

# Performance tuning considerations in your application server environment

Learn what makes a Web application tick, and how to make it tick faster

Skill Level: Intermediate

[Sean Walberg \(sean@ertw.com\)](mailto:sean@ertw.com)

Network engineer

P.Eng

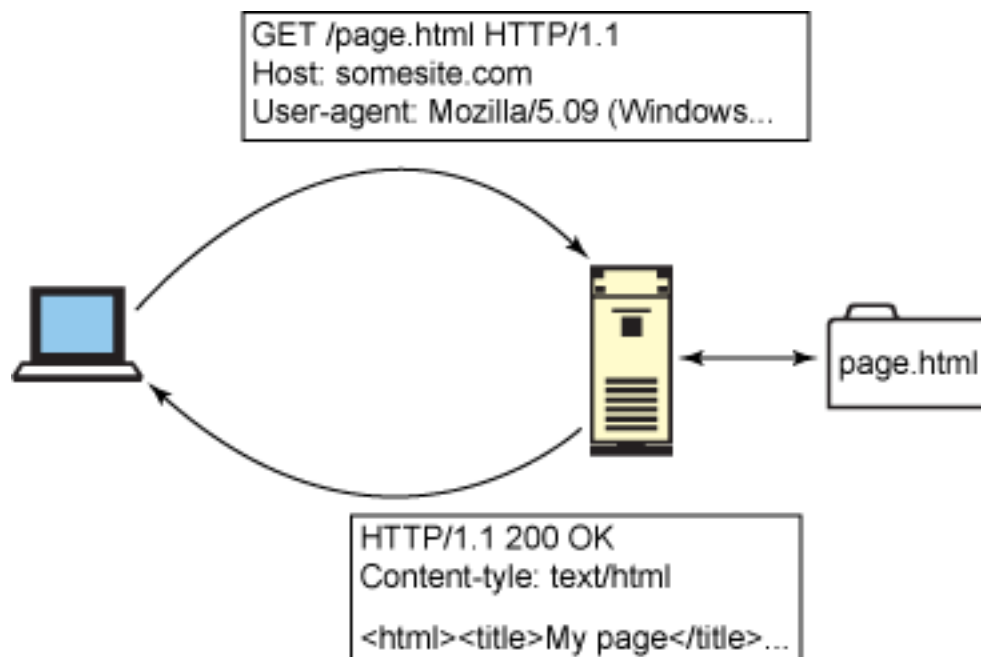
27 Jan 2009

Understand how the various components of a Web application interact, and where you might find performance bottlenecks. Both developers and administrators will benefit from knowing this because performance is everyone's responsibility.

Dynamic Web applications enable you to make vast stores of information instantly accessible to your users through a familiar interface. As your applications become more popular, however, you may find it harder to serve your requests as fast as you could earlier. An understanding of how Web requests are served, along with a grasp of the do's and don't's of Web application development, will ensure less trouble later on.

A static Web request, such as the one shown in Figure 1, is easy to grasp. A client connects to the server, usually on TCP port 80, and makes a simple request using the HTTP protocol.

**Figure 1. A client requesting a static file over HTTP**



The server parses the request, which is mapped to a file on a filesystem. The server then sends the client some headers that describe the payload (such as a Web page or image), and finally sends the file along to the client.

