Web server architecture

antoni_drudis@hp.com
john_mendonca@hp.com

Hewlett-Packard Company
11000 North Wolfe Rd. 47 LAX
Cupertino, CA 95014
tel/fax (408) 447-5109

Architecture of web applications

The distributed nature of web applications requires a flexible, scalable, and fault-tolerant architecture that is not adequately served by traditional transaction-processing engines. While the basic building blocks of software tools have not changed much over time, the way these blocks are put together reflects the new requirements on the overall solution.

The generic architecture of the client-server model is just the first step towards the design of the solution. Performance considerations such as communication bandwidth requirements, transaction processing methodology, and taking advantage of parallel computing and multithreading technology on the server platform determine the perceived quality of products such as database, web, and application servers.

Software design is an evolving discipline where every major milestone is the consolidation of technical innovations from previous steps with the solution for new application requirements in terms of resilience, development costs, and functionality. The introduction of client-server architectures was marked by the formal delegation of the application intelligence to distributed objects. For example, in the client-server model, the graphical user interfaces manage the interaction between the user and the application, and local editing reduces the need to use valuable server resources to perform tasks that can be done as well at the client side. That strategy was already used in the conventional mainframe architecture supporting intelligent terminals. What is different in the client-server paradigm is that the application is no longer centralized but distributed among specialized servers.

Since many functions that used to be centralized in the mainframe were moved towards autonomous servers, some of the design paradigms had to be changed accordingly. For example, the need for a uniform resource identification policy for objects led to name servers and the need to protect services from unauthorized access led to secure services and encryption techniques. From the user's perspective, client-server architectures provided a lower-cost, flexible, and scalable solution where services and processing power could be added or modified without a major rewrite of the application software.

The web is an extension of the client-server paradigm

The web can be architected as a client-server model where web servers deliver multimedia contents to browsers and other clients such as applets and search robots. The web has changed the scale of the global network and the patterns of server access. Concurrent users are not counted in thousands but in tens or hundreds of thousands, the environment is heterogeneous and constantly changing, transactions are stateless and difficult to track, and the process may integrate applications from different environments. These four differentiators for web applications highlight the need to build a new design paradigm and the reason to explore the implications of the requirements in transaction processing applications.

Scaling can be achieved by replacing the current processing platforms by faster systems but, specially, by distributing services and replicating servers. That strategy not only provides increasing computing power without bringing down the whole network but it also increments the resilience of the solution and allows for planned and unplanned down time in each server.

Standardization -either formal or de-facto- of application components facilitates the interoperability of servers and clients on the web. The transport protocol is standard, markup languages are being standardized, and file formats such as pdf are being used across operating systems and computing platforms.

A basic principle in software design is the ability to map the logical requirements of the users into the physical implementation of a solution that satisfies these needs. While users formalize their requirements in a high-level language and programmers map that language into an architecture-oriented application, the execution of the program does not offer a simple platform-independent mapping between the user's view and what code is running in the platform at a given time.

Transactions and sessions

The concept of transaction narrows the gap between logical and physical views of how data is processed. A transaction is a set of operations that, if fully executed, guarantee to leave the data in a consistent state. For example, in a low-level transaction such as a disk I/O, the programmer is guaranteed that there is no need to recover from a hardware malfunction. The read o write operation will return a value to indicate the success or failure of the operation but the operation itself will take care of unwanted concurrent access at this level. At a higher-level transaction such as a database update, a set of logical operations is required to execute as a single unit to ensure consistency. For example, to transfer money between two accounts, the program has to subtract a given amount from the balance of the first account and add the same amount to the balance of the second account. That transaction spans several lower-level transactions that have to be undone if any of them fail.

The architecture of software applications evolves as new technologies allow a change of scale in the number of transactions these applications process. Software designers make assumptions about the architecture of the platform and the performance ranges of the components of the solution. For example, the programmer may assume that access to memory is much faster than access to a disk and consequently minimize the I/O requests. A change in the order of magnitude of the volume of data to be processed often forces the designer not only to reconsider the speed requirements of the hardware components but also how these components relate to each other. In the example, if the memory requirements exceed the available memory, the application may incur in frequent page faults and lose the theoretical advantage of using faster memory access.

