CHAPTER

3

Internal Server Architectures

Often, it is important to understand how software works internally in order to fully understand why it functions the way it does, and how to configure it best to gain optimum performance. This chapter covers some of the different internal server architectures from the point of view of the implementation and processing paradigm.

It is by no means a complete lesson on server programming, as there are a lot of subtleties and performance features which contribute to the high performance seen in today's server products. Some of these features are too complex for the scope of this book and may be trade secrets of their respective companies.

This chapter is not specific to proxy servers, and the principles can be applied to any information server architecture. UNIX systems allow for all of these different variants; on NT the operating system architecture and programming design rules out multiprocess architectures, so these variants are pertinent to UNIX systems only.

Architectural issues related to caching are covered separately in Chapter 10.

SINGLE-PROCESS SERIALIZED SERVER ARCHITECTURE

The simplest type of server is one that sequentially accepts a request and services it to the end before taking on the next request. Obviously, this is an unacceptable alternative in the Internet server world where the number of requests is overwhelming and they must be serviced in parallel. It is unacceptable to let a client wait until some other client request is serviced.

This section is here for completeness, and to clarify the point that Internet servers need to be able to handle parallel requests. There are several different ways to accomplish this:

- by forking a new process for each request
- by keeping a pool of separate server processes that continuously accept requests and process them
- by spawning a new thread, instead of a process, to handle each request
- by keeping a pool of separate threads, instead of processes, around to handle the load of requests
- by using an asynchronous I/O server architecture that is capable of managing multiple parallel connections from within a single process/thread

FORKING

Admittedly, the simplest way to implement a server capable of servicing multiple requests in parallel is the forking model [1]. In this model the master process sits in a loop simply accepting new connections, and for each new received connection forking a new process to handle it. The new process will handle the request and exit upon completion.

A benefit of this architecture is that the master process can be very simple, and, therefore, stable. Another advantage is that the child processes don't have to worry about memory leaks because each process will exit upon completing the response, and the memory will be automatically cleaned up by the operating system.

In most early Web and proxy servers [2], the base architecture was so simple that the authors were able to focus heavily on developing the actual Web technology, the HTTP protocol, and server features, such as CGI—still to date (in 1997) the only standard server application interface.

The dawn of commercial application of Web technology soon rendered these forking servers inefficient to handle the load generated by the boom of the Internet era. Namely, *forking a new process involves considerable overhead*. Performance of these early forking servers, typically in the range from a few requests to a few dozen requests per second, was only a fraction of that of modern servers utilizing new, more efficient architectures. Modern Web servers can handle hundreds of requests per second.

PROCESS MOB ARCHITECTURE

The first breakthrough in high-performance Web servers was the introduction of the so-called *process mob* [3] architecture. In this model a set of processes are preforked at the server startup time. These processes remain resident, servicing requests in parallel. After each response, the process will simply proceed to the next request.

The mob process model eliminates the overhead of the `fork()` system call. Processes are created once during startup time, and the same processes get reused over and over again. The mob process model has been in use in the Netscape Proxy Server since its first version.

Dealing with Memory Leaks

The process mob model requires the server software to be written carefully so that the persistent processes don't corrupt their address space by programming errors causing crashes and don't clutter the memory by memory leaks. Despite diligence, memory leaks may still be a problem. Some [older versions of] operating systems may have standard system libraries that unfortunately leak memory. Also, since it is possible for users to add on their own server plugins using server programming APIs, the user code may introduce a memory leak.

Two solutions exist to control memory leak related problems:

Chapter3 Page 32 Friday, January 26, 2001 2:41 PM

- limiting the lifetime of each server child process
- memory pools

Limiting Child Process Lifetime

By limiting the lifetime of each server child process the processes are forced to eventually exit (freeing any leaked memory) and get respawned by the master process. Even though this reintroduces forking, it will have minimal performance impact since there is only a small fraction of forking compared to the number of requests processed. A typical process lifetime is on the order of hundreds of requests.

Memory Pools

Memory pools are a creative way to prevent memory leaks while at the same time enhancing performance. The standard dynamic memory management routines `malloc()` and `free()` involve certain overhead that in the Web server environment can be avoided by introducing a new clever memory allocation routine. This new memory allocator returns the memory from a larger pool of memory preallocated using `malloc()` and associates with the data structures containing information about the request. Instead of having to worry about freeing all the allocated memory, this system doesn't have an explicit memory freeing routine: all memory is freed automatically upon the completion of request processing.

From the system's point of view, there was only one call to `malloc()` at the beginning of the request, and one call to `free()` at the end. In between, the server application handles application routines' requests for memory allocation by giving out memory slots from the large memory pool. Only a single pointer is retained to keep track of where the allocated memory area ends and free area begins. Also, if dynamic memory consumption is high, another pool may have to be allocated.

MULTITHREADED, SINGLE-PROCESS ARCHITECTURE

Another approach to servicing parallel requests is to use multithreading instead of multiple processes. The simpler version of this approach is to create a new thread for each incoming connection and destroy the thread upon completion of the request service. This corresponds to the forking server model (the section on forking on page 30), but, instead of creating new processes, new threads are created.

