree of consistency selected also determines the mechanisms used to maintain it, and the overheads incurred by both the server and the intermediary.

4.1. Degrees of Consistency

In general the degrees of consistency that an intermediary cache can support fall into the following four categories.

- *strong consistency*: A cache consistency level that always returns the results of the latest (committed) write at the server is said to be strongly consistent. Due to the unbounded message delays in the Internet, no cache consistency mechanism can be strongly consistent in this idealized sense. Strong consistency is typically implemented using a two-phase message exchange along with timeouts to handle unbounded delays.
- *delta consistency*: A consistency level that returns data that is never outdated by more than δ time units, where δ is a configurable parameter, with the last committed write at the server is said to be delta consistent. In practice the value of delta should be larger than t which is the network delay between the server and the intermediary at that instant, i.e., $t < \delta \leq \infty$.
- *weak consistency*: For this level of consistency, a read at the intermediary does not necessarily reflect the last committed write at the server but some correct previous value.
- *mutual consistency*: A consistency guarantee in which a group of objects are mutually consistent with respect to each other. In this case some objects in the group cannot be more current than the others. Mutual consistency can co-exist with the other levels of consistency.

Strong consistency is useful for mirror sites that need to reflect the current state at the server. Some applications based on financial transactions may also require strong consistency. Certain types of applications can tolerate stale data as long as it is within some known time bound. For such applications delta consistency is recommended. Delta consistency assumes that there is a bounded

Overheads	Polling	Periodic polling	Invalidates	Leases	TTL
File Transfer	W'	$W' - \delta$	W'	W'	W'
Control Msgs.	$2R - W'$	$2R/t - (W' - \delta)$	$2W'$	$2W'$	W'
Staleness	0	t	0	0	0
Write delay	0	0	notify(all)	$\min(t, \text{notify}(all_t))$	0
Server State	None	None	All	All_t	None

Table 1. Overheads of Different Consistency Mechanisms. Key: t is the period in periodic polling or the lease duration in the leases approach. W' is the number of non-consecutive writes. All consecutive writes with no interleaving reads is counted as a single write. R is the number of reads. δ is the number of writes that were not notified to the intermediary as only weak consistency was provided.

communication delay between the server and the intermediary cache. Mutual consistency is useful when a certain set of objects at the intermediary (e.g., the fragments within a sports score page, or within a financial page) need to be consistent with respect to each other. To maintain mutual consistency the objects need to be atomically invalidated such that they all either reflect the new version or maintain the earlier stale version.

Most intermediaries deployed in the Internet today provide only weak consistency guarantees [29, 62]. Until recently, most objects stored on Web servers were relatively static and changed infrequently. Moreover, this data was accessed primarily by humans using browsers. Since humans can tolerate receiving stale data (and manually correct it using browser reloads), weak cache consistency mechanisms were adequate for this purpose. In contrast, many objects stored on Web servers today change frequently and some objects (such as news stories or stock quotes) are updated every few minutes [7]. Moreover, the Web is rapidly evolving from a predominantly read-only information system to a system where collaborative applications and program-driven agents frequently read as well as write data. Such applications are less tolerant of stale data than humans accessing information using browsers. These trends argue for augmenting the weak consistency mechanisms employed by today's proxies with those that provide strong consistency guarantees in order to make caching more effective. In the absence of such strong consistency guarantees, servers resort to marking data as uncacheable, and thereby reduce the effectiveness of proxy caching.

4.2. Consistency Mechanisms

The mechanisms used by an intermediary and the server to provide the degrees of consistency described earlier fall into 3 categories: i) *client-driven*, ii) *server-driven*, and iii) *explicit* mechanisms .

Server-driven mechanisms, referred to as *server-based invalidation*, can be used to provide strong or delta consistency guarantees [69]. Server-based invalidation, requires the server to notify proxies when the data changes. This approach substantially reduces the number of control messages exchanged between the server and the intermediary (since messages are sent only when an object is modified). However, it requires the server to maintain per-object state consisting of a list of all proxies that cache the object; the amount of state maintained can be significant especially at popular Web servers. Moreover, when an intermediary is unreachable due to network failures, the server must either delay write requests until it receives all the acknowledgments or a timeout occurs, or risk violating consistency guarantees. Several new protocols have been proposed recently to provide delta and strong consistency using server-based invalidations. Web cache invalidation protocol (WCIP) is one such proposal for propagating server invalidations using application-level multicast while providing delta consistency [43]. Web content distribution protocol (WCDP) is another proposal that supports multiple consistency levels using a request-response protocol that can be scaled to support distribution hierarchies [65].

