# High-Performance Web Site Design Techniques

ARUN IYENGAR, JIM CHALLENGER, DANIEL DIAS, AND PAUL DANTZIG
*IBM T.J. Watson Research Center*

Performance and high availability are critical at Web sites that receive large numbers of requests. This article presents several techniques—including redundant hardware, load balancing, Web server acceleration, and efficient management of dynamic data—that can be used at popular sites to improve performance and availability. We describe how we deployed several of these techniques at the official Web site for the 1998 Olympic Winter Games in Nagano, Japan, which was one of the most popular sites up to its time. In fact, the *Guinness Book of World Records* recognized the site on 14 July 1998 for setting two records:

- "Most Popular Internet Event Ever Recorded," based on the officially audited figure of 634.7 million requests over the 16 days of the Olympic Games; and
- "Most Hits on an Internet Site in One Minute," based on the officially audited figure of 110,414 hits received in a single minute around the time of the women's freestyle figure skating.

The site visibility, large number of requests, and amount of data made performance and high availability critical design issues. The site's architecture was an outgrowth of our experience designing and implementing the 1996 Olympic Summer Games Web site. In this article, we first describe general techniques that can be used to improve performance and availability at popular Web sites. Then we describe how several of these were deployed at the official Web site in Nagano.

This article presents techniques for designing Web sites that need to handle large request volumes and provide high availability. The authors present new techniques they developed for keeping cached dynamic data current and synchronizing caches with underlying databases. Many of these techniques were deployed at the official Web site for the 1998 Olympic Winter Games.

## ROUTING TO MULTIPLE WEB SERVERS

To handle heavy traffic loads, Web sites must use multiple servers running on different computers. The servers can share information through a shared file system, such as Andrew File System (AFS) or Distributed File System (DFS), or via a shared database; otherwise, data can be replicated across the servers.

### RR-DNS

The Round-Robin Domain Name Server (RR-DNS)[1,2] approach, which NCSA used for its server,[3] is one method for distributing requests to multiple servers. RR-DNS allows a single domain name to be associated with multiple IP addresses, each of which could represent a different Web server. Client requests specifying the domain name are mapped to servers in a round-robin fashion. Several problems arise with RR-DNS, however.

**Server-side caching.** Caching name-to-IP address mappings at name servers can cause load imbalances. Typically, several name servers cache the resolved name-to-IP-address mapping between clients and the RR-DNS. To force a mapping to different server IP addresses, RR-DNS can specify a time-to-live (TTL) for a resolved name, such that requests made after the specified TTL are not resolved in the local name server. Instead, they are forwarded to the authoritative RR-DNS to be remapped to a different HTTP server's IP address. Multiple name requests made during the TTL period would be mapped to the same HTTP server.

Because a small TTL can significantly increase network traffic for name resolution, name servers will often ignore a very small TTL given by the RR-DNS and impose their own minimum TTL instead. Thus, there is no way to prevent intermediate name servers from caching the resolved name-to-IP address mapping—even by using small TTLs. Many clients, such as those served by the same Internet service provider, may share a name server, and may therefore be pointed to a single specific Web server.

**Client-side caching.** Client-side caching of resolved name-to-IP address mappings can also cause load imbalances. The load on the HTTP servers cannot be controlled, but rather will vary with client access patterns. Furthermore, clients make requests in bursts as each Web page typically involves fetching several objects including text and images; each burst is directed to a single server node, which increases the skew. These effects can lead to significant imbalances that may require the cluster to operate at lower mean loads in order to handle peak loads (see Dias et al.[4]).

Another problem with RR-DNS is that the round-robin method is often too simplistic for providing good load balancing. We must consider factors such as the load on individual servers. For instance, requests for dynamic data constructed from many database accesses at a server node can overload a particular Web server.

**Node failures.** Finally, client and name server caching of resolved name-to-IP-address mappings make it difficult to provide high availability in the face of Web server node failures. Since clients and name servers are unaware of the failures, they may continue to make requests to failed Web servers. Similarly, online maintenance may require bringing down a specific Web server node in a cluster. Again, giving out individual node IP addresses to the client and name servers makes this difficult. While we can configure backup servers and perform IP address takeovers for detected Web server node failures, or for maintenance, it is hard to manage these tasks. Moreover, an active backup node may end up with twice the load if a primary node fails.

## TCP Routing

Figure 1 illustrates a load-balancing method based on routing at the TCP level (rather than standard IP-level routing). A node of the cluster serves as a so-called *TCP router*, forwarding client requests to the Web server nodes in the cluster in a round-robin (or other) order. The router's name and IP address are public, while the addresses of the other nodes in the cluster are hidden from clients. (If there is more than one TCP router node, RR-DNS maps a single name to the multiple TCP routers.) The client sends requests to the TCP router node, which forwards all packets belonging to a particular TCP connection to one of the server nodes. The TCP router can use different load-based algorithms to select which node to route to, or it can use a simple round-robin scheme, which often is less effective than load-based algorithms. The server nodes bypass the TCP router and respond directly to the client. Note that because the response packets are larger than the request packets, the TCP router adds only small overhead.