Applications running on low-performance computing platforms often force a simplification in the external processes. For example, on these platforms, transactions can be serialized if the number of concurrent users is small. In many cases, entry-level machines perform critical tasks using a single-user paradigm. Attempts to parallelize the process often start by replicating lower-cost resources so the usage of the most expensive resources is maximized. For example, the move from traditional batch processing computers to the time-share minicomputers and transaction servers was based on the much faster increase of performance in computing power than the increase in performance -or its perceived need- of disk files and terminals. In a traditional transaction-processing server, programmers may indicate the logical bounds of a transaction and the middleware or the operating system will perform the necessary actions to preserve the ACID requirements of the transaction.

Two architectures used for traditional transaction processing applications are mainframes and client-server. Mainframes typically allow concurrent users by serializing the access to critical resources such as data on disks, cache, and I/O buffers. By allowing multiple levels of resource locking -at a global, file, entry, or data item level- multiple users may work concurrently on unrelated data. Transaction serialization is done using application paradigms such as two-phase commit.

On simple client-server applications, concurrency can be achieved by using the communications protocol features to serialize the access to critical resources. For example, by delegating the management of user requests to sockets, the server program can process a transaction at a time while all the other clients are sending their requests through sockets. A key differentiator of traditional transaction process is the fact that transactions are processed as unrelated to each other. There is no state kept from one transaction to the next and the environment is stable.

The classic paradigm of transaction processing needs to be extended for the web. Both on servers and mainframes, transaction monitors are based on the assumption that the client or the remote terminal will send well-behaved transactions. These transactions are short, serializable, most frequently stateful, and homogeneous. In the initial instances of the web usage to publish data, that concept fits the requirements of the web server: users send requests to get data from a file or out of a software module, and the server processes each transaction on a best-effort basis. The goal of the systems architect is to optimize the resources. On transaction processing engines, optimization is reached by minimizing the waiting time for each transaction.

Instead, web servers for database access, such in business to business or business to consumer transactions, introduce the concept of *session*, where a set of transactions have to be processed using the same criteria about the user environment. As opposed to typical transaction processing environments, clients may pause in a session for long periods of time while the user interacts with the application.

For example, a user accesses a book catalog on a web bookstore. That type of access can be seen as a simple transaction where the user sends the query criteria and the web server communicates with the application server to request database entries that satisfy the query conditions. Next, the user selects one or more books and decides to buy them. Because the web runs non-stop, the store may have changed the price of the book while the user was deciding whether to buy it, but the buyer still expects the old price to be applied to this sale. Next, the buyer places the order and the web server talks to the application server which in turn may place an order to the publisher's server, request space in the warehouse, start a shipping order, verify and charge the customer's credit card, and send an e-mail confirmation. All these individual transactions on different servers and subnets form a single session because all of them have to use a uniform set of data from the environment. While there are many transaction engines commercially available, so far sessions are managed by individual applications. This is an area of growth both from the theoretical perspective and product offerings. At this time, state preservation, limited scalability, and application resilience are the main roadblocks to the expansion of the web in commercial applications.

The Java language is making its inroads as a pervasive computing environment at the client side, thus reducing the need

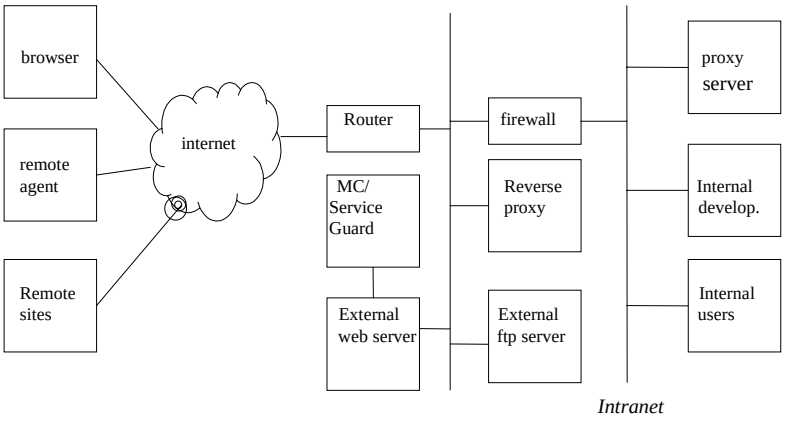

Figure 1: architecture of a typical web server

for communication bandwidth and server processing. Efficient queuing management tied to a multithreading paradigm provides a first level of smoothing out the rate variations in client accesses, and solutions at the client side such as cookies and the server side such as application servers provide some state preservation in a fundamentally stateless environment. Some specialized applications such as publishing -where data is stable, access is repetitive, and the cost of local storage is smaller than cost of bandwidth- can use data and server caching and platform replication as a means to reduce the current strain on servers.

