CHAPTER 9

Could They Do It?: Synthetic Monitoring

We measure end user experience using two complementary approaches: synthetic testing, which involves testing a website by simulating visitor requests; and real user monitoring (RUM), which involves watching actual user interactions with the site.

Basic synthetic testing of websites is easy to implement, and there are many free options available to operators of fledgling websites. It should be the first kind of performance and availability monitoring you deploy for any web application. While it can't give you the granularity and accountability that comes from watching actual users, it offers peace of mind and an understanding of the percentage of time your site is online and how long it takes to retrieve specific pages.

In this chapter, we'll look at some of the fundamentals of synthetic testing and how to leverage what you already know from your web analytics data to best configure synthetic tests.

The most basic distinction in synthetic testing is between internal tests run behind your own firewall and external tests run from locations around the Internet. While the bulk of this chapter will focus on synthetic testing done outside your data center by a third party, it's important to understand the basics of internal testing to know what data you already have on hand and don't need to use testing services for.

Monitoring Inside the Network

Internal tests are those you run within your own data center to ensure all of your machines are working properly. You can run simple, simulated transactions to each server to verify that everything's working, as shown in Figure 9-1. Many web operators rely on commercial monitoring software, such as HP Sitescope, or open source tools, like Nagios, to do this.

Because you're running your own test systems over a LAN connection with lots of spare capacity, you can afford to generate large numbers of tests every few seconds. This will

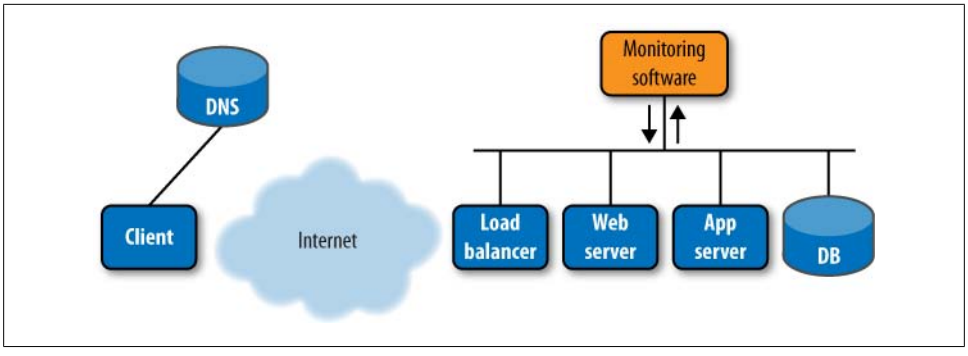


Figure 9-1. Internal testing of website infrastructure components

give you more complete "coverage" of your website, since you'll have smaller intervals between each test during which something can go wrong.

Internal testing is an essential tool for any IT operator. It may take the form of simple "Are you there?" up/down checks run every minute to each machine, or that of more comprehensive HTTP requests that check each machine's response for the correct content.

Using the Load Balancer to Test

For websites whose infrastructure includes a load balancer, a second—and increasingly common—option is to use this device to generate internal tests.

Load balancers provide redundancy by detecting server failures and taking the offending machines out of rotation. To do this, load balancers need to know when a server is broken. They determine this by running their own small tests, as shown in Figure 9-2. Since they're already testing each server, you can use load balancers for monitoring—to a degree. While they're good for up/down alerting, they won't provide long-term performance baselines and trending, although some monitoring tools can extract test results from them and graph them over time using programs like Cacti (*www.cacti.net/*) or MRTG.

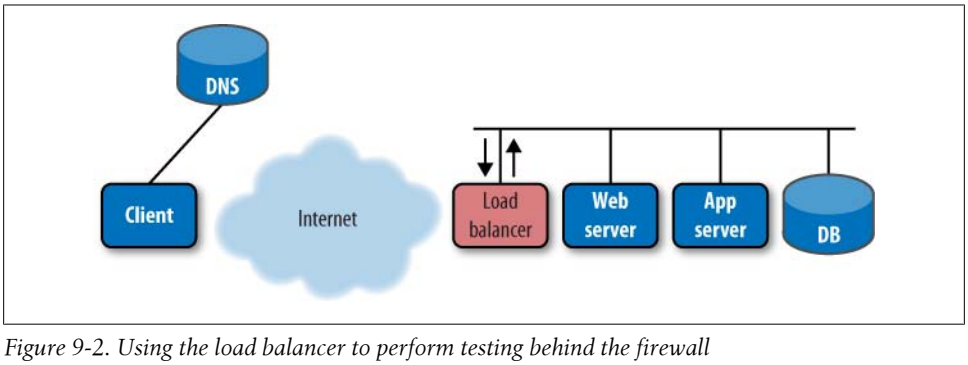


Figure 9-2. Using the load balancer to perform testing behind the firewall

A load balancer can monitor the network, TCP, and HTTP services on the machines across which it is distributing traffic. Any health check sent to a server comes back as "working" or "broken." Because they're constantly sending traffic to servers, load balancers are often the first to know when a bad response comes back. Some go as far as to inject JavaScript into pages as they go past in order to extract performance metrics from user visits.

You should always run internal tests. The load balancer's job is to hide broken servers from the outside world, and as a result, no external test will see a failed server when the load balancer is functioning properly. Internal tests fill in the gaps in your external monitoring, and because you run the tests yourself, you save money by reducing the number of external tests you need to pay for. However, internal device monitoring tools aren't able to properly simulate your visitors' conditions and shouldn't be used as a substitute for external monitoring.

Monitoring from Outside the Network

The Internet can fail in many creative and hard-to-pinpoint ways. Even though these aren't your fault, they're still your problem. To detect these problems, you need external tests that can watch your site from around the world.

A synthetic monitoring solution can:

- Alert you when your site is unreachable or unacceptably slow
- Detect localized outages limited to a region or a carrier
- Identify performance issues specific to a particular segment of your visitors, such as those using a certain web browser or a specific operating system
- Baseline your performance and availability around the world
- Predict whether you'll need to use content delivery networks when entering new markets
- Localize errors and slowdowns to a particular component of your infrastructure
- Generate additional traffic to test the capacity of your system or evaluate a new code release before it goes into production
- Test things you don't control, like mashup content or third-party web services
- Compare your performance to that of your competitors
- Estimate required capacity going forward by simulating load

A Cautionary Tale

A web startup we know had invested heavily in bottom-up device monitoring tools. It had also deployed several web analytics tools, and employees were confident they could

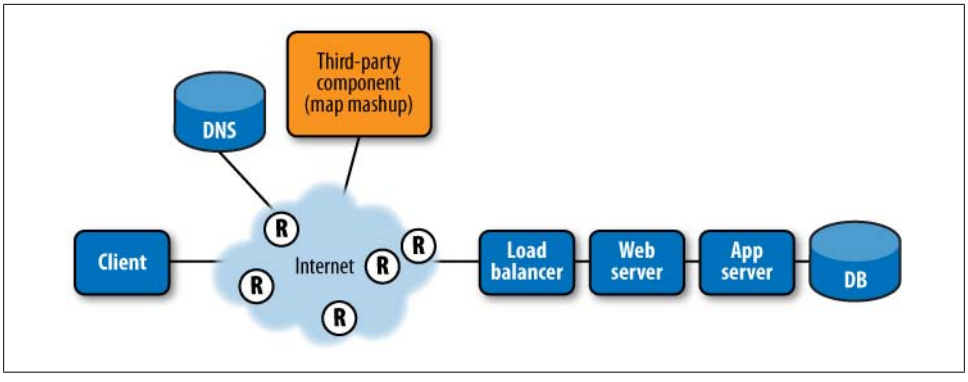


Figure 9-3. Some of the components on which a website depends—and which you need to monitor

finally sleep well at night, knowing that a series of well-designed alerts would notify them immediately if something went wrong.

The company had also been through a dramatic reduction in its operational staff. Management, emboldened by its recent investments in monitoring, decided that the humans weren't important now that the tools were in place.

Early one morning, one of the few remaining IT employees sent a panicked email message to the entire company to let them know that their site wasn't working. The IT team immediately brought up its dashboard, which showed that all servers were operating properly. The dashboard suggested that the infrastructure was, if anything, unusually healthy: web, application, and database servers were faster than normal.

Unfortunately, the company was under siege. A well-orchestrated denial-of-service attack had overloaded some of the core routers linking the company's data center to Europe, where nearly half of its customers were located.

The attack had a significant impact on revenues, costing the company a substantial amount in SLA refunds. That quarter, the company missed its revenue targets, and this ultimately resulted in delayed financing of their startup, at terms that were much less favorable for the founders and managers. Perhaps worse, management lost faith in the team's ability to properly monitor the infrastructure. Even though the problem was resolved in only three hours, the fact that it was discovered almost by accident was unforgivable.

What Can Go Wrong?

Testing your site from outside your own firewall means looking at the many things that can break between a visitor and your website. As you saw in Chapter 8, a web application relies on many components: DNS, routers, CDNs, load balancers, servers, third-party components, client-side scripts, and browser add-ons (Figure 9-3). A failure of any of these components will break or negatively impact the user's experience.

You need to verify the health of all these systems, all the time, from all locations that matter to your business.

Why Use a Service?

It would be expensive for you to set up and maintain dozens of servers in various countries and carriers around the world for the sole purpose of generating requests to your site. While some large organizations can afford their own testing networks, most companies rely on hosted synthetic testing services. There are several reasons for using a third-party service:

Cost
: It's cheaper than building and running your own servers.

No setup time
: These companies offer predefined reports and tools that make it easier to set up tests and report performance.

Viewed as impartial
: As independent organizations, the results of their tests are considered impartial and trusted—something you'll care about when you're using a performance report to resolve a dispute with a customer.

Competitive analysis
: The testing services can show you how your site stacks up against other businesses or even competitors.

Visibility into backbone health
: Because they have many points of presence, hosted services can test connections between their own servers and report on the health of carrier-to-carrier communications. Portals such as Keynote Systems' Internet Health Report (*www.internetpulse.net*) tell you which Tier 1 ISPs are currently experiencing issues. For example, in Figure 9-4, there is increased latency between Verizon and both AT&T and Sprint.

Synthetic testing services are relatively simple to use. Figure 9-5 shows how a basic service works.

1. You configure the test through a web user interface.
2. The service rolls out the test to nodes around the world.
3. The nodes run the test at regular intervals, recording performance.
4. The results are collected in reports you can view or receive by email.
5. If an error occurs, the nodes detect it, and may even capture a copy of the error.
6. Alerts are sent to a mobile device or email client.

Depending on the sophistication of the synthetic testing service, the system may perform additional diagnostics or capture the page error for review, which makes pinpointing and correcting the issue easier.