The TCP router scheme allows for better load balancing than DNS-based solutions and avoids

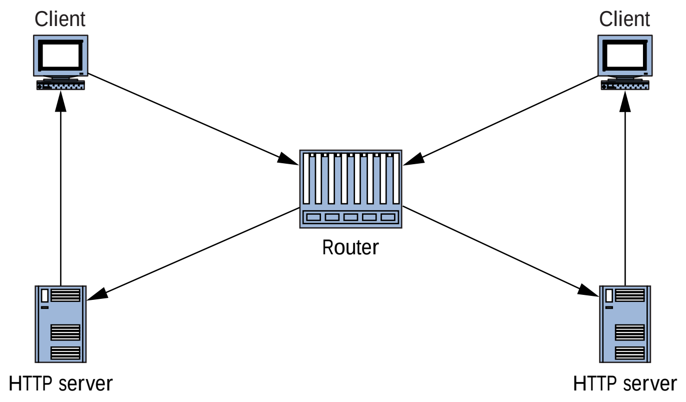

**Figure 1. The TCP router routes requests to various Web servers. Server responses go directly to clients, bypassing the router.**

the problem of client or name server caching.[4] The routers can use sophisticated load-balancing algorithms that take individual server loads into account. A TCP router can also detect Web server node failures and route user requests to only the available Web server nodes. The system administrator can change the TCP router configuration to remove or add Web server nodes, which facilitates Web server cluster maintenance. By configuring a backup TCP router, we can handle failure of the TCP router node, and the backup router can operate as a Web server during normal operation. On detecting the primary TCP router's failure, the backup would route client requests to the remaining Web server nodes, possibly excluding itself.

**Commercially available TCP routers.** Many TCP routers are available commercially. For example, IBM's Network Dispatcher (ND) runs on stock hardware under several operating systems (OS), including Unix, Sun Solaris, Windows NT, and a proprietary embedded OS.[5]

An embedded OS improves router performance by optimizing the TCP communications stack and eliminating the scheduler and interrupt processing overheads of a general-purpose OS. The ND can route up to 10,000 HTTP requests per second when running under an embedded OS on a uniprocessor machine, which is a higher request rate than most Web sites receive.

Other commercially available TCP routers include Radware's Web Server Director (http://www.radware.co.il/) and Resonate's Central Dispatch (http://www.resonate.com/products/central_dispatch/data_sheets.html). Cisco Systems' LocalDirector (http://www.cisco.com/warp/public/cc/cisco/mkt/scale/locald/) differs from the TCP router approach because packets returned from servers go through the LocalDirector before being returned to clients. For a comparison of different load-balancing approaches, see Cardellini, Colajanni, and Yu.[6]

**Combining TCP routing and RR-DNS.** If a single TCP router has insufficient capacity to route requests without becoming a bottleneck, the TCP-router and DNS schemes can be combined in various ways. For example, the RR-DNS method can be used to map an IP address to multiple router nodes. This hybrid scheme can tolerate the load imbalance produced by RR-DNS because the corresponding router will route any burst of requests mapped by the RR-DNS to different server nodes. It achieves good scalability because a long TTL can ensure that the node running the RR-DNS does not become a bottleneck, and several router nodes can be used together.

**An embedded OS improves router performance by optimizing the TCP communications stack.**

**Special requirements.** One issue with TCP routing is that it spreads requests from a single client across Web server nodes. While this provides good load balancing, some applications need their requests routed to specific servers. To support such applications, ND allows requests to be routed with an affinity toward specific servers. This is useful for requests encrypted using Secure Sockets Layer (SSL), which generates a session key to encrypt information passed between a client and server. Session keys are expensive to generate. To avoid regenerating one for every SSL request, they typically have a lifetime of about 100 seconds. After a client and server have established a session key, all requests between the specific client and server within that lifetime use the same session key.

In a system with multiple Web servers, however, one server will not know about session keys generated by another. If a simple load-balancing scheme like round-robin is used, it is highly probable that multiple SSL requests from the same client will be sent to different servers within the session key's lifetime—resulting in unnecessary generation of additional keys. ND avoids this problem by routing two SSL requests received from the same client within 100 seconds of each other to a single server.