Replication and caching are further utilized with static data to increase the tolerance of the overall network to both planned and unplanned maintenance tasks. Communication from the client is no longer an all or nothing proposition but a guaranteed operation when the access parameters stay in a given range. System and network administrators can configure the resources to provide a fail-over mechanism that prevents catastrophic consequences when a key component fails.

Local solutions extend the usability of components but a new definition of the principles that govern web applications is needed

Optimizing individual components of the e-commerce solution may prove to be a good short-term strategy for product vendors to accelerate their market penetration. But on a global scale, integrators of the hardware and software platforms where the web, application, and database servers run are still faced with the formidable task to translate these component-level performance improvements into a predictable end-user experience.

Since network communications are bound by the slower component, solution architects often design redundant solutions with replicated components such as network segments and computer systems. These topologies scale only if the appropriate algorithms run on the nodes to divert the traffic into the most efficient branch. Protocols such as ATM based on these algorithms may provide network level quality of service.

The main factors in solution performance are providing adequate bandwidth and balancing the distributed components of the application in terms of I/O, memory and processing power. For example, by moving services to a different system, the designer may free up some scarce resources at the cost of increasing the bandwidth needed for the additional network traffic. This is a continuous task because of the increasing requirements on data traffic for multimedia applications.

The web introduces news challenge to application architects and stresses the need to design up front the security, scalability, manageability, and quality of service that users will require.

- Users are no longer the trusted and well trained employees who used to access the company's database but semi-anonymous users who may engage in non-cooperative or destructive behavior that increases resource consumption and endangers the integrity of the database. The distributed and stateless nature of the application makes it difficult to maintain efficient and secure communications between the server and its clients.

- The designer of the overall solution needs to anticipate the access patterns. While demand in a controlled environment is bound by an upper limit that can be architect into the application, the web is characterized by an unpredictable demand. Web access patterns changes during the hours of the day, the days of the week, and other seasonal variations, external events. Also, unexpected events such as breaking news that make users to access the server such as an on-line fashion show that produces a sudden increase of visitors to the company web site or a marketing campaign that produces a steady increase in the traffic- originate peaks in resource consumption. Disk space, processing power, and network bandwidth may reach their limit and all users experience bad response time and diminished throughput.

- When the number of users is larger than expected, the server needs to differentiate users by giving priority to certain users who are perceived to add the most value to the service supplier.

- Even for the same user, there might be a need to differentiate services at the system level. For example, complex queries to a large database consume more resources than just accessing a static web page or filling a simple order form. The administrator of the application may want to provide premium service to the users who engage in commercial transactions through the server by penalizing non income-producing activities such as catalog browsing.

- While servers are designed as independent entities, the performance of the application depends on how the user environment fits into the expected network topology in terms of servers, services, and systems.

Programming paradigms for transaction servers

The single transaction instance model may fit into one of the two main server paradigms:

- In real-time systems, the transaction is completed before the application gives control to the next request. This model can be used when transactions are short in nature, they have no logical interrupts such as waiting for a response from another module, and throughput is deemed more important than the response time of individual transactions.

- Time-sharing systems allow for concurrent processing of transactions lined-up in different queues. In this case, the system allows the transaction to be interrupted so other transactions are not starved out from critical resources. In time-sharing systems, the programmer defines the bounds of the transaction and often has to define the recovery methods when a transaction cannot be completed. This model is typically used when transactions are long in nature and have logical interrupts such as waiting for a response from the user or an I/O device, and the response time of individual transactions is deemed more important than overall throughput

For example, an ATM machine may use the more efficient real-time transaction model because the individual transactions are simple and independent of each other. Users wait for a few seconds while their requests are being lined up in the transaction queue in the server. Once the transaction is executed, the user resets his or her expectations about the contents of the data and the performance of the terminal. Instead, an airline reservation system may use the time-sharing paradigm because a typical transaction may require many steps where the decision at a given point depends from the response of the system to the previous inquiry. The combination of the application rules for record locking and the user training to set the right expectations on the accuracy of the data at a given time allows a large percentage of transactions to be executed without rejection from the server.