The client-driven approach, also referred to as *client polling*, requires that intermediaries poll the server on *every read* to determine if the data has changed [69]. Frequent polling imposes a large message overhead and also increases the response time (since the intermediary must await the result of its poll before responding to a read request). The advantage, though, is that it does not require any state to be maintained at the server, nor does the server ever need to delay write requests (since the onus of maintaining consistency is on the intermediary).

Most existing proxies provide only weak consistency by (i) explicitly providing a server specified lifetime of an object (referred to as the *time-to-live (TTL)* value), or (ii) by *periodic polling* of the the server to verify that the cached data is not stale [14, 29, 62]. The TTL value is sent as part of the HTTP response in an `Expires` tag or using the `Cache-Control` headers. However, *a priori* knowledge of when an object will be modified is difficult in practice and the degree of consistency is dependent on the clock skew between the server and the intermediaries. With periodic polling the length of the period determines the extent of the object staleness. In either case, modifications to the object before its TTL expires or between two successive polls causes the intermediary to return stale data. Thus both mechanisms are heuristics and provide only weak consistency guarantees. Hybrid approaches where the server specifies a time-to-live value for each object and the intermediary polls the server only when the TTL expires also suffer from these drawbacks.

Server-based invalidation and client polling form two ends of a spectrum. Whereas the former minimizes the number of control messages exchanged but

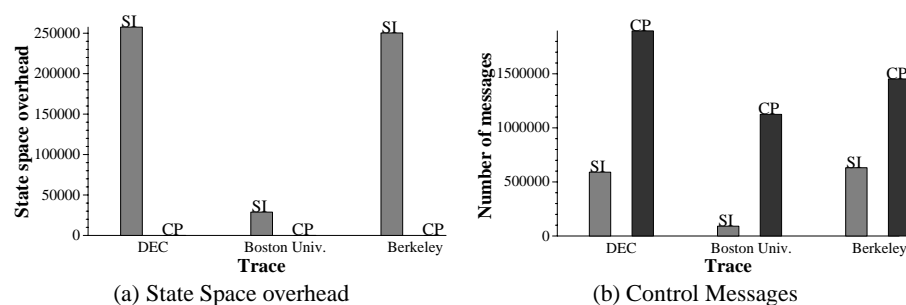


Figure 9. Efficacy of server-based invalidation and client polling for three different trace workloads (DEC, Berkeley, Boston University). The figure shows that server-based invalidation has the largest state space overhead; client polling has the highest control message overhead

may require a significant amount of state to be maintained, the latter is stateless but can impose a large control message overhead. Figure 9 quantitatively compares these two approaches with respect to (i) the server overhead, (ii) the network overhead, and (iii) the client response time. Due to their large overheads, neither approach is appealing for Web environments. A strong consistency mechanism suitable for the Web must not only reduce client response time, but also balance both network and server overheads.

One approach that provides strong consistency, while providing a smooth tradeoff between the state space overhead and the number of control messages exchanged, is *leases* [27]. In this approach, the server grants a lease to each request from an intermediary. The lease duration denotes the interval of time during which the server agrees to notify the intermediary if the object is modified. After the expiration of the lease, the intermediary must send a message requesting renewal of the lease. The duration of the lease determines the server and network overhead. A smaller lease duration reduces the server state space overhead, but increases the number of control (lease renewal) messages exchanged and vice versa. In fact, an infinite lease duration reduces the approach to server-based invalidation, whereas a zero lease duration reduces it to client-polling. Thus, the leases approach spans the entire spectrum between the two extremes of server-based invalidation and client-polling.

The concept of a lease was first proposed in the context of cache consistency in distributed file systems [27]. Recently some research groups have begun investigating the use of leases for maintaining consistency in Web intermediary caches. The use of leases for Web proxy caches was first alluded to in [11] and was subsequently investigated in detail in [69]. The latter effort focused on the design of *volume leases* – leases granted to a collection of objects – so as to reduce (i) the lease renewal overhead and (ii) the blocking overhead at the server due to unreachable proxies. Other efforts have focused on extending leases to hierarchical proxy cache architectures [70, 71]. The adaptive leases

effort described analytical and quantitative results on how to select the optimal lease duration based on the server and message exchange overheads [21].

