RC22845 (W0307-113) July 15, 2003
Computer Science

IBM Research Report

Web Caching, Consistency, and Content Distribution

Arun K. Iyengar, Erich M. Nahum, Anees A. Shaikh, Renu Tewari
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Contents

0-8493-0052-5/00/$0.00+$.50
© 2003 by CRC Press LLC

1

Web Caching, Consistency, and Content Distribution

Arun Iyengar, Erich Nahum, Anees Shaikh, Renu Tewari
IBM Research

CONTENTS

1.1 Introduction

Caching has been widely deployed to improve Web performance by reducing client-observed latency and network bandwidth usage in addition to improving server scalability by reducing the load on the servers. Web caches can be deployed at various points in the network. Forward proxy caches are deployed close to the client at network entry points by ISPs to reduce the network bandwidth usage and improve client latency by caching frequently accessed data. Such caches can be either transparent to the client or be manually configured. With transparent caching the packets are intercepted by an intermediate router (layer 4 or layer 7 switch) and transparently routed to a cache which in turn responds to the client directly [cis]. Manual configuration requires the client to explicitly configure the browser to go via a proxy cache. In addition to forward proxies, caches can be deployed as a front-end to a server farm to reduce server load and increase server scalability. Such caches called reverse proxies are useful in eliminating the load of a hot-set from impacting the server performance. Typically reverse proxies are in the same administrative domain as the server.

As with any caching system, Web caches need to use a cache replacement policy to decide what to keep in the cache and a consistency mechanism on how to keep it current. Various cache replacement algorithms from LRU to Greedy-dual size have been studied in the context of the Web to improve cache performance in terms of client response times and server throughput. For maintaining consistency, Web objects may have explicit 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. Furthermore, expiration times are not sufficient for applications which have strong consistency requirements. Without expiration times the proxy cache needs to always check the staleness of the data with the server using if-modified-since messages, thereby, increasing client response times. Stale cached data and the inability in many cases to cache dynamic and personalized data limit the effectiveness of caching. Numerous proposals have been made to extend the support for consistency such that stronger requirements can be met [Li et al., 2000].

Simple proxy caching is limited by the space and processing capacity of a single caching server. To further improve performance caching can be extended to include a group of cooperating caches deployed in the network either in a hierarchical or distributed manner. Hierarchical caches such as the NLANR Squid [Squ, 1997] cache consist of a single tree with parent child relationship, whereas other organizations include meshes with hierarchical or centralized directories [Wolman et al., 1999]. A further extension of distributed caching are content distribution networks (CDNs) that supplement the client side proxy caching to other points in the network controlled by the CDN service provider. A CDN is a shared network of servers or caches that deliver content to users on behalf of content providers by using various request routing techniques. 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. In doing so CDN's also reduce the load on origin servers.

This chapter examines several issues related to cache management consistency maintenance and the overall architecture and techniques for routing requests in CDNs. We also provide insight into the performance improvements typically achieved by CDN's.

1.2 Practical Issues in the Design of Caches

Web caches can be implemented at the application level [Iyengar, 1999], kernel level [Joubert et al., 2001], or under an embedded operating system [Song et al., 2002]. Application-level caches are the easiest to design and have the potential for the most features. Kernel-level caches are harder to design but have the potential for better performance. Caches can also be designed for embedded operating systems which may be optimized for certain features such as communication. Such caches may offer comparable performance to kernel-level caches. A problem with using embedded operating systems is that as processor technology improves, it may not be feasible for the embedded operating system to keep up with new processors. This means that over time, the advantage achieved by a cache running under an embedded operating system may decrease.

HTTP provides a standard interface for applications to utilize caches. An HTTP interface alone is limiting, however, and doesn't provide adequate support for explicitly managing the contents of a cache. It is also not the most efficient interface and can be cumbersome for applications to use. It is therefore preferable for the cache to define an interface which an application program can use to explicitly add, delete, and update cached objects [Iyengar, 1999].

The number of transactions per unit time that a Web cache has to perform in order to achieve good performance is orders of magnitude less than that needed by a processor cache. Therefore, Web caches can employ more sophisticated consistency and replacement policies. Cache replacement policies are applied when a cache becomes full and it is necessary to determine which objects in the cache to keep. The least recently used algorithm (LRU) has been used for caching across a broad range of disciplines. In LRU, the object which was accessed the farthest in the past is selected for removal when the cache becomes full. LRU has the advantage that it is easy to implement. A doubly linked list is used to order objects by access times. Whenever an object is accessed, it is moved to the front of the list.