Multi-user transaction models, used for example in workflow collaboration programs, define an environment where the users log into the application, which keeps track of each transaction. Under that model, the parallelism does not only apply to the I/O components of the application but also to the shareable critical components. There are several variants of this model, depending on where the queues are set up. For example, in a simple multi-threaded application, users may access the server using the socket library, effectively serializing the requests from the clients. Once a request reaches the server, the application may assign a given thread to that request. At this point, the programming paradigm for the client is the same as the time-sharing model.

Server implementation

Servers are typically implemented as daemon processes (services) that may use standard mechanisms for communication with their clients such as CORBA or the socket library to support application features such as guaranteed delivery and serialization of requests. While some servers are meant to be lightweight, others try to use all available resources to process concurrent requests from hundreds or thousands of users. Foot print, scalability, transaction latency and software fault tolerance are just a few of the many characteristics of a server.

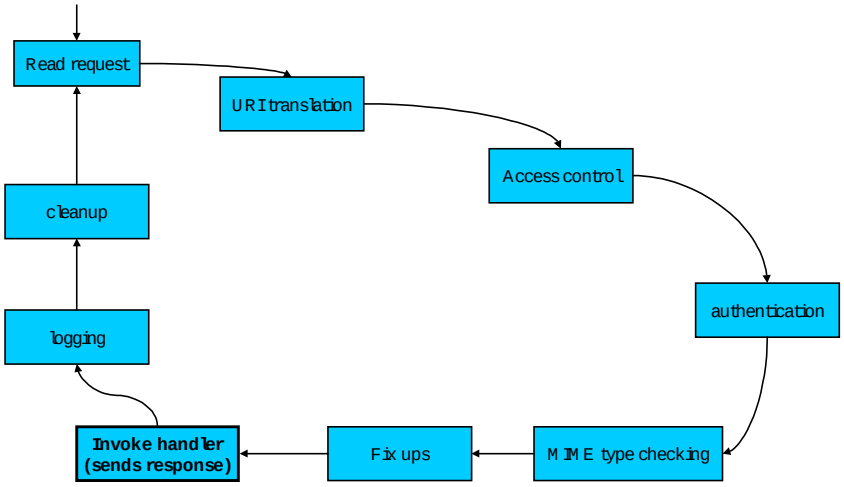

Figure 2: web server functions

Most web servers are meant to be general-purpose applications where the user can plug specific tools to extend the functionality of the server in areas such as multimedia processing, encryption and authorization, or resource management. While this strategy leads to applications with a high degree of compatibility between versions and easy migration paths for the user, it does not lead to a product tuned to the specific requirements of a given company.

By analyzing the design objectives of the application and published performance data for the server, the user can determine which brand better fits his or her requirements. The analysis of a web server can suggest implementation strategies that can be used for other programs that use the client-server paradigm. For example, some programmers may analyze the Apache web server to see how to use plug-in technology and other programmers might be interested in designing servers that operate at peak performance such as the Zeus web server or support a wide set of functions such as the Netscape Enterprise Server.

The basic purpose of a serve is to attend unexpected requests from its clients, process these requests in a predictable manner, and return the result to the client. To accomplish its objective, the server can be structured in a three-layer model, namely I/O, concurrency control, and request processing.

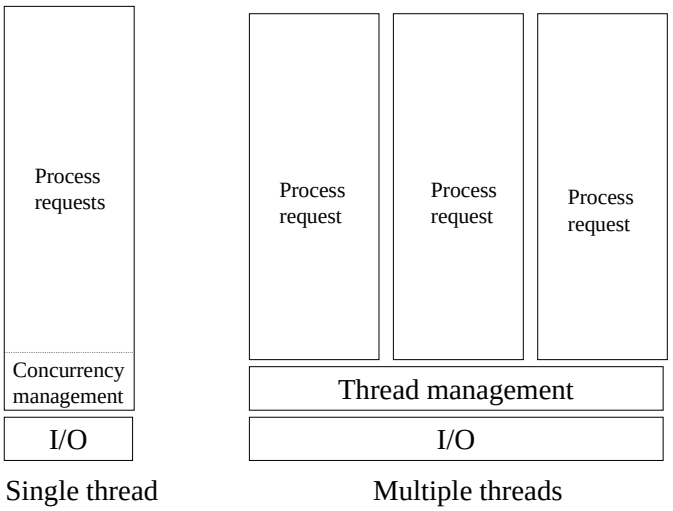

Figure 3: single-threaded and multi-threaded applications

At the I/O level, the server can serialize the requests to simplify the programming paradigm. Seemingly concurrent requests from clients are queued using an algorithm (first-come, first-served, priority-based, LIFO, etc.) that fits the application requirements (top performance, quality of service, optimization of a given component, etc).