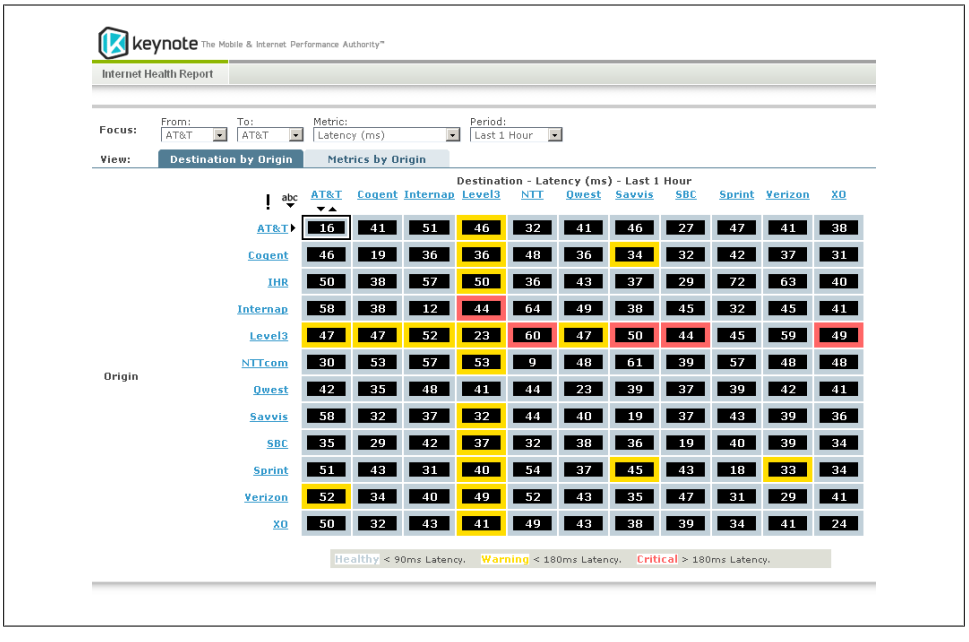


Figure 9-4. A Keynote.com Internet Health Report on latency between different monitoring locations

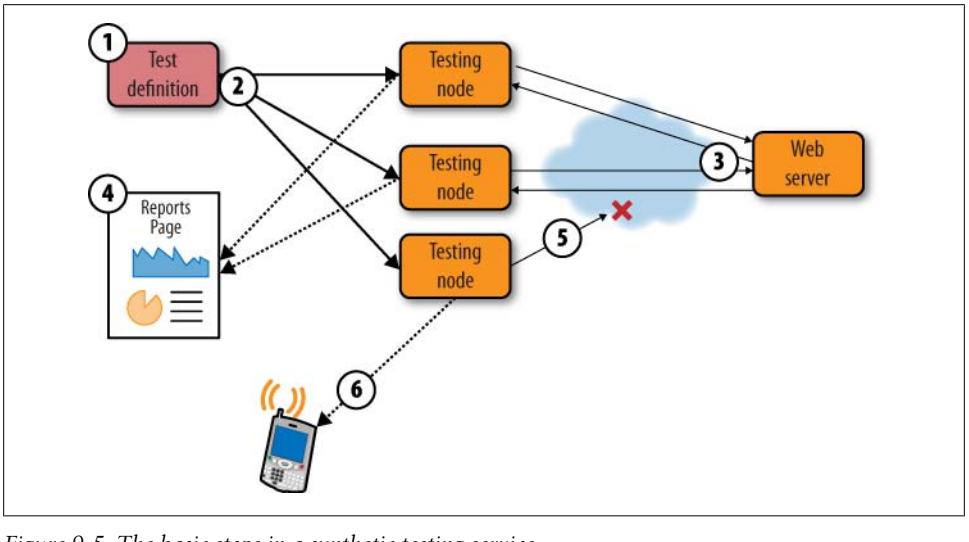


Figure 9-5. The basic steps in a synthetic testing service

Different Tests for Different Tiers

Different synthetic tests probe different layers of the Internet infrastructure we reviewed earlier. Each kind of test provides information on the health of a component or a service, and all are useful for determining where issues lie.

Testing DNS

As you've seen, DNS is an important source of latency for mashups and a common culprit when sites aren't accessible. DNS services are also a common point of attack for hackers, who try to "poison" domain name listings, which can misdirect visitors to other locations. In other words, you need to watch:

- The response time for DNS lookups
- DNS resolution of every site involved in building a page, not just your own
- Whether the DNS lookup returns the correct IP addresses

If you're using a CDN to speed up the delivery of your web pages to the far reaches of the Internet, the CDN may be operating your DNS on your behalf. This isn't an excuse not to test DNS resolution; in fact, you may want to watch more closely to be sure that IP addresses don't change without your approval. However, CDNs use resilient DNS services and Global Server Load Balancing (GSLB), which significantly improve the performance and availability of DNS resolution.

In most cases, you won't test DNS by itself. It will be a part of a synthetic test, and will be shown as the first element of latency in a test's results. Verifying that the content of a page is what you expect it to be will also let you know if your page has been hacked or if users are being redirected elsewhere because of a poisoned DNS.

Getting There from Here: Traceroute

In the previous chapter, we looked at traceroute as a tool for measuring the round-trip time between a client and a server. Traceroutes let you peer into the inner workings of the Internet. But because of the way traceroute collects information—sending several packets to every intervening device—it places a lot of load on the Internet's routers. As the traceroute manpage explains:

```
This  program  is intended for use in network testing,
measurement and management.  It should be used primarily for manual fault
isolation.  Because of the load it could impose on the network, it is un
wise to use traceroute during normal operations or from automated scripts.
```

Don't run traceroutes via automated scripts. Instead, use them as a diagnostic tool to tell where something has gone wrong across the Internet devices between a client and a website.

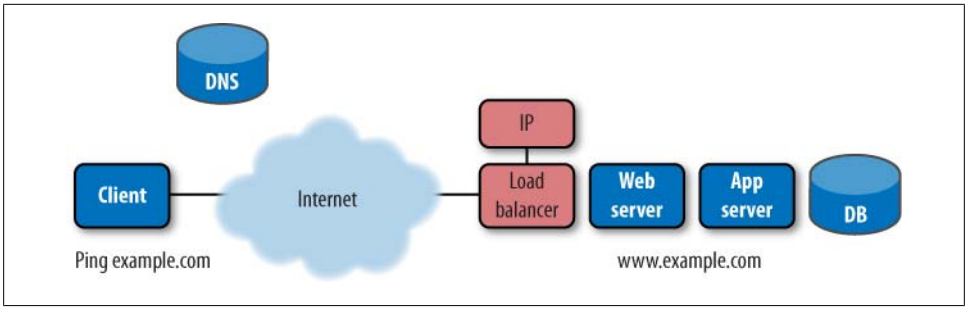


Figure 9-6. Ping testing measures IP functionality on the external device of the website

Recall that it's possible to use TCP, UDP, or ICMP traceroutes; you should use TCP traceroutes because they'll be treated in the same way that HTTP or HTTPS traffic would, while many parts of the Internet treat ICMP traffic differently. Also, don't expect traceroutes to always look clean; some devices won't return information about themselves, or will block downstream devices entirely.

Several synthetic testing services will perform automated traceroutes when they detect an outage, and include this information with the alert they send.

Testing Network Connectivity: Ping

Once you've confirmed that DNS can resolve the IP address of a site, and that the route across the Internet is clear, the most basic test you can run is to send a single packet to a website and receive a response. This is known as a *ping*, or ICMP ECHO. The device that receives the ping responds in kind (Figure 9-6). In most web applications, the responding device is a firewall or load balancer.

A transcript of a ping looks like this:

```
macbook:~ sean$ ping failblog.org
PING failblog.org (72.233.69.8): 56 data bytes
64 bytes from 72.233.69.8: icmp_seq=0 ttl=50 time=57.457 ms
64 bytes from 72.233.69.8: icmp_seq=1 ttl=50 time=58.432 ms
64 bytes from 72.233.69.8: icmp_seq=2 ttl=50 time=56.762 ms
64 bytes from 72.233.69.8: icmp_seq=3 ttl=50 time=56.780 ms
64 bytes from 72.233.69.8: icmp_seq=4 ttl=50 time=58.273 ms
64 bytes from 72.233.69.8: icmp_seq=5 ttl=50 time=58.555 ms
^C
--- failblog.org ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max/stddev = 56.762/57.710/58.555/0.751 ms
```

The ping contains the number of bytes received, the address of the host, and a sequence number, which can be used to identify any lost packets. The hop count is in the direction of the response (one-way) while the time measurement is the round-trip time (back and forth).

On a less reliable network, ping helps us to understand the quality of the connection. Here's a ping run from a wireless network on a train:

```
macbook:~ sean$ ping vangogh.cs.berkeley.edu
PING vangogh.cs.berkeley.edu (128.32.112.208): 56 data bytes
64 bytes from 128.32.112.208: icmp_seq=0 ttl=41 time=296.634 ms
64 bytes from 128.32.112.208: icmp_seq=1 ttl=41 time=431.799 ms
64 bytes from 128.32.112.208: icmp_seq=2 ttl=41 time=352.870 ms
64 bytes from 128.32.112.208: icmp_seq=3 ttl=41 time=479.129 ms
64 bytes from 128.32.112.208: icmp_seq=4 ttl=41 time=503.164 ms
64 bytes from 128.32.112.208: icmp_seq=5 ttl=41 time=291.246 ms
64 bytes from 128.32.112.208: icmp_seq=7 ttl=41 time=777.717 ms
64 bytes from 128.32.112.208: icmp_seq=8 ttl=41 time=391.574 ms
64 bytes from 128.32.112.208: icmp_seq=9 ttl=41 time=722.543 ms
64 bytes from 128.32.112.208: icmp_seq=10 ttl=41 time=1770.265 ms
64 bytes from 128.32.112.208: icmp_seq=12 ttl=41 time=588.587 ms
64 bytes from 128.32.112.208: icmp_seq=13 ttl=41 time=1114.075 ms
64 bytes from 128.32.112.208: icmp_seq=15 ttl=41 time=2086.454 ms
64 bytes from 128.32.112.208: icmp_seq=16 ttl=41 time=1809.736 ms
^C
--- vangogh.cs.berkeley.edu ping statistics ---
19 packets transmitted, 14 packets received, 26% packet loss
round-trip min/avg/max/stddev = 291.246/829.699/2086.454/595.160 ms
```

A ping test ends with a summary of the results. In this example, over a quarter of all packets that were sent were lost entirely, and the average packet took 829.699 milliseconds. This kind of latency and packet loss effectively renders the Internet unusable.

Ping is a general-purpose tool used for testing reachability, packet loss, and latency for any kind of application, including voice, video, file transfers, and so on. It's the most basic test of Internet reachability. Ping tests have some important limitations, however.

The Internet may treat pings differently from web traffic:

- In some cases, firewalls and load balancers simply won't respond to pings, or will block them entirely.
- Some networks may prioritize pings differently, so the results you get won't be representative of HTTP traffic.
- Because a ping only tests whether the server's Internet connection is working, and does not check the actual web service, it won't detect web problems such as missing content, a locked database, or an Apache service that isn't responding correctly.

Nevertheless, ping tests are easy to run and are the backbone of up/down monitoring on the Internet. Some synthetic testing portals offer hosted services that ping an IP address at regular intervals. This may be your only monitoring strategy for nonweb devices that you need to monitor.

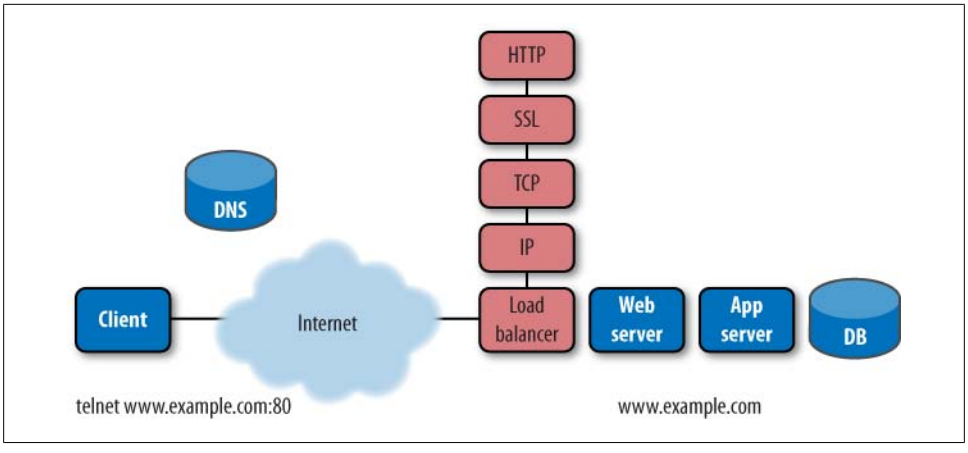


Figure 9-7. HTTP GETs test whether the web service is able to handle a request for a static object

Asking for a Single Object: HTTP GETs

Knowing that your web server is reachable doesn't mean users are getting the right content. Your DNS, network, and Internet-facing devices may be fine, but your website can still be down. The only way to tell whether the web service is working properly is to ask it for something and see what happens.

