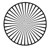
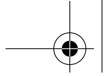
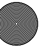
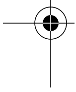
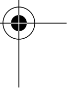


CHAPTER 7

Identifying Your Performance Problems

You have been assigned to improve the performance of your company's web service. The hardest thing is to get started. How should you tackle this task? And how do you sort out the insignificant issues and identify those that will make a difference once resolved?

In this chapter, we look at this problem from different angles. Only after you understand the problem should you start looking for solutions. Don't search for a solution before the problem has been precisely identified, or you'll end up wasting a lot of time concentrating on trivial issues. Instead, try to identify where you can make the biggest difference in performance.

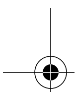
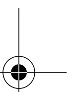
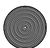
Note that in this book, we use the term "web service" to mean the whole aggregate that provides the service: the machine, the network, and the software. Don't confuse this with web services such as SOAP and XML-RPC.

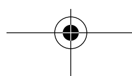
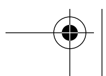
Looking at the Big Picture

To make the user's web-browsing experience as painless as possible, every effort must be made to wring the last drop of performance from the server. Many factors affect web site usability, but one of the most important is speed. (This applies to any web server, not just Apache.)

How do we measure the speed of a server? Since the user (and not the computer) is the one that interacts with the web site, one good speed measurement is the time that elapses between the moment the user clicks on a link or presses a Submit button, and the time when the resulting page is fully rendered in his browser.

The requests and resulting responses are broken into packets. Each packet has to make its own way from one machine to another, perhaps passing through many interconnection nodes. We must measure the time starting from when the request's first packet leaves our user's machine to when the reply's last packet arrives back there.

A request may be made up of several packets, and a response may contain a few hundred (typical for a GET request). Remember that the Internet standard for Maximum Transmission Unit (MTU), which is the size of a TCP/IP packet, is 576 bytes. While the packet size can be 1,500 bytes or more, if it crosses a network where the MTU is 576, it will be broken into smaller packets.

It is also possible that a request will be made up of many more packets than its response (typical for a POST request where an uploaded file is followed by a short confirmation response). Therefore, it is important to optimize the handling of both the input and the output.

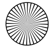
A web server is only one of the entities the packets see on their journey. If we follow them from browser to server and back again, they may travel via different routes through many different entities. For example, here is the route the packets may go through to reach *perl.apache.org* from our machine:

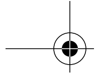
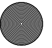
```
% /usr/sbin/traceroute -n perl.apache.org

traceroute to perl.apache.org (63.251.56.142), 30 hops max, 38 byte packets
 1  10.0.0.1            0.847 ms    1.827 ms    0.817 ms
 2  165.21.104.1        7.628 ms   11.271 ms   12.646 ms
 3  165.21.78.37        8.613 ms    7.882 ms   12.479 ms
 4  202.166.127.28     10.131 ms    8.686 ms   12.163 ms
 5  203.208.145.125     9.033 ms    7.281 ms    9.930 ms
 6  203.208.172.30    225.319 ms  231.167 ms  234.747 ms
 7  203.208.172.46    252.473 ms           *  252.602 ms
 8  198.32.176.29     250.532 ms  251.693 ms  226.962 ms
 9  207.136.163.125   232.632 ms  231.504 ms  232.019 ms
10  206.132.110.98    225.417 ms  224.801 ms  252.480 ms
11  206.132.110.138   254.443 ms  225.056 ms  259.674 ms
12  64.209.88.54      227.754 ms  226.362 ms  253.664 ms
13  63.251.63.71      252.921 ms  252.573 ms  258.014 ms
14  64.125.132.18     237.191 ms  234.256 ms *
15  63.251.56.142     254.539 ms  252.895 ms  253.895 ms
```

As you can see, the packets travel through 14 gateways before they reach *perl.apache.org*. Each of the hops between these gateways may slow down the packet.

Before they are processed by the server, the packets may have to go through proxy servers, and if the request contains more than one packet, packets might arrive at the server by different routes and at different times. It is possible that some packets may arrive out of order, causing some that arrive earlier to have to wait for other packets before they can be reassembled into a chunk of the request message that can then be read by the server. The whole process is then repeated in the opposite direction as response packets travel back to the browser.

