CHAPTER 10

Could They Do It?: Real User Monitoring

A company once ran a beautiful monster of a marketing campaign.

The campaign was an attempt to drive traffic to its e-commerce site. Shortly after launching the campaign, sales dropped by nearly half. Synthetic tests suggested everything was fine. Web analytics reported an increase in visits, but a huge drop in conversions across every visitor segment. It looked like the campaign had appealed to a large number of visitors who came to the site but didn't buy anything.

Management was understandably annoyed. The official response amounted to, "Don't ever do that again, and fire the guy who did it the first time."

Fortunately, one of the company's web operators was testing out new ways of monitoring end user performance at this time. He noticed something strange: a sudden spike in traffic, followed by the meltdown of much of the payment infrastructure on which the site depended. This payment system wasn't part of the synthetic tests the company was running.

As it turned out, the company had hit upon an incredibly successful promotion that nearly killed the system. So many people were trying to buy that the checkout page took over 20 seconds, and often didn't load at all. Nearly all of the visitors abandoned their purchases. Once the company responded by adding servers, upgrading the payment system, and fixing some performance bottlenecks, they tripled monthly revenues.

It's one thing to know your site is working. When your synthetic tests confirm that visitors were able to retrieve a page quickly and without errors, you can be sure it's available. While you know it's working for your tests, however, there's something you don't know: *is it broken for anyone, anywhere?*

Just because a test was successful doesn't mean users aren't experiencing problems:

- The visitor may be on a different browser or client than the test system.
- The visitor may be accessing a portion of the site you're not testing, or following a navigational path you haven't anticipated.

- The visitor's network connection may be different from that used by the test for a number of reasons, including latency, packet loss, firewall issues, geographic distance, or the use of a proxy.
- The outage may have been so brief that it occurred in the interval between two tests.
- The visitor's data—such as what he put in his shopping cart, the length of his name, the length of a storage cookie, or the number of times he hit the Back button—may cause the site to behave erratically or to break.
- Problems may be intermittent, with synthetic testing hitting a working component while some real users connect to a failed one. This is particularly true in a load-balanced environment: if one-third of your servers are broken, a third of your visitors will have a problem, but there's a two-thirds chance that a synthetic test will get a correct response to its HTTP request.

In other words, there are plenty of ways your site can be working and still be broken. As one seasoned IT manager put it, "Everything could be blinking green in the data center with no critical events on the monitoring tools, but the user experience was terrible: broken, slow, and significantly impacting the business." To find and fix problems that impact actual visitors, you need to watch those visitors as they interact with your website.

Real user monitoring (RUM) is a collection of technologies that capture, analyze, and report a site's performance and availability from this perspective. RUM may involve sniffing the network connection, adding JavaScript to pages, installing agents on end user machines, or some combination thereof.

Three Ways to Watch Visitors

Three of the technologies we cover in this book overlap considerably. We've already looked at web analytics, which records visitor requests and shows you what visitors did. And we've seen WIA, which shows how those visitors interacted with pages and forms. RUM collects performance and availability information from user traffic.

Because these three technologies all capture information about a visitor's session, it's common for vendors to combine analytics, WIA, and RUM features in a single product. So, for example, you might purchase a RUM product that allows you to diagnose a usability issue. While looking at POST parameters for visitor sessions you notice that the form field for "quantity" often contains five-digit zip codes. This is a usability problem, discovered by a RUM solution.

There's overlap in the industry, which inevitably leads to vendors jockeying for position and overreaching their claims in order to compete with one another. Ultimately, any solution that gives you better visibility into end users is a good thing, and hopefully by understanding the technology behind these tools, you can ask smarter questions and choose a solution that's right for your business.