A number of cache replacement algorithms have been proposed which result in higher cache hit rates than LRU. One of the most commonly used such algorithm is the GreedyDual-Size algorithm [Cao and Irani, 1997]. The GreedyDual-Size algorithm associates a cost $C(o)$ with each object o. The cost would typically be associated with how expensive it is to fetch or create the object. It is preferable to cache more expensive objects because doing so results in greater savings in the event of a cache hit. GreedyDual-Size divides $C(o)$ by the size of o, $S(o)$, in order to arrive at an estimate $H(o)$ of the savings per unit of cache memory which would be achieved by caching the object.

When object o is first brought into the cache, $H(o)$ is set to $C(o)/S(o)$. When the cache becomes full and an object needs to be removed, the object with the lowest H value, H_{min}, is removed, and all objects reduce their H values by H_{min}. When an object is accessed, its H value is restored to $C(o)/S(o)$. That way, objects which are accessed frequently will on average have higher H values and are therefore less likely to be replaced.

A naive implementation would require $n-1$ subtractions every time an object is replaced to update H values for the remaining cached objects, where n is the number of cached objects. This is inefficient. Instead, an inflation value, L, is maintained. When an object o is accessed, $H(o)$ is set to $C(o)/S(o)+L$. By adding L to compute the H value of an accessed object, it becomes unnecessary to reduce H values for all remaining objects when an object is replaced. L is initially set to 0. Whenever an object is replaced, L is updated to the H value of the replaced object.

The cost function C depends on resources the cache is trying to minimize. If the objective is to maximize cache hit rates, then the cost function should be a constant for each object. If the objective is to minimize time consumed fetching remote objects, then $C(o)$ could be the expected latency for fetching o. For a dynamic Web object, the CPU cycles consumed for creating the object may have the most significant effect on performance. The cost function for such an object could thus be proportional to the CPU cycles for creating the object.

Caches can be implemented using both main memory and disk storage. Main memory offers better performance. In some cases, however, disk storage is essential. If the cache size exceeds the main memory size, it may be desirable to store colder objects on disk instead of deleting the objects to keep the cache within memory limits. Disk storage is also essential for persistence when a cache must be shut down and later restarted. If the cache is totally purged each time the machine is shut down, then performance is likely to be poor while the machine is being brought to a warm state after start up. If, on the other hand, cached information is maintained on disk before the shutdown, the cache can be brought to a warm state right after the system is restarted. Disk storage is also important for fault tolerance. When a cache fails, if hot objects are maintained on disk, then the cache can be quickly brought to a warm state after the failure.

File systems and databases can be used for persistently storing cached data. A key problem with file systems and databases is that they can be inefficient. For Web caches, the rate at which objects are added to and deleted from caches can be high [Markatos et al., 1999]. If a file system is used and a different file is used to store each object, the overhead for creating and deleting files can be significant. Customized disk storage allocators can often achieve much better performance. Good performance for Web workloads has been achieved by maintaining multiple objects in a single file and efficiently managing the storage space within the file [Iyengar et al., 2001]. A portable disk storage allocator we have built in Java achieves considerably better performance than both file systems and databases.

1.3 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
Online auctions: Consider a Web server that offers online auctions over the Internet. For each item being sold, the server maintains information 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.

1.3.1 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 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 [Gwertzman and Seltzer, 1996; Squ, 1997]. 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

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, notify(all$_t$))	0
Server State	None	None	All	All$_t$	None

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 are 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. 'All' means all of the subscribers for server-driven invalidation. 'All$_t$' means all of the servers within lease duration t.

stock quotes) are updated every few minutes [Barford et al., 1999]. 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.

