IBM Research Report

Improving Web Site Performance

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

IBM

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Contents

1

Improving Web Site Performance

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

CONTENTS

1.1 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 chapter 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 for 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.

We also discuss Web server workload characterization and show properties of the workloads related to performance such as document sizes, popularities, and protocol versions used. Understanding these properties about a Web site is critically important for optimizing performance.

1.2 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 1.1 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 or other back-end systems
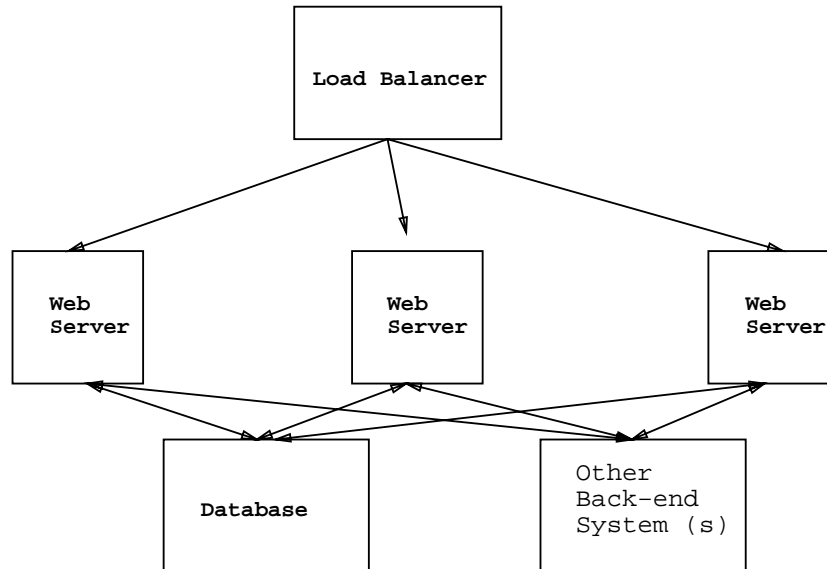
FIGURE 1.1
Architecture of a scalable Web site. Requests are directed from the load balancer to one of several Web servers. The Web servers may access one or more databases or other back-end systems for creating content.

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 [Kwan et al., 1995]. 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.

1.2.1   Load Balancing

1.2.1.1   Load Balancing via DNS

The load balancer in Figure 1.1 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 Web server. DNS servers can select one of these servers using a policy such as round robin [Brisco].

There are other approaches which can be used for DNS load balances which offer some advantages over simple round robin [Cardellini et al., 1999b]. 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 [Beck and Moore, 1998].

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 [Dias et al., 1996]. 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 [Shaikh et al., 2001]. 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 [Cardellini et al., 1999a]. 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.

1.2.1.2 Load Balancing via Connection Routers

Another approach to load balancing is using a connection router (also known as "Dispatchers", "Web switches", "content switches") 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.

In such environments a front-end connection router directs incoming client requests to one of the physical server machines, as shown in Figure 1.2. The physical servers often share one or more virtual IP addresses so that any server can respond to client requests. In other scenarios, the servers have only private addresses, so the connection router accepts all connections destined for the site virtual address. The request-routing decision can be based on a number of criteria, including server load, client request, or client identity.

Connection routers are typically required to perform several functions related to the routing decision:

- monitor server load and distribute incoming requests to balance the load across servers

- examine client requests to determine which server is appropriate to handle the request

- identify the client to maintain session affinity with a particular server for e-business applications

In addition, many commercial connection routers provide functions important in a production data center environment. These include:

- failover to a hot standby to improve availability
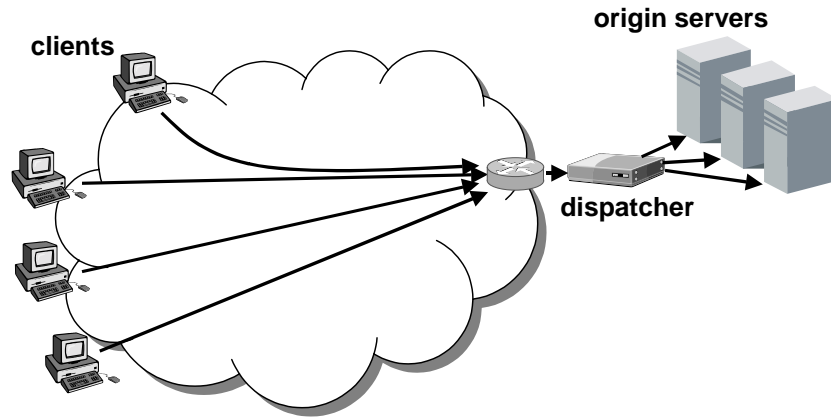
FIGURE 1.2
A server-side connection router (labelled *dispatcher*) directs incoming client Web requests to one of the physical servers in the cluster.

- detection and avoidance of many common denial-of-service attacks

- SSL acceleration to improve the performance of secure applications

- simplified configuration and management (e.g., Web browser-based configuration interface)

A variety of networking equipment and software vendors offer connection routers, including Cisco Systems [cis, b,a], Nortel Networks [nor], IBM [ibm], Intel [int], Foundry Networks [fou], and F5 Networks [f5:].

Request-routing may be done primarily in hardware, completely in software, or with a hardware switch combined with control software. For example, several of the vendors mentioned above offer dedicated hardware solutions consisting of multiple fast microprocessors, several Fast Ethernet and Gigabit Ethernet ports, and plenty of memory and storage. Others offer software-only solutions that can be installed on a variety of standard platforms.