At the request process level, the server may use a best-effort model where the request (transaction) is fully processed and returned to the client before starting the process of the next request. Sometimes, when the requests consume highly variable amount or resources, the server may use a concurrent model where transactions have to coexist.

In any case, the server has an implicit or explicit module for concurrency control to ensure data integrity and guarantee a reasonable performance. Figure 3 shows the two-layer architecture of a single-threaded server.

When transactions have to be processed in parallel, the server may take advantage of the hardware architecture of the platform and assign a different thread to each transaction. In the example shown in figure 3, the server consists in three logical layers. At the I/O level, this server receives the requests from multiple clients and serializes them. Next, at the thread management layer, the server assigns each request to a particular thread created and destroyed either as they are needed by the server or created initially using some configuration database and assigned dynamically to the request. In both cases, the intelligence of the server about concurrency guidelines resides in that specific layer.

At the request process layer, the server could be somewhat simpler than in the single-thread model at the expense of minimizing global data shared by all threads.

Multithreaded architectures are the norm in web servers, even on single-processor platforms, because the diverse level or resources needed to process each request and the independence of a request from other requests from other clients. When the web server is connected to an application server, the advantages of the multithreaded architecture for single-processor platforms may not be that significative.

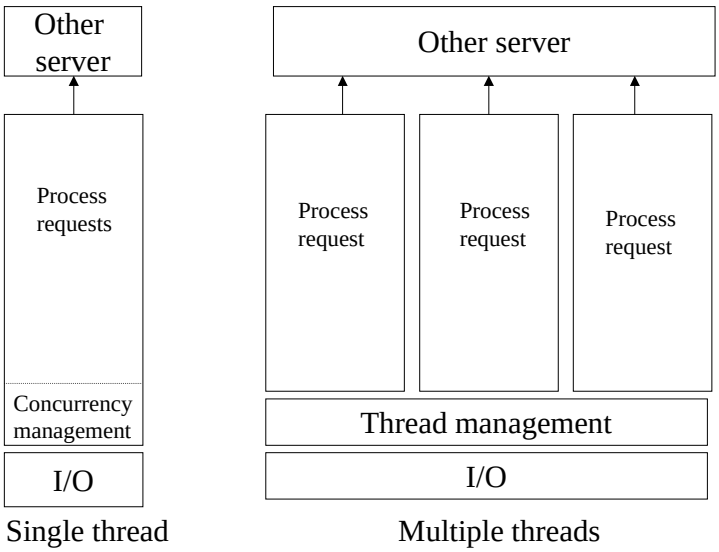

Figure 4: bottlenecks single-threaded and multi-threaded applications

Servers using multi-threaded architecture may have better performance on multi-processor platforms if the services invoked by each *process request* module (the middle layer in the figure) perform faster than these individual threads. Otherwise the performance improvements of the multi-threaded architecture may be lost and single-thread and single-process servers provide the functionality at a lower cost.

Example: simple administration server for a network-based application

In some instances, simple single-purpose programs yield a better performance than full-fledged general-purpose servers that use more modern technologies do. For example, figure 5 shows the architecture of a single-threaded management server. That server is a daemon process waiting for requests from clients such as GUI programs and applications running on the network. Clients send requests to the server to read or update the configuration of the network topology and distributed applications parameters. Clients may also send requests to notify events and communicate the status of each component. One of these servers is the logging daemon. Clients send asynchronous messages to the logging server and may occasionally request configuration data such as log file names and sizes.

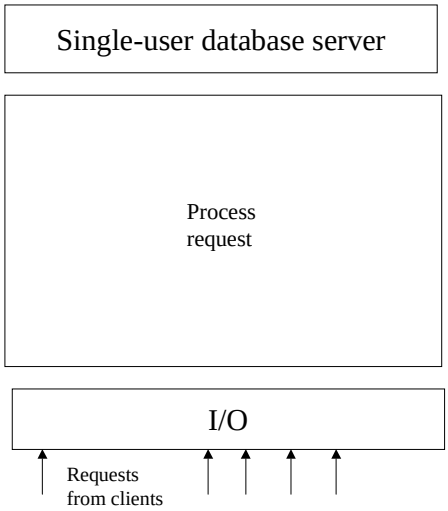

Figure 5: management server

If the management server communicates with its clients through message queues or sockets, requests are serialized. Because of the dependencies between requests (for example, the meaning of an event depends on the configuration of the component) the most efficient architecture is to use a single thread that processes the request on a best-effort basis. Access to the underlying database does not require a locking mechanism and the throughput be maximized.