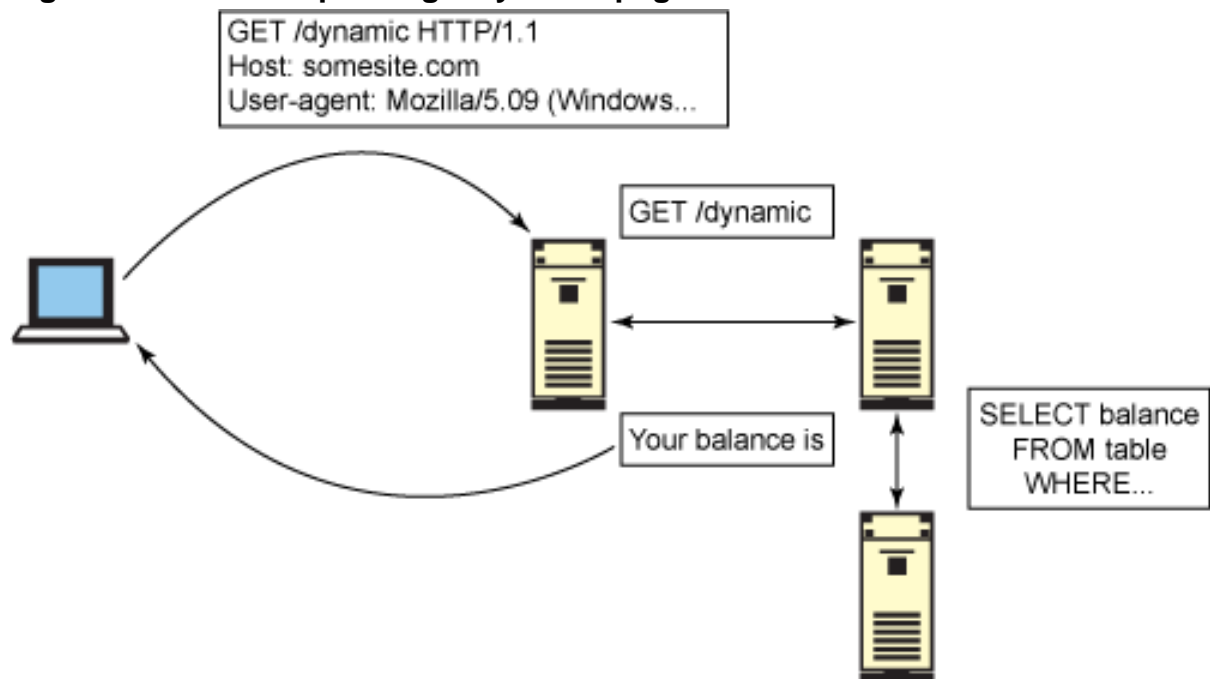
There are few bottlenecks in the above scenario. If the requests are varied enough to make ineffective use of the operating system's disk cache, then the server's disks will become busy and at some point slow the whole process down. If the network pipe feeding the clients becomes saturated, then all clients will suffer. But these conditions aside, the "receive request, send file" process is fairly efficient.

You could get a rough idea of the performance of the static server by making some assumptions. With a request service time of 10ms (largely constrained by having to seek disk heads), you can expect to serve about 100 requests per second before the disks become saturated ( $10\text{msec/request} / 1 \text{ second} = 100 \text{ requests/second}$ ). With a 10K document you would be serving about 8mbit/sec of Web traffic ( $100 \text{ requests/second} * 10 \text{ KBytes/request} * 8\text{bits/byte}$ ). If files can be served from memory cache, then that decreases the average service time, and hence increases the number of connections per second the server can handle. Should you have some real numbers for your disk service time or average request latency, you can plug them in to the calculations above to get better performance estimates.

Now that the capacity of a server has been shown to be the inverse of the average request service time, it follows that doubling the service time halves the server's capacity in terms of number of connections served per second. With this in mind, turn your attention to dynamic applications.

The flow of a dynamic application depends on the application, but generally looks like Figure 2.

**Figure 2. A client requesting a dynamic page over HTTP**



Much like the client from the previous example, the client from [Figure 2](#) starts the process by making a request. Nothing in the request differentiates a static request from a dynamic request (sometimes, extensions like `.php` or `.cgi` give you a hint, but they can be misleading). It is up to the Web server to determine what to do with the request.