IBM's Network Dispatcher [Hunt et al., 1998] 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. Servers with a given weight are given priority in receiving new connections over servers with a lesser weight. Consequently, 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). SSL is commonly used for encryption on the Web. It can add significant overhead, however. 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, particularly 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.

1.2.1.3 Content-based Routing

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 [Pai et al., 1998]. 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. In order to examine the contents of a request, the router must terminate the connection with the client.

The use of layer-4 or layer-7 routers depends on the request-routing goal. Load-balancing across replicated content servers, for example, typically does not require knowledge about the client request or identity, and thus is well-suited to a layer-4 approach. Simple session affinity based on client IP address, or directing requests to servers based on application (e.g., port 80 HTTP traffic vs. port 110 POP3 traffic) is also easily accomplished by examining layer-3/4 headers of packets while in transit through the router. For example, the router may peek at the TCP header flags to determine when a SYN packet arrives from a client indicating a new connection establishment. Then, once a SYN is identified, the source and destination port numbers and IP addresses may be used to direct the request to the right server. This decision is recorded in a table so that subsequent packets arriving with the same header fields are directed to the same server.

Layer-4 routers, due to their relative simplicity, are often implemented as specialized hardware since they need not perform any layer-4 protocol processing or maintain much per-connection state. Although traffic from the clients must be routed via the router, the response traffic from the server, which accounts for the bulk of the data in HTTP transactions, can bypass the router, flowing directly back to the client. This is typically done by configuring each server to respond to traffic destined for the virtual IP address(es), using IP aliasing, for example.

Although layer-4 routers are usually deployed as front-end appliances, an alternative is to allow back-end servers to perform load-balancing themselves by redirecting connections to relatively underloaded machines [Bestavros et al., 1998]. However, even without such an optimization, hardware-based layer-4 routers are able to achieve very high scalability and performance.

Request-routing based on the URL (or other application-layer information), on the other hand, requires the router to terminate the incoming TCP connection and receive enough information to
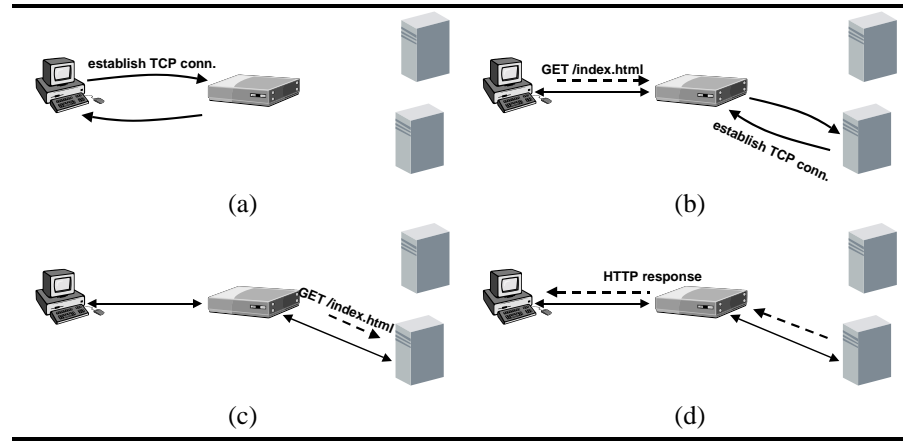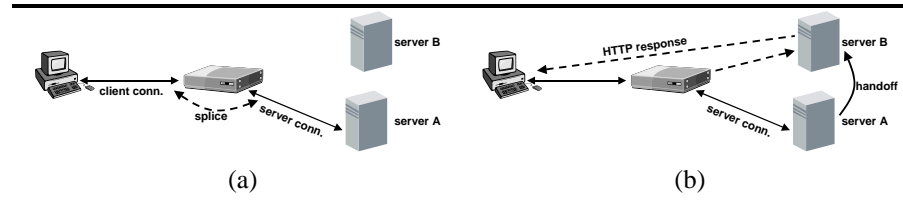
FIGURE 1.3
The connection router accepts the TCP connection transparently) from the client (a). Next, in (b) the client sends a `GET` request on the established connection, prompting the router to open a new TCP connection to the appropriate server. In (c) the router forwards the client request to the server, and, in (d), returns the response to the client.

make a routing decision. In the case of Web traffic, for example, the router must accept the incoming TCP connection and then wait for the client to send an HTTP request in order to view application-layer information such as the requested URL, or HTTP cookie. Once enough information to make a routing decision is received, the router can create a new connection to the appropriate server and forward the client request. The server response is then passed back to the client via the router on the client's original connection. Figure 1.3 outlines these steps.

In the simplest realization, a layer-7 router may be implemented as a software application-level gateway that transparently accepts incoming client connections (destined for port 80 to the server virtual IP address) and reads the requests. After deciding which server should handle the request, the application can forward it on a new or pre-established connection to the server. The router serves as a bridge between the two connections, copying data from one to the other. From a networking point of view, the router behaves much like a forward Web proxy installed at an enterprise site, though the forward proxy's primary function lies primarily in filtering and content caching, rather than request routing. In this approach, the router can quickly become a bottleneck, since it must perform connection termination and management for a large number of clients [Aron et al., 2000; Cohen et al., 1999]. This limits the overall scalability of the data center in the number of clients it can support simultaneously.