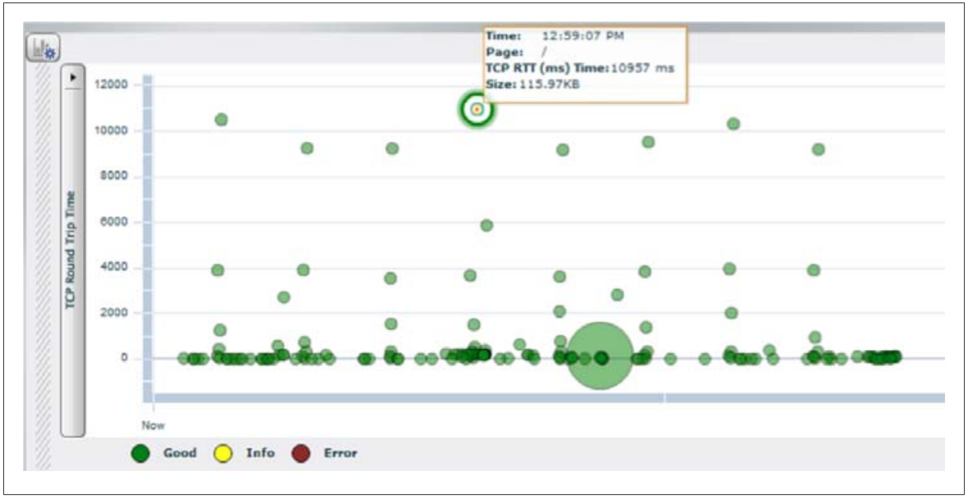


Figure 10-1. A scatterplot of page requests over time in Coradiant's TrueSight, showing relative TCP round-trip time

RUM and Synthetic Testing Side by Side

For this book, we're using a simple distinction between synthetic testing and RUM. If you collect data every time someone visits your site, it's RUM. This means that if you have 10 times the visitors, you'll collect 10 times the data. On the other hand, with the synthetic testing approaches we saw in the previous chapter, the amount of data that you collect has nothing to do with how busy the site is. A 10-minute test interval will give you six tests an hour, whether you have one or a thousand visitors that hour.

Here's a concrete example of RUM alongside synthetic data. Figure 10-1 shows page requests to an actual website across an hour. Each dot in the figure is an HTTP GET. The higher the dot, the greater the TCP round-trip time; the bigger the dot, the larger the request.

While requests happen throughout the hour for which the data was collected, there are distinct stacks of dots at regular intervals. These columns of requests, which occurred at five-minute intervals, are actually synthetic tests from the Alertsite synthetic testing service, coming from Australia, Florida, and New York.

Figure 10-2 highlights these five-minute intervals. The tests from each of the three locations have "bands" of latency—tests from Australia had the highest round-trip time, as we'd expect. Notice that there would have been no data on Australia without synthetic data. Also notice that the only way to discover the excessively large request (the big dot) was to watch actual visitors—there's no way for a synthetic test to detect this. Finally, notice the dozens of requests that happen in those five minutes—an eternity of Internet time.

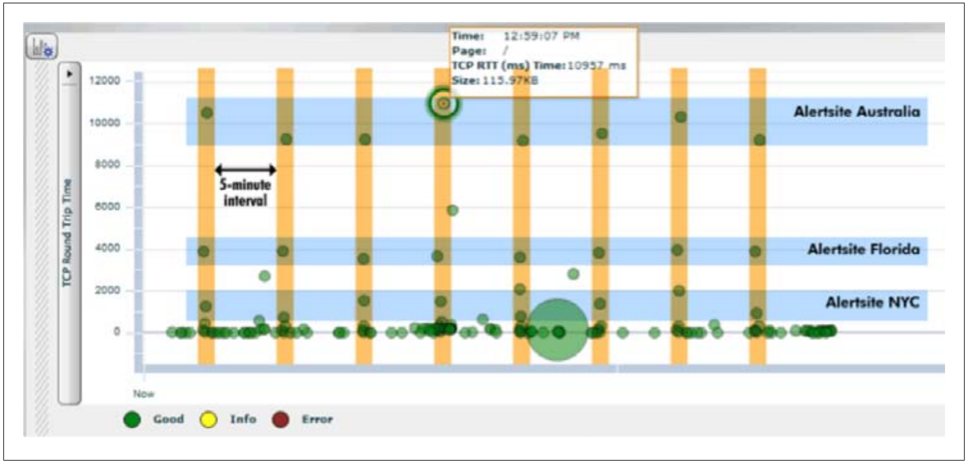


Figure 10-2. The same scatterplot in Figure 10-1, with synthetic tests identified