Even if you work hard to fine-tune your web server's performance, a slow Network Interface Card (NIC) or a slow network connection from your server might defeat it all. That is why it is important to think about the big picture and to be aware of possible bottlenecks between your server and the Web.



Of course, there is little you can do if the user has a slow connection. You might tune your scripts and web server to process incoming requests ultra quickly, so you will need only a small number of working servers, but even then you may find that the server processes are all busy waiting for slow clients to accept their responses.

There are techniques to cope with this. For example, you can compress the response before delivery. If you are delivering a pure text response, *gzip* compression will reduce the size of the sent text by two to five times.

You should analyze all the components involved when you try to create the best service for your users, not just the web server or the code that the web server executes.

```
              _____
                |
         A web service is
       like      a      car,
      if        one     of the
     parts or mechanisms  is  broken
     the car may ~ not ~  run  smoothly;
     it can even stop dead if pushed too
       far without  first  fixing  it.
          \___/         \___/
```

If you want to have success in the web service business, you should start worrying about the client's browsing experience, not only how good your code benchmarks are.

Asking the Right Questions

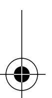
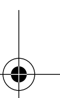
There is much more to the web service than writing the code, and firing the server to crunch this code. But before you specify a set of questions that will lead you to the coverage of the whole mechanism and not just a few of its components, it is hard to know what issues are to be checked, what components are to be watched, and what software is to be monitored. The better questions you ask, the better coverage you should have.

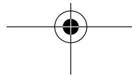
Let's raise a few questions and look at some possible answers.

Q: *How long does it take to process each request? What is the request distribution?*

A: Obviously you will have more than one script and handler, and each one might be called in different modes; the amount of processing to be done may be different in every case. Therefore, you should attempt to benchmark your code, using all the modes in which it can be executed. It is good to learn the average case, as well as to learn the edges—the worst and best cases.

It is also very important to find out the distribution of different requests relative to the total number of requests. You might have only two handlers: one very slow and the other very fast. If you optimize for the average case without finding out the request distribution, you might end up under-optimizing your server, if in fact the slow request handler has a much higher call rate than the fast one. Or you might



have your server over-optimized, if the slow handler is used much less frequently than the fast handler.

Remember that users can never be trusted not to do unexpected things such as uploading huge core dump files, messing with HTML forms, and supplying parameters and values you didn't consider. Which leads us to two things. First, it is not enough to test the code with automatic offline benchmarking, because chances are you will forget a few possible scenarios. You should try to log the requests and their execution times on the live server and watch the real picture. Secondly, after everything has been optimized, you should add a safety margin so your server won't be rendered unusable when heavily hit by the worst-case usage load.

Q: *How many requests can the server process simultaneously?*

A: The number of simultaneous requests you can handle is equal to the number of web server processes you can afford to run. This all translates to the amount of main memory (RAM) available to the web server. Note that we are not talking about the amount of RAM installed on your machine, since this number is misleading. Each machine is running many processes in addition to the web server processes. Most of these don't consume a lot of memory, but some do. It is possible that your web servers share the available RAM with big memory consumers such as SQL engines or proxy servers. The first step is to figure out what is the real amount of memory dedicated to your web server.

Q: *How many simultaneous requests is the site expected to service? What is the expected request rate?*

A: This question sounds similar to the previous one, but it is different in essence. You should know your server's abilities, but you also need to have a realistic estimate of the expected request rate.

Are you really expecting eight million hits per day? What is the expected peak load, and what kind of response time do you need to guarantee? Doing market research would probably help to identify the potential request rates, and the code you develop should be written in a scalable way, to allow you to add a few more machines to accommodate the possibility of rising demand.

Remember that whatever statistics you gathered during your last service analysis might change drastically when your site gains popularity. When you get a very high hit rate, in most cases the resource requirements grow exponentially, not linearly!

Also remember that whenever you apply code changes it is possible that the new code will be more resource-hungry than the previous code. The best case is when the new code requires fewer resources, but generally this is not the case.