In [Figure 2](#), the request is destined to an application server, such as a Solaris system running a Java™ application. The application server does some processing, but then consults a database for more information. With this information in hand, the application server generates an HTML page, which is relayed to the client by the Web server. The service time for this request is now the sum of the components. If the database access cost 7ms, the application server 13ms, and the Web server 5ms, then the service time of the Web page becomes 25ms. Using the inverse rule learned above, the capacity of the components are 142, 77, and 200 requests per second, respectively. The bottleneck is then the application server, which limits this system to 77 connections per second before the Web server is forced to wait, and to queue connections.

It is important to note, though, that just because the system can dispatch 77 connections per second and that a single connection takes 25ms to process, each application user will not have their requests served in that 25ms. Each component can only process one connection at a time, so at peak load, requests will have to wait their turn to get on the CPU of each step. In the example above, the average request time ends up being over 1.1 seconds once you factor in the queuing time and the 25ms processing time. See the [Resources](#) for information on solving these queuing problems.

From these examples, you can draw several conclusions:

- Putting more steps between the user and the final page makes things slower and decreases system capacity.
- This effect can be noticeable, especially as the page request rate increases.
- The architecture decisions made at the beginning of a project also affect the ability of a site to handle loads.

The rest of this article will reinforce these points.

## N-tier architectures for dynamic sites

The architecture of applications, including dynamic Web applications, is often described in terms of tiers. A static site can be considered to have *one tier* -- the Web server. If you then had a scripting language running in-process with the Web server, such as PHP, that connected to a database, this would be described as *two tiers*. The example from the previous section has *three tiers*, which are the front-end Web server, the application server, and the database.

Depending on who you are talking to, a single piece of software might consist of multiple tiers. For example, a PHP script might use a template engine to separate business logic from the presentation, and be considered two separate tiers. A Java application might have a Java servlet performing presentation tasks, which talks with an Enterprise Java Bean (EJB) to perform business logic, which then connects to a database for further information. Therefore one 3-tier architecture may look different from another, especially when different toolsets are involved.

### Common architectures

Even though an application architecture varies from application to application, some common trends emerge. Generally, an application needs the functionality of four tiers:

- A client tier
- A presentation tier
- A business logic tier
- A data tier

In a Web application, the client tier is handled by the Web browser. The browser renders HTML and executes Javascript (and, optionally, Javaapplets, ActiveX, or Flash applets) in order to show and collect information from the user. The

presentation tier is the interface from the servers to the client, and is responsible for formatting the output so that it can be displayed on the client. The business logic tier enforces the rules of the business, such as calculations and workflow, that drive the application. Finally, the data access layer is a persistent data store, such as a database or file storage.

Most applications will need functionality from each of these four tiers, even though they may not implement them distinctly and completely.

Another popular architecture is **Model-View-Controller**, which is a pattern for separating components of an application. In MVC, the model encapsulates the business logic tier, and together with the framework, the data tier. The view handles the presentation of data to the client. The controller's job is to control the application flow.

## Scaling the tiers

To scale a Web application means to grow it so that it can handle more traffic. One aspect of scaling is how hardware is deployed to match demand. Another aspect is how the application responds to this new hardware environment. Conceptually, it is easy to imagine throwing a bigger server at the problem, but the nature of the application may be such that other bottlenecks present themselves first. Breaking down the application into a series of tiers helps to reduce the scope of the problem, and allows for easier scaling.

Application bottlenecks aside for now, scaling the application's hardware is usually described as being done **horizontally** or **vertically**. Horizontal scaling means that scaling is done by adding more servers to a tier. The earlier example that had an application server bottleneck of 77 requests per second might be solved by a second application server, with the load shared between the two. This would give a theoretical capacity of 154 requests per second, pushing the bottleneck to the database.

Vertical scaling, on the other hand, means that a bigger machine is used. You could use a larger machine that could run two instances of the application server, or one that could run requests in half the time.