Synthetic tests give you an idea of what users *might* experience, but RUM tells you what actually happened—whether users could accomplish the things they tried to do. In this respect, RUM is the natural complement to web analytics. However, you cannot use RUM on its own: it's useless if users don't visit the site, because there are no visits to analyze.

How We Use RUM

We've already covered why you need to look at end user experience (Chapter 8), but here are some specific uses of RUM technology:

- Using performance records to prove you met service-level targets with customers
- Supporting users and resolving disputes based on a record of what actually happened
- Speeding up problem resolution with "first-cause" analysis to localize the issue
- Helping to configure and verify synthetic tests
- Defining testing scripts from actual user visits

Proving That You Met SLA Targets

When a service provider and its customers argue, it's usually because they don't have all the facts. Instead, they resort to anecdotes and recrimination, each providing incomplete evidence to support the view that they're right and the other is wrong.

This is especially true for SaaS websites. When you're the service provider, you need to be gentle. If you're in the wrong, you'll be issuing a refund and apologizing soon. If the problem is the customer's fault, you have the opportunity to fix it and help her save face.

You need to know what actually happened, which is where RUM excels. If you have reports on what the end user experience was like, you can tell subscribers precisely where delays occurred.

To make the most of dispute resolution, your RUM solution must segment traffic by the application being used, by subscriber (usually the company that's paying the bill), and by individual user. It must also generate reports by elements of latency and by type of error. By distributing this data to sales and support teams on a regular basis, you'll be well equipped to prove that you did what you said you would.

Supporting Users and Resolving Disputes

While dispute resolution normally happens with aggregate data, call centers work with individuals. If your call center has access to RUM information, call center personnel can bring up a user's session and see what went wrong.

If the problem is on the user's side, you can add the issue to a FAQ, modify troubleshooting processes to help customers serve themselves in the future, or let the design team know where users are getting stuck, all of which will reduce call center volumes. On the other hand, if the user has encountered a legitimate problem that must be fixed, the session records will be invaluable in convincing engineering that there's an error and helping QA to test for the problem in future releases.

"First-Cause" Analysis

RUM data will seldom diagnose a problem completely—there are simply too many components in a web transaction to pinpoint the root cause by looking at web traffic. Rather, RUM will tell you where in your delivery infrastructure to look. In this respect, it is a "first-cause" rather than a root-cause approach.

Consider, for example, RUM data broken down by type of request: a static image, a static page, a dynamic page, and a page that writes to the database. If there's a sudden performance problem with dynamic pages, it's likely that the application server is to blame. If that data is then segmented by server address and URL, you know where to start looking.

This kind of segmentation is how IT operations teams solve problems naturally. When you adopt a RUM solution, you need to make it an integral part of your problem resolution workflow and escalation procedures, with the new data in order to reap all of the benefits.

There are, however, emerging end-to-end passive analysis tools, like ExtraHop, that monitor not only HTTP, but other protocols as well, and can drill down into many of the backend systems behind the web tier to help with troubleshooting.

Helping to Configure Synthetic Tests

When you're developing your synthetic test strategy, you need to know what an acceptable response time is. In the previous chapter, we looked at how you can use data from web analytics to help ensure that your synthetic tests are watching the right parts of your site. RUM data can help you determine what the acceptable results should be for those tests.

Imagine, for example, that you want to test the login process. You've got a synthetic test for *http://www.example.com/login.jsp* that you'd like to run. You can take the RUM data for the 95th percentile of all logins, add a standard deviation, and you will have a good estimate of a "normal" performance range for that page.