The simplest synthetic web test asks a server for content (using the HTTP GET method), times the response, and checks for an HTTP 200 status code confirming that the request was handled. In doing so, it tests not only the server's network layer (IP), but also the TCP layer (which manages end-to-end sessions) and any encryption (if present), as shown in Figure 9-7.

The HTTP GET is the workhorse of synthetic monitoring. It retrieves a single object, and in doing so verifies that many systems are functioning correctly. By timing the various milestones in the request and response, a testing service can blame poor performance on the correct culprit, whether it's DNS, network latency, or a slow server. Similarly, it can determine whether an error is caused by a bad network, a broken server, or missing content.

Beyond Status Codes: Examining the Response

Having sent a request and received an HTTP 200 OK in return, you might be tempted to pronounce the test a success. After all, you asked for content and received it. Some synthetic testing scripts stop here—particularly the free ones.

However, if you look deeper into the resulting response you'll see that there's a problem:

```
<div id="bd"><h1>Sorry, the page you requested was not found.</h1>
<p>Please check the URL for proper spelling and capitalization. If
 you're having trouble locating a destination on Yahoo!, try visiting the
```

```
 <strong><a href="http://us.rd.yahoo.com/default/*http://www.yahoo.com">Yahoo!
 home page</a></strong> or look through a list of
 <strong><a href="http://us.rd.yahoo.com/default/*http://docs.
yahoo.com/docs/family/more/">Yahoo!'s online services</a>
</strong>. Also, you may find what you're looking for if you try searching below.</p>
```

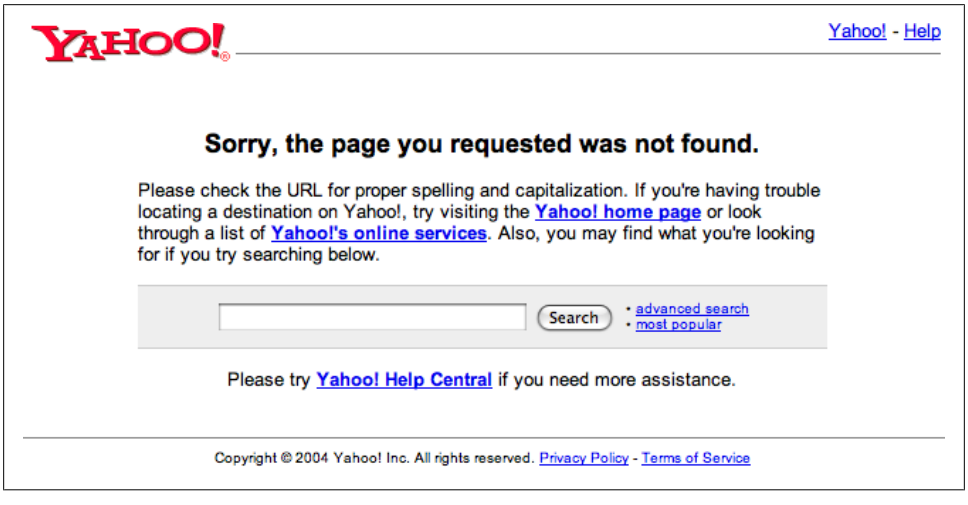


Figure 9-8. An apology page response for missing content that was served with an HTTP 200 OK status code

Despite the HTTP 200 OK response, the page you asked for didn't exist. Instead, you received a polite apology shown in Figure 9-8. In the early days of the Web, the ubiquitous "HTTP 404 Not Found" made it clear that you had asked for something nonexistent. Today, however, apology pages are commonplace, and they can hide errors.

Parsing Dynamic Content

Dynamic content further complicates the detection of errors. When you return to a portal, it probably welcomes you back by name. Your version of that page is different from someone else's version. Similarly, for a media site, news changes constantly. Even when part of the site is broken, much of the page will still render properly, and only one frame or section will indicate a problem (as shown in Figure 9-9). In other words, you'll never retrieve exactly the same page twice. So how does a synthetic test know that it's retrieved a valid copy of *index.html* when every copy is different?

To properly understand the performance and availability of a web application, you need to monitor dynamic content despite the fact that it's changing, as shown in Figure 9-10.

Figure 9-9. An error in a dynamic website, surrounded by properly loaded components

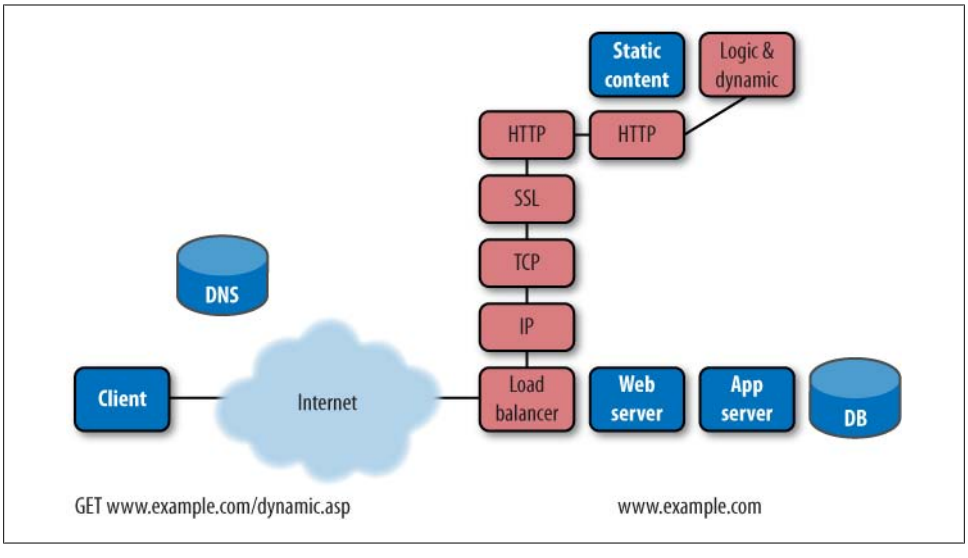


Figure 9-10. Checking for more than just an HTTP status code ensures you measure whether dynamic applications are working

The right way to check for HTTP content is to do the following:

- Check that the site returns an HTTP status code you want (generally a 200 OK).
- Check for content that *should* be there. Not all of a dynamic page changes every time. Often, changes are limited to a particular `DIV` tag, or certain cells of the table. The synthetic test needs to look for specific strings that are always present to confirm that the page is working. In Figure 9-10, this might be the text "Customize your reddit." You need to configure known keywords for each page you want to test and then make sure the application team doesn't remove them, which would lead to false alarms.
- Check for content that *shouldn't* be there. In Figure 9-10, this would involve making sure you don't see the string "There doesn't seem to be anything here." This is easiest to implement—you just need to be sure you're not getting a specific set of text—but it's error-prone because it won't detect errors that fail to give you the string you're looking for (as in cases of site vandalism.)

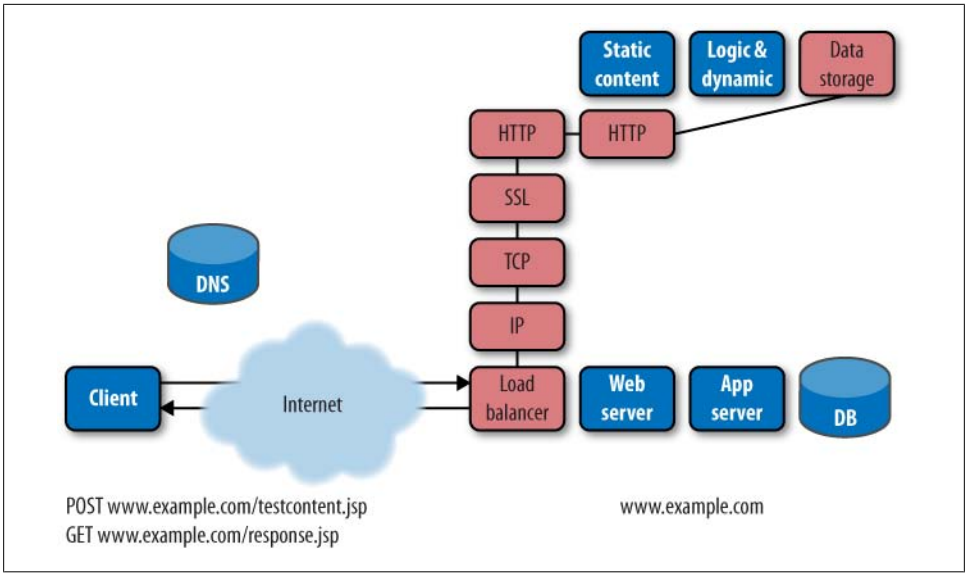


Figure 9-11. To test backend systems, a request must move data to or from data storage

While an HTTP GET for a static page will verify that the network, load balancer, and HTTP service are functioning correctly, it won't include the processing delay that comes from building dynamic pages. To properly measure performance, it's important to simulate what visitors see, even the parts that make the server work. You therefore need to test dynamic pages to measure real site performance.

The time it takes the server to send back the first byte of a response (known as *time-to-first-byte* in many testing reports) is roughly equivalent to the time the server takes to prepare the response, plus one network round trip. Because of this, if you have measurements of static and dynamic pages side by side, you can measure the difference and determine just how long the server is taking to process dynamic content.

Checking data persistence: Database access and backend services

Simply checking a dynamic page may not reveal the delay that backend data services are causing. To test the performance and availability of backend systems, you need to request content that forces the application to communicate with the database, as shown in Figure 9-11.

One way to isolate the database tier so you can test it is to have your developers build a page that exercises the backend of the application, writing data to a record and retrieving it before displaying a message indicating success. This might be a specific URL that, when requested, responds with considerable detail about the health of backend systems.

Here's an example of this kind of page:

```
macbook:~ alistair$ telnet www.bitcurrent.com 80
Trying 67.205.65.12...
Connected to bitcurrent.com.
Escape character is '^]'.
GET backendtest.cgi
<HTML><HEAD></HEAD>
<BODY>
Authentication check: <B>Passed</B> - 60ms
Database read: <B>Passed</B> - 4ms
Database write: <B>Passed</B> - 12ms
Partner feed: <B>NO RESPONSE</B> - N/A
</BODY>
<HTML
```

This is a trivially small response from a server, but it includes tests for authentication, database read and write, and access to a third-party server. You can set up monitoring services to look for the specific confirmation messages and to alert when an error (such as NO RESPONSE) occurs. If your monitoring service captures the page when an error occurs, you'll also have data from the responses at hand.

You can measure the performance of the database tier by comparing the performance of this page to that of a dynamic page that doesn't involve database access, which will help you to anticipate the need for more database capacity. In fact, this approach of creating a custom page that checks a particular backend element, then configuring a synthetic test, can be applied to any service on which your application depends, such as credit card transaction processing, currency conversion, hosted search, and so on. Just be sure to secure the test pages with proper authentication so that the backend service doesn't become a point of attack for hackers.

Beyond a Simple GET: Compound Testing

We've looked at testing various "depths" in the tiers of the web application, from DNS all the way back to the application server and database. Single-object HTTP requests still leave many questions unanswered, though. How long do whole pages take to load? Can users complete multistep transactions? Are page components functioning correctly?

Getting the Whole Picture: Page Testing

The majority of synthetic tests retrieve a page, then all of its components. A standard web page lists all of the components it contains; the testing system parses this page, then retrieves each component. The number of concurrent connections between the client and the server affects the total time the page takes to load, and the result is a "cascade" of objects that ultimately drive the total page load time, as Figure 9-12 shows.