## WEB SERVER ACCELERATORS

In satisfying a request, a Web server often copies data several times across layers of software. For example, it may copy from the file system to the application, again to the operating system kernel during transmission, and perhaps again at the device-driver level. Other overheads, such as OS scheduler and interrupt processing, can add further inefficiencies. One technique for improving Web site performance is to cache data at the site so that frequently requested pages can be served from a cache with significantly less overhead than a Web server. Such caches are known as

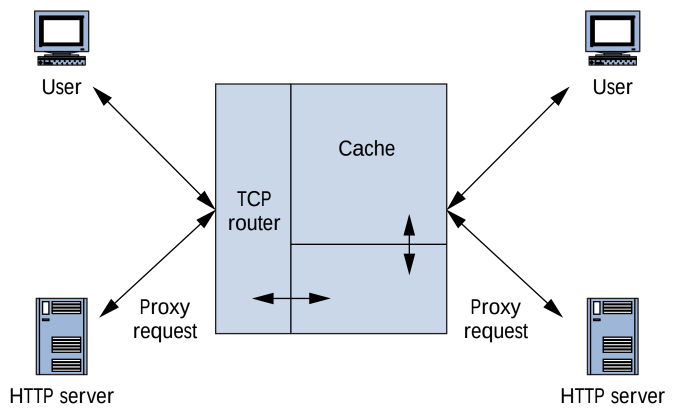

**Figure 2. Web server acceleration. The cache significantly reduces server load.**

HTTPD accelerators or Web server accelerators.[7]

## Our Accelerator

We have developed a Web server accelerator that runs under an embedded operating system and can serve up to 5,000 pages per second from its cache on a uniprocessor 200-MHz PowerPC.[8] This throughput is up to an order of magnitude higher than typically attainable by a high-performance Web server running on similar hardware under a conventional OS. Both the Harvest and Squid (http://squid.nlanr.net/Squid/) caches include HTTP daemon (HTTPD) accelerators, but ours performs considerably better than either.[8] Novell also sells an HTTPD accelerator as part of its BorderManager.[9]

Our system's superior performance results largely from the embedded operating system. Buffer copying is kept to a minimum. With its limited functionality, the OS is unsuitable for general-purpose software applications, such as databases or online transaction processing. Its optimized support for communications, however, makes it well suited to specialized network applications like Web server acceleration.

A key difference between our accelerator and others is that our API allows application developers to explicitly add, delete, and update cached data, which helps maximize hit rates and maintain current caches. We allow caching of dynamic as well as static Web pages because applications can explicitly invalidate any page whenever it becomes obsolete. Dynamic Web page caching is important for improving performance at Web sites with significant dynamic content. We are not aware of other accelerators that allow caching of dynamic pages.

As Figure 2 illustrates, the accelerator front-ends a set of Web server nodes. A TCP router runs on the same node as the accelerator (although it could also run on a separate node). If the requested page is in the cache, the accelerator returns the page to the client. Otherwise, the TCP router selects a Web server node and sends the request to it. Persistent TCP connections can be maintained between the cache and the Web server nodes, considerably reducing the overhead of satisfying cache misses from server nodes. Our accelerator can reduce the number of Web servers needed at a site by handling a large fraction of requests from the cache (93 percent at the Wimbledon site, for example).

As the accelerator examines each request to see if it can be satisfied from cache, it must terminate the connection with the client. For cache misses, the accelerator requests the information requested by the client from a server and returns it to the client. In the event of a miss, caching thus introduces some overhead as compared to the TCP router in Figure 1 because the accelerator must function as a proxy for the client. By keeping persistent TCP connections between the cache and the server, however, this overhead is significantly reduced. In fact, the overhead on the server is lower than it would be if the server handled client requests directly.

The cache operates in one of two modes: transparent or dynamic. In transparent mode, data is cached automatically after cache misses. The webmaster can also set cache policy parameters to determine which URLs get automatically cached. For example, different parameters determine whether to cache static image files, static nonimage files, and dynamic pages, as well as their default lifetimes. HTTP headers included in a server response can then override a specific URL's default behavior as set by the cache policy parameters.

In dynamic mode, the cache contents are explicitly controlled by applications executed on either the accelerator or a remote node. API functions allow programs to cache, invalidate, query, and specify lifetimes for URL contents. While dynamic mode complicates the application programmer's task, it is often required for optimal performance. Dynamic mode is particularly useful for prefetching popular objects into caches and for invalidating objects whose lifetimes were not known at the time they were cached.

## Performance

Because caching objects on disk would slow the accelerator too much, all cached data is stored in memory.

Thus, cache sizes are limited by memory sizes. Our accelerator uses the least recently used (LRU) algorithm for cache replacement. Figure 3 shows the throughput that a single accelerator can handle as a function of requested object size. Our test system had insufficient network bandwidth to drive the accelerator to maximum throughput for requested objects larger than 2,048 bytes. The graph shows the numbers we actually measured as well as the projected numbers on a system where network throughput was not a bottleneck. A detailed analysis of our accelerator's performance and how we obtained the projections in Figure 3 is available in Levy et al.[8]

Our accelerator can handle up to about 5,000 cache hits per second and 2,000 cache misses per second for an 8-Kbyte page when persistent connections are not maintained between the accelerator and back-end servers. In the event of a cache miss, the accelerator must request the information from a back-end server before returning it to the client. Requesting the information from a server requires considerably more instructions than fetching the object from cache. Cache miss performance can therefore be improved by maintaining persistent connections between the accelerator and back-end servers.