At this point it would be tempting to dismiss vertical scaling entirely because it is usually cheaper to buy more smaller machines than it is to continually buy bigger servers. However, there are many cases where vertical scaling is the better option. If you have something like an IBM® Power® server that supports partitioning of hardware through logical partitions (LPAR), you may already have spare capacity to add to your application server tier.

Your application's requirements may also drive you towards vertical partitioning. On one server it is easy to share the user's session state in a shared memory segment.

Once you have two servers, you need to find a new way to share state, such as a database. The database will be slower than the memory access, so two servers will not be twice as fast as one.

Databases are another example of a place where vertical scaling is often used. To split your dataset across different servers requires a lot of work at the application level, such as joining columns across two databases and making sure data is consistent. It would be far easier to build a bigger database server, and push off the need to rebuild your application to support the sharding of data.

## Modeling a Web application in terms of queues

From the previous discussion on application architectures, it becomes apparent that a Web request passes through different phases, each phase taking a certain amount of time to execute. Requests line up to pass through each of these steps, and once the step has completed, the request lines up for the next step. Each of these steps is much like how people line up at a grocery store to go through the check out.

A Web application can be modeled in terms of these steps, called queues. Each component of your application is a queue. A typical WebSphere application is shown in Figure 3, drawn as a series of queues.

**Figure 3. A WebSphere application shown as a queuing network**

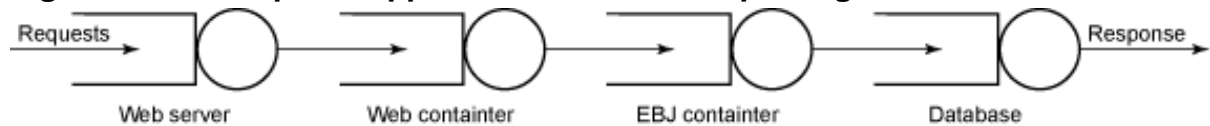


Figure 3 shows how requests wait to be serviced by the Web server, then they wait for the Web container, and so forth. If the rate of requests entering a particular queue exceeds the rate at which the queue can process requests, then the requests back up. When requests back up, the service time is unpredictable, and your users will be seeing stalled browser sessions. The queuing in Figure 3 represents the worst-case scenario, because some requests will be satisfied by the Web server, or without having to hit the database.

Queues are common in the UNIX® environment. The operating system queues disk requests when the applications issue them faster than the disks can return the data, optionally reordering them to minimize seeking. The run queue, which holds an ordered list of processes waiting to run, is another queue. Applications wait for their turn on a limited resource (the CPU) before being served.

Tuning the queues is therefore a balancing act. Too small, and you'll drop users when you still have capacity. Too big, and you'll try to serve more users than your system can handle, causing poor performance for everyone.

To complicate the matter, these queuing slots aren't free. There is a memory

overhead to hold the spot, and in the case of an application server, memory for the thread that's handling the request. It is then rarely a good idea to allow yourself to queue at the application server. Instead, the recommended approach is to queue any users before the application server, such as on the Web server. This means that the Web server keeps the connection open with the Web client and issues the request once the next available application server slot is freed. The application server is tuned to handle only enough requests per second that it can dispatch in a timely manner.

IBM has documented their recommended approach for laying out a Web application, along with methods to tune various queues. Of note, though, is their guidance that you should avoid queuing inside WebSphere. What this means is that you want to only have requests that you are able to immediately service directed at the WebSphere application server. Your Web server (or even better, a proxy server in front of the Web server) should be holding on to the excess connections as they wait their turn to be processed. This ensures that the heavier application server queues are able to spend their time servicing a limited number of requests rather than trying to serve everyone at once.

## Notes for developers

As a developer, there are general principles that you can follow in your applications to help them scale. These are high-level things that are applicable to most Web applications.