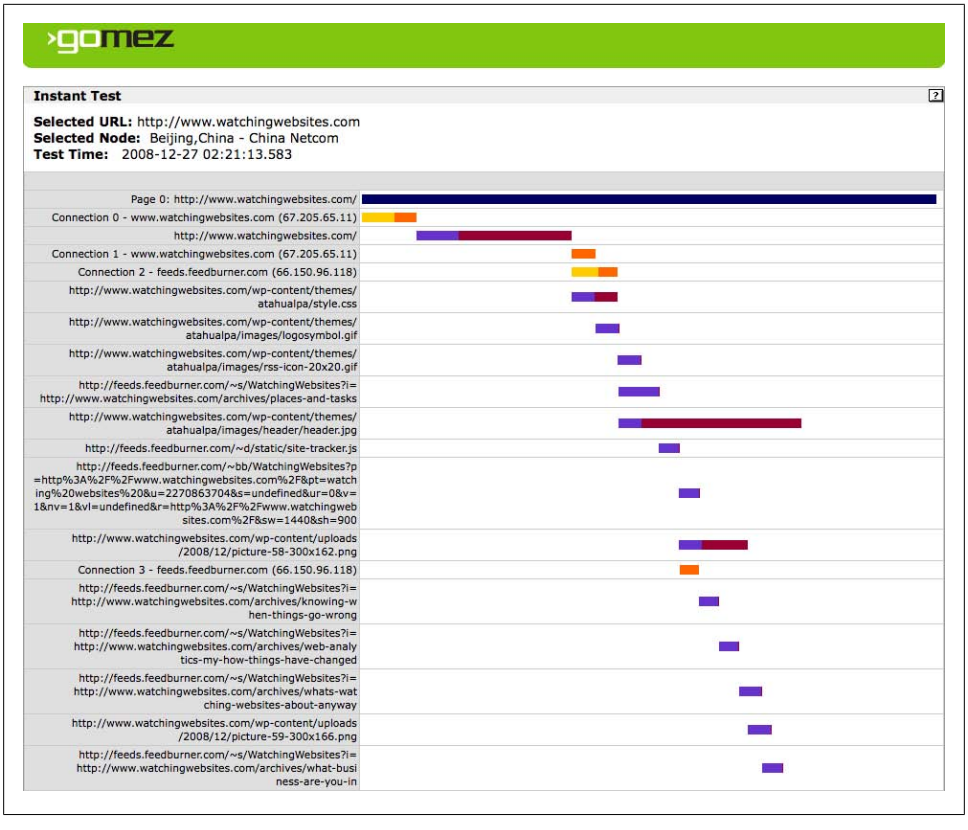


Figure 9-12. A cascade diagram from Gomez showing watchingwebsites.com and all its component objects being loaded from Beijing, China

In the modern Web, a site is only as good as its weakest component. Small components—JavaScript elements, Flash plug-ins, map overlays, analytics, survey scripts, and so on—can limit the speed with which the page loads, or even affect whether it loads at all, as shown in Figure 9-13.

If you're using a compound monitoring service, you'll be testing these components each time you check your own site. Synthetic testing services perform the test from several locations and networks using several browser types, at regular intervals, and report the data in aggregate charts like the one in Figure 9-14.

What if you want to measure a multistep process? It's time to record and replay a transaction.

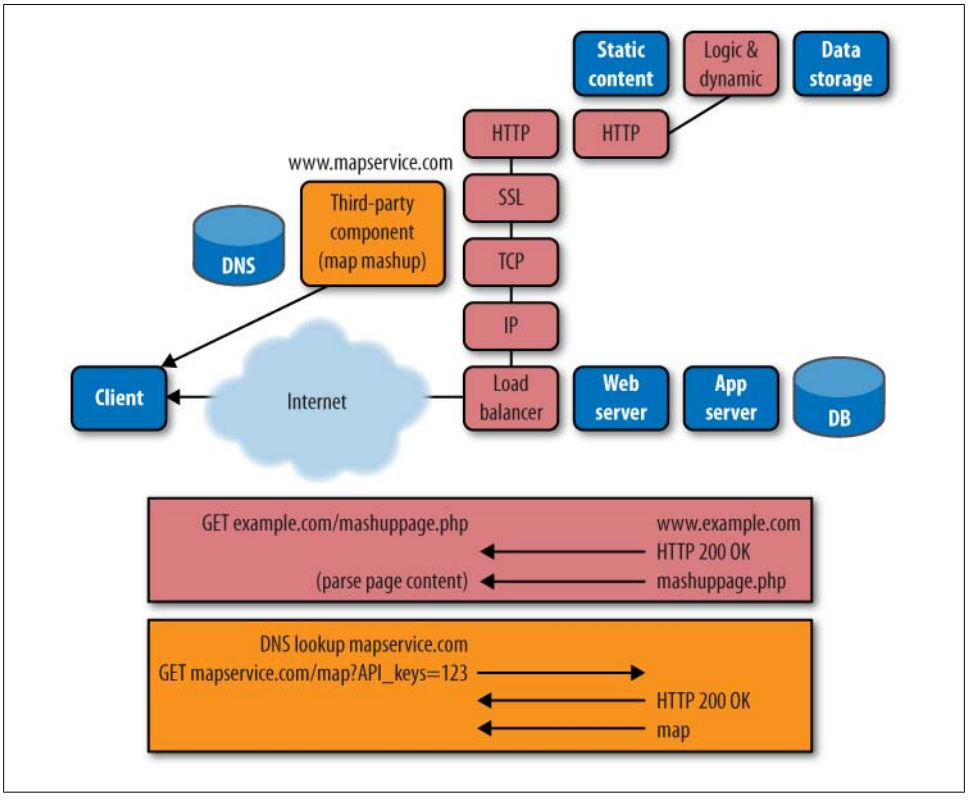


Figure 9-13. A complex page retrieval that includes data from a third-party site such as a mapping service

Monitoring a Process: Transaction Testing

Many synthetic testing services allow you to record a series of transactions and then repeat the sequence at regular intervals. The result is an aggregation of the total time it takes to complete a transaction. Figure 9-15 shows an example of this kind of data for Salesforce.com.

This is particularly useful if various steps in a user's experience put load on different systems. Imagine, for example, a purchase on a music site: the visitor searches for a song, posts a review about the band, adds an album to her cart, provides her payment information, and confirms the purchase. By testing a transaction, you see which components are slowest or most error-prone. You can also segment the results to see if certain visitors are having a particularly bad experience, as shown in Figure 9-16.

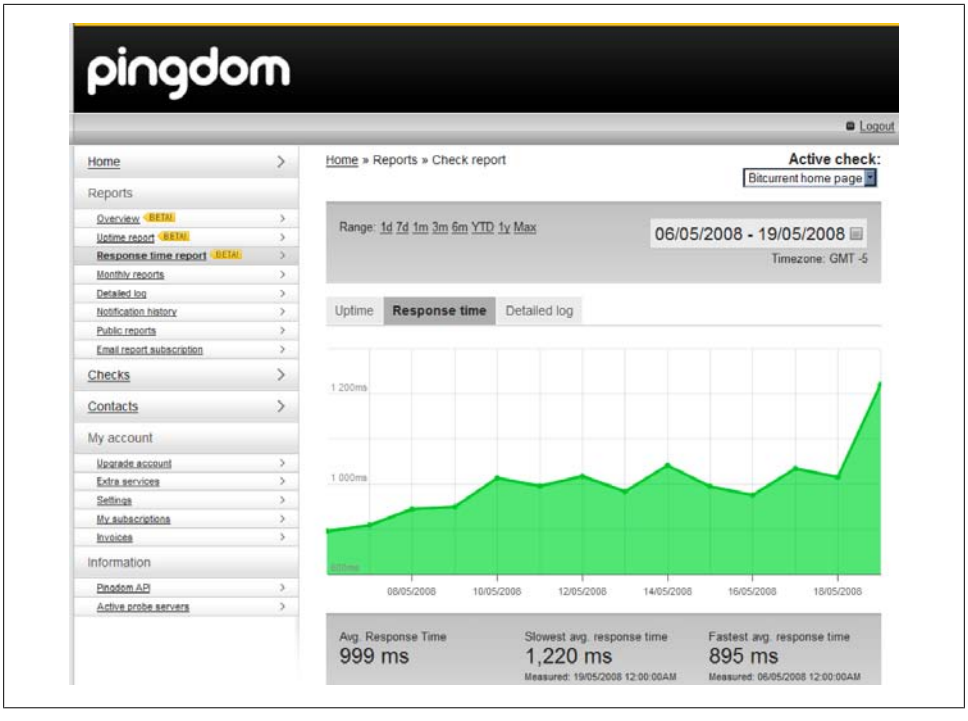


Figure 9-14. An aggregate performance report resulting from many tests of a website

Figure 9-15. A Webmetrics multistep transaction performance report showing the performance of three transaction steps in Salesforce.com

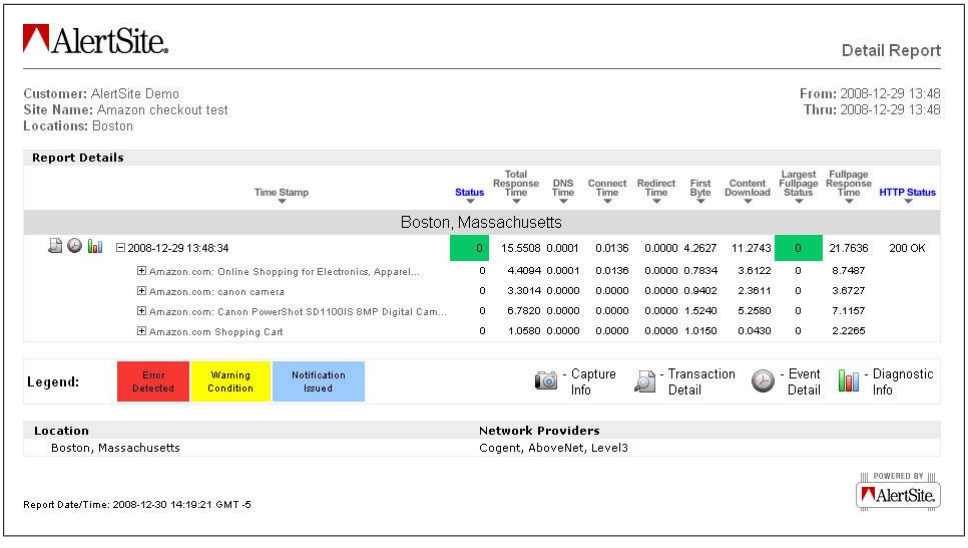


Figure 9-16. Segmenting transaction data by a geographic region or network provider using AlertSite

There are some things you can't monitor with synthetic testing, however. If one step of a process commits to an action, such as charging a credit card for a pair of shoes, the synthetic tests will have all-too-real consequences.

When Bad Things Happen to Good Tests

One publicly traded e-commerce vendor we spoke with described a test that tried to buy a PalmPilot stylus in order to verify its shopping cart process. The company ran the test from dozens of locations, simulating several browsers, every five minutes. Unfortunately, the development team hadn't properly disconnected the test account from the purchasing system. Before this was discovered, the test had purchased hundreds of thousands of styluses, having a material impact on the firm's revenues, requiring the finance team to notify securities regulators of the error.

In another case at a large ISP, the testing team set up a user with the name "test test." The test account hadn't been properly flagged within the provisioning software, resulting in a thousand modems being sent to a new customer named "test."

The moral of the story is that test URLs and test accounts that have been "neutered"—disconnected from the actual transaction engine—are essential for testing, but the creation and maintenance of these accounts must be part of the development process.

One way around this is to work with developers to set up a "dummy" user account and neuter its ability to execute real transactions. However, the synthetic test can't actually verify that the real process works, since it's not using a real credit card. This is one reason it's essential to have both synthetic testing and RUM as part of your EUEM strategy.