Chances are, however, that you'll deploy a synthetic testing solution well before you deploy RUM, so a more likely use of RUM is to validate that your synthetic tests are working properly and that your test results match what real users are seeing.

As Content for QA in New Tests

Session records provide the raw material for new tests. Because they record every HTTP transaction, you can feed them into load testing systems to test capacity. What's more, if you share a copy of a problematic visit with the release management group, the release team can add the offending test to its regression testing process to make sure the issue is addressed in subsequent releases.

Capturing End User Experience

Armed with thousands of measurements of individual page requests, you can answer two important questions: could visitors use the website? What were their experiences like?

The first question involves problem diagnosis and dispute resolution. When someone calls with a problem, you can quickly see what happened. You can even detect and resolve a problem before she calls, because your RUM tool has seen the error occur.

The second question involves segmentation and reporting. You can look across all requests of a certain type—a specific browser, a geographic region, a URL, and so on—and analyze the performance and availability of the website for capacity planning or to understand how the site's performance affects business outcomes.

Both questions are vital to your web business. To answer them, you first need to collect all those page measurements. The work you'll need to do depends heavily on which RUM approach you use.

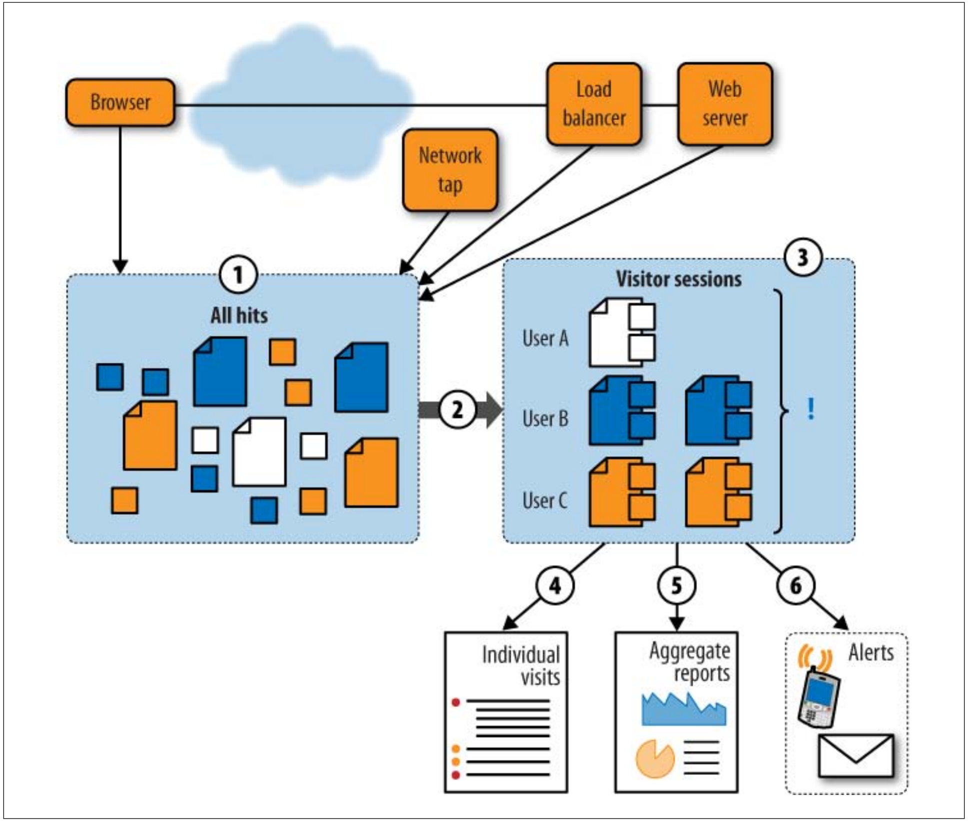


Figure 10-3. The basic steps in all RUM solutions

How RUM Works

RUM solutions vary widely, but always involve the following basic steps, as shown in Figure 10-3.