If you machine runs the service perfectly well under normal loads, but the load is subject to occasional peaks—e.g., a product announcement or a special offer—it is possible to maintain performance without changing the web service at all. For

example, some services can be switched off temporarily to cope with a peak. Also avoid running heavy, non-urgent processes (backups, cron jobs, etc.) during the peak times.

Q: *Who are the users?*

A: Just as it is important for a public speaker to know her audience in order to provide a successful presentation and deliver the right points, it is important to know who your users are and what can be expected from them.

If you are administering an Intranet web service (internal to a company, publicly inaccessible), you can tell what connection speed most of your users have, the number of possible users, and therefore the maximum request rate. You can be sure that the service will not gain a sudden popularity that will drive the demand rate up exponentially. Since there are a known number of users in your company, you know the expected limit. You can optimize the Intranet web service for high-speed connections, but don't forget that some users might connect to the Intranet with a slower dial-up connection. Also, you probably know at what hours your users will use the service (unless your company has branches all over the world, which requires 24-hour server availability) and can optimize service during those hours.

If you are administering an Internet web service, your knowledge of your audience is very limited. Depending on your target audience, it can be possible to learn about usage patterns and obtain some numerical estimates of the possible demands. You can either attempt to do the research by yourself or hire professionals to do this work for you. There are companies who release various survey reports available for purchase.

Once your service is running in the ideal way, know what to expect by keeping up with the server statistics. This will allow you to identify possible growth trends. Certainly, most web services cannot stand the so-called *Slashdot Effect*, which happens when some very popular news service (Slashdot, for instance) releases an exotic report on your service and suddenly all readers of this news service are trying to hit your site. The effect can be a double-edged sword: on one side you gain free advertising, but on the other side your server may not be able to withstand the suddenly increased load. If that's the case, most clients may not succeed in getting through.

Just as with the Intranet server, it is possible that your users are all located in a given time zone (e.g., for a particular country-specific service), in which case you know that hardly any users will be hitting your service in the early morning. The peak will probably occur during late evening and early night hours, and you can optimize your service during these times.

Q: *How can we protect ourselves from the Slashdot Effect?*

A: Use mod_throttle. mod_throttle allows you to limit the use of your server based on different metrics, configurable per vhost/location/file. For example, you can limit requests for the URL */old_content* to a maximum of four connections per

second. Using mod_throttle will help you prioritize different parts of your server, allowing smart use of limited bandwidth and limiting the effect of spikes.

Q: *Does load balancing help in this area?*

A: Yes. Load balancing, using mod_backhand, Cisco LocalDirector, or similar products, lets you wring the most performance out of your servers by spreading the load across a group of servers.

Q: *How can we deal with the situation where we can afford only a limited amount of bandwidth but some of the service's content is large (e.g., streaming media or large files)?*

A: mod_bandwidth is a module for the Apache web server that enables the setting of server-wide or per-connection bandwidth limits, based on the directory, size of files, and remote IP/domain.

Also see Akamai, which allows you to cache large content in regionally specific areas (e.g., east/west coast in the U.S.).

The given list of questions is in no way complete, and each specific project will have a different set of questions and answers. Some will be retained from project to project; others will be replaced by new ones. Remember that this is not a one-size-fits-all glove. While partial functionality can generally be optimized using the same method, you will have to go through this question-and-answer process each time from scratch if you want to achieve the best performance.

References

- *http://slashdot.org/* is a site for geeks with news interesting to geeks. It has become very popular and gathers large crowds of people who read the posted articles and participate in various discussions. When a news story posted on this site appeals to a large number of Slashdot readers, the site mentioned in the news story often suddenly becomes a new mecca during the day the story was posted and the next few days. If the site's owner has just a small machine and never expected to gain such popularity in so little time, the server is generally unable to supply the demand and often dies. This is known as the Slashdot Effect.
- *Web Performance Tuning*, by Patrick Killelea (O'Reilly).
- The mod_throttle home page: *http://www.snert.com/Software/mod_throttle/*.
- The mod_bandwidth home page: *http://www.cohprog.com/mod_bandwidth.html*.
- The mod_backhand home page: *http://www.backhand.org/mod_backhand/*.