### Instrumentation

Your application should have some way of providing measurements back to a collection system (even if the collection system is just a log file). These measurements could be regarding how often a certain function in your application is accessed, or how long a request took to handle. This won't make your application any faster, but it helps you understand when things are running slowly, and which parts of your code are taking the most time. Understanding when certain functions are being called can help you correlate events you see on your system (CPU spikes, high disk activity) with something meaningful to your application (image uploading).

Being able to know what's going on in your site is the key to being able to scale it. Parts of your code that you think are not properly optimized may not end up being a problem. Only proper measurement can tell you what's going on.

### Sessions

The Web is inherently stateless. Each request you make is independent on the last one. Applications, however, are usually stateful. You must log in to an application to identify yourself, you might have a shopping cart for the life of your visit, or you might

just be filling in a profile for later use. Keeping track of sessions can be an expensive operation, especially as you grow beyond one server.

A Web application running on a single server can keep session information in memory, which can be accessed by any instance of the Web application running on the server. Users are usually given a token that identifies the session in this memory bucket. Consider what happens when a second application server is introduced. If the user's first request is to one server, and the second request is to the second server, then two separate sessions exist, and each is different.

The common solution to this problem is to store the session in a database instead of memory. The problem that this causes is that you've introduced another database read, and potentially a database write, per request. This database is also required by each Web application server.

One solution is to only use sessions where you need them. Instead of having your application load the session on each request, only have it load the session when the session is needed. This reduces the number of requests to the backend database.

Another approach is to encrypt the session data and send it back to the client, eliminating the need to store the session locally. You cannot store an infinite amount of data in the user's cookie, but RFC 2109 specifies that clients should be able to store at least 20 cookies per domain name, each a minimum of 4K bytes long.

If database back-ed sessions do prove to become a bottleneck for you, and you can't get rid of them, you should be able to split them onto separate databases, or even multiple databases. For example, you can have even session IDs on one database, an odd session IDs on another.

## Caching

Changes in your data happen more in some parts of your application than they do others. A news site might only change its list of top-level categories once a month. Therefore, it would be wasteful to query a database on every request to get the latest list of categories. Similarly, a page representing a news story might only change a couple of times in its life, so there is no need to regenerate it on every request.

Caching means to store the results of an expensive request for later use. You could cache the list of categories, or even the whole page.

When considering caching, ask yourself "Does this information have to be fresh?" If not, it might be a candidate for caching. It might be important to be able to change the news story on a moment's notice, but even if you check for changes once a minute and serve from cache the rest of the time, it may be considered fresh enough.

A complementary method is to invalidate the cached entry when the underlying data changes. If the news story is changed, one of the actions performed on saving might be to remove the cached version. The next request would see that there is no cached version, and generate a new entry.

When caching, you must be careful of what happens when the cached entry expires, or is removed. If many requests are being made for the cached item, you will have many people regenerating the cached item when the cached entry expires. To get around this, you could only have the first request regenerate the cache, and every one else use the stale item until the new one is available.

*memcached* is a distributed memory caching system popular with applications deployed in UNIX environments. Servers run instances of the memcache daemon, which allocates a block of RAM that's accessible over a simple network protocol. Applications wishing to store or retrieve a piece of data within memcache first hash the key, which tells them which server out of the memcache pool to use. The data can be checked or stored by connecting to the server, which is much quicker than a disk or database access.

When looking for things to cache, you might also consider asking if the information is really needed in the first place. Do you need to display the user's shopping cart on every page? What about just the total? What about a simple link to "view the contents of your cart?"

*Edge-Side Includes* (ESI) is a markup language that allows you to break apart your Web page into separate, cachable, entities. Your application is responsible for generating the HTML document that contains the ESI markup, and also for generating the components. The proxy cache in front of your Web application reassembles the final document based on the parts, and is responsible for caching some components and making the request for others. Listing 1 shows an example of an ESI document.