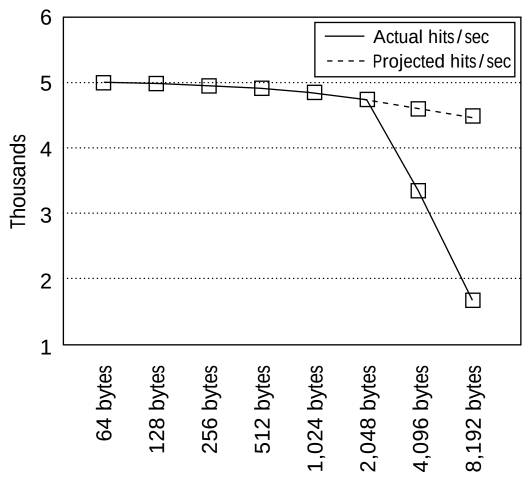

**Figure 3. Our accelerator achieved a maximum of 5,000 cache hits per second. Network bandwidth limitations in our test configuration restricted the accelerator's maximum throughput for page sizes larger than 2,048 bytes.**

## EFFICIENT DYNAMIC DATA SERVING

High-performance uniprocessor Web servers can typically deliver several hundred static files per second. Dynamic pages, by contrast, are often delivered at rates orders of magnitude slower. It is not uncommon for a program to consume over a second of CPU time to generate a single dynamic page. For Web sites with a high proportion of dynamic pages, the performance bottleneck is often the CPU overhead associated with generating them.

Dynamic pages are essential at sites that provide frequently changing data. For example, the official 1998 U.S. Open tennis tournament Web site averaged up to four updated pages per second over an extended time period. A server program that generates pages dynamically can return the most recent version of the data, but if the data is stored in files and served from a file system, it may not be feasible to keep it current. This is particularly true when numerous files need frequent updating.

### Cache Management Using DUP

One of the most important techniques for improving performance with dynamic data is caching pages the first time they are created. Subsequent requests for an existing dynamic page can access it from cache rather than repeatedly invoking a program to generate the same page. A key problem with this technique is determining which pages to cache and when they become obsolete. Explicit cache management by invoking API functions is again essential for optimizing performance and ensuring consistency.

We have developed a new algorithm called Data Update Propagation (DUP) for precisely identifying which cached pages have been obsoleted by new information. DUP determines how cached Web pages are affected by changes to underlying data. If a set of cached pages is constructed from tables belonging to a database, for example, the cache must be synchronized with the database so that pages do not contain stale data. Furthermore, cached pages should be associated with parts of the database as precisely as possible; otherwise, objects whose values are unchanged can be mistakenly invalidated or updated after a database change. Such unnecessary cache updates can increase miss rates and hurt performance.

DUP maintains correspondences between *objects*—defined as items that might be cached—and *underlying data*, which periodically change and affect object values. A program known as a *trigger monitor* maintains data dependence information between the objects and underlying data and determines when data have changed. When the system becomes aware of a change, it queries the stored dependence information to determine which

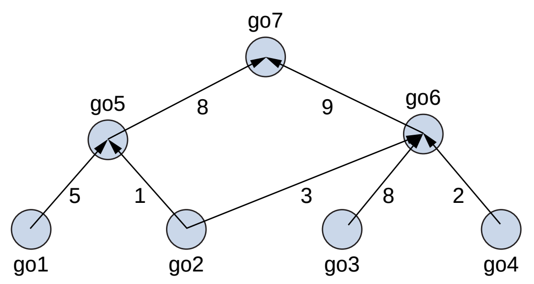

**Figure 4. Object dependence graph (ODG). Weights are correlated with the importance of data dependencies.**

cached objects are affected and should be invalidated or updated.

Dependencies are represented by a directed graph known as an object dependence graph (ODG), wherein a vertex usually represents an object or underlying data. An edge from a vertex *v* to another vertex *u* indicates that a change to *v* also affects *u*. The trigger monitor constructed the ODG in Figure 4 from its knowledge of the application. If node *go*2 changed, for example, the trigger monitor would detect it. The system uses graph traversal algorithms to determine which objects are affected by the change to *go*2, which in this case include *go*5, *go*6, and *go*7. The system can then invalidate or update cached objects it determines to be obsolete.

Weights can be associated with edges to help determine how obsolete underlying data changes have rendered an object. In Figure 4, the data dependence from *go*1 to *go*5 is more important than the dependence from *go*2 to *go*5 because the former edge is five times the weight of the latter. Therefore, a change to *go*1 would generally affect *go*5 more than a change to *go*2. We have described the DUP algorithm in detail in earlier work.[10]

### Interfaces for Creating Dynamic Data

The interface for invoking server programs that create dynamic pages has a significant effect on performance. The Common Gateway Interface (CGI) works by creating a new process to handle each request, which incurs considerable overhead. Once the most widely used interface, CGI is being replaced with better performing mechanisms. For instance, Open Market's FastCGI (http://www.fastcgi.com/) establishes long-running processes to which a Web server passes requests. This avoids the overhead of process creation but still requires some communication overhead between the Web server and the FastCGI process. FastCGI also imposes some limits on the number of simultaneous connections a server can handle by the number of physical processes that a machine can handle.