A qualitative comparison of the overheads of the different consistency mechanisms is shown in Table 1. The message overheads of an invalidation-based or lease-based approach is smaller than that of polling especially when reads dominate writes, as in the Web environment.

4.3. Invalidates and Updates

With server-driven consistency mechanisms, when an object is modified, the origin server notifies each “subscribing” intermediary. The notification consists of either an invalidate message or an updated (new) version of the object. Sending an invalidate message causes an intermediary to mark the object as invalid; a subsequent request requires the intermediary to fetch the object from the server (or from a designated site). Thus, each request after a cache invalidate incurs an additional delay due to this remote fetch. An invalidation adds to 2 control messages and a data transfer (an invalidation message, a read request on a miss, and a new data transfer) along with the extra latency. No such delay is incurred if the server sends out the new version of the object upon modification. In an update-based scenario, subsequent requests can be serviced using locally cached data. A drawback, however, is that sending updates incurs a larger network overhead (especially for large objects). This extra effort is wasted if the object is never subsequently requested at the intermediary. Consequently, cache invalidates are better suited for less popular objects, while updates can yield better performance for frequently requested small objects. Delta encoding techniques have been designed to reduce the size of the data transferred in an update by sending only the changes to the object[40]. Note that delta encoding is not related to delta consistency. Updates, however, require better security guarantees and make strong consistency management more complex. Nevertheless, updates are useful for mirror sites where data needs to be “pushed” to the replicas when it changes. Updates are also useful for pre-loading caches with content that is expected to become popular in the near future.

A server can dynamically decide between invalidates and updates based on the characteristics of an object. One policy could be to send updates for objects whose popularity exceeds a threshold and to send invalidates for all other objects. A more complex policy is to take both popularity and object size into account. Since large objects impose a larger network transfer overhead, the server can use progressively larger thresholds for such objects (the larger an object, the more popular it needs to be before the server starts sending updates).

The choice between invalidation and updates also affects the implementation of a strong consistency mechanism. For invalidations only, with a strong consistency guarantee, the server needs to wait for all acknowledgments of the invalidation message (or a timeout) to commit the write at the server. With updates, on the other hand, the server updates are not immediately committed at the intermediary. Only after the server receives all the acknowledgments (or a timeout) and then sends a commit message to all the intermediaries is the new update version committed at the intermediary. Such two-phase message exchanges are expensive in practice and are not required for weaker consistency guarantees.

4.4. Consistency Management for CDNs

An important issue that must be addressed in a CDN is that of *consistency maintenance*. The problem of consistency maintenance in the context of a single proxy used several techniques such as time-to-live (TTL) values, client-polling, server-based invalidation, adaptive refresh [63], and leases [68]. In the simplest case, a CDN can employ these techniques at each individual CDN server or proxy – each proxy assumes responsibility for maintaining consistency of data stored in its cache and interacts with the server to do so independently of other proxies in the CDN. Since a typical CDN may consist of hundreds or thousands of proxies (e.g., Akamai currently has a footprint of more than 14,000 servers), requiring each proxy to maintain consistency independently of other proxies is not scalable from the perspective of the origin servers (since the server will need to individually interact with a large number of proxies). Further, consistency mechanisms designed from the perspective of a single proxy (or a small group of proxies) do not scale well to large CDNs. The leases approach, for instance, requires the origin server to maintain per-proxy state for each cached object. This state space can become excessive if proxies cache a large number of objects or some objects are cached by a large number of proxies within a CDN.

A cache consistency mechanism for hierarchical proxy caches was discussed in [71]. The approach does not propose a new consistency mechanism, rather it examines issues in instantiating existing approaches into a hierarchical proxy cache using mechanisms such as multicast. They argue for a fixed hierarchy (i.e., a fixed parent-child relationship between proxies). In addition to consistency, they also consider pushing of content from origin servers to proxies. Mechanisms for scaling leases are studied in [68]. The approach assumes volume leases, where each lease represents *multiple objects* cached by a stand-alone proxy. They examine issues such as delaying invalidations until lease renewals and discuss prefetching and pushing lease renewals.

