



Web Server Software Architectures

Daniel A. Menascé • George Mason University • menasce@cs.gmu.edu

Web site scalability depends on several things – workload characteristics,¹ security mechanisms,² Web cluster architectures³ – as I’ve discussed in previous issues. Another important item that can affect a site’s performance and scalability is the Web server software architecture.

In this column, I provide a classification of Web server architectures, offer a quantitative analysis of some possible software architectural options, and discuss the importance of software contention on overall response time.

Software Architecture Schemes

A Web server’s software architecture can affect performance significantly. Two dimensions generally characterize the architecture: the processing model and pool-size behavior. The processing model describes the type of process or threading model used to support a Web server operation; pool-size behavior specifies how the size of the pool of processes or threads varies over time and with workload intensity.

Processing Models

The main options for a processing model are process-based, thread-based, or a hybrid of the two. Software architectures on process-based servers consist of multiple single-threaded processes, each of which handles one request at a time (see Figure 1a). We can see implementations of this model, also called *prefork* in Apache, in the Unix version of Apache version 1.3, and in Apache version 2.0’s multiprocessing (MPM) *prefork* module (see also <http://httpd.apache.org/docs-2.0/mod/prefork.html>). In a thread-based architecture, the server consists of a single multithreaded server;

each thread handles one request at a time (see Figure 1b). We can see an implementation of this model in Windows NT’s version of Apache 1.3. The hybrid model consists of multiple multithreaded processes, with each thread of any process handling one request at a time (see Figure 1c). The Apache 2.0 Worker MPM implements an example of this type of approach (see <http://httpd.apache.org/docs-2.0/mod/worker.html>).

One advantage of a process-based architecture is stability. The crash of any process generally does not affect the others, so the Web server continues to operate and serve other requests even when one of its processes must be killed and restarted. The architecture’s drawbacks relate to performance: creating and killing processes overloads the Web server, mainly because of address-space management operations. Moreover, high-volume Web sites require many processes, which leads to non-negligible memory requirements and increased context-switching overhead (see <http://httpd.apache.org/docs/misc/perf-tuning.html#preforking>).

A thread-based architecture is not as stable as a process-based one. A single malfunctioning thread can bring the entire Web server down because all threads share the same address space. However, this type of server architecture’s memory requirements are much smaller than a process-based one’s. Spawning threads of the same process is much more efficient than forking processes because the new threads don’t need additional address space. One additional advantage is that the various threads can easily share data structures such as caches.

The hybrid architecture combines the advantages of both methods and reduces their disadvantages. For example, suppose that a Web server