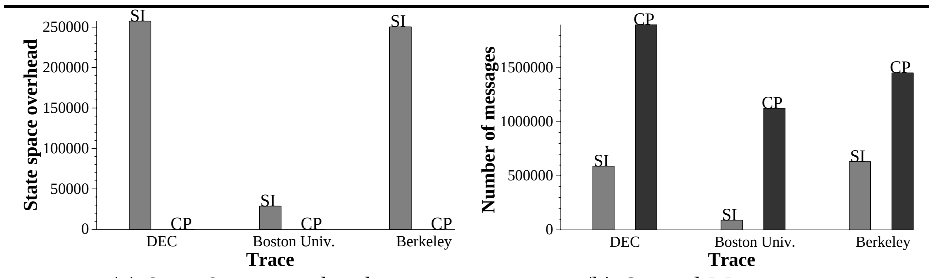
1.3.2 Consistency Mechanisms

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 [Yin et al., 1999b]. 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 [Li et al., 2000]. 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 [Tewari et al., 2002].

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 [Yin et al., 1999b]. 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



(a) State Space overhead (b) Control Messages

FIGURE 1.1
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

the server to verify that the cached data is not stale [Cate, 1992; Gwertzman and Seltzer, 1996; Squ, 1997]. 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 may require a significant amount of state to be maintained, the latter is stateless but can impose a large control message overhead. Figure 1.1 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* [Gray and Cheriton, 1989]. 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 [Gray and Cheriton, 1989]. The use of leases for Web proxy caches was first alluded to in [Liu and Cao, 1997] and was subsequently investigated in detail in [Yin et al., 1999b]. 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 [Yin et al., 1999a; Yu et al., 1999]. 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 [Duvvuri et al., 2000].

A qualitative comparison of the overheads of the different consistency mechanisms is shown in Table 1.1. 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.

1.3.3 Invalidates and Updates

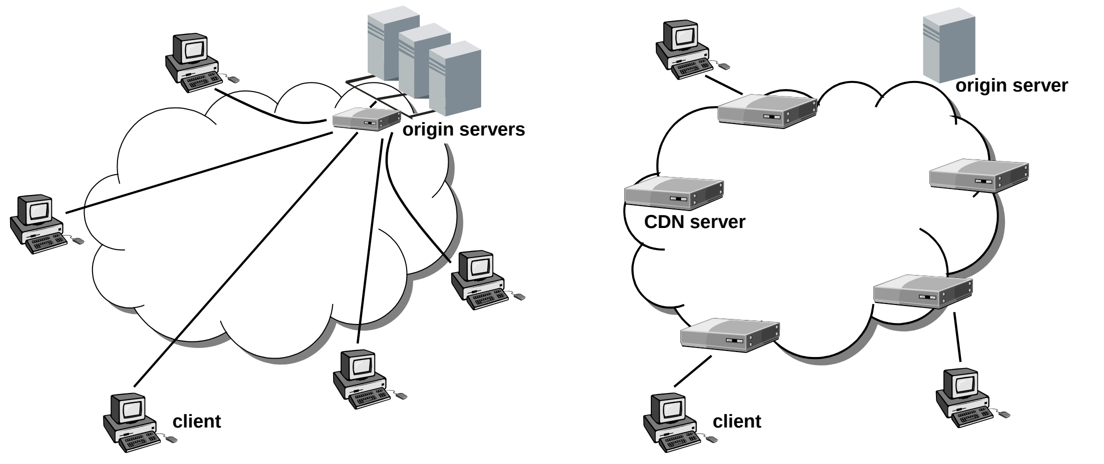
With server-driven consistency mechanisms, when an object is modified, the origin server notifies each “subscribing” intermediary. The notification consists of either an invalidate message or an updated (new) version of the object. Sending an invalidate message causes an intermediary to mark the object 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[Krishnamurthy and Wills, 1997]. 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.

1.4 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



(a) Traditional centralized architecture (b) Distributed CDN architecture

FIGURE 1.2
Content-serving architectures

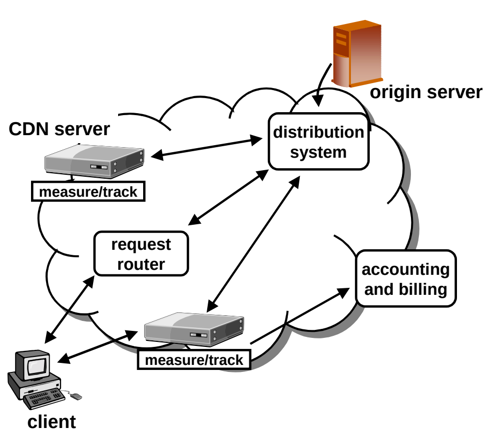
factors not related to performance such as cost, reliability, and ease of management. In a traditional content-serving architecture all clients request content from a single location, as shown in Figure 1.2(a). In this architecture, scalability and performance are improved by adding servers, without 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.