Rather than forking off new processes each time a server program is invoked, or communicating with prespawned processes, Web servers such as Apache, the IBM Go server, and those by Netscape and Microsoft provide interfaces to invoke server extensions as part of the Web server process itself. Server tasks are run in separate threads within the Web server and may be either dynamically loaded or statically bound into the server. IBM's Go Web server API (GWAPI), Netscape's server application programming interface (NSAPI), and Microsoft's InternetServer API (ISAPI), as well as Apache's low-level "modules," are all examples of this approach. Also, careful thread management makes it possible to handle many more connections than there are servicing threads, reducing overall resource consumption as well as increasing capacity. Unfortunately, the interfaces presented by GWAPI, NSAPI, ISAPI, and Apache modules can be rather tricky to use in practice, with issues such as portability, thread safety, and memory management complicating the development process.

More recent approaches, such as IBM's Java Server Pages (JSP), Microsoft's Active Server Pages (ASP), as well as Java Servlets, and Apache's mod_perl hide those interfaces to simplify the Web author's job through the use of Java, Visual Basic, and Perl. These services also hide many of the issues of thread safety and provide built-in garbage collection to relieve the programmer of the problems of memory management. Although execution can be slightly slower than extensions written directly to the native interfaces, ease of program creation, maintenance and portability join with increased application reliability to more than make up for the slight performance difference.

## 1998 OLYMPIC GAMES SITE

The 1998 Winter Games Web site's architecture was an outgrowth of our experience with the 1996 Olympic Summer Games Web site. The server logs we collected in 1996 provided significant insight that influenced the design of the 1998 site. We determined that most users had spent too much time looking for basic information, such as medal standings, most recent results, and current news stories. Clients had to make at least three Web server requests to navigate to a result page. Browsing patterns were similar for the news, photos, and sports sections of the site. Furthermore, when a client reached a leaf page, there were no direct links to pertinent information in other sections. Given this hierarchy, intermediate navigation pages were among the most frequently accessed.

Hit reduction was a key objective for the 1998 site because we estimated that using the design from the 1996 Web site in conjunction with the additional content provided by the 1998 site could result in more than 200 million hits per day. We thus redesigned the pages to allow clients to access relevant information while examining fewer Web pages. A detailed description of the 1998 Olympic Games Web site is available in our earlier work.[11]

The most significant changes were the generation of a new home page for each day and the addition of a top navigation level that allowed clients to view any previous day's home page. We estimate that improved page design reduced hits to the site by at least three-fold. Web server log analysis suggests that more than 25 percent of the users found the information they were looking for with a single hit by examining no more than the current day's home page.

## Site Architecture

The Web site utilized four IBM Scalable Power-Parallel (SP2) systems at complexes scattered around the globe, employing a total of 143 processors, 78 Gbytes of memory, and more than 2.4 Tbytes of disk space. We deployed this level of hardware to ensure high performance and availability because not only was this a very popular site, but the data it presented was constantly changing. Whenever new content was entered into the system, updated Web pages reflecting the changes were available to the world within seconds. Clients could thus rely on the Web site for the latest results, news, photographs, and other information from the games. The system served pages quickly even during peak periods, and the site was available 100 percent of the time.

We achieved high availability by using replicated information and redundant hardware to serve pages from four different geographic locations. If a server failed, requests were automatically routed to other servers, and if an entire complex failed, requests could be routed to the other three. The network contained redundant paths to eliminate single points of failure. It was designed to handle at least two to three times the expected bandwidth in order to accommodate high data volumes if portions of the network failed.

Dynamic pages were created via the FastCGI interface, and the Web site cached dynamic pages using the DUP algorithm. DUP was a critical component in achieving cache hit rates of better than 97 percent. By contrast, the 1996 Web site used an earlier version of the technologies and cached dynamic pages without employing DUP. It was thus difficult to precisely identify which pages had changed as a result of new information. Many current pages were invalidated in the process of ensuring that all stale pages were removed, which caused high miss rates after the system received new information. Cache hit rates for the 1996 Web site were around 80 percent.[12] Another key component in achieving near 100-percent hit rates was prefetching. When hot pages in the cache became obsolete, new versions were updated directly in the cache without being invalidated. Consequently, these pages had no cache misses.

Because the updates to underlying data were performed on different processors from those serving pages, response times were not adversely affected during peak update times. By contrast, processors functioning as Web servers at the 1996 Web site also performed updates to the underlying data. This design combined with high cache miss rates during peak update periods to increase response times.