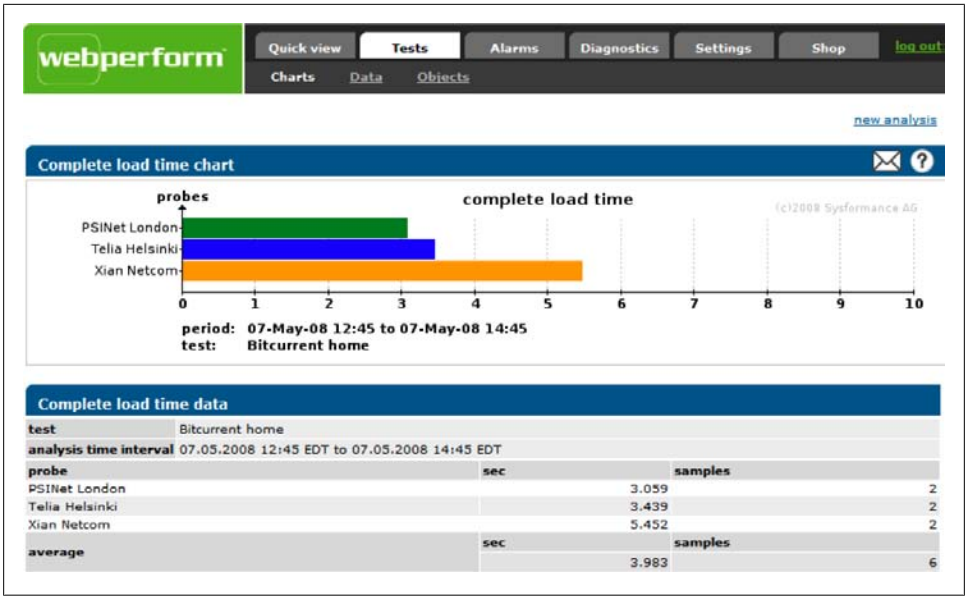


Figure 9-17. Page-level granularity report in Gomez's free Webperform testing service

Data Collection: Pages or Objects?

You've seen how you can probe deep within a site using synthetic testing, and you are now familiar with the way in which synthetic testing systems aggregate HTTP requests into pages and sessions. Different solutions offer different levels of detail in what they report.

Page detail

Some services, such as the one shown in Figure 9-17, report performance only at the page level. They may simply report total page load time, or they may break latency up into its component parts, including DNS latency, time to first byte, connection time, content time, and redirect time.

Object detail

Other services break the page down into its component parts, showing you the latency and problems within the individual pages. This information is more useful to operations and engineering teams, can sometimes lead to quicker answers and resolutions, but also costs more.

Error recording

Some testing services keep a copy of pages that had problems so you can see the error that actually occurred. In addition to showing you firsthand what the error looked like,

error recording makes it easier to troubleshoot your synthetic testing setup and identify problems that are causing false alarms, such as mistyped URLs.

There are significant differences between vendors here, and it's the basis of much competition in the monitoring industry. Some vendors capture container objects and their components separately, then reassemble them after the fact. Others capture screenshots of the pages as rendered in a browser, and may even record images of pages leading up to an error.

In other words, you get what you pay for. Ask lots of questions about how errors are captured and reported.

Is That a Real Browser or Just a Script?

One of the main differences between synthetic testing vendors is whether they simulate a browser or run a real browser.

Browser simulation

The simplest way to run an HTTP test is to do so through a script. When you open a command line and use `telnet` to connect to port 80, you're simulating a browser, and you get an answer. Emulating a browser by sending HTTP requests and parsing the responses is an efficient way to perform many tests by simply writing scripts.

Many lower-end testing services rely on this *browser simulation* approach, shown in Figure 9-18.

1. The test service creates simple instructions containing lists of sites and pages.
2. The script interpreter builds properly formed HTTP requests that it sends to the server.
3. The test service examines the resulting responses.
4. The service pronounces the test finished.

While browser simulation is straightforward and consumes few resources on the testing machine, it has important limitations that a second approach—using actual browsers—doesn't face.

Browser puppetry

The other main way to run tests is by manipulating an actual copy of a web browser, illustrated in Figure 9-19, which we call *browser puppetry*.

1. Instead of sending `GET index.html`, the script tells the browser, "Click on the fourth button."
2. The browser then performs the action, which results in a message to the server.
3. The next page is loaded.

4. The script can then examine the browser's DOM to determine whether the request worked.

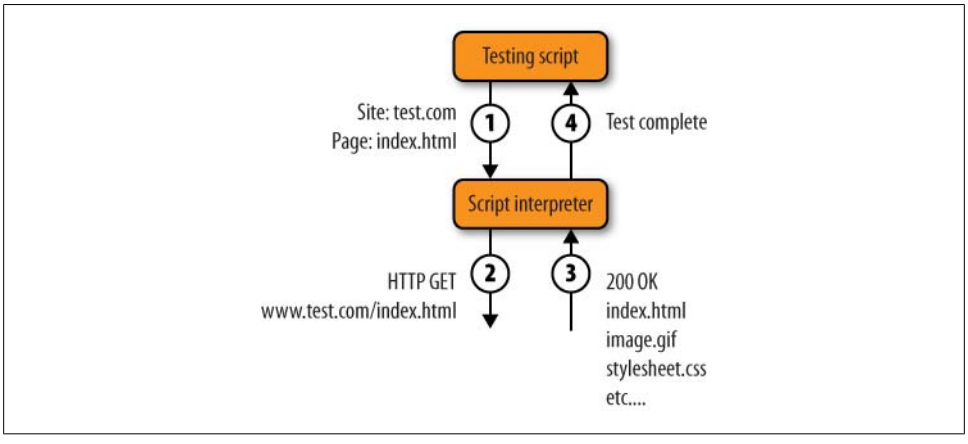


Figure 9-18. How browser simulation works

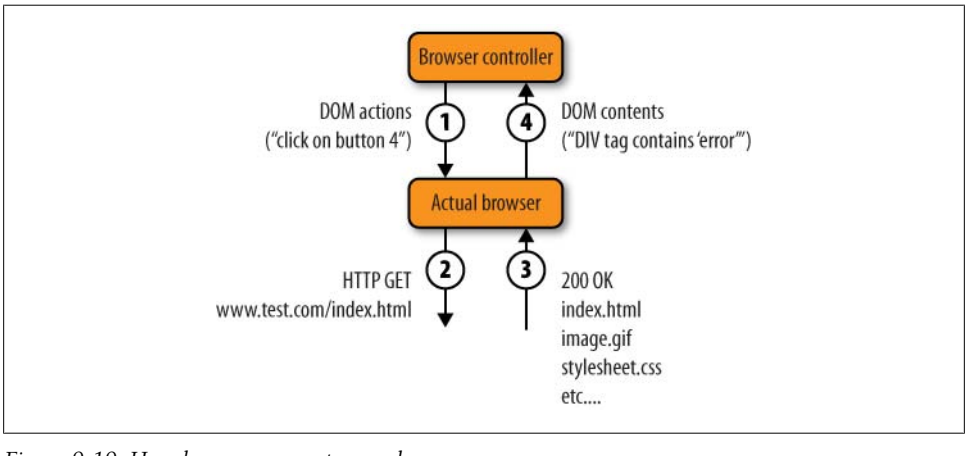


Figure 9-19. How browser puppetry works

This is much more burdensome for the testing service (it must run thousands of actual browsers in memory on its testing platform), but the benefits are significant:

- In a puppetry model, the service can examine all elements of the user experience, including things like text in the status bar, or cookies, by looking at the DOM, rather than just parsing the server's response.
- The puppet browser will include all the headers and cookies in a request that a browser would normally send. Scripted tests require that you manually supply these headers, and also require logic to maintain cookies.

- The puppet browser will automatically handle caching properly, whereas simulation will likely load all objects each time a test is run.
- Forms and complex navigation are much harder to emulate with simulation than they are with a puppet browser, where the test script simply has to say, "Put value A into form B."

The most important advantage of browser puppetry, however, comes from client-side content. In many modern web pages, JavaScript within the page dynamically generates URLs after the page is loaded. Consider, for example, a simple script that randomly loads one of five different pictures. Each time someone visits the site, the container page loads, then JavaScript decides which image to retrieve and builds the URL for that picture.

There's no way to know which image to retrieve without actually running the JavaScript, so this is very difficult for a simulated browser to test. With browser puppetry, the controller simply tells the browser, "Go and get this page." The browser does so, runs the JavaScript, and picks one of the five images. For dynamic, rich websites, puppetry is the right choice.

Configuring Synthetic Tests

Setting up synthetic testing is relatively easy, provided that you have a working knowledge of your web application. To test dynamic features, you'll also need a "dummy" user account that can place orders, leave messages, and log in and out without having an impact on the real world.

Also consider how you name your test accounts and your test scripts. The name "testscript5" will be much less meaningful than "westcoast-retail-orderpage" when you're configuring alerts or trying to tie an outage to a business impact.

Testing services generally bill per test. Four main factors affect how much your bill will be at the end of the month:

Test count
: The number of tests you want to run

Test interval
: How often you want to run them

Client variety
: The number of different clients you want to mimic in terms of browser type, network connection, desktop operating system, and so on

Geographic distributon
: The number of locations you want to test from, both in terms of geography and in terms of the Internet backbones and data centers

You'll also pay more for some of the advanced features outlined earlier, such as browser puppetry, multipage transactions, error capture, and object drill-down.

Test Count: How Much Is Too Much?

You should test every revenue-generating or outcome-producing component of your site, and perform comparative tests of a static page, a dynamic page, and a database-access page. If you rely on third-party components, you may want to test them, too.

That sounds like a lot of tests, which is bad news. Adding tests doesn't just cost more money, it also places additional load on your infrastructure that may impact real users. So you need to be judicious about your monitoring strategy. Here's how.

It's not necessary to run every test from every location. If you already know your site's performance and availability from 20 locations, there's no incremental value to testing the database from each of those 20 locations unless the database somehow varies by region. The trick is to identify *which tests can check functionality from just one location* and *which have to be run from many sources* to give you good coverage.

- *Functional tests* examine a key process, such as posting a comment on a blog. They only need to be run from a few locations. You will have many functional tests—sometimes one test for every page on your site—verified from few locations. They're the things that don't vary by visitor.
- *Coverage tests* examine aspects of your website that vary by visitor segment, such as web performance or a functioning DNS within a particular ISP. You will have few coverage tests—sometimes just a single page such as the landing page—verified from many networks and carriers.

Resist the temptation to run a wide range of functional tests from many locations. You'll make your site slower and your monitoring bill bigger without improving detection or adding to diagnostic information.

Test Interval: How Frequently Should You Test?

Now that you've separated your functional tests from your coverage tests, you need to decide how often to run them.

Availability testing checks to see whether something is working, while performance testing collects measurements for use in trending and capacity planning. Because of this, availability testing needs to run often—every five minutes or less—while baselining tests can run less often.

Problem detection: Availability testing

Operational data is tactical, intended for use by your networking and system administration teams to detect and repair problems, and by your marketing and development teams to verify whether a recent change works as soon as it's released. In the former

case, it will be tied to alerting, and in the latter, it will be tied to a particular change, such as a software release or a marketing campaign.

The goal of availability testing is to identify issues and help to localize problems so that you can fix them more quickly. You're likely to change tests whose primary goal is availability testing more often, according to business decisions and release calendars. As a result, you won't be doing long-term trending of this data—a test may only exist in the system for a few weeks before it is replaced with a new one.

Baselining and planning: Performance testing

Performance tests won't change as frequently. You'll keep test results as baselines for months or even years, comparing one holiday season to another, for example. Data from performance-focused tests is aggregated and displayed through dashboards to business decision makers. It tends to focus on availability percentages rather than on individual incidents, and on performance against a service target rather than sudden slowdowns.

You may also use baselining data like this for capacity-planning purposes, helping you to see a gradual slowdown in performance that may require additional servers, and for setting thresholds and alerts in more tactical, operational tests.

Client Variety: How Should I Mimic My Users?

Some desktops are underpowered, which means they take a long time to load and execute JavaScript. Some browsers use older versions of the HTTP protocol or connect across sluggish dial-up links. In other words, not all clients are equal. If you're hoping to understand end user experience, you have to make sure your tests mimic the conditions of your target audience so that the data you're capturing is representative of your end users' experience.