1. **Capture**. The monitoring system captures page and object hits from several sources—JavaScript on a browser, a passive network tap, a load balancer, or a server and its logfiles.
2. **Sessionization**. The system takes data about these hits and reassembles it into a record of the pages and components of individual visits, along with timing information.
3. **Problem detection**. Objects, pages, and visits are examined for interesting occurrences—errors, periods of slowness, problems with navigation, and so on.
4. **Individual visit reporting**. You can review individual visits re-created from captured data. Some solutions replay the screens as the visitors saw them; others just present a summary.

5. **Reporting and segmentation**. You can look at aggregate data, such as the availability of a particular page or the performance on a specific browser.
6. **Alerting**. Any urgent issues detected by the system may trigger alerting mechanisms.

These six steps seem fairly basic, but as with most monitoring technologies, the devil's in the details. How much information the RUM solution records, and what you can do with it, depends on what information that solution collects, how it decides what's "interesting," and how it stores and retrieves user data.

Server-Side Capture: Putting the Pieces Together

Collecting performance at the server's side of the connection takes a bottom-up approach. First, your RUM tool gathers the individual HTTP objects, then organizes them into pages and visits, then calculates performance metrics. This is the approach many server agents, logfiles, and passive analyzers use, including those from Computer Associates, Coradiant, HP, and Tealeaf.

Often, these objects have something in common that lets the RUM tool group them together. It may be a cookie unique to that visit (known as a *session cookie*) or some other piece of information, such as an IP address or a browser. The RUM tool uses this information to assemble all of the objects into a user's visit.

Within those objects, some are containers (such as those ending in *.html*) and some are components (such as those ending in *.gif*). This allows the RUM solution to identify where pages begin and end, though in practice, it's hard to do this well and there are many rules and tweaks that a RUM tool uses to reassemble a user session properly.

Once a RUM tool has grouped a visit into pages, it uses the timing of the various objects to determine page load time.

Client-Side Capture: Recording Milestones

If you're collecting performance at the browser's side of the connection using JavaScript, you're recording milestones that occur as a page loads in the browser. JavaScript embedded in the page records key events, such as the start of a page or the moment that all the objects it contains have been retrieved from the server. The JavaScript then sends the timing of these milestones and some metadata about the page to a collector.

With this model, you don't need to worry about reassembling the individual objects that make up a page. In fact, you may ignore them entirely. You also won't need to work out which sessions belong to which visitors—after all, you're running on one visitor's browser, so you only see one visit and can mark page requests on behalf of the service to associate them with one another.

Gomez and Keynote both offer this model, and many large websites (such as Netflix, Yahoo!, Google, and Whitepages.com) have deployed homegrown client-side collection. The developers of the Netflix solution have made their work available as an open source project called Jiffy (for more information on how Netflix is instrumenting page performance, see *http://looksgoodworkswell.blogspot.com/2008/06/measuring-user-experience-performance.html*).

You may not have to choose between server-side and client-side collection. Some vendors offer a hybrid approach that collects user experience data at both the client and the server for a more complete perspective on performance.

What We Record About a Page

There are dozens of facts about a web page that you will want to record. All of them are useful for diagnostics, and many of them are good dimensions along which to segment data within reports. These facts include performance metrics, headers and DOM information, error conditions, page content, correlation data, and external metadata.

Performance metrics

You can track the following timing metrics for every page:

TCP round-trip time
: The round-trip time between client and server. This is the time it takes a packet to travel across the Internet from the client to the server and back.

Encryption time
: The time to negotiate encryption (SSL, for example) if needed.

Request time
: The time for the client to send the server a request.

First byte
: The time for the server to respond to the request with a status code and the requested object (known as the first byte or host time).

Redirect time
: The time taken to redirect a browser to another object, if applicable.

Network time
: The time it takes to deliver the object to the client.

TCP retransmissions
: The number of TCP segments that were lost (and had to be retransmitted) during the delivery.

TCP out-of-order segments
: The number of TCP segments that arrived out of order and had to be reordered by the client.