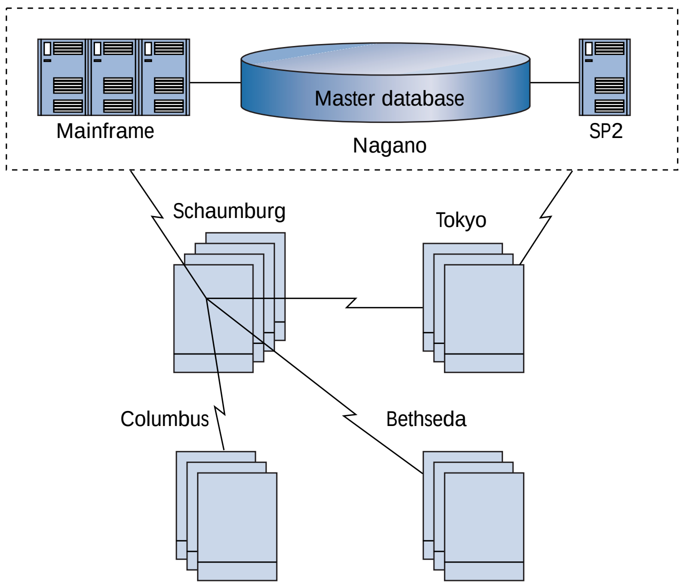

**Figure 5. Data flow from the master database to the Internet servers.**

## System Architecture

Web pages were served from four locations: Schaumburg, Illinois; Columbus, Ohio; Bethesda, Maryland; and Tokyo, Japan. Client requests to http://www.nagano.olympic.org were routed from

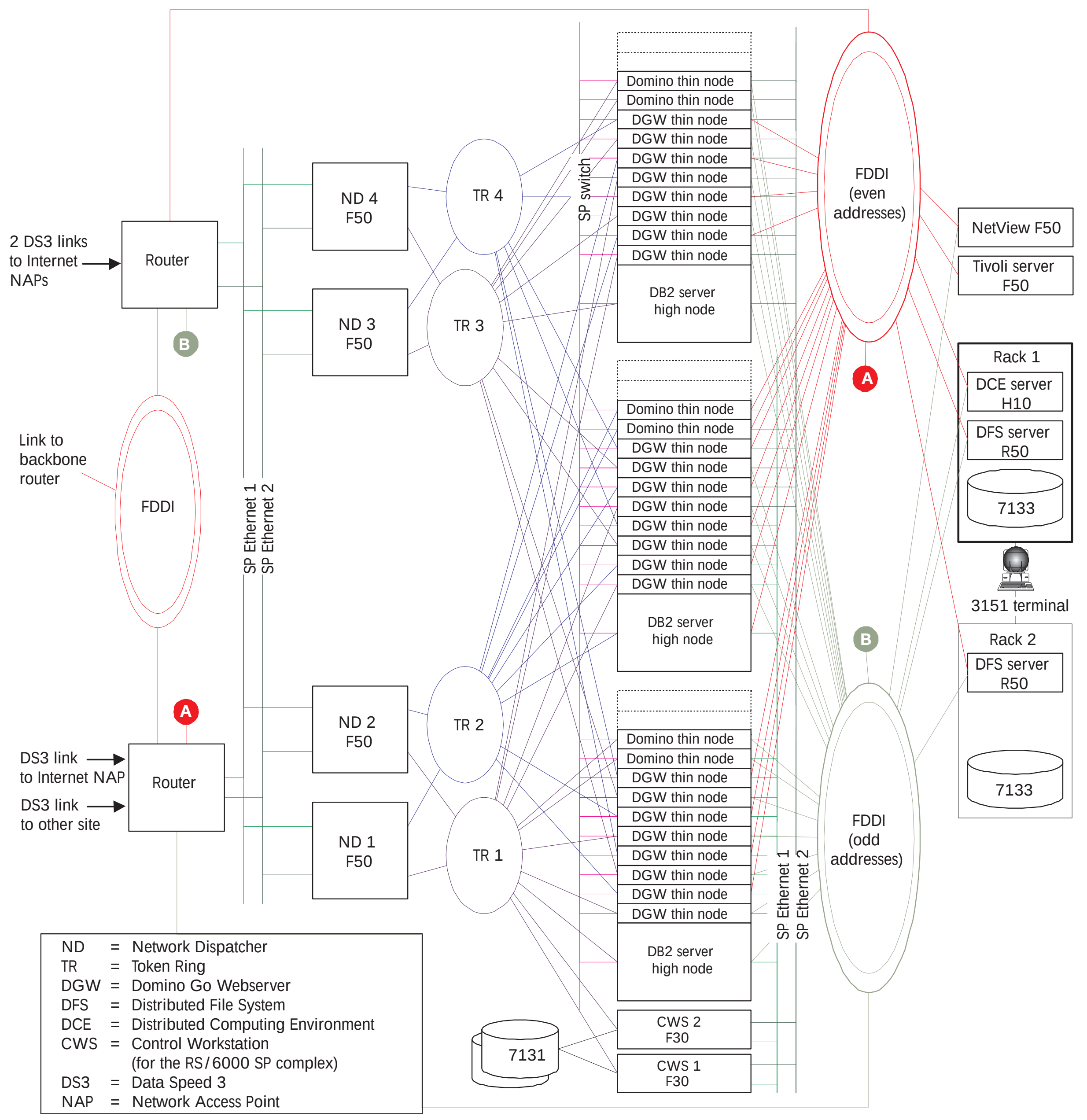

**Figure 6. Network diagram for each of the four complexes serving Web pages.**

the local ISP into the IBM Global Services network, and IGS routers forwarded the requests to the server location geographically nearest to the client.