### Listing 1. ESI example

```
<html>
<head>
</head>
<body>
<p>This is static content</p>
<esi:include src="/stories/123" />
<p>The line above just told the proxy to request /stories/123 and insert
it in the middle of the page </p>
</body>
</html>
```

Though a very simple example, [Listing 1](#) shows how two documents can be spliced together, each with their own caching rules.

## Asynchronous processing

Loosely tied to the "Does this information have to be fresh?" question is, "Does the information have to be updated by the time this request finishes?" There are many times that you can take the data submitted by the user and delay the processing a matter of seconds, rather than process the information while the user waits for his page to load. This is called asynchronous processing. One common approach is to have the application send the data into a message queue, such as IBM WebSphere MQ, to be processed when the resources are available. This lets you immediately return a page to the user, even though the results of the data processing may not be known.

Consider an e-commerce application where the user submits an order. You may decide that it is important to immediately return the credit card verification, but you may not need to get confirmation from your ordering system that all the components are available. Instead, you could put the order in a queue to be processed, which might happen within seconds. If an error happens, this can be relayed to the user through email, or even inserted into her session should she still be on your site. Another example is reporting. Rather than wait for the report to be generated, you could return a "please check the reports page in a few minutes" message, while the report is farmed out to a separate server asynchronously.

## Summary

Applications are usually written in a tiered fashion. Presentation logic is separated from business logic, which is further separated from the persistent storage. This allows for easier maintainability of the code, but introduces overhead. Scaling an application involves understanding the flow of data in this tiered environment, and looking for bottlenecks.

Techniques such as caching and asynchronous processing can reduce the workload on the application by reusing previous work or by farming the bulk of the work to another machine. Providing instrumentation hooks in your application lets you zero in on your hotspots in near real time.

The application server environment operates much like a queuing network, and it is important to manage the size of the queues so that one tier isn't overwhelmed by another. IBM recommends that you queue as far before the application server as possible, such as on the external Web server or proxy cache.

Applications rarely scale only by throwing more hardware at the problem. Usually these techniques must be put into place so that the new hardware can be used effectively.

# Resources

## Learn

- ["LPI exam 301 prep, Topic 306: Capacity planning"](#) (developerWorks, Apr 2008) has two sections on modeling a computer system as a series of queues, and using some Perl code to solve the queuing equations.
- [RFC 2109](#) describes how cookies are used to keep state in Web applications.
- [Edge Side Includes](#) can be used to split up the cost of page generation, and to cache fragments of a page.
- This document explains [what your computer does while you wait](#), using an Intel computer as an example. The article continues by comparing the cost of CPU cycles vs disk access, which is important to keep in mind when you're writing dynamic applications.
- [High Scalability](#) is an interesting blog that posts case studies about how well-known Web sites are built to scale. The site also looks at software that is designed to help with scaling.
- Wikipedia has some good architecture documentation, especially regarding [Three Tier Architecture](#) and the [Model-View-Controller Pattern](#).
- If you are working with WebSphere, you must read IBM's [WebSphere Application Server V6 Scalability and Performance Handbook](#). It's 1100 pages of advice on how to tune WebSphere and other components, and how to write your applications with scalability in mind.
- Another IBM RedBook is the [Running IBM Websphere Application Server on System p and AIX](#) guide, which describes horizontal and vertical scaling as it applies to a WebSphere application. The document also discusses ways to tune your application server and operating system.
- Browse the [technology bookstore](#) for books on these and other technical topics.

## Get products and technologies

- [memcached](#) is used by many large sites to cache various objects in a distributed memory system, rather than having to read from disk or a database.
- Two Open Source front end proxy caches are [Varnish](#) and [Squid](#). They both support varying levels of Edge-Side Includes.
- Download [IBM product evaluation versions](#) and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

## Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

## About the author

Sean Walberg

Sean Walberg has been working with Linux and UNIX systems since 1994 in academic, corporate, and Internet service provider environments. He has written extensively about systems administration over the past several years. You can contact him at [sean@ertw.com](mailto:sean@ertw.com).