Page render time
: The time it takes for the browser to process the returned object—generally, the time taken to render a page.

Application-specific milestones
: The start of a rich media component, the moment a visitor drags a map, or other timing events specific to a particular application that are recorded as part of the page's performance.

End-to-end time
: The total time taken to request, receive, and display a page.

Visitor think time
: The time the visitor takes once the page is loaded, before beginning the next action.

Some of this data may not be available, depending on how you're capturing user experience. Page render time, for example, is something only client-side RUM can measure. Similarly, low-level networking statistics like TCP retransmissions aren't visible to JavaScript running on the browser.

Headers and DOM information

Every web page—indeed, every HTTP object—includes request and response information in the form of headers sent between the browser and the web server. Because HTTP is an extensible protocol, the number of possible headers is unlimited. Most RUM solutions will collect the following data about the request:

Browser type (user agent)
: The browser requesting the object.

URL
: The requested object.

POST or URI parameters
: Any data the browser sent to the server.

Referrer
: The referring URL that triggered the browser's request.

Cookies
: Any cookies the browser sent that it acquired on previous visits.

The server's response contains additional information about the object being delivered, which the RUM solution can capture:

MIME type
: The kind of object that the server is returning.

Object size
: The size of the requested object.

HTTP status code
: The server's response (200 OK, 404, etc.).

Last-modified date
: The time it takes for the browser to process the returned object—generally, the time taken to render a page.

Compression type
: Whether the object is compressed, and if so, how.

MIME type
: What type of object is being delivered. This helps the browser to display the object.

Response content
: The object itself, or specific strings within the object.

Cachability information
: Details on whether the object can be cached, and if so, for how long.

New cookies
: Any cookies the server wants to set on the browser.

Metadata is important. If the server says an object is big, but the actual object is much smaller, it's a sign that something was lost. Similarly, an unusual MIME type may mean that content can't be displayed on some clients. As a result, RUM tools often capture this kind of data for segmentation ("What percentage of requests are compressed?") or to help with diagnostics.

Server-side RUM tools will collect this data from actual requests, while client-side tools will assemble it from the browser's DOM, so the metadata you can collect will depend on how you're capturing page performance.

Error conditions

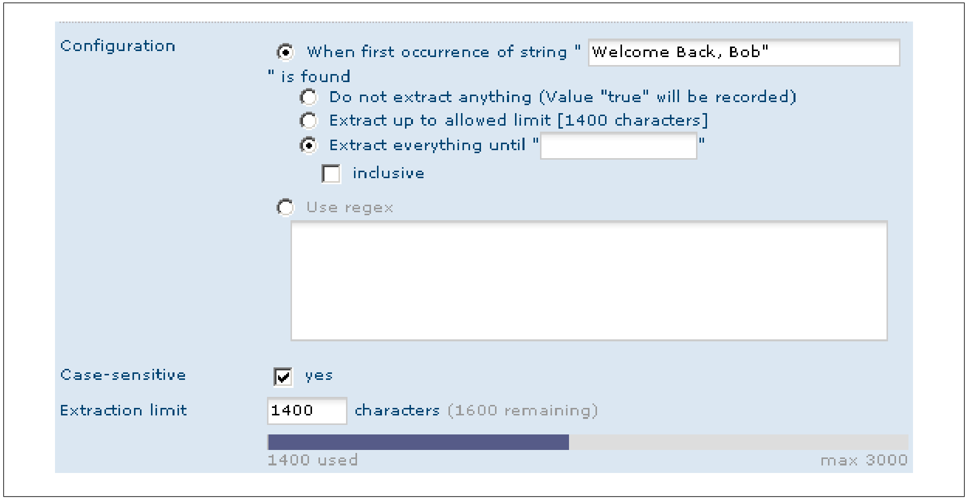
RUM tools are on the lookout for everything from low-level network errors to bad HTTP status codes to specific content in a page. Most watch for a built-in list of error conditions. Again, the RUM approach you choose will determine which errors you can detect. In particular, client-side RUM can't detect errors that happen before the client-side JavaScript has loaded (because the monitoring tool isn't capturing data yet), so it's used less often than server-side RUM for error detection.