Pages were created by, and served from, IBM SP2 systems at each site. Each SP2 was composed of multiple frames, each containing ten RISC/6000 uniprocessors, and one RISC/6000 8-way symmetric multiprocessor. Each uniprocessor had 512 Mbytes of memory and approximately 18 Gbytes of disk space. Each multiprocessor had one Gbyte of memory and approximately six Gbytes of disk space. In all, we used 13 SP2 frames: four at the SP2 in Schaumburg and three at each of the other sites. Numerous machines at each location were also dedicated to maintenance, support, file serving, networking, routing, and various other functions.

We had calculated the hardware requirements from forecasts based on 1996 site traffic, capacity, performance data; rate of increase in general Web traffic over the preceding year and a half; and the need for 100 percent site availability for the duration of the games. In retrospect, we deployed significantly more hardware than required. This overestimate was partly due to the fact that our caching algorithms reduced server CPU cycles by more than we had expected.

Results data was collected directly from the timing equipment and information keyed into computers at each venue. The scoring equipment was connected via token ring LAN at each venue to a local DB2 database that transferred its data to the master (mainframe-attached) database in Nagano. This master database served both the on-site scoring system and the Internet.

Figure 5 shows how data was replicated from the master database to the SP2 complexes in Tokyo and Schaumburg. From Schaumburg, the data was again replicated to the machines in Bethesda and Columbus. For reliability and recovery purposes, the Tokyo site could also replicate the database to Schaumburg.

## Local Load Balancing and High Availability

Load balancers (LB) were an essential component in implementing the complex network management techniques we used. While we chose IBM NDs, we could also have used other load balancers, such as those from Radware or Resonate. The LBs accepted traffic requests from the Internet, routed them to a complex, and forwarded them to available Web servers for processing. As illustrated in Figure 6 (previous page), four LB servers sat between the routers and the front-end Web servers at each complex in the U.S. Each of these four LB servers was the primary source of three of the 12 addresses and the secondary source for two other addresses. Secondary addresses for a particular LB were given a higher routing cost.[11]

The LB servers ran the gated routing daemon, which was configured to advertise IP addresses as routes to the routers.[13] We assigned each LB a different cost based on whether it was the primary or secondary server for an IP address. The routers then redistributed these routes into the network. With knowledge of the routes being advertised by each of the complexes, routers could decide where to deliver incoming requests to the LB with the lowest cost. This was typically the LB that was the primary source for the address assigned to incoming requests at the closest complex.

The request would only go to the secondary LB for a given address if the primary LB were down for some reason. If the secondary LB also failed, traffic would be routed to the primary LB in a different complex. This design gave LB server operators control of load balancing across complexes. Moreover, the routers required no changes to support the design because they learned routes from the LB servers via a dynamic routing protocol.

Each LB server was connected to a pool of front-end Web servers dispersed among the SP2 frames at each site. Traffic was distributed among Web servers based on load information provided by so-called *advisors* running at the Web server node. If a Web node went down, the advisors immediately pulled it from the distribution list.

**Since the 1998 Olympics, our technology has been deployed at several other highly accessed sites.**

This approach helped ensure high availability by avoiding any single failure point in a site. In the event of a Web server failure, the LB would automatically route requests to the other servers in its pool, or if an SP2 frame went down, the LB would route requests to the other frames at the site. The router would send requests to the backup if an LB server went down, and if an entire complex failed, traffic was automatically routed to a backup site. In this way we achieved what we call elegant degradation, in which various failure points within a complex are immediately accounted for, and traffic is smoothly redistributed to system elements that are still functioning.

Since the 1998 Olympics, our technology has been deployed at several other highly accessed sites including the official sites for the 1998 and 1999 Wimbledon tennis tournaments. Both of these sites received considerably higher request rates than the 1998 Olympic games site due to continuous growth in Web usage. The 1999 Wimbledon site made extensive use of Web server acceleration technology that was not ready for the 1998 Olympic games. The site received 942 million hits over 14 days, corresponding to 71.2 million page views and 8.7 million visits. Peak hit rates of 430 thousand per minute and 125 million per day were observed. By contrast, the 1998 Olympic Games Web site received 634.7 million requests over 16 days with peak hit rates of 110 thousand per minute and 57 million per day.