Another effort describes *cooperative consistency* along with a mechanism, called cooperative leases, to achieve it [52]. Cooperative consistency enables proxies to cooperate with one another to reduce the overheads of consistency maintenance. By supporting delta consistency semantics and by using a single lease for multiple proxies, the cooperative leases mechanism allows the notion of leases to be applied in a scalable manner to CDNs. Another advantage of the approach is that it employs application-level multicast to propagate server notifications of modifications to objects, which reduces server overheads. Experimental results show that cooperative leases can reduce the number of server messages by a factor of 3.2 and the server state by 20% when compared to original leases, albeit at an increased proxy-proxy communication overhead.

Finally, numerous studies have focused on specific aspects of cache consistency for content distribution. For instance, piggybacking of invalidations [40], the use of deltas for sending updates [48], an application-level multicast framework for Internet distribution [26] and the efficacy of sending updates versus invalidates [22].

References

- [1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of PDIS'96: The IEEE Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.
- [2] The Apache Project. The Apache WWW server. <http://httpd.apache.org>.
- [3] M. F. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, May/June 2000.
- [4] M. F. Arlitt and C. L. Williamson. Internet Web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–646, Oct 1997.
- [5] G. Banga, J. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Technical Conference*, Monterey, CA, June 1999.
- [6] A. Barbir, B. Cain, F. Douglis, M. Green, M. Hofmann, R. Nair, D. Potter and O. Spatscheck. Known CDN request-routing mechanisms. IETF Internet-Draft, February 2002.
- [7] P. Barford, A. Bestavros, A. Bradley, and M. E. Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *World Wide Web Journal*, 1999.
- [8] M. Beck and T. Moore. The Internet2 Distributed Storage Infrastructure Project: An Architecture for Internet Content Channels. In *Proceedings of the 3rd International Web Caching Workshop*, 1998.
- [9] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. IETF RFC 1945, May 1996.

- [10] T. Brisco. DNS Support for Load Balancing. IETF RFC 1794, April 1995.
- [11] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.
- [12] V. Cardellini, M. Colajanni, and P. Yu. DNS Dispatching Algorithms with State Estimators for Scalable Web Server Clusters. *World Wide Web*, 2(2), July 1999.
- [13] V. Cardellini, M. Colajanni, and P. Yu. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing*, pages 28–39, May/June 1999.
- [14] V. Cate. Alex: A Global File System. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12, May 1992.
- [15] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of IEEE INFOCOM 2000*, March 2000.
- [16] M. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, Nov 1997.
- [17] C. R. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of www client-based traces. Technical Report CS 95-010, Boston University Computer Science Department, Boston, MA, June 1995.
- [18] M. Day, B. Cain, G. Tomlinson, and P. Rzewski. A model for content internetworking (CDI). Internet Draft (draft-ietf-cdi-model-01.txt), February 2002.
- [19] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.
- [20] A. Downey. The structural cause of file size distributions. In *Proceedings of the Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Cincinnati, OH, Aug 2001.
- [21] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of the IEEE Infocom'00, Tel Aviv, Israel*, March 2000.
- [22] Z. Fei. A Novel Approach to Managing Consistency in Content Distribution Networks. In *Proceedings of the 6th Workshop on Web Caching and Content Distribution, Boston, MA*, June 2001.
- [23] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar. A Novel Server Selection Technique for Improving the Response Time of a Replicated Service. In *Proceedings of IEEE INFOCOM'98*, 1998.
- [24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. IETF RFC 2068, January 1997.
- [25] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. IETF RFC 2616, June 1999.

- [26] P. Francis. Yoid: Extending the Internet Multicast Architecture. Technical report, AT&T Center for Internet Research at ICSI (ACIRI), April 2000.
- [27] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.
- [28] M. Gritter and D.R. Cheriton. An Architecture for Content Routing Support in the Internet. In *Proceedings of the USENIX Symposium on Internet Technologies, San Francisco, CA*, March 2001.
- [29] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.
- [30] J. C. Hu, S. Mungee, and D. C. Schmidt. Techniques for developing and measuring high-performance Web servers over ATM networks. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, San Francisco, CA, Mar 1998.
- [31] J. C. Hu, I. Pyarali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on Web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference (held as part of GLOBECOM '97)*, Phoenix, AZ, Nov 1997.
- [32] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proceedings of the 7th International World Wide Web Conference*, April 1998.
- [33] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [34] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2), March/April 2000.
- [35] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek. The measured performance of content distribution networks. In *International Web Caching and Content Delivery Workshop (WCW)*, Lisbon, Portugal, May 2000. <http://www.terena.nl/conf/wcw/Proceedings/S4/S4-1.pdf>.
- [36] P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [37] M. Koletsou and G. M. Voelker. The Medusa proxy: A tool for exploring user-perceived web performance. In *Proceedings of International Web Caching and Content Delivery Workshop (WCW)*, Boston, MA, June 2001. Elsevier.
- [38] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice*. Addison Wesley, 2001.
- [39] B. Krishnamurthy and C. Wills. Proxy Cache Coherency and Replacement—Towards a More Complete Picture. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