Fortunately, web analytics provides a rich resource for understanding your visitors' environments, letting you define tests that mimic their browsing experience as well as possible.

The folks at *www.eurotrip.com* were kind enough to share their analytics data with us, and we will use it liberally in this chapter. The site's primary focus is on casual European backpackers, and while it is an ad-driven media site, it has some collaborative elements in which visitors share their travel experiences.

Browser type

Figure 9-20 shows that over 80 percent of Eurotrip's visitors are using Internet Explorer and Firefox browsers running on Windows.

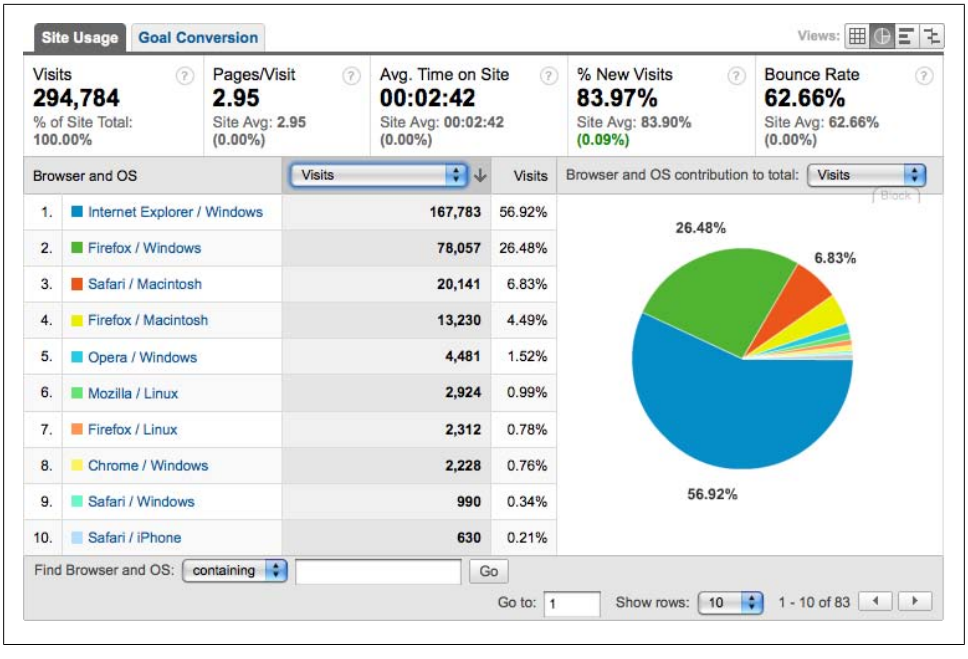


Figure 9-20. Segmentation of Eurotrip.com traffic by browser type

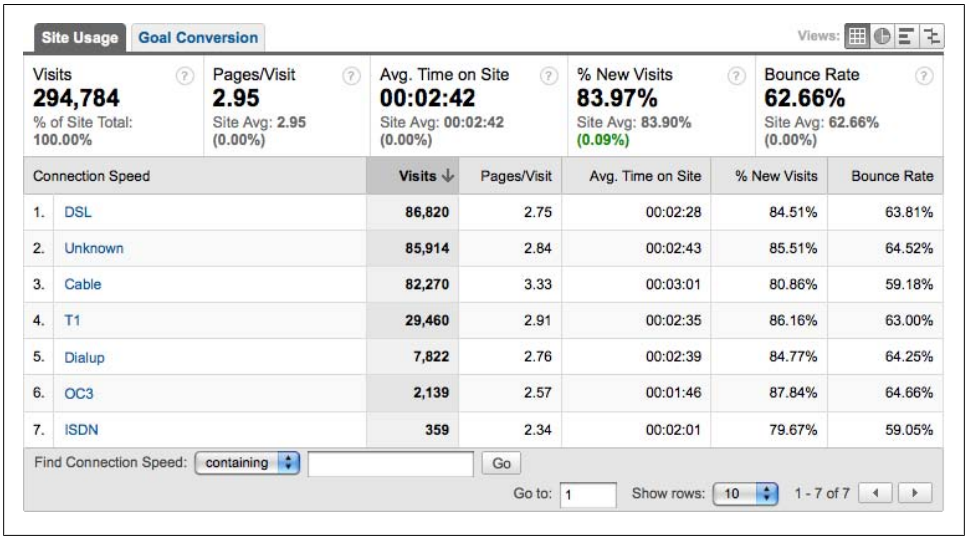
Due to this finding, Eurotrip's tests should probably simulate a Windows-based desktop. If Eurotrip was willing to pay for additional tests, it might also run tests that represent Internet Explorer and Firefox environments.

End user bandwidth

Your analytics tools can estimate what kinds of network connectivity your visitors enjoy. This is a rough estimate at best, because the analytics system can't tell whether delays are related to the core of the network or the edge devices, but it's a good enough approximation for you to use it to define testing.

Some synthetic services will allow you to simulate dial-up connections. Some even have "panels" of thousands of volunteer computers running small agents that can collect performance data from their home networks, businesses, universities, and Internet cafes, giving you an extremely accurate understanding of your site's health from domestic dial-up and residential broadband. This last-mile experiential monitoring can be a good way to determine how an application will function in a new market or new geographic region.

As Figure 9-21 shows, most visitors going to Eurotrip are using DSL or cable modems. With this information, the company can now provision, collocate or request the appropriate carrier types for our synthetic tests.



Site Usage | Goal Conversion | Views:

Visits	Pages/Visit	Avg. Time on Site	% New Visits	Bounce Rate
294,784	2.95	00:02:42	83.97%	62.66%
% of Site Total: 100.00%	Site Avg: 2.95 (0.00%)	Site Avg: 00:02:42 (0.00%)	Site Avg: 83.90% (0.09%)	Site Avg: 62.66% (0.00%)

	Connection Speed	Visits ↓	Pages/Visit	Avg. Time on Site	% New Visits	Bounce Rate
1.	DSL	86,820	2.75	00:02:28	84.51%	63.81%
2.	Unknown	85,914	2.84	00:02:43	85.51%	64.52%
3.	Cable	82,270	3.33	00:03:01	80.86%	59.18%
4.	T1	29,460	2.91	00:02:35	86.16%	63.00%
5.	Dialup	7,822	2.76	00:02:39	84.77%	64.25%
6.	OC3	2,139	2.57	00:01:46	87.84%	64.66%
7.	ISDN	359	2.34	00:02:01	79.67%	59.05%

Find Connection Speed: containing Go

Go to: 1 Show rows: 10 1 - 7 of 7

Figure 9-21. Eurotrip.com users by connection speed

Geographic Distribution: From Where Should You Test?

Internet latency is often correlated with geographic distance, and you need to test from the places your visitors are based. Knowing what performance is like for visitors from other countries is essential. For one thing, it will help you figure out when an overzealous shark has chewed up a transatlantic backbone on which your users rely. Measuring remote performance is also useful if you're trying to decide whether to deploy a CDN to speed up application performance.

Once again, Eurotrip's web analytics data shows you where its visitors are coming from.

As Figures 9-22 and 9-23 show, most of Eurotrip's visitors are located in the United States, Europe (UK, Germany, the Netherlands, and France), Australia, and Canada. This is consistent with the business goals of the site. If the site were only targeting customers in a particular region, it could ignore testing from other locations and save some money. For example, a used car dealership in San Jose could safely ignore performance from mainland China.

You can get similar data for visits by network or carrier, which may be important if you have customers on one carrier while you're collocated on another. For segmenting performance by carrier, you will want to know whether your tests come from dedicated or shared network connections, and whether they're in a data center or a last-mile location, such as a consumer desktop. You may also need to deploy private testing agents to measure performance from branch offices or important customers if your business needs this data.

Site Usage | Goal Conversion

Views:

Visits	Pages/Visit	Avg. Time on Site	% New Visits	Bounce Rate
294,784	**2.95**	**00:02:42**	**83.97%**	**62.66%**
% of Site Total: 100.00%	Site Avg: 2.95 (0.00%)	Site Avg: 00:02:42 (0.00%)	Site Avg: 83.90% (0.09%)	Site Avg: 62.66% (0.00%)

Detail Level: Country/Territory	Visits ↓	Pages/Visit	Avg. Time on Site	% New Visits	Bounce Rate
1. United States	116,959	3.27	00:02:58	81.26%	59.52%
2. United Kingdom	37,172	2.50	00:02:08	88.45%	67.03%
3. Canada	26,995	4.45	00:03:46	74.42%	54.20%
4. Australia	11,686	3.55	00:03:59	80.89%	60.03%
5. Germany	8,109	2.21	00:01:44	91.10%	67.90%
6. Netherlands	6,052	2.02	00:01:37	91.01%	66.26%
7. France	5,522	1.96	00:01:26	90.69%	71.35%
8. Spain	5,360	1.92	00:01:28	91.57%	68.68%
9. Ireland	4,765	2.05	00:01:28	92.44%	76.79%
10. Italy	4,759	1.90	00:01:26	89.49%	69.36%

Find Country/Territory: containing Go

Go to: 1 Show rows: 10 1 - 10 of 199

Figure 9-22. Visits by countries/territories in Google Analytics

Detail Level: City	Visits ↓	Pages/Visit	Avg. Time on Site	% New Visits	Bounce Rate
1. London	11,578	2.29	00:01:55	87.56%	68.62%
2. (not set)	8,751	2.62	00:02:39	86.96%	65.52%
3. New York	3,799	2.82	00:02:12	83.15%	62.83%
4. London	3,449	2.70	00:02:22	85.79%	63.99%
5. Sydney	2,965	3.13	00:03:41	82.19%	61.96%
6. Dublin	2,797	1.78	00:01:12	94.74%	80.51%
7. Toronto	2,660	3.17	00:03:09	80.98%	61.80%
8. Melbourne	2,215	3.56	00:03:40	83.16%	62.35%
9. Vancouver	2,186	3.56	00:03:10	63.91%	47.90%
10. Amsterdam	2,007	1.92	00:01:38	91.83%	64.42%

Figure 9-23. Top visits by city in Google Analytics

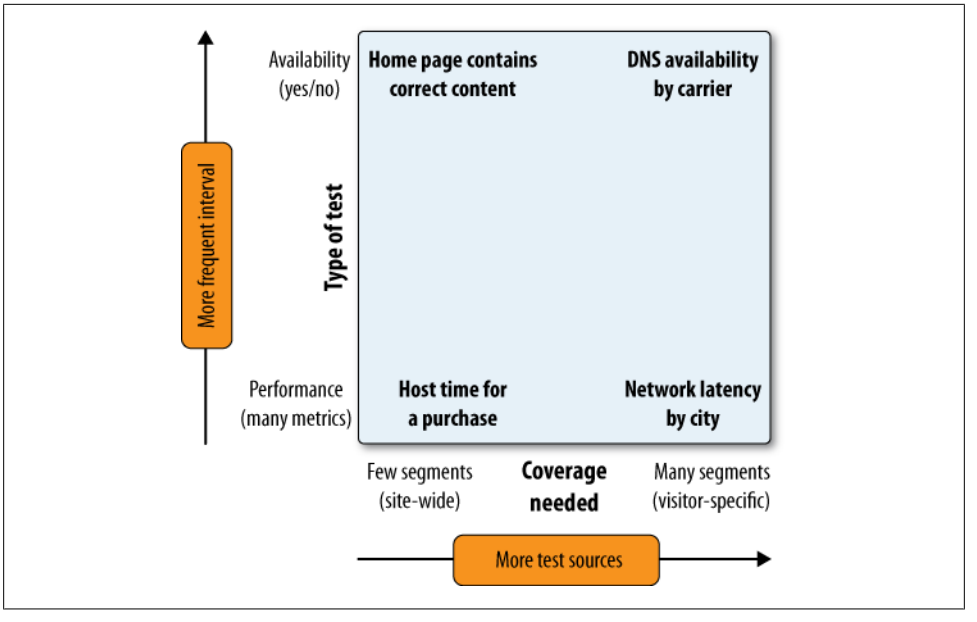


Figure 9-24. Determining test frequency based on the type of test and the coverage needed

Putting It All Together

Knowing which tests are focused on availability versus performance, and which tests need to be segmented for analysis, you can set your monitoring strategy. Some examples of tests across these two dimensions are shown in Figure 9-24.