We have developed a sophisticated publishing system for dynamic Web content for the 2000 Olympic Games Web site, which will also make extensive use of Web server acceleration. A description of this publishing system is scheduled to appear in the *Proceedings of InfoCom 2000*.[14] ■

## ACKNOWLEDGMENTS

Many people made valuable contributions to the 1998 Olympic Games' site. Cameron Ferstat, Paul Reed, and Kent Rankin contributed to the network design, and John Thompson, Jerry Spivak, Michael Maturo, Kip Hansen, Brian Taylor, Elin Stilwell, and John Chiavelli, to the Web site design. Kim Benson provided us useful information. Glen Druce contributed to the news, photos, and Lotus Notes design. Eric Levy and Junehwa Song made key contributions to the Web server accelerator described in this article.

## REFERENCES

1. T. Brisco, "DNS Support for Load Balancing," Tech. Report RFC 1974, Rutgers Univ., N.J., April 1995.
2. P. Mockapetris, "Domain Names—Implementation and Specification," Tech. Report RFC 1035, USC Information Sciences Inst., Los Angeles, Calif., Nov. 1987.
3. T.T. Kwan, R.E. McGrath, and D.A. Reed, "NCSA's World Wide Web Server: Design and Performance," *Computer*, Vol. 28, No.11, Nov. 1995, pp. 68-74.
4. D. Dias et al., "A Scalable and Highly Available Web Server," *Proc. 1996 IEEE Computer Conf.* (CompCon), IEEE Computer Soc. Press, Los Alamitos, Calif., 1996.
5. G. Hunt et al., "Network Dispatcher: A Connection Router for Scalable Internet Services," *Proc. 7th Int'l World Wide Web Conf.*, 1998; available online at http://www7.scu.edu.au/programme/fullpapers/1899/com1899.htm.
6. V. Cardellini, M. Colajanni, and P. Yu, "Dynamic Load Balancing on Web-Server Systems," *IEEE Internet Computing*, Vol. 3, No. 3, May/June 1999, pp. 28-39.
7. A. Chankhunthod et al., "A Hierarchical Internet Object Cache," *Proc. 1996 Usenix Tech. Conf.*, Usenix Assoc., Berkeley, Calif., 1996, pp. 153-163.
8. E. Levy, et al., "Design and Performance of a Web Server Accelerator," *Proc. IEEE InfoCom 99*, IEEE Press, Piscataway, N.J., 1999, pp. 135-143.
9. R. Lee, "A Quick Guide to Web Server Acceleration," white paper, Novell Research, 1997; available at http://www.novell.com/bordermanager/accel.html.
10. J. Challenger, A. Iyengar, and P. Dantzig, "A Scalable System for Consistently Caching Dynamic Web Data," *Proc. IEEE InfoCom 99*, IEEE Press, Piscataway, N.J., 1999, pp.294-303.
11. J. Challenger, P. Dantzig, and A. Iyengar, "A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites," *Proc. ACM/IEEE Supercomputing 98* (SC 98), ACM Press, N.Y., 1998; available at http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Challenger602/index.htm.
12. A. Iyengar and J. Challenger, "Improving Web Server Performance by Caching Dynamic Data," *Proc. Usenix Symp. Internet Tech. and Systems*, Usenix Assoc., Berkeley, Calif., 1997, pp. 49-60.
13. D.E. Comer, *Internetworking with TCP/IP*, Prentice Hall, Englewood Cliffs, N.J., sec. ed., 1991.
14. J. Challenger et al., "A Publishing System for Efficiently Creating Dynamic Web Content," to be published in *Proc. InfoCom 2000*, IEEE Press, Piscataway, N.J., Mar. 2000.

Arun Iyengar is a research staff member at IBM's T.J. Watson Research Center. His research interests include Web performance, caching, parallel processing, and electronic commerce. He has a BA in chemistry from the University of Pennsylvania, and an MS and PhD in computer science from the Massachusetts Institute of Technology.

Jim Challenger is a senior programmer at the T.J. Watson Research Center. His research interests include development and deployment of highly scaled distributed computing systems, caching, and highly scaled Web servers. He has a BS in mathematics and an MS in computer science from the University of New Mexico.

Daniel Dias manages the parallel commercial systems department at the T.J. Watson Research Center. His research interests include scalable and highly available clustered systems including Web servers, caches, and video servers; Web collaboratory systems; frameworks for business-to-business e-commerce; and performance analysis. He received the B.Tech. degree from the Indian Institute of Technology, Bombay, and an MS and a PhD from Rice University, all in electrical engineering.

Paul Dantzig is manager of high-volume Web serving at the T.J. Watson Research Center and adjunct professor of computer science at Pace University, New York. His research interests include Web serving, digitial library and search, speech application software, content management, Internet protocols, and electronic commerce. He has a BS interdepartmental degree in mathematics, statistics, operations research, and computer science from Stanford University and an MS in computer science and computer engineering from Stanford University.

Readers can contact the authors at {aruni, challngr, dias, pauldant}@us.ibm.com.