Several techniques have been proposed to improve the performance and scalability of application-level gateways used in various contexts, including as HTTP proxies. TCP connection splicing is one such optimization in which packets are forwarded from one connection to the other at the network layer, avoiding traversal of the transport layer and the user-kernel protection boundary [Maltz and Bhagwat, 1998; Spatscheck et al., 2000; Cohen et al., 1999]. TCP splicing mechanisms are usually implemented as kernel-level modifications to the operating system protocol stack with an interface to allow applications to initiate the splice operation between two connections. Once the TCP splice is completed, data is relayed from one connection to the other without further intervention by the application. Figure 1.4(a) depicts the operation of TCP splicing. With splicing, care must be taken to ensure that TCP header fields such as sequence numbers, checksums, and options, are correctly relayed. TCP splicing has been shown to improve the performance of application-layer gateways to the level of software IP routers. Variations to the kernel-based implementation include implemen-

FIGURE 1.4

With TCP splicing (a), the connection router splices the two connections after determining that server B should handle the request. The response must be sent back through the router. In the TCP handoff approach (b), the router, a simpler layer-4 device, initially forwards the connection request to server A. After receiving the request, server A hands off the connection state to server B. The response traffic can then flow directly back to the client, bypassing the router. Client acknowledgements, however, still come through the router, and must be forwarded to the server B.

tation in the kernel socket library (as opposed to the network layer) [Rosu and Rosu, 2002] and in a hardware switch [Apostolopoulos et al., 2000].

Though TCP splicing can improve the scalability of a layer-7 router it is still limited by the fact that a centralized node must terminate incoming connections, examine application-layer information, and make request-routing decisions before initiating a splice. In addition, all traffic to and from the servers must pass through the router to allow the header translation to occur. To address these limitations, an alternate scheme using connection handoff was proposed [Pai et al., 1998; Aron et al., 2000; Song et al., 2002]. In this approach, each back-end server can function as a router, effectively distributing the content inspection and request-routing operations to multiple nodes. Client connections are initially routed to any one of the servers, perhaps using a fast hardware switch. If the initial server decides that another server is better suited to handle the request, it transfers the TCP connection state to the alternate server. Using the transferred state, the new server can resume the connection with the client without requiring that data pass through a front-end router. Figure 1.4(b) shows the operation of TCP connection handoff.

While the connection handoff approach does remove the bottleneck of connection termination at a single front-end, its scalability and performance are ultimately limited by the number and overhead of TCP handoff operations. Furthermore, it still requires a special front-end layer-4 router, since incoming packets (e.g., TCP acknowledgements) must be forwarded to the appropriate server after the connection is handed off. Finally, TCP handoff requires kernel modifications to the server operating systems to support handoff. The splicing approach, on the other hand, is transparent to both servers and clients.

As Web application requirements evolve, there will be a need for more sophisticated connection routing, based on a variety of application information. This trend implies that the layer-4 approach of examining only transport-layer headers provides insufficient functionality. But layer-7 routers, while more sophisticated, suffer from the limitations on scalability and performance described above.

1.2.1.4 Client-based Load Balancing

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 [Mosedale et al., 1997]. 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

www$i$.netscape.com.

Another client-based technique is to use the client's DNS [Fei et al., 1998; Rabinovich and Spatscheck, 2002]. 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).

1.2.2   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 [Iyengar et al., 2000]. 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 [Market].

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 [Iyengar and Challenger, 1997]. 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 [Challenger et al., 2000]. 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.

1.3   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 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.

1.3.1   Process-Based Servers

Processes are perhaps the most common form of providing concurrency. The original NCSA server and the widely-known Apache server [Project] 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.

1.3.2   Thread-Based Servers

Threads are the next most common form of concurrency. Servers that use threads include JAWS [Hu et al., 1997] and Sun's Java Web Server [Inc., b]. 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 such as a file I/O, the whole address space will be stopped.

1.3.3 Event-Driven Servers

The third form of concurrency is known as the *event-driven* architecture. Servers that use this method include Flash [Pai et al., 1999] and Zeus [Inc., c]. 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 [Corporation, 1999]. 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.

1.3.4 In-Kernel Servers

The fourth and final form of server architectures is the *in-kernel* approach. Servers that use this method include AFPA [Joubert et al., 2001] and Tux [Inc., a]. 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
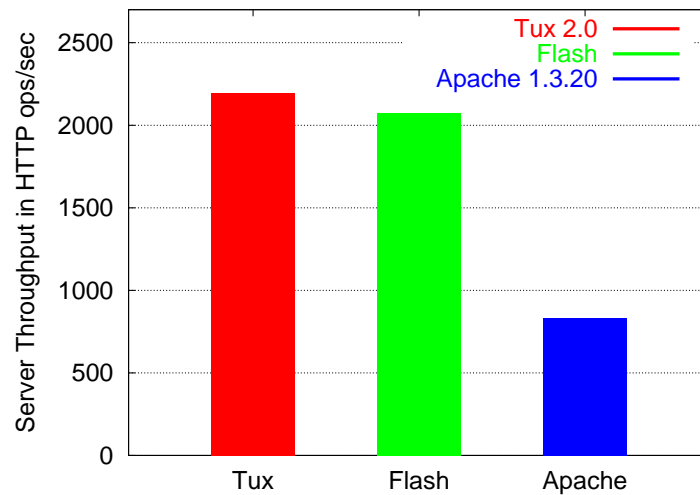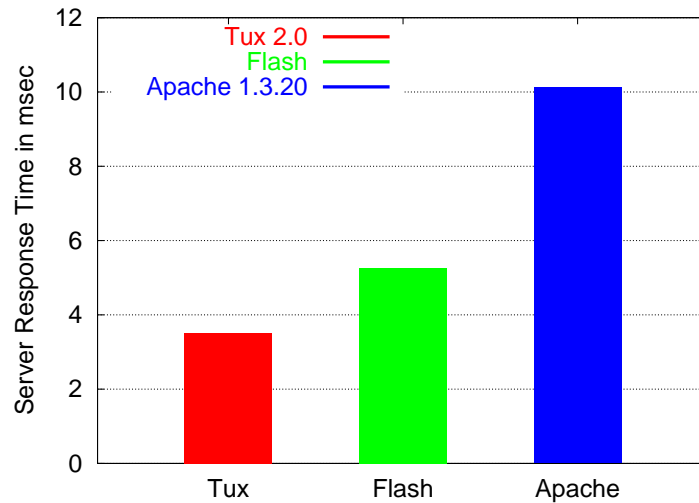
FIGURE 1.5
Server Throughput

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 [Corporation, 1999] 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.