Now it's time to make your test plan using these two dimensions. For coverage tests, identify the segmentation you need: browser types, bandwidth tiers, geographic regions, networks, and so on. Table 9-1 shows an example of this kind of test plan.

Table 9-1. Determining test frequency based on the type of test and the coverage needed

Test type	Segmentation	Frequency
Functional		Low
	Key pages	Low
	Tiers (i.e. database)	Low
	Third-party components	Medium
	Key transactions	Medium
Coverage		
	Browser types	High
	Bandwidth tiers	High
	Geographic areas	High
	Networks	High

Your plan will allow you to estimate your monthly service costs, as well as the features you need from a synthetic testing vendor.

It should be clear by now that for sites of any size, test management is a full-time job. Tests are created, modified, and retired as the website changes. You need to track test versions and manage which reports are sent to whom, and should treat them as part of the software release cycle.

This is particularly important if those tests rely on specific pass/fail text on a page that may change across releases, or use "dummy accounts" that require maintenance. Tests grow stale quickly, and you'll need to maintain thresholds and alerts as the site's designers change functionality and content.

Setting Up the Tests

Now that you've decided what, where, and how often to test, you need to configure testing. Single-page testing is relatively simple: you provide the URL, the type of test (network, TCP, DNS, and so on), interval, geographic distribution, and any other information that helps the tests to mimic your visitors, as shown in Figure 9-25.

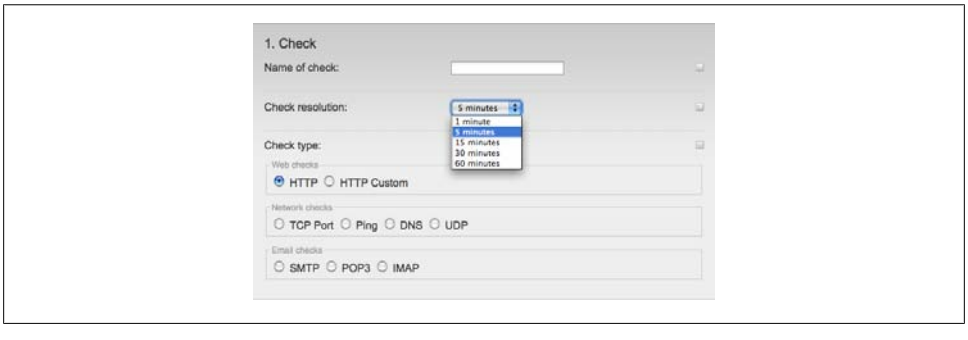


Figure 9-25. Configuring test parameters with the Pingdom testing service

Transactional testing is more complex to configure, and there are many more chances to get things wrong. Fortunately, monitoring service companies often include test recording tools that remove much of the guesswork from setting up tests. Figure 9-26 shows an example of this—you navigate within the target site in a frame, and the recording tool records the steps you take as part of the test.

Once recorded, you can edit the scripts to correct errors made during recording, modify timings, and define what should be used to verify that a page was loaded properly.

Setting Up Alerts

When one of your availability tests fails, it will alert you via email, SMS, or voicemail. Alerts can also be directed to software in the network operations center for escalation

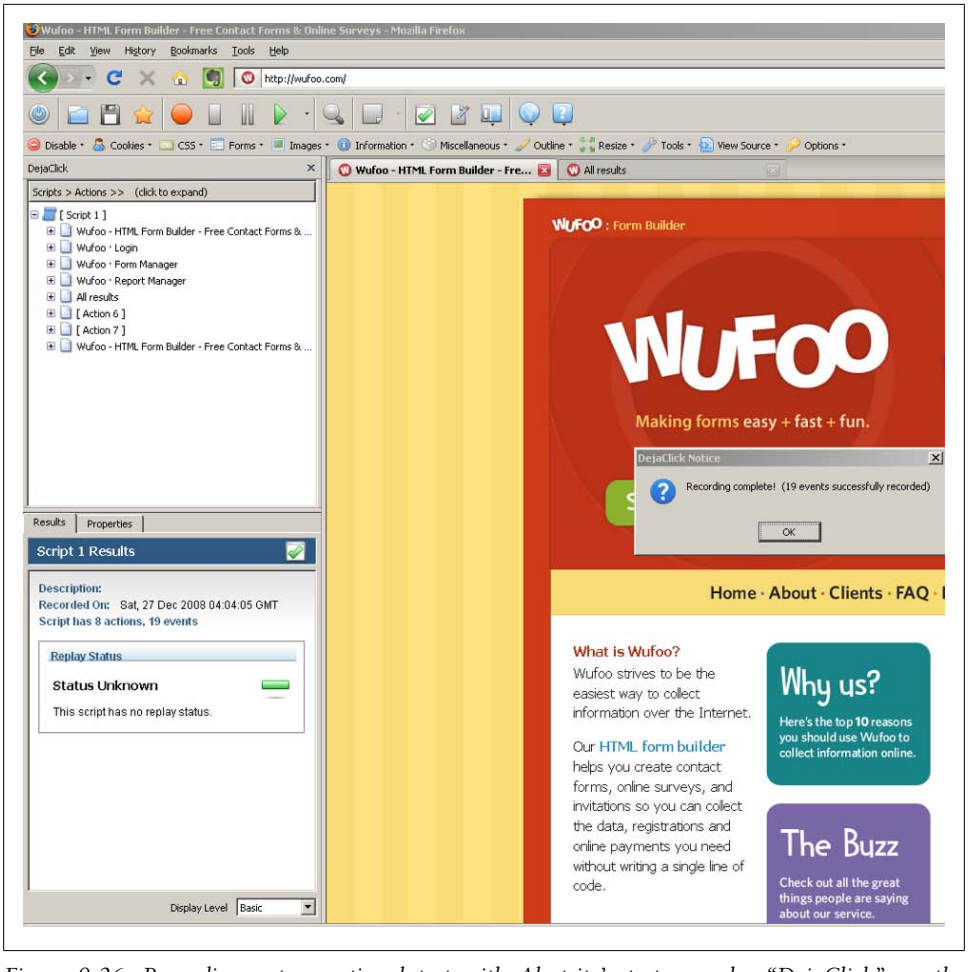


Figure 9-26. Recording a transactional test with Alertsite's test recorder "DejaClick" on the Wufoo.com website

and correlation with device metrics. Some synthetic monitoring vendors also send SNMP traps to enterprise management software tools.

Alerts fall into one of two main categories. *Hard errors* occur when a test encounters a problem, for example, a 404 Not Found message when retrieving a page. By contrast, *threshold violations* occur when a metric, such as host latency, exceeds some threshold for a period of time.

Figure 9-27 shows an alarm configuration screen that defines both hard (availability) and threshold (performance) violations. Setting up a threshold violation alarm requires several details.

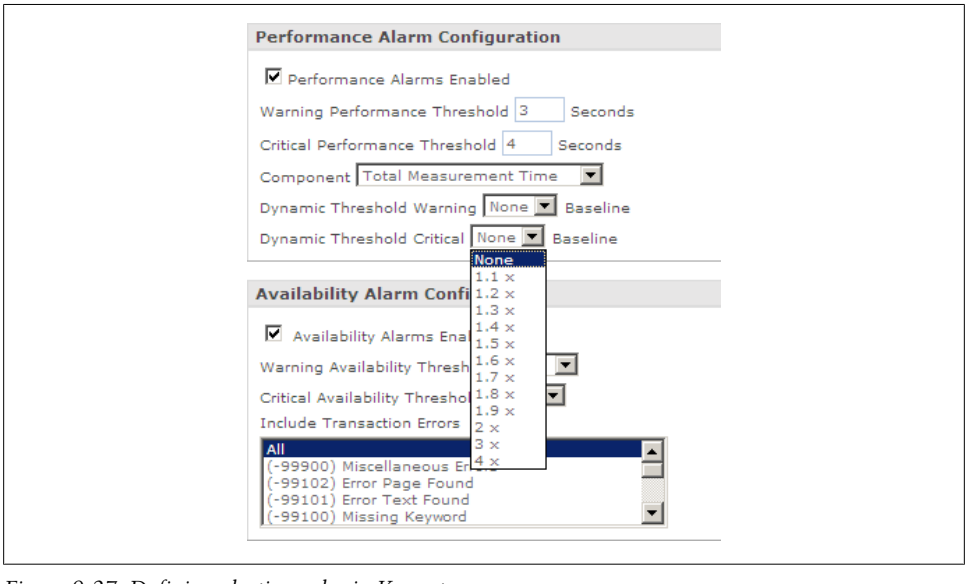


Figure 9-27. Defining alerting rules in Keynote

- The performance threshold. In this example, there are two thresholds, one for "warning" and one for "critical."
- Which metric to look at (in this case, it's the total time to load a page).
- A dynamic threshold violation (this alerts you if the page performance exceeds the baseline for "normal" performance at this time).

For hard error alerts, you simply need to define which kinds of availability problems to notify about. When defining any kind of alert, you may also have to configure rules about suppression (so that the service only tells you when it's broken and when it's recovered, rather than flooding you with error notifications) and verification (so that the service checks several times to be sure the error is real before telling you about it).

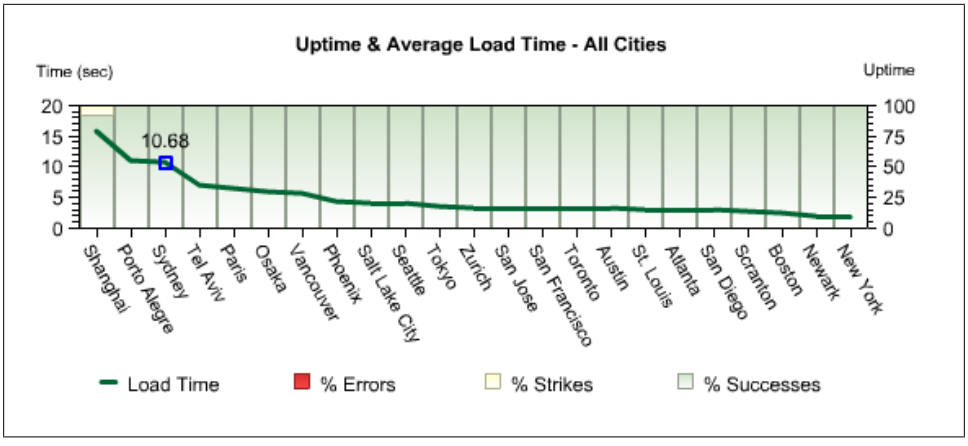


Figure 9-28. Webmetrics transaction performance report by geographic region for a site on the East Coast of North America

A good alerting system lets you sleep at night knowing that if there's an issue with your website (availability or performance), you'll be alerted. That system loses its effectiveness if there are too many false positives, but tuning an alerting system can take time. For example, most sites will exhibit different response times from different geographic locations, even within the same country. Setting a single response time alert threshold across all the geographies from which you're testing from will either cause you to miss critical alerts (if set too high) or get too many false positives (if set too low).

Some services have far more complex alerting systems, including escalation, problem severity options, time zones, maintenance windows during which to ignore downtime, and so on.

Whichever option you choose, your alerts should contain as much business information as possible, such as how many users were affected, the change in website conversions during the outage, or whether the incident happened during a planned adjustment to the site. *The perfect alert contains all of the information you need to localize, diagnose, and repair the problem, and to assess its impact on your business*.

Aggregation and Visualization

We've seen several examples of how synthetic tests report data. Let's look at a few other common representations of end user experience. One way of reporting performance is to segment it by region or carrier to identify the worst-served parts of the Internet, as shown in Figure 9-28.