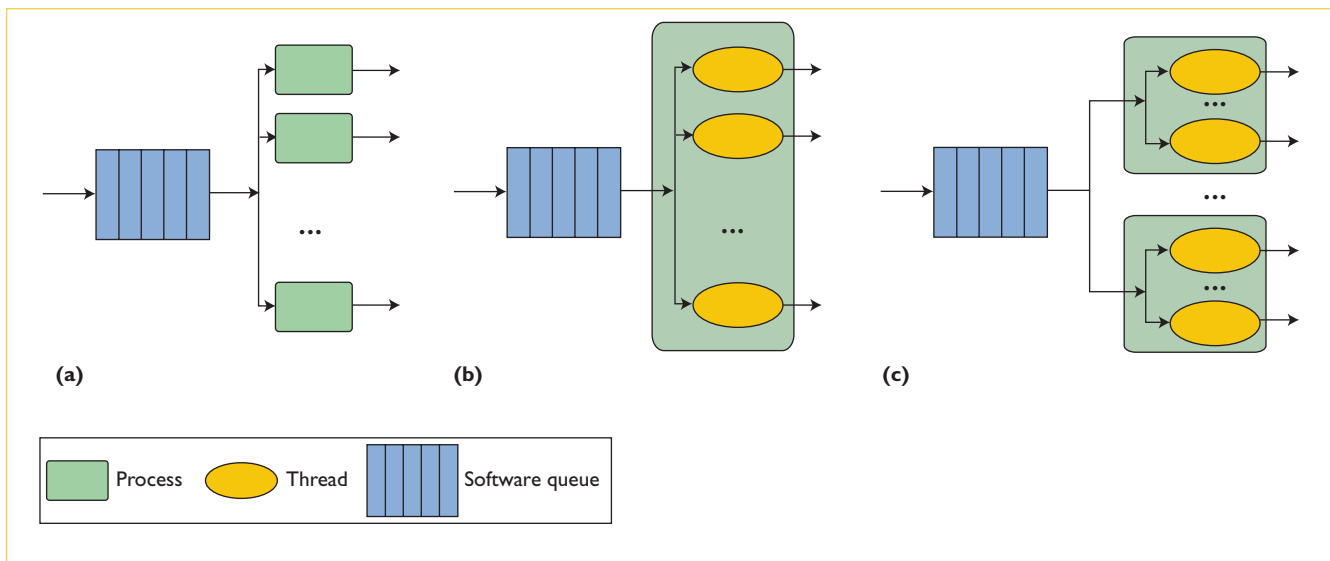


Figure 1. Web server processing models. (a) Process-based servers have multiple single-threaded processes, (b) thread-based servers have a single multithreaded process, and (c) hybrid servers use multiple multithreaded processes.

has p processes with n threads each. So, up to $p \times n$ requests can execute simultaneously. If a thread crashes, it can bring down the process in which it runs, but all other $(p - 1) \times n$ threads continue to process their requests. Less memory is required in this approach to handle $p \times n$ concurrent requests than to run the same number of requests in the process-based architecture.

Pool-Size Behavior

The other dimension of Web-server software architecture is the process or thread's pool-size behavior. Two approaches are possible: static or dynamic.

For static pools, the Web server creates a fixed number of processes or threads at startup time. Consider a process-based server and assume that p processes are created when the server is started. If a request arrives and finds p requests being processed, it waits in a queue until one of the requests completes execution (see Figure 2's queue). As the arrival rate of requests to a Web server increases, the queue for processes or threads increases, which increases a request's response time. If server load is low, most of the processes or threads will be idle.

For dynamic pools, the number of

processes or threads varies with load. As load increases, so does pool size, letting more requests be processed concurrently and reducing queue. In periods of low loads, the number of processes or threads reduces to free up more memory.

Apache provides an example of a dynamic pool-size implementation. The Web server's parent process initially forks a number p of child processes, established as a configuration parameter. The child processes handle requests directly; the parent process only monitors the load to decide if processes should be forked or killed. Another configuration parameter determines the minimum number m of idle processes (those processes that aren't handling requests). As load increases, the number of idle processes could fall below its minimum value m ; if so, the main process creates more processes. If the number of idle processes exceeds a value set in the configuration file, Apache's parent process kills the excess child processes.

You can also regulate pool size in Apache's configuration file by specifying the maximum number of requests a child can process before it dies. Once a process has reached the parameter-specified value, it exits. You

can also specify an infinite value for the parameter (by making the parameter value equal to zero). An infinite or high value is appropriate for high-volume Web sites subject to bursty traffic, whereas a lower value is more appropriate for low-volume Web sites. One advantage of limiting a process's lifetime is that it reduces the amount of lost and unusable memory that accumulates via memory leaks.

Performance Considerations

We can decompose a Web server request's total response time into the following components:⁴

- *Service time*, which is the total time a request spends at the physical-resource level (such as CPU and disk). This time does not include any time spent waiting to use any of the Web server's physical resources.
- *Physical contention*, which is the total time a request spends waiting to use any of the Web server's physical resources.
- *Software contention*, which is the total time a request spends waiting in a queue for a process or thread to become available to handle the request.