1.3.5 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 a experimental testbed setup and evaluate the performance using a synthetic workload generator [Nahum et al., 2001] 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 of their architecture: Apache as a process-based server, Flash as an event-driven server, and Tux as an in-kernel server.

Figure 1.5 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 1.6 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 content distribution (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.

FIGURE 1.6
Server Response Time

Name:	Chess 1997	Olympics 1998	IBM 1998	World Cup 1998	Dept. Store 2000	IBM 2001
Desc.:	Kasparov-Deep Blue Event Site	Sporting Event Site	Corporate Presence	Sporting Event Site	Online Shopping	Corporate Presence
Period:	2 weeks in May 1997	1 day in Feb. 1998	1 day in June 1998	31 days in June 1998	12 days in June 2000	1 day in Feb. 2001
Hits:	1,586,667	11,485,600	5,800,000	1,111,970,278	13,169,361	12,445,739
KBytes:	14,171,711	54,697,108	10,515,507	3,952,832,722	43,960,527	28,804,852
Clients:	256,382	86,021	80,921	2,240,639	254,215	319,698
URLs:	2,293	15,788	30,465	89,997	11,298	42,874

Logs used in examples

1.4   Web Server Workload Characterization

Workload characterization is frequently necessary in order to better understand the performance of a Web site. Typical workload questions include: What do requests look like? How popular are some documents versus others? How large are Web transfers? What level of HTTP protocol deployment exists on the Web?

   Over time, Web server workload characterization has answered some of those questions, which we provide an overview of here. In this section we describe various characteristics or performance metrics identified in the literature. To help illustrate the characteristics, we also present examples derived from several logs. Table 1.1 gives an overview of the logs used in the examples, several of which are taken from high-volume Web sites that were managed by IBM. One log, taken from an online department store, is taken from a site hosted by but not designed or managed by IBM. We also include most of the 1998 World Cup logs [Arlitt and Jin, 2000], which are publicly available at the Internet Traffic Archive [Laboratory]. Due to the size of these logs, we limit our analysis to the

Request Method	Chess 1997	Olymp. 1998	IBM 1998	W. Cup 1998	Dept. 2000	IBM 2001
GET	92.18	99.37	99.91	99.75	99.42	97.54
HEAD	03.18	00.08	00.07	00.23	00.45	02.09
POST	00.00	00.02	00.01	00.01	00.11	00.22

HTTP Request Methods (percent)

busiest four weeks of the trace, June 10th through July 10th (days 46 through 76 on the web site).

Since our analysis is based on Web logs, certain interesting characteristics cannot be examined. For example, persistent connections, pipelining, network round-trip times and packet loss all have significant effects on both server performance and client-perceived response time. These characteristics are not captured in Apache Common Log format and typically require more detailed packet-level measurements using a tool such as tcpdump. These sorts of network-level measurements are difficult to obtain due to privacy and confidentiality requirements.

An important caveat worth reiterating is that any one Web site may not be representative of a particular application or workload. For example, the behavior of a very dynamic Web site such as eBay, which hosts a great deal of rapidly changing content, is most likely very different from an online trading site like Schwab, which conducts most of its business encrypted using the Secure Sockets Layer (SSL). Several example Web sites given here were all run by IBM, and thus may share certain traits not observed by previous researchers in the literature. As we will see, however, the characteristics from the IBM sites are consistent with those described in the literature.

Dynamic content [Amza et al., 2002; Cecchet et al., 2002; Challenger et al., 2000; Iyengar and Challenger, 1997] is becoming a central component of modern transaction-oriented Web sites. While dynamic content generation is clearly a very important issue, there is currently no consensus as to what constitutes a "representative" dynamic workload, and so we do not present any characteristics of dynamic content here.

1.4.1   Request Methods

The first trait we examine is how frequent different *request methods* appear in server workloads. Several methods were defined in the HTTP 1.0 standard [Berners-Lee et al., 1996] (e.g., HEAD, POST, DELETE), and multiple others were added in the 1.1 specification [Fielding et al., 1997, 1999] (e.g., OPTIONS, TRACE, CONNECT). GET requests are the primary method by which documents are retrieved; the method "means retrieve whatever information ... is identified by the Request-URI" [Berners-Lee et al., 1996]. The HEAD method is similar to the GET method except that only meta-information about the URI is returned. The POST method is a request for the server to accept information from the client, and is typically used for filling out forms and invoking dynamic content generation mechanisms. The literature has shown [Krishnamurthy and Rexford, 2001] that the vast majority of methods are GET requests, with a smaller but noticeable percentage being HEAD or POST methods. Table 1.2 shows the percentage of request methods seen in the various logs. Here, only those methods which appear a non-trivial fraction are shown, in this case defined as greater than one hundredth of a percent. While different logs have slightly varying breakdowns, they are consistent with the findings in the literature.