Page content

Many pages your website serves contain additional context about the content of the page. Perhaps it's the price of a purchase, the type of subscriber (for example, "premium"), or maybe the visitor's account number.

Some RUM tools extract this kind of information from pages and associate it with a visit. With a server-side RUM tool, you specify which HTML content it should extract as it goes by, using an interface like the one shown in Figure 10-4. Every time page content matches those specifications, the RUM tool captures the content and stores it with the page.

One common piece of data to extract from page content is the title of the page. Most HTML objects have a `<Title>` tag that provides the name of the page in plain language. This name is often different from and more readable than the page's actual URL. Instead of talking about "page4.aspx," you're now discussing "the checkout page." If you



Configuration

[✓ When first occurrence of string "] Welcome Back, Bob" " is found

- Do not extract anything (Value "true" will be recorded)
- Extract up to allowed limit [1400 characters]
- [✓ Extract everything until "] "
  - inclusive

Use regex

Case-sensitive [✓ yes]

Extraction limit 1400 characters (1600 remaining)

1400 used max 3000

Figure 10-4. *Configuring content extraction in Coradiant's TrueSight*

capture additional page metadata, such as the total amount spent, you can make the visit record even more relevant and tangible.

Page content is also useful for segmentation, allowing you to ask questions like, "How much did sales go down for users whose transactions took longer than five seconds?"

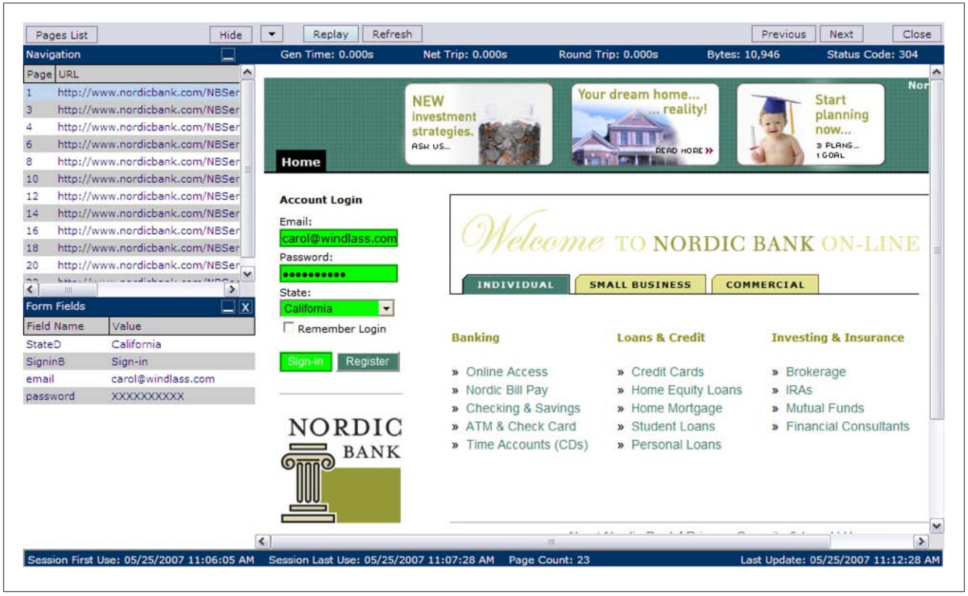
If you're using client-side RUM collection, metadata collection happens differently. The JavaScript used for performance instrumentation can also extract information from the DOM, such as the page title or some text in the page, and include that data in its report on the page's retrieval, just as an analytics tool records page tags.

Some tools can even capture the entire page rather than just specific strings. This lets you search through pages after the fact, and is useful for problem diagnosis, particularly when problems are related to content or to user input (Figure 10-5).