Some ISPs address performance bottlenecks in the network by deploying caching proxies near clients to reduce network traffic and improve client performance. Caching proxies are limited, however, since they operate based only on user demand for a very large and diverse set of content. Most proxy cache studies, for example, find they achieve only a 20–40% hit rate [IRCache Project Daily Reports, 2002; Wolman et al., 1999].

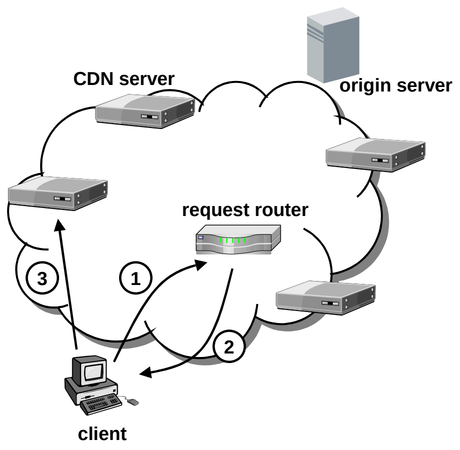
Another way to address poor performance due to network congestion, or flash crowds at servers, is to distribute popular content to servers or caches located closer to the edges of the network, as shown in Figure 1.2(b). Such a distributed network of servers comprises a content distribution network (CDN). A CDN is simply a network of servers or caches that delivers 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. Since CDN servers receive requests only for hosted content, cache misses typically occur only for compulsory misses due to the initial request for some content.

CDNs have several advantages over traditional centralized content-serving architectures, including [Verma, 2002]:

- 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
- reducing content provider costs by reducing the need to invest in more powerful servers or more bandwidth as user population increases



(a) CDN architectural elements



(b) CDN request-routing

FIGURE 1.3
CDN architecture and request-routing

- improving site availability by replicating content in many distributed locations

Content distribution service providers (CDSPs) manage and operate the CDN, thus freeing content providers from the tasks of maintaining the servers themselves. Some network service providers offer a CDN service in addition to network access service (e.g., AT&T and Cable&Wireless). Other CDSPs focus primarily on providing a variety of CDN services (e.g., Akamai and Speedera).

CDN servers may be configured in tree-like hierarchies [Yu et al., 1999] or clusters of cooperating proxies that employ content-based routing to exchange data [Gritter and Cheriton, 2001]. 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 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 [Krishnamurthy et al., 2001]. 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.

1.4.1 CDN Architectural Elements

As illustrated in Figure 1.3(a), 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 [Day et al., 2002]. The distribution system is responsible for moving content from origin servers into CDN servers and ensuring data consistency. Section 1.4.4 describes some techniques used to maintain consistency in CDNs. The accounting/billing system collects logs of client accesses and 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. The request-routing system may also interact with the distribution system to keep an up-to-date view of which content resides on which CDN servers.

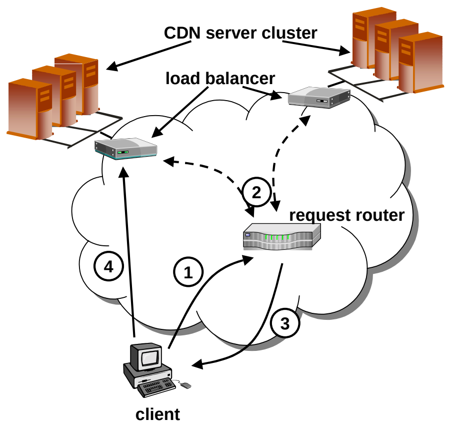


FIGURE 1.4
Interaction between request router and CDN servers

The request-routing system operates as shown in Figure 1.3(b). 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).

1.4.2 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 what content is being requested, for example if request-routing is done in the context of name resolution. In this case the request contains only a server hostname (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 or connection router, it may be possible to query a server-side agent for server load information, as shown in Figure 1.4. After the client makes its request, the 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 determine which CDN server is closest to the client and then respond to the client appropriately.