Thread Pool Model

The use of threads can be refined in the same way the forking process model was refined into the preforked process mob model: there is a pool of prespawned threads that handle the mass of incoming connections.

In practice, there is typically a single so-called *accept thread*, which, as its name suggests, sits waiting for new connections and accepts them as they come in. It will then queue the connection and notify the so-called *worker threads* of the new connection. One of the worker threads dequeues the connection, reads the request from the connection, services it, and then moves on to the next request (with persistent connections), or dequeues a new connection.

MULTITHREADED, MULTIPROCESS ARCHITECTURE

The multiprocess and multithreaded architectures can be combined: the result is a pool of preforked processes, each containing a pool of prespawned threads. As an example, this architecture was deployed in Netscape Netsite Server 1.1.

SINGLE-PROCESS, ASYNCHRONOUS I/O ARCHITECTURE

In the face of the extremely high loads that proxy servers may have to cope with, even threading may have too much overhead associated with it. The management of—and the context switches between—the hundreds of threads may take up a considerable portion of the processing power.

Most of the duration of a request service cycle is spent waiting for a (slow) remote server to respond. During this time the thread (or process) is idle but tied up with that request and cannot do anything useful. Once data is streaming in, the proxy will simply pass it to the client, possibly doing some content filtering and writing to the cache.

In the asynchronous I/O architecture, the sockets are marked nonblocking [4]. This causes `read()` calls to return immediately with a return status indicating that the call would block [5] if there is no data, instead of waiting for data to arrive. This allows the software to perform other tasks (service other requests) while the connection is idle. Similarly, calls to `write()` will return a status code indicating that the call will block if the receiving end of the connection is not yet ready to receive more data (that is, internal buffers are full and the application should wait for the destination to read more data). Normally, the `write()` call will block waiting for the data to be delivered, but with the asynchronous I/O enabled the software can continue with other tasks and deliver the remaining data later when the socket is ready for more writing.

The overall architecture whirls around the so-called *select loop*, which is named after the `select()` call. `select()` is given an array representing socket descriptors, and it blocks until one or some of them are ready for reading and/or writing. On return the bits of the array are modified to indicate which sockets are ready for either read or write (or both). The software can then match the socket with the task (request) that it is performing (servicing) and figure out what data is to be written to the socket, and what is to be read from the socket, and where to pass it.

After all sockets have been handled, sockets that ended up indicating a blocking state are then set in the descriptor array and `select()` is called again.

This asynchronous I/O engine architecture is employed by the Harvest [6] and Squid [7] proxy servers.

MIXED ASYNCHRONOUS I/O WITH THREADS ARCHITECTURE

Asynchronous I/O can be combined with the multithreaded architecture—and it actually simplifies the implementation significantly. In this design one thread runs the asynchronous I/O engine (the *I/O worker thread*), while the remaining *worker threads* handle requests in the normal fashion. However, once they reach the point of simple data pumping between two sockets, they pass the socket descriptors to the I/O worker thread.

This way the regular worker threads can handle the more complex steps of request processing which may block—such as authentication or custom API functions—and are thus harder to rearchitect to be completely non-blocking. Only once these steps are completed is the request processing passed to the asynchronous I/O worker thread which will take over the processing for the more mechanical data pumping part.

This is ideal—the longest (wallclock) time is spent doing I/O while it usually requires only little CPU cycles but would cause a lot of context switches (two or more for every new buffer of data received). The first part of the request processing consists of various mappings, checkings, authentication, authorization, and cache lookup, all of which are harder to implement with the non-blocking I/O model—so it is natural to perform these initial steps in a dedicated worker thread which is allowed to block.

This mixed thread and asynchronous architecture model is used by, for example, the Netscape Enterprise Server.

SUMMARY

This chapter concludes the overview part of this book. The following parts study each major area of proxy server operation: protocols, caching, performance, filtering, monitoring, access control, and security. You do not have to proceed in this order; you may read the parts and chapters you are interested in, and leave the rest for reference. However, the next chapter on the HTTP protocol is recommended reading in order to get an understanding about how HTTP actually works, and how the various proxy server features relate to the HTTP protocol.

Endnotes

1. "Forking" means the creation of a new process in UNIX. It is accomplished via the `fork()` system call.

2. Among the first Web servers were CERN `httpd` and NCSA `httpd`, both forking UNIX servers. CERN `httpd` could act as a proxy server as well.

3. The term "process mob" comes from having this "mob," or crowd, of processes that are all competing to grab and service new connections. It was introduced for Web servers by Netscape's Netsite Server 1.0 in 1994.

4. The non-blocking I/O for a socket descriptor is enabled using `ioctl()`, by setting the `FIONBIO` attribute.

5. Return value `-1`, with `errno` set to `EWOULDBLOCK`.

6. `http://excalibur.usc.edu.`

7. `http://squid.nlanr.net.`