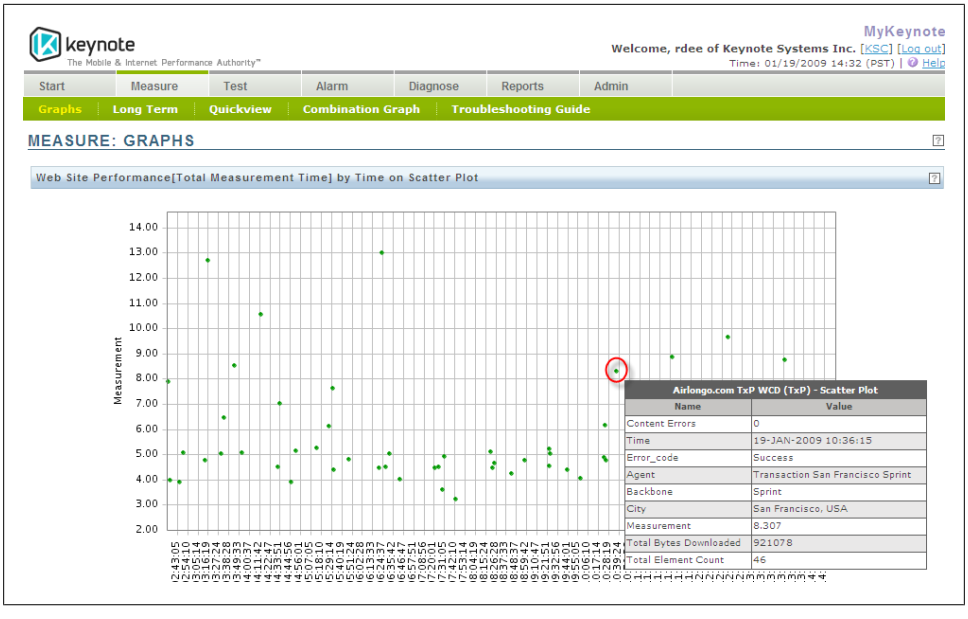


Figure 9-29. A scatterplot of individual test results

Segmented reports show you the slowest or least-available tests along a particular dimension. But what if you want to focus on a particular element of latency? Synthetic tests collect a great deal of data, and you need to look across all of it to properly understand what normal performance is like.

By displaying a metric—such as total transaction time—as a histogram, you can quickly see what kind of performance is most typical. On the other hand, you may not want to aggregate data, but rather see individual tests to better understand a particular period of time. In this case, a scatterplot visualization like the one shown in Figure 9-29 works well.

This kind of report is only available from services that keep the raw test data for considerable lengths of time. Some lower-cost services will aggregate this information into averages to reduce their storage requirements.

Finally, matrix reports are ways of comparing two segments to one another. In the case of Figure 9-30, these segments are locations and websites.

Matrix reports are useful for identifying problem areas at a glance, such as two carriers that aren't forwarding packets to one another well. From a matrix report you can drill into the history of a specific region or site.

As you define and prepare these reports, be sure to share them with managers and members of other monitoring groups, including marketing, QA, and web design, to accustom them to the information that synthetic monitoring can provide.

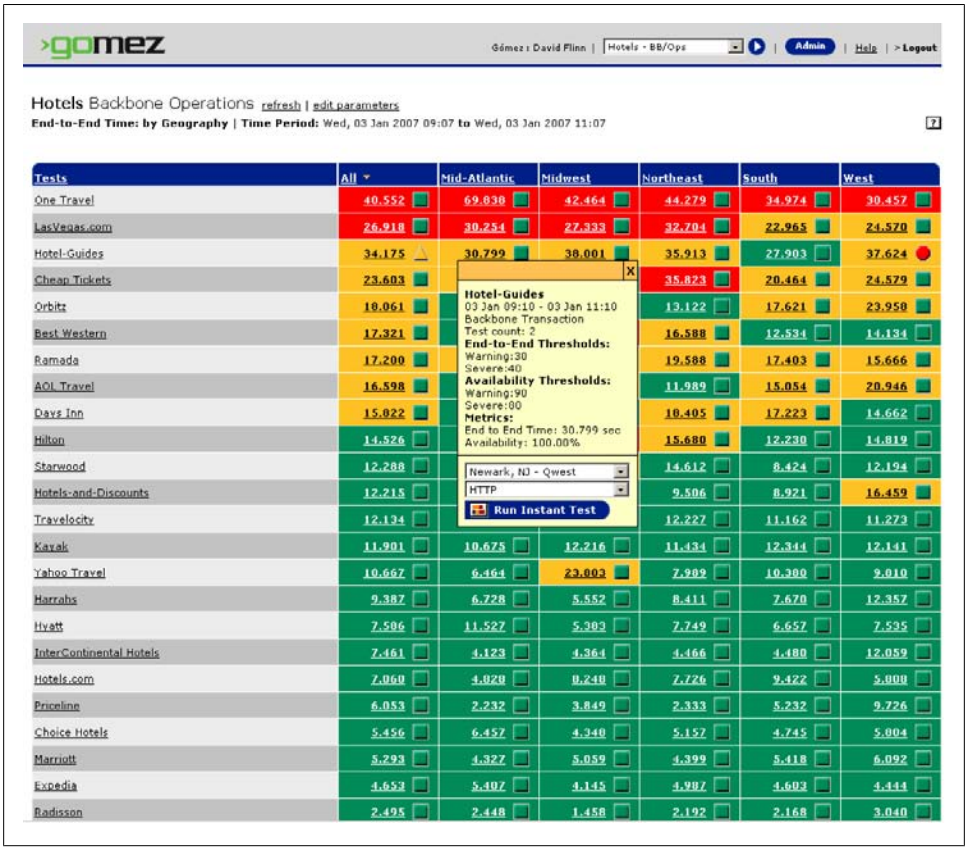


Figure 9-30. Gomez matrix report of websites by region

Advantages, Concerns, and Caveats

Synthetic testing is your first answer to the question, "Could they do it?" It's a reliable, controllable, repeatable measurement that you can use regardless of how much or how little traffic there is on your website. It also has some significant shortcomings.

No Concept of Load

Synthetic testing services don't know how much traffic is on your site. A testing service is blissfully unaware of whether your site is experiencing a deluge of traffic or is so slow that it has idle servers. This means they're missing an important possible cause of web latency, because the more visitors you have, the longer the site takes to respond. Alerts and thresholds on synthetic testing systems can't take into account how busy the site is.

You should, of course, be concerned if your site becomes slow when nobody's using it, but a synthetic testing service won't alert you to that fact as long as the latency is within acceptable limits. Dynamic baselining, in which the service learns what "normal" latency is like at a certain time of day, is somewhat of a proxy for load, assuming your website gets the same loads at the same times of day.

Muddying the Analytics

Synthetic tests generate traffic. If you're running tests on a site, exclude the synthetic tests from overall analytics *before you start testing* or your visitor count will be artificially high, as shown in Figure 9-31.

Figure 9-31. The sudden drop in traffic on October 27 is the result of excluding synthetic testing from web analytics measurements

Checking Up on Your Content Delivery Networks

If you're using a CDN to speed up traffic, you should test the performance of retrieving a page from the CDN, but also of retrieving the same page from your own servers. To accomplish this, you'll need a second domain name that bypasses the CDN. By comparing both test results, you'll see how much the CDN is helping with performance and whether you're getting your money's worth.

Rich Internet Applications

Most of this chapter has dealt with monitoring HTML-centric applications. The modern Web is changing that in two important ways. First, more and more applications rely on rich clients written in JavaScript, Flex, and so on. These clients do much of the presentation and display on the end user's computer. Instead of retrieving whole pages, the client retrieves smaller nuggets of information from the server, often structured in XML or JSON. The testing service must be able to emulate these kinds of calls.

This is an area where browser puppetry works better, because there's a real browser executing the tests. Nevertheless, rich clients have destroyed much of the standardization that we enjoyed with traditional HTML-centric, page-by-page web design, forcing us to rethink web monitoring.

In some cases, the user may also install client-side software, such as a streaming video helper or a plug-in that uses a non-HTTP protocol like the Real-Time Streaming Protocol (RTSP) or even UDP instead of TCP. If this is the case, you need to extend your synthetic testing to test these new protocols, and this will limit the synthetic testing vendors you can use.

Video monitoring is a special case that will change which metrics matter to you. Rather than response times, you'll care about startup time (how long video takes to begin) and rebuffering ratio (what percentage of visitors saw a message saying "rebuffering..." while playing the video). You may even measure details like frame rate and effective bit rate that can only be collected within a media player. If you want to measure this synthetically, you'll need "media player puppetry" from your testing provider.

Site Updates Kill Your Tests

Changes to a website are the most common cause of false alarms. Test scripts are inherently "brittle"—when the site changes, the tests stop working properly. It may be a transaction that that no longer follows the same navigation path through the site, a new URL, or a different response that the service interprets as an error.

There are no simple technical solutions to this problem. They're human issues, and you need good processes like change control management to overcome them. Remember, however, that tests using browser puppetry are less likely to falsely report errors when you change a page: if a URL is altered but the button's name remains the same, the test will still work.

Generating Excessive Traffic

Synthetic tests still consume resources on your servers. We've seen bad test plans that caused over 60 percent of all the site's traffic via the testing itself. This is a tremendous waste of money and makes it even harder to identify real users who are having problems amidst the noise.

Data Exportability

Synthetic testing data contains historical information that you may want to use elsewhere. When selecting a synthetic testing platform, you should be aware of how much data the system stores and for how long. This is particularly true if your organization relies heavily on data warehousing or business intelligence systems. Data should be downloadable via XML, CSV, or a real-time feed.

Competitive Benchmarking

Synthetic tests let you keep tabs on your competitors. You can monitor their performance and availability and use it to set your own service-level targets. For SaaS companies, knowing that your competitors are slower or less reliable than you are is an important selling point.

Some synthetic testing companies offer comparative benchmarks that rank members of a particular industry segment against one another.

Tests Don't Reflect Actual User Experience

It's easy to let synthetic traffic lull you into a false sense of security. Often, synthetic tests will be fine, even as users are suffering. This happens for two main reasons:

- Your tests don't simulate the network, region, bandwidth, browser, or desktop properly.
- Users are doing something on the site that you aren't testing for.

This is one of the main reasons that you should always use synthetic testing in conjunction with RUM, which we'll cover in the next chapter.

Synthetic Monitoring Maturity Model

You need synthetic transaction monitoring as early as possible, preferably while you're still building and testing your application. The less real traffic you have on your site, the more valuable synthetic testing is. Once you've got more traffic, you can reduce the number and frequency of your tests and look at actual user performance instead.

Many companies limit the use of synthetic testing to IT only, without considering its broader impact on other departments. As an organization starts to share synthetic data, it shifts from testing machines to testing business processes, as well as using properly structured tests to speed up problem resolution. Eventually, synthetic testing becomes a key performance indicator for marketing campaigns and competitive benchmarking.

Maturity level	Level 1	Level 2	Level 3	Level 4	Level 5
Focus	Technology: make sure things are alive	Local site: make sure people on my site do what I want them to	Visitor acquisition: make sure the Internet sends people to my site	Systematic engagement: Make sure my relationship with my visitors and the Internet continues to grow	Web strategy: Make sure my business is aligned with the Internet age
Who?	Operations	Merchandising manager	Campaign manager/SEO	Product manager	CEO/GM

Maturity level		Level 1	Level 2	Level 3	Level 4	Level 5
EUEM	**Synthetic**	Availability and performance: Checking to see if the site is available from multiple locations, and reporting on performance	Transactions and components: Multistep monitoring of key processes, tests to isolate tiers of infrastructure	Testing the Internet: Monitoring of third-party components and communities on which the application depends	Correlation & competition: Using the relationship between load and performance; comparing yourself to the industry and public benchmarks	Organizational planning: Using performance as the basis for procurement; uptime objectives at the executive level; quantifying outages or slowdowns financially