A common strategy used in CDN request-routing is to choose a server "nearby" 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.

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 schemes [Barbir et al., 2002]. 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 acknowledgments) 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., status 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 [Krishnamurthy et al., 2001]. 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 or NS record) to the CDSP’s 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. This type of URL rewriting may also be done dynamically as the base page is retrieved, though this may increase client response time.

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 [Shaikh et al., 2001; Mao et al., 2002; Barbir et al., 2002].

One problem is that request-routing is done on the granularity of DNS domains, rather than per-object, thus limiting the ability to make object-specific server selection decisions. A second problem is that requests usually come to the DNS server not from clients, but from their local nameservers. Hence, the CDN server is chosen based on the local nameserver address instead of the client, which may lead to poor decisions if clients and their local nameservers are not proximal. Finally, as mentioned earlier, DNS request routers return answers with small TTLs to facilitate fine-grained load balancing. This may actually increase Web access latency because clients must contact the DNS server more frequently to refresh the name-to-address mapping.

1.4.3 Request-Routing Metrics and Mechanisms

Request-routing systems use a number of metrics and techniques in deciding which CDN server is best suited for a given client. This section describes some specific metrics and techniques used in commercially available load-balancing and request-routing products. Note that these techniques are not necessarily limited to CDNs – they are applicable to load-balancing and request-routing in many replicated content-serving architectures.

Determining server availability and load

Server availability is often the most critical criterion used in request routing. Availability is usually determined using "health checks" initiated by the request-router. These probes may be implemented at layer-3 with ICMP (Internet Control Message Protocol) ping, or layer-4 by checking that TCP connections can be established, for example. In addition, the request router is often configured to perform application-layer health checks, such as retrieving a specified file using HTTP or FTP, or interacting with an IMAP mail server or telnet server. Application-layer checks are important to detect cases when a host machine may be operational, but a mission-critical application is not, hence making the server unsuitable for handling client requests.

As described in Section 1.4.2, the request router may consult a local load-balancing switch at each site to determine the relative load at candidate server sites. The local load balancer typically keeps track of statistics like the number of active client connections, the aggregate packet and connection arrival rates, and number of available servers. Using agents that reside on the servers themselves, the local load balancer may also collect information such as per-server CPU load and memory usage. All or some of these statistics can be queried by the request-router to assess the relative load of each server or server cluster.

In most vendor solutions the request router is tightly integrated with an agent at the server-side load-balancer which reports statistics, or an aggregate "score." This scheme usually requires that the request router and load-balancer are from the same vendor since they often communicate using proprietary protocols. Limited support may also be available for communicating with heterogeneous local load balancers or servers. This is often done using the Simple Network Management Protocol (SNMP), since most products and operating systems support SNMP queries of information such as packet arrival rate or number of active concurrent connections. The request router may also use the responsiveness of application-layer health checks as an indication of the site or server load. These checks appear as normal client requests and thus do not require special protocols.

Determining network proximity and performance

Since network performance plays an important role in overall end-to-end Web performance, the request router tries to direct clients to the nearest server in terms of geographic or topological location, or network latency. In a typical DNS-based request-routing system, however, this is complicated by several factors. The network performance (e.g., delay, loss, throughput) may change dynamically and dramatically over time, requiring that the notion of "nearest" be updated regularly. Also, the client's actual location may be difficult to determine if the local nameserver that sends DNS requests on behalf of the client is not nearby the client. Finally, the network performance must be determined from the point of view of each server site, rather than from the request router.

One approach is for the request router to ask candidate CDN servers to measure network latency to the client (or its nameserver) using ICMP echo (i.e., ping) and report the measured values. The request router then responds to the client with the address of 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. Measurement results are reported back to the request router and can be cached for a short time to serve subsequent requests from the same or nearby clients. On the other hand, this technique can introduce additional latency for the client as the request router waits for responses from the CDN servers.

In a slightly different approach the request router can forward the request to agents at several sites, each of which then respond directly to the client. The client uses the response that reaches it first, thus automatically choosing the nearest site. For a fair "race", the request router must know its one-way latency to each site, and delay the forwarding accordingly, to ensure that each site receives the forwarded request at the same time. This approach avoids actively probing the client nameserver from each server site, but it does require that each responding agent spoof the IP address of the request router (to which the request was originally sent). Otherwise the client may not accept the response.