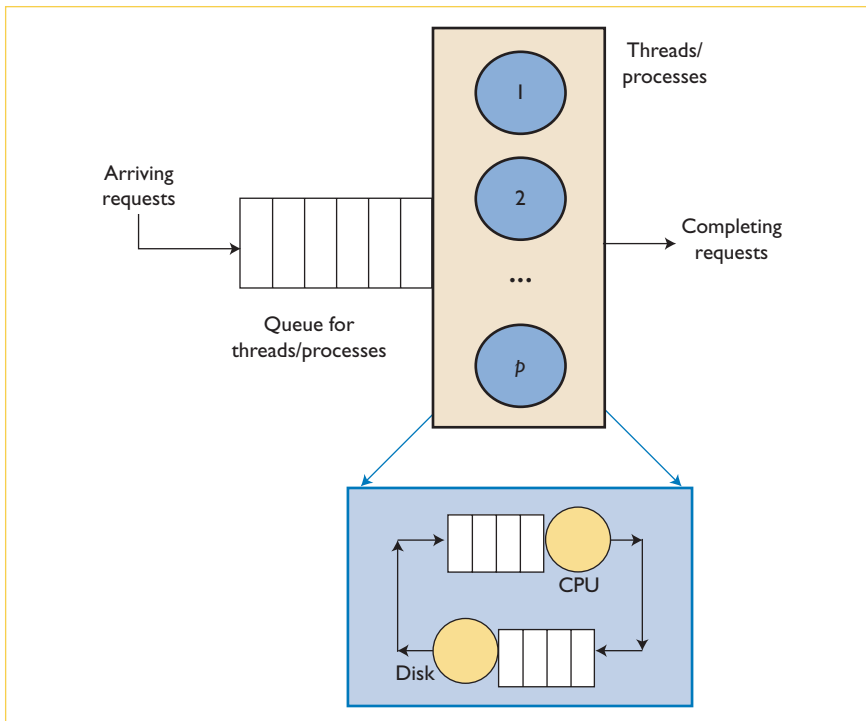


Figure 2. Software and physical contention. While a request is waiting to be processed, it doesn't use the CPU or I/O or any other physical resources.

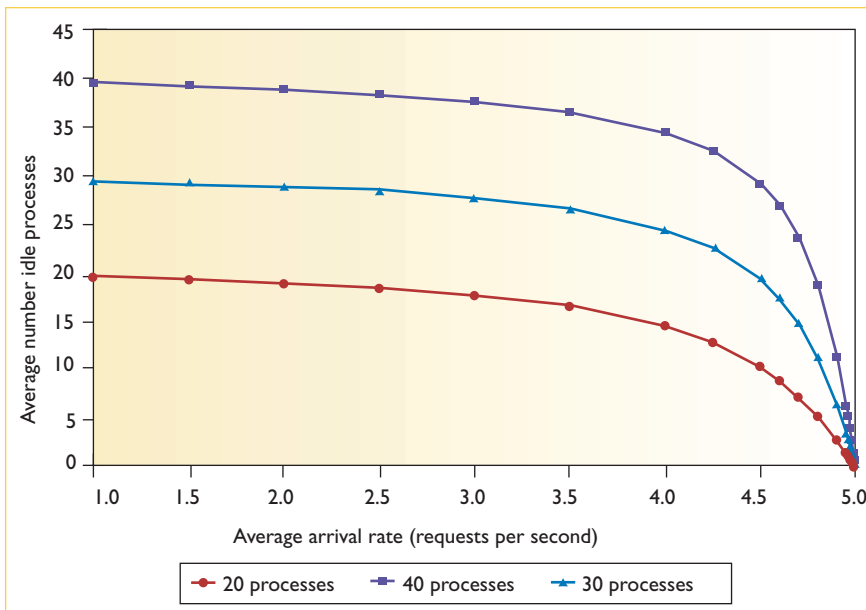


Figure 3. Average number of idle processes versus the average arrival rate of requests. At low loads, almost all processes tend to be idle. At high loads, the number of idle processes falls to zero very quickly.

As Figure 2 illustrates, when a request waits for a software resource (such as a thread or process), it is not using or waiting for any physical resource. During processing, a request

typically alternates between periods in which it is using a physical resource and waiting to use one.

To explore some of Figure 2's architecture's performance characteristics,

we use the results of a combined Markov chain model with a queuing network (QN) model.⁵ The QN model computes the Web server's throughput $X_0(k)$, $k = 1, \dots, p$ given the incoming requests' service demands on physical resources. In other words, the QN models the Web server's physical resources.

Next, we can use a Markov chain model to represent the software architecture. In this model, the state k of the Markov chain represents the number of requests in the system (either waiting for a process or being handled by one). Transitions from state k to state $(k + 1)$ occur at the rate at which new requests arrive at the server. Transitions from state k to state $(k - 1)$ indicate request completion. The completion rate is given by $X_0(k)$ for states $k = 1, \dots, p$ and $X_0(p)$ for all states $k > p$.