1.4.2   Response Codes

The next characteristic we study are the *response codes* generated by the server. Again, the HTTP specifications define a large number of responses, the generation of which depends on multiple factors such as whether or not a client is allowed access to a URL, whether or not the request is

| Response Code	Chess 1997	Olymp. 1998	IBM 1998	W.Cup 1998	Dept. 2000	IBM 2001
200 OK	85.32	76.02	75.28	79.46	86.80	67.73
206 Partial Content	00.00	00.00	00.00	00.06	00.00	00.00
302 Found	00.05	00.05	01.18	00.56	00.56	15.11
304 Not Modified	13.73	23.25	22.84	19.75	12.40	16.26
403 Forbidden	00.01	00.02	00.01	00.00	00.02	00.01
404 Not Found	00.55	00.64	00.65	00.70	00.18	00.79

Server Response Codes (percent)
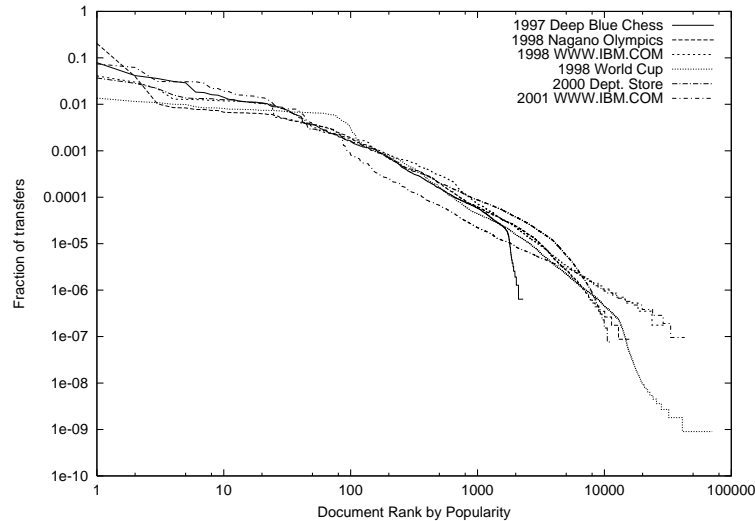
FIGURE 1.7
Document Popularity

properly formed, etc. However, certain responses are much more frequent than others.

Table 1.3 shows those responses seen in the logs that occur with a non-trivial frequency, again defined as greater than one hundredth of a percent. We see that the majority of the responses are successful transfers, i.e., the 200 OK response code.

Perhaps the most interesting aspect of this data is, however, the relatively large fraction of 304 Not Modified responses. This code is typically generated in response to a client generating a GET request with the If-Modified-Since option, which provides the client's notion of the URL's last-modified time. This request is essentially a cache-validation option and asks the server to respond with the full document if the client's copy is out of date. Otherwise, the server should respond with the 304 code if the copy is OK. As can be seen, between 12 and 23 percent of responses are 304 codes, indicating that clients re-validating up-to-date content is a relatively frequent occurrence, albeit in different proportions at different Web sites.

Other responses, such as 403 Forbidden or 404 Not Found, are not very frequent, on the order of a tenth of a percent, but appear occasionally. The IBM 2001 log is unusual in that roughly 15 percent of the responses use the 302 Found code, which is typically used as a temporary redirection facility.
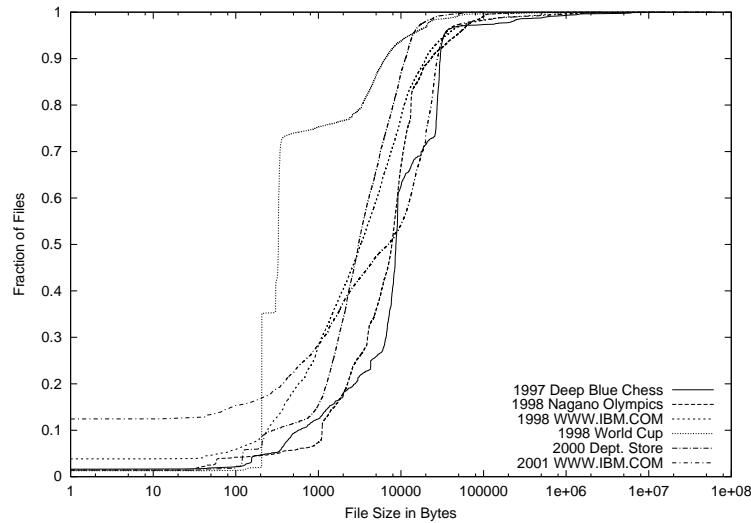
FIGURE 1.8
Document Size (CDF)

1.4.3   Object Popularity

Numerous researchers [Almeida et al., 1996; Arlitt and Williamson, 1997; Crovella and Bestavros, 1997; Padmanabhan and Qui, 2000] have observed that, in origin Web servers, the relative probability with which a web page is accessed follows a Zipf-like distribution. That is,

$$p(r) \approx C/r^\alpha$$

where $p(r)$ is the probability of a request for a document with rank r, and C is a constant (depending on α and the number of documents) that ensures that the sum of the probabilities is one. Rank is defined by popularity; the most popular document has rank 1, the second-most popular has rank 2, etc. When α equals 1, the distribution is a true Zipf; when α is another value the distribution is considered "Zipf-like." Server logs tend to have α values of one or greater; proxy server logs have lower values ranging from 0.64 to 0.83 [Breslau et al., 1999].