Note that the server depicted in figure 5 can be abstracted into a single object where methods are used to configure and retrieve configurations and to notify and select miscellaneous events and statistics. From the client point of view, this server has a behavior similar to traditional transaction servers: when the client issues a request, the request is placed on an input queue to the server and the initial response time depends on the current length of that queue. When the request is received, it is processed with no interruptions from other clients –reducing concurrent access to the database- and then it is returned to the client. Single-processor servers may benefit from that architecture because of its simplicity of programming and overall throughput. Note that when the program supports dialog with the user, this model might not be appropriate because of the long response time when the input queue is longer than expected.

In this example, if the server is multithreaded, transactions sent to and from the clients will have to go through a multiplexing and a demultiplexing process. First, all client requests sent to the server are queued by the socket library. Next, the server reads the transactions and processes them on a separate thread. Then, the transaction will have to compete with transactions on other threads: the database access will be locked for each transaction and the advantages of using multiple threads will be lost.

Counter-example: an alternative to the simple management server

Requirements on commercial-grade servers often include capabilities far beyond the raw performance provided by the server in the example. For example:

- Software-based fault-tolerance, which is typically implemented by using a heart-beat that checks if the server responds in the required time period and a dispatcher processes that switch to a backup server when the heart beat is not received.
- Manageability through OpenView or other management tools
- On-the-fly software upgrades, which require component versioning and dynamic binding
- Auto-discovery, self-configuration, and other intelligent initialization of the status of the server

These capabilities may be designed into the base product or they may be coded later on as add-ons when the architecture of the server allows simultaneous execution of the components. Because of this, multithreaded architectures are the norm more than the exception on modern servers.

Where to draw the line: distributed applications architecture evaluation criteria

Successful software applications satisfy a limited set of requirements but they satisfy these requirements without flaws. In the case of distributed applications, the set of requirements is still small. Typically, application requirements include:

- Scalability, which is achieved by replicating resources such as services and client applications.
- Flexibility, which is a consequence of an intelligent partition of functions between the elements of the distributed application. For example, by separating database access from business logic, the engineering team may take advantage of performance improvements when a new version of the database server is released while being able to adapt the application functionality to the requirements of the user.
- Resilience, which is a consequence of quality of the design and implementation and the replication of resources to minimize the impact of application component failures. For example, if the example server described above uses a heart beat to detect the presence of a backup server and take over in case of failure, the application will have a better chance to stay up when unplanned events occur.
- Raw performance, typically in the case of web and application servers where throughput and response time can be easily mapped into revenue for the company running the server.
- Efficiency, which has to be balance against the other criteria to determine the best cost/performance solution that fits into the requirements of the application.

In addition to these requirements, the implementation team has to take into consideration the classic principles of software design:

- Serial processes run at the speed of the slower component (all CPUs, no matter how fast, wait at the same speed)
- While parallel processes augment the capacity (throughput) of the application, the response time for each individual branch still depends on the implementation of each parallel instance.
- Optimization makes sense for critical resources. In distributed applications, servers are always critical. Network bandwidth, user-interface performance and customization may also determine the success or failure of the application.

- From the software architecture's point of view, speed is achieved by caching intermediate results, thus reducing the need for resources. On the other hand, flexibility is achieved by minimizing the interdependencies that cached solutions require. The balance between replication and raw performance on a particular server determines its applicability to a specific problem.

Acknowledgements

The ideas presented in this paper are based on the discussions the authors had while working on WebQoS web server tools. Brian McCracken and Kim Scott provided many useful insights into the architecture of the management server and David Dalton made extensive contributions in the area of caching and performance. Stuart Cain, Frank Leong and Frank Lawrence provided the support for the development and presentation of this paper at the HP-World conference.

References

If you can buy only one book this year and you want to design a server, that book should be "*Unix Network Programming, Volume 1"* by Richard Stevens. If you have more time and a bigger budget buy any good *pthreads* book, download the Apache web server and you'll have a good basis for developing a good server.