Figure 3 shows the average number of idle processes (those not handling requests) as a function of the average arrival rate of requests for three different values of the number of active processes p . The figure indicates, as expected, that for very light loads, the number of idle processes is almost equal to the total number of processes. As load increases, the average number of idle processes decreases slightly at the beginning and precipitously as the arrival rate achieves its maximum possible value, which is equal to $X_0(p)$. This maximum value, which is equal to five requests per second in Figure 3, represents the point at which the Web server becomes unstable – that is, the queue for processes grows without bounds.

Figure 3 also indicates how a Web server with a dynamic pool size can vary the size of its pool of processes as load increases. Consider Table 1's scenarios and assume that we want to have an average of 10 to 11 idle processes to handle surges in requests with a good response time. In scenario 1, when the arrival rate is 4.5 requests/sec, the average number of idle processes is close to 10 when there are 20 processes. If the arrival rate increases to 4.8 requests/sec (scenario 2), the average number of idle processes decreases to 5.3. To keep the num-

Table 1. Scenarios of different numbers of idle processes versus arrival rate variations.

Scenario	Number of processes	Arrival rate (requests per second)	Average number of idle processes
1	20	4.5	10.3
2	20	4.8	5.3
3	30	4.8	11.3
4	30	4.9	6.6
5	40	4.9	11.4

ber of idle processes at around 10 to 11, we must increase the number of processes to 30. Suppose now that the average arrival rate increases to 4.9 requests/sec (scenario 4). The average number of idle processes falls to 6.6. We must then fork 10 additional processes (scenario 5) to restore the average number of idle processes to a value close to 11.

Final Remarks

Software contention and architectures can significantly affect Web server performance. Poorly designed and configured software architectures might even generate high response times while the physical resources display low utilization.

Figure 3 and Table 1’s scenarios indicate that computer systems, Web servers included, could use analytic

performance models as a guide to dynamically adjust configuration parameters such as the number of active processes.⁶ This line of research has potentially important practical applications for complex, high-volume Web and e-commerce sites for which it is virtually impossible to have humans adjust configuration parameters to meet QoS goals.⁷ □

References

1. D.A. Menascé, “Workload Characterization,” *IEEE Internet Computing*, vol. 7, no. 5, 2003, pp. 89–92.
2. D.A. Menascé, “Security Performance,” *IEEE Internet Computing*, vol. 7, no. 3, 2003, pp. 84–87.
3. D.A. Menascé, “Trade-Offs in Designing Web Clusters,” *IEEE Internet Computing*, vol. 6, no. 5, 2002, pp. 76–80.
4. D.A. Menascé and V.A.F. Almeida, *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*, Prentice

Hall, 2000.

5. D.A. Menascé and V.A.F. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*, Prentice Hall, 2002.
6. D.A. Menascé and M. Bennani, “On the Use of Performance Models to Design Self-Managing Computer Systems,” *Proc. 2003 Computer Measurement Group Conf.*, Computer Measurement Group, 2003.
7. D.A. Menascé, “Automatic QoS Control,” *IEEE Internet Computing*, vol. 7, no. 1, 2003, pp. 92–95.

Daniel A. Menascé is a professor of computer science, the codirector of the E-Center for E-Business, and the director of the MS in E-Commerce program at George Mason University. He received a PhD in computer science from UCLA and wrote *Capacity Planning for Web Services* and *Scaling for E-Business* (Prentice Hall, 2002 and 2000). He is a fellow of the ACM and a recipient of the A.A. Michelson Award from the Computer Measurement Group.

IEEE Transactions on Mobile Computing

A revolutionary new quarterly journal that seeks out and delivers the very best peer-reviewed research results on mobility of users, systems, data, computing information organization and access, services, management, and applications. *IEEE Transactions on Mobile Computing* gives you remarkable breadth and depth of coverage ...



- Architectures**
- Support Services**
- Algorithm/Protocol Design and Analysis**
- Mobile Environment**
- Mobile Communication Systems**
- Applications**
- Emerging Technologies**



To subscribe:
<http://computer.org/tmc>
 or call
 USA and CANADA:
+1 800 678 4333
 WORLDWIDE:
+1 732 981 0060