Figure 1.7 shows the fraction of references based on document rank generated from the sample Web logs. Note that both the X and Y axes use a log scale. As can be seen, all the curves follow a Zipf-like distribution fairly closely, except towards the upper left of the graph and the lower right of the graph.

This Zipf property of document popularity is significant because it shows the effectiveness of document caching. For example, one can see that by simply caching the 100 most popular documents, assuming these documents are all cacheable, the vast majority of requests will find the document in the cache, avoiding an expensive disk I/O operation.

1.4.4   File Sizes

The next characteristic we examine is the range of sizes of the URLs stored on a Web server. File sizes give a picture of how much storage is required on a server, as well as how much RAM might be needed to fully cache the data in memory. Which distribution best captures file size characteristics has been a topic of some controversy. There is consistent agreement that sizes range over multiple orders of magnitude and that the body of the distribution (i.e., that excluding the tail) is Log-Normal.

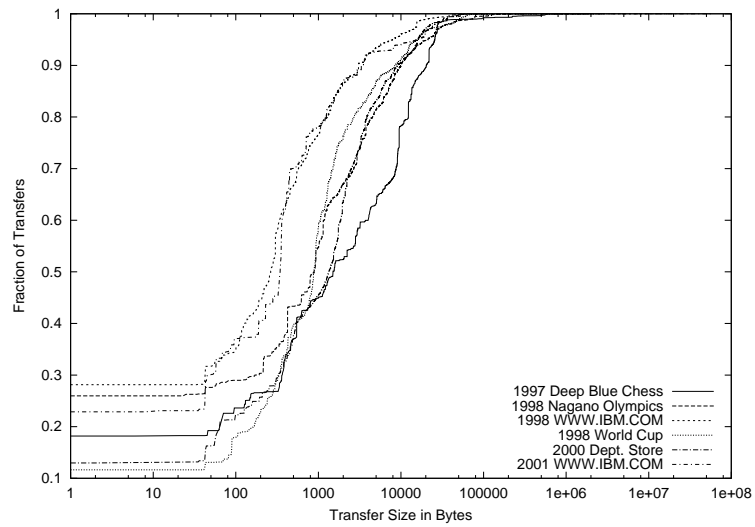| Statistic	Chess 1997	Olympics 1998	IBM 1998	W. Cup 1998	Dept. 2000	IBM 2001
Min	1	23	1	1	2	1
Median	8,697	7,757	3,244	328	3,061	7,049
Mean	45,012	12,851	20,114	6,028	4,983	29,662
Max	8,723,751	2,646,058	17,303,027	64,219,310	99,900	61,459,221
Std Dev	384,175	44,618	193,892	253,481	6,115	394,088

File Size Statistics (bytes)
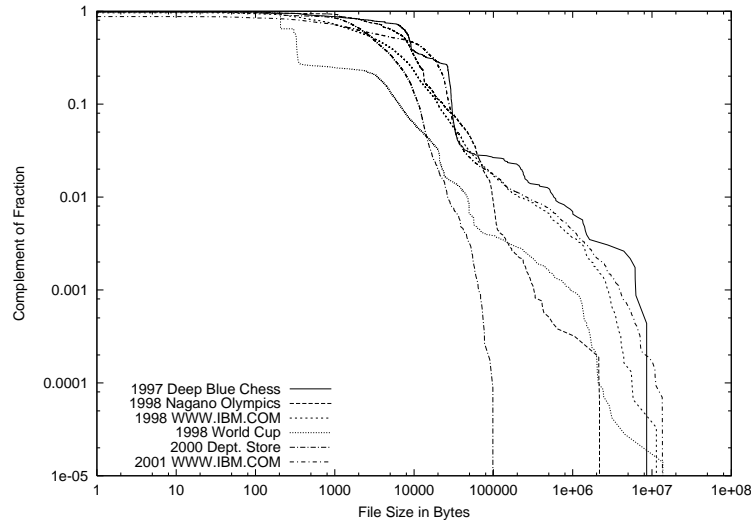
FIGURE 1.9
Document Size (CCDF)

However, the shape of the tail of the distribution has been debated, with claims that it is Pareto [Crovella and Bestavros, 1997], Log-Normal [Downey, 2001], and even that the amount of data available is insufficient to statistically distinguish the the two [Gong et al., 2001]. Figure 1.8 shows the CDF of file sizes seen in the logs. Note that the X axis is in log scale. Table 1.4 presents the minimum, maximum, median, mean, and standard deviation of the file sizes. As can be seen, sizes range from a single byte to over 64 megabytes, varying across several orders of magnitude. In addition, the distributions show the rough 'S' shape of the Log-Normal distribution.

As mentioned earlier, a metric of frequent interest in the research community is the "tail" of the distribution. While the vast majority of files are small, the majority of bytes transferred are found in large files. This is sometimes known as the "Elephants and Mice" phenomenon. To illustrate this property, we graphed the complement of the cumulative distribution function, or CCDF, of the logs. These are shown in Figure 1.9. The Y values for this graph are essentially the complement of the corresponding Y values from Figure 1.8. Unlike Figure 1.8, however, note here that the Y-axis uses a log scale to better illustrate the tail. We observe that all the logs have maximum files in the range of 1 to 10 MB, with the exception of the Department Store log, which has no file size greater than 99990 bytes.