Another alternative approach is to passively monitor client connections to the CDN servers to build a performance database that can be consulted by the request router when making its decision. For example, the local load balancer can capture and examine TCP packets to estimate the round-trip time between the site and a particular client. Using these estimates, the request router can determine which site has the lowest delay to some group of clients. This technique must address several issues, however, such as how to collect a sufficient number of samples at each CDN server, and how to aggregate client performance statistics. It also requires tight integration between the request router and the performance monitoring entity at each server. Note that all three of these approaches for determining client network proximity have been used in vendor products.

In addition to dynamic metrics such as network latency, request routing systems often depend on more static notions of network proximity, based either on hopcount or geographic location. A hopcount-based metric may be implemented by simply using a UDP-based traceroute from each server site to the client nameserver, similar to the ICMP echo technique described above. If the request router has access to network routers at the server sites, it can consult interdomain routing tables at each site to find out the distance between the site and the client subnet in terms of AS-hops. This requires a specialized agent or protocol on the network routers, however. Moreover, several studies have shown hopcount to be a poor predictor of network latency [Crovella and Carter, 1995; Obraczka and Silva, 2000].

Many request routing systems attempt to direct clients to the geographically nearest site, often based on coarse notions of regions (e.g., U.S. East coast) or continents (e.g., Asia-Pacific clients). Determining geographic proximity based on IP addresses remains an active and open research topic and though a number of heuristics have been developed, they are not always accurate [Moore et al., 2000; Padmanabhan and Subramanian, 2001]. Nevertheless, it is possible to use information published by regional Internet registries to obtain rough per-country address block allocations [ian, 2003]. These can be used to determine, to some extent, the location of the client in order to direct it to the nearest site. Most request-routing products also offer the ability to manually specify IP addresses and their associated geographic regions. This is useful, for example, when the requests are anticipated from known clients (e.g., remote branch offices).

1.4.4 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 [Srinivasan et al., 1998], and leases [Yin et al., 2001]. 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 [Yu et al., 1999]. 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 [Yin et al., 2001]. 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 [Ninan et al., 2002]. 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 [Krishnamurthy and Wills, 1997], the use of deltas for sending updates [Mogul et al., 1997], an application-level multicast framework for Internet distribution [Francis, 2000] and the efficacy of sending updates versus invalidates [Fei, 2001].

1.4.5 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 [Johnson et al., 2000]. 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 [Koletsou and Voelker, 2001]. 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 [Krishnamurthy et al., 2001]. 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.

Acknowledgement

Some of the material in this chapter appeared in the paper *Enhancing Web Performance* by the same authors in Proceedings of the 2002 IFIP World Computer Congress (Communication Systems: The State of the Art), edited by Lyman Chapin, published by Kluwer Academic Publishers, Boston, copyright 2002, International Federation for Information Processing.

References

Cisco LocalDirector 400 series. `http://www.cisco.com/warp/public/cc/pd/cxsr/400/`.

Squid Internet object cache users guide. `http://squid.nlanr.net`, 1997.

IP address services. `http://www.iana.org/ipaddress/ip-addresses.htm`, January 2003.

Abbie Barbir, Brad Cain, Fred Douglis, Mark Green, Markus Hofmann, Raj Nair, Doug Potter, and Oliver Spatscheck. Known CDN request-routing mechanisms. Internet Draft (draft-ietf-cdi-known-request-routing-00.txt), February 2002.

Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: Characteristics and caching implications. In *Proceedings of the World Wide Web Journal*, 1999.

Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

Vincent Cate. Alex: A global file system. In *Proceedings of the 1992 USENIX File System Workshop*, May 1992.

Mark E. Crovella and Robert L. Carter. Dynamic server selection in the Internet. In *Proc. of IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS '95)*, 1995.

Mark Day, Brad Cain, Gary Tomlinson, and Phil Rzewski. A model for content internetworking (CDI). Internet Draft (draft-ietf-cdi-model-01.txt), February 2002.