- [40] B. Krishnamurthy and C. Wills. Study of Piggyback Cache Validation for Proxy Caches in the WWW. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems, Monterey, CA*, pages 1–12, December 1997.
- [41] B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, November 2001.
- [42] T. T. Kwan, R. E. McGrath, and D. A. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer*, 28(11):68–74, November 1995.
- [43] D. Li, P. Cao, and M. Dahlin. WCIP: Web Cache Invalidation Protocol. IETF Internet Draft, November 2000.
- [44] B. Mah. An empirical model of HTTP network traffic. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Kobe, Japan, Apr 1997.
- [45] Z. Morley Mao, C. D. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang. A precise and efficient evaluation of the proximity between web clients and their local DNS servers. In *Proceedings of USENIX Annual Technical Conference*, June 2002.
- [46] J. C. Mogul. Clarifying the fundamentals of HTTP. In *Proceedings of WWW 2002 Conference*, Honolulu, HA, May 2002.
- [47] J. C. Mogul. Network behavior of a busy Web server and its clients. Technical Report 95/5, Digital Equipment Corporation Western Research Lab, Palo Alto, CA, October 1995.
- [48] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of ACM SIGCOMM Conference*, 1997.
- [49] D. Mosedale, W. Foss, and R. McCool. Lessons Learned Administering Netscape's Internet Site. *IEEE Internet Computing*, 1(2):28–35, March/April 1997.
- [50] E. M. Nahum, T. Barzilai, and D. Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(2):2–11, Feb 2002.
- [51] E. M. Nahum, M. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on WWW server performance. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Cambridge, MA, June 2001.
- [52] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *Proceedings of the World Wide Web conference (WWW2002)*, May 2002.
- [53] Open Market. FastCGI. <http://www.fastcgi.com/>.
- [54] V. N. Padmanabhan and L. Qui. The content and access dynamics of a busy web site: findings and implications. In *SIGCOMM*, pages 111–123, 2000.
- [55] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-Aware Request Distribution in Cluster-based Network Services. In *Proceedings of ASPLOS-VIII*, October 1998.

- [56] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, Monterey, CA, June 1999.
- [57] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O Lite: A copy-free UNIX I/O system. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.
- [58] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison-Wesley, 2002.
- [59] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of IEEE INFOCOM 2001*, 2001.
- [60] J. Song, A. Iyengar, E. Levy, and D. Dias. Architecture of a Web Server Accelerator. *Computer Networks*, 38(1), 2002.
- [61] The Standard Performance Evaluation Corporation. SpecWeb99. <http://www.spec.org/osg/web99>, 1999.
- [62] *Squid Internet Object Cache Users Guide*. Available on-line at <http://squid.nlanr.net>, 1997.
- [63] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining Temporal Coherency of Virtual Warehouses. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, Madrid, Spain, December 1998.
- [64] Sun Microsystems Inc. The Java Web server. <http://www.sun.com/software/jwebserver/index.html>.
- [65] R. Tewari, T. Niranjana, and S. Ramamurthy. WCDP: Web Content Distribution Protocol. IETF Internet Draft, March 2002.
- [66] Red Hat Inc. The Tux WWW server. <http://people.redhat.com/mingo/TUX-patches/>.
- [67] D. C. Verma. *Content Distribution Networks: An Engineering Approach*. John Wiley & Sons, 2002.
- [68] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Server-driven Consistency for Large-scale Dynamic Web Services. In *Proceedings of the 10th World Wide Web Conference, Hong Kong*, May 2001.
- [69] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases for Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, January 1999.
- [70] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the Usenix Symposium on Internet Technologies (USITS'99)*, Boulder, CO, October 1999.
- [71] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of the ACM SIGCOMM'99, Boston, MA*, September 1999.
- [72] Zeus Inc. The Zeus WWW server. <http://www.zeus.co.uk>.