FIGURE 1.10
Transfer Size (CDF)

Statistic	Chess 1997	Olympics 1998	IBM 1998	W. Cup 1998	Dept. 2000	IBM 2001
Median	1506	886	265	889	1339	344
Mean	10847	4851	1856	4008	3418	2370
Std Dev	100185	31144	29134	32945	6576	35986

Transfer Size Statistics (bytes)
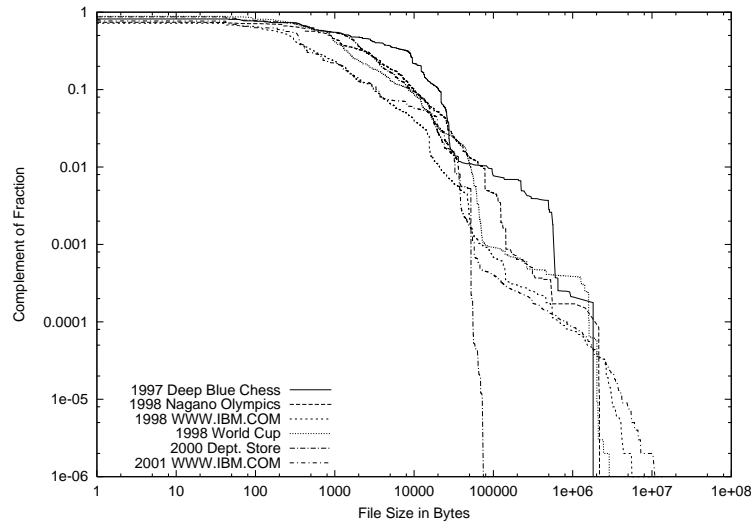
FIGURE 1.11
Transfer Size (CCDF)

1.4.5   Transfer Sizes

A metric related to Web file sizes is Web *transfer sizes*, or the size of the objects sent "over the wire." Transfer sizes are significant since they connote how much bandwidth is used by the Web server to respond to clients. In Apache Common Log Format, transfer size is based on the amount of *content* sent, and does not include the size from any HTTP headers or lower-layer bytes such as TCP or IP headers. Thus, here transfer size is based on the size of the content transmitted. The distribution of transfer sizes is thus influenced by the popularity of documents requested, as well as by the proportion of unusual responses such as 304 Not Modified and 404 Not Found.

Figure 1.10 shows the CDF of the object transfers from the logs. Table 1.5 presents the median, mean, and standard deviation of the transfer sizes. As can be seen, transfers tend to be small; for example, the median transfer size from the IBM 2001 log is only 344 bytes! An important trend is to note that a large fraction of transfers are for *zero* bytes, as much as 28 percent in the 1998 IBM log. The vast majority of these zero-byte transfers are the 304 Not Modified responses noted above in Section 1.4.2. When a conditional GET request with the If-Modified-Since option is successful, a 304 response is generated and no content is transferred. Other return codes, such as 403 Forbidden and 404 Not Found, also result in zero-byte transfers, but they are significantly less common. The exception is the IBM 2001 log, where roughly 15 percent of the 302 Found responses contribute to the fraction of zero-byte transfers.

Figure 1.11 shows the CCDF of the transfer sizes, in order to illustrate the "tail" of the distributions. Note again that the Y axis uses a log scale. The graph looks similar to Figure 1.9, perhaps since these transfers are so uncommon that weighting them by frequency does not change the shape of the graph, as it does with the bulk of the distribution in Figure 1.10.

1.4.6   HTTP Version

Another question we are interested in is what sort of HTTP protocol support is being used by servers. While HTTP 1.1 was first standardized in 1997 [Fielding et al., 1997], the protocol has undergone some updating [Fielding et al., 1999; Krishnamurthy and Rexford, 2001] and in some

| Protocol | Chess | Olymp. | IBM | W. Cup | Dept. | IBM |
Version	1997	1998	1998	1998	2000	2001
HTTP 1.0	95.30	78.56	77.22	78.62	51.13	51.08
HTTP 1.1	00.00	20.92	18.43	21.35	48.82	48.30
Unclear	04.70	00.05	04.34	00.02	00.05	00.06

HTTP Protocol Versions (percent)

ways is still being clarified [Krishnamurthy et al., 1999; Mogul, 2002]. The transition from 1.0 to 1.1 is a complex one, requiring support from browsers, servers, and any proxy intermediaries as well.

Table 1.6 shows the HTTP protocol version that the server used in responding to requests. A clear trend is that over time, more requests are being serviced using 1.1. In the most recent logs, from 2000 and 2001, HTTP 1.1 is used in just under half the responses.

Given the depth and complexity of the HTTP 1.1 protocol, the numbers above only scratch the surface of how servers utilize HTTP 1.1. Many features have been added in 1.1, including new mechanisms, headers, methods, and response codes. How these features are used in practice is still an open issue, and as mentioned earlier, server logs are insufficient to fully understand HTTP 1.1 behavior.

1.4.7   Summary

This work has presented some of the significant performance characteristics observed in Web server traffic. An interesting observation is that many of them have not fundamentally changed over time. Some, such as HTTP 1.1 support, have changed as we would expect, albeit more slowly than we might anticipate. Still, certain characteristics seem to be invariants that hold both across time and across different Web sites.

Acknowledgement

References

Alteon ACEdirector. `http://www.nortelnetworks.com/products/01/acedir`.

BIG-IP controller. `http://www.f5.com/f5products/bigip/`.

Cisco CSS 1100. `http://www.cisco.com/warp/public/cc/pd/si/11000/`, a.

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

Foundry ServerIron. `http://www.foundrynet.com/products/webswitches/serveriron`.

Intel NetStructure 7175 traffic director. `http://www.intel.com/network/idc/products/director_7175.htm`.

Websphere edge server. `http://www.ibm.com/software/webservers/edgeserver/`.

Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana 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.

Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Alan Cox, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Bottleneck characterization of dynamic Web site benchmarks. Technical Report TR02-388, Rice University Computer Science Department, February 2002.

George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, and Debanjan Saha. Design, implementation and performance of a content-based switch. In *Proceedings of IEEE INFOCOM*, March 2000.

Martin F. Arlitt and Tai Jin. Workload characterization of the 1998 World Cup Web site. *IEEE Network*, 14(3):30–37, May/June 2000.

Martin F. Arlitt and Carey L. Williamson. Internet Web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–646, Oct 1997.

Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of USENIX Annual Technical Conference*, San Diego, CA, June 2000.

Micah Beck and Terry Moore. The Internet2 distributed storage infrastructure project: An architecture for Internet content channels. In *Proceedings of the 3rd International Web Caching Workshop*, 1998.

Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext transfer protocol – HTTP/1.0. In *IETF RFC 1945*, May 1996.

Azer Bestavros, Mark Crovella, Jun Liu, and David Martin. Distributed packet rewriting and its application to scalable server architectures. In *Proceedings of IEEE International Conference on*

Network Protocols, Austin, TX, October 1998.

Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, New York, NY, March 1999.

Thomas P. Brisco. DNS support for load balancing. Number IETF RFC 1974, April 1995.

Valeria Cardellini, Michele Colajanni, and Philip S. Yu. DNS dispatching algorithms with state estimators for scalable Web server clusters. *World Wide Web*, 2(2), July 1999a.

Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic load balancing on Web-server systems. *IEEE Internet Computing*, pages 28–39, May/June 1999b.

Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. A comparison of software architectures for e-business applications. Technical Report TR02-389, Rice University Computer Science Department, February 2002.

Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic Web content. In *Proceedings of IEEE INFOCOM*, March 2000.

Ariel Cohen, Sampath Ragarajan, and Hamilton Slye. On the performance of TCP splicing for URL-aware redirection. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, October 1999.

The Standard Performance Evaluation Corporation. SPECWeb99. `http://www.spec.org/osg/web99`, 1999.

Mark Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, Nov 1997.

Dan Dias, William Kish, Rajat Mukherjee, and Renu Tewari. A scalable and highly available Web server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.

Allen 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.

Zongming Fei, Samrat Bhattacharjee, Ellen Zegura, and Mustapha Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceedings of IEEE INFOCOM*, 1998.

Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. In *IETF RFC 2068*, January 1997.

Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. In *IETF RFC 2616*, June 1999.

Weibo Gong, Yong Liu, Vishal Misra, and Don Towsley. On the tails of Web file size distributions. In *Proceedings of the 39th Allerton Conference on Communication, Control, and Computing*, Monticello, Illinois, Oct 2001.

James C. Hu, Irfan Pyarali, and Douglas 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.

Guerney Hunt, German Goldszmidt, Richard King, and Rajat Mukherjee. Network dispatcher: A connection router for scalable Internet services. In *Proceedings of the 7th International World Wide Web Conference*, April 1998.

Red Hat Inc. The Tux WWW server. `http://people.redhat.com/˜mingo/ TUX-patches/`, a.

Sun Microsystems Inc. The Java Web server. `http://wwws.sun.com/software/ jwebserver/index.html`, b.

Zeus Inc. The Zeus WWW server. `http://www.zeus.co.uk`, c.

Arun Iyengar and Jim Challenger. Improving Web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

Arun Iyengar, Jim Challenger, Daniel Dias, and Paul Dantzig. High-performance Web site design techniques. *IEEE Internet Computing*, 4(2), March/April 2000.

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.

Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. In *Proceedings of WWW-8 Conference*, Toronto, Canada, May 1999.

Balachander Krishnamurthy and Jennifer Rexford. *Web Protocols and Practice*. Addison Wesley, 2001.

Thomas T. Kwan, Robert E. McGrath, and Daniel A. Reed. NCSA's World Wide Web server: Design and performance. *IEEE Computer*, 28(11):68–74, November 1995.

Lawrence Berkeley National Laboratory. The Internet traffic archive. `http://ita.ee.lbl. gov/`.

David Maltz and Pravin Bhagwat. TCP splicing for application layer proxy performance. Technical Report RC 21139, IBM TJ Watson Research Center, 1998.

Open Market. FastCGI. http://www.fastcgi.com/.

Jeffrey C. Mogul. Clarifying the fundamentals of HTTP. In *Proceedings of WWW 2002 Conference*, Honolulu, HA, May 2002.

Dan Mosedale, William Foss, and Rob McCool. Lessons learned administering Netscape's Internet site. *IEEE Internet Computing*, 1(2):28–35, March/April 1997.

Erich M. Nahum, Marcel Rosu, Srinivasan Seshan, and Jussara 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.

Venkata N. Padmanabhan and Lili Qui. The content and access dynamics of a busy Web site: Findings and implications. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 111–123, 2000. URL `citeseer.nj.nec.com/ padmanabhan00content.html`.

Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, Monterey, CA, June 1999.

Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998. URL `citeseer.nj.nec.com/article/pai98localityaware.html`.

The Apache Project. The Apache WWW server. `http://httpd.apache.org`.

Michael Rabinovich and Oliver Spatscheck. *Web Caching and Replication*. Addison-Wesley, 2002.

Marcel-Catalin Rosu and Daniela Rosu. An evaluation of TCP splice benefits in Web proxy servers. In *Proceedings of the 11th International World Wide Web Conference (WWW2002)*, Honolulu, Hawaii, May 2002.

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

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

Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 8(2):146–157, April 2000.