Venkata Duvvuri, Prashant Shenoy, and Renu Tewari. Adaptive leases: A strong consistency mechanism for the World Wide Web. In *Proceedings of IEEE Infocom*, Tel Aviv, Mar 2000.

Zongming Fei. A novel approach to managing consistency in content distribution networks. In *Proceedings of the 6th Workshop on Web Caching and Content Distribution*, Boston, Jun 2001.

Paul Francis. Yoid: Extending the Internet multicast architecture. In *Technical report, AT&T Center for Internet Research at ICSI (ACIRI)*, Apr 2000.

Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.

Mark Gritter and David R. Cheriton. An architecture for content routing support in the Internet. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, San Francisco, Mar 2001.

James Gwertzman and Margo Seltzer. World-Wide Web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference*, Jan 1996.

IRCache Project Daily Reports. `http://www.ircache.net/Statistics/Summaries/`

0-8493-0052-5/00/$0.00+$.50

`Root/`, April 2002.

Arun Iyengar. Design and performance of a general-purpose software cache. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99)*, February 1999.

Arun Iyengar, Shudong Jin, and Jim Challenger. Efficient algorithms for persistent storage allocation. In *Proceedings of the 18th IEEE Symposium on Mass Storage Systems*, April 2001.

Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans 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`.

Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, and John Tracey. High-performance memory-based Web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

Mimika Koletsou and Geoffrey 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.

Bala Krishnamurthy and Craig Wills. Study of piggyback cache validation for proxy caches in the WWW. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, Dec 1997.

Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the use and performance of content distribution networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, November 2001.

Dan Li, Pei Cao, and Mike Dahlin. WCIP: Web cache invalidation protocol. In *IETF Internet Draft*, Nov 2000.

Chengjie Liu and Pei Cao. Maintaining strong cache consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

Zhuoqing Morley Mao, Charles D. Cranor, Fred Douglis, Michael Rabinovich, Oliver Spatscheck, and Jia 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.

Evangelos Markatos, Manolis Katevenis, Dionisis Pnevmatikatos, and Michael Flouris. Secondary storage management for Web proxies. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

Jeffrey C. Mogul, Fred Douglis, Anya Feldmann, and Bala Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM*, Sep 1997.

David Moore, Ram Periakaruppan, and Jim Donohoe. Where in the world is netgeo.caida.org? In *Proceedings of the Internet Society Conference (INET)*, 2000. `http://www.caida.org/outreach/papers/2000/inet_netgeo/`.

Anoop Ninan, Purushottam Kulkarni, Prashant Shenoy, Krithi Ramamritham, and Renu Tewari. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *Proceedings of the Eleventh International World Wide Web Conference (WWW2002)*, May 2002.

Katia Obraczka and Fabio Silva. Network latency metrics for server proximity. In *Proceedings of*

IEEE GLOBECOM, pages 421–427, 2000.

Venkata N. Padmanabhan and Lakshminarayanan Subramanian. An investigation of geographic mapping techniques for Internet hosts. In *Proceedings of ACM SIGCOMM*, San Diego, CA, August 2001.

Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the effectiveness of DNS-based server selection. In *Proceedings of IEEE INFOCOM*, Anchorage, AK, April 2001.

Junehua Song, Arun Iyengar, Eric Levy, and Daniel Dias. Architecture of a Web server accelerator. *Computer Networks*, 38(1), 2002.

Raghav Srinivasan, Chao Liang, and Krithi Ramamritham. Maintaining temporal coherency of virtual warehouses. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, Madrid, Dec 1998.

Renu Tewari, Thirumale Niranjan, and Srikanth Ramamurthy. WCDP: Web content distribution protocol. In *IETF Internet Draft*, Mar 2002.

Dinesh C. Verma. *Content Distribution Networks: An Engineering Approach.* John Wiley & Sons, 2002.

Alex Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–31, December 1999.

Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Arun Iyengar. Engineering server-driven consistency for large-scale dynamic Web services. In *Proceedings of the 10th World Wide Web Conference*, Hong Kong, May 2001.

Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Hierarchical cache consistency in a WAN. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems (USITS'99)*, Boulder, Oct 1999a.

Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):563–576, 1999b.

Haobo Yu, Lee Breslau, and Scott Shenker. A scalable Web cache consistency architecture. In *Proceedings of the ACM SIGCOMM*, Boston, Sep 1999.