Capturing all page content also lets you search across all visits for specific occurrences of a problem. We know of a case in which a disgruntled employee vandalized a site's pages on his last day on the job. When an outraged customer first reported the problem to the call center, the company had to determine how many others had seen the changed content. Nobody knew to look for the string in question until it had happened, but being able to search through historical data to find all instances of the string of expletives the employee had left let the company defuse the situation.

Correlational data

If you're planning to combine your RUM data with other data sources, you may need to extract strings that will help you correlate those sources.



Gen Time: 0.000s   Net Trip: 0.000s   Round Trip: 0.000s   Bytes: 10,946   Status Code: 304

Page	URL
1	http://www.nordicbank.com/NBSer
3	http://www.nordicbank.com/NBSer
4	http://www.nordicbank.com/NBSer
6	http://www.nordicbank.com/NBSer
8	http://www.nordicbank.com/NBSer
10	http://www.nordicbank.com/NBSer
12	http://www.nordicbank.com/NBSer
14	http://www.nordicbank.com/NBSer
16	http://www.nordicbank.com/NBSer
18	http://www.nordicbank.com/NBSer
20	http://www.nordicbank.com/NBSer

Form Fields

Field Name	Value
StateD	California
SigninB	Sign-in
email	carol@windlass.com
password	XXXXXXXXXX

Account Login

Email: carol@windlass.com

Password: ••••••••••

State: California

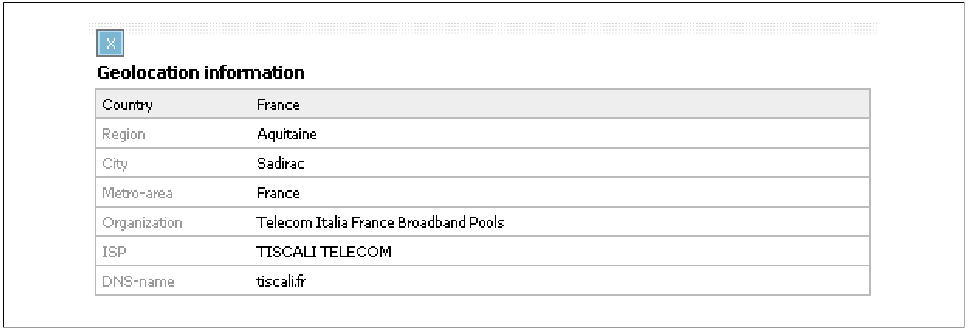
☐ Remember Login

Session First Use: 05/25/2007 11:06:05 AM   Session Last Use: 05/25/2007 11:07:28 AM   Page Count: 23   Last Update: 05/25/2007 11:12:28 AM

Figure 10-5. Replaying an entire user visit from a complete HTML capture with Tealeaf

- *Timing information* is the most common form of correlation. It lets you line up RUM with synthetic tests and other forms of analytics, such as conversion rates. For this, the RUM tool needs to synchronize its clock with that of the other data collectors. You can then merge records of user visits by timestamp to understand how user experience affects conversions.
- *Visit-specific information* (such as session cookies) is even more useful, because it lets you associate an individual visit with other systems, such as a record of an individual visit in a WIA tool or an individual customer survey in a VOC tool. This also lets you segment individual conversions (in analytics) by the actual experience that visitor had on the site (in RUM).
- *Personally identifiable data*, such as an account number or full name, can help you track customers as they move from a website to a call center or even a retail outlet. With this data, you can bring up a visitor's online session when he calls support or sales, and offer better service by seeing what happened, just as you do with WIA tools.

Actually joining RUM data to other information in a data warehouse is quite another matter, which we'll address in Chapter 17, but it's wise to collect correlation information ahead of time in case you need it in the future.



Geolocation information

Country	France
Region	Aquitaine
City	Sadirac
Metro-area	France
Organization	Telecom Italia France Broadband Pools
ISP	TISCALI TELECOM
DNS-name	tiscali.fr

Figure 10-6. Additional visit metadata based on a visitor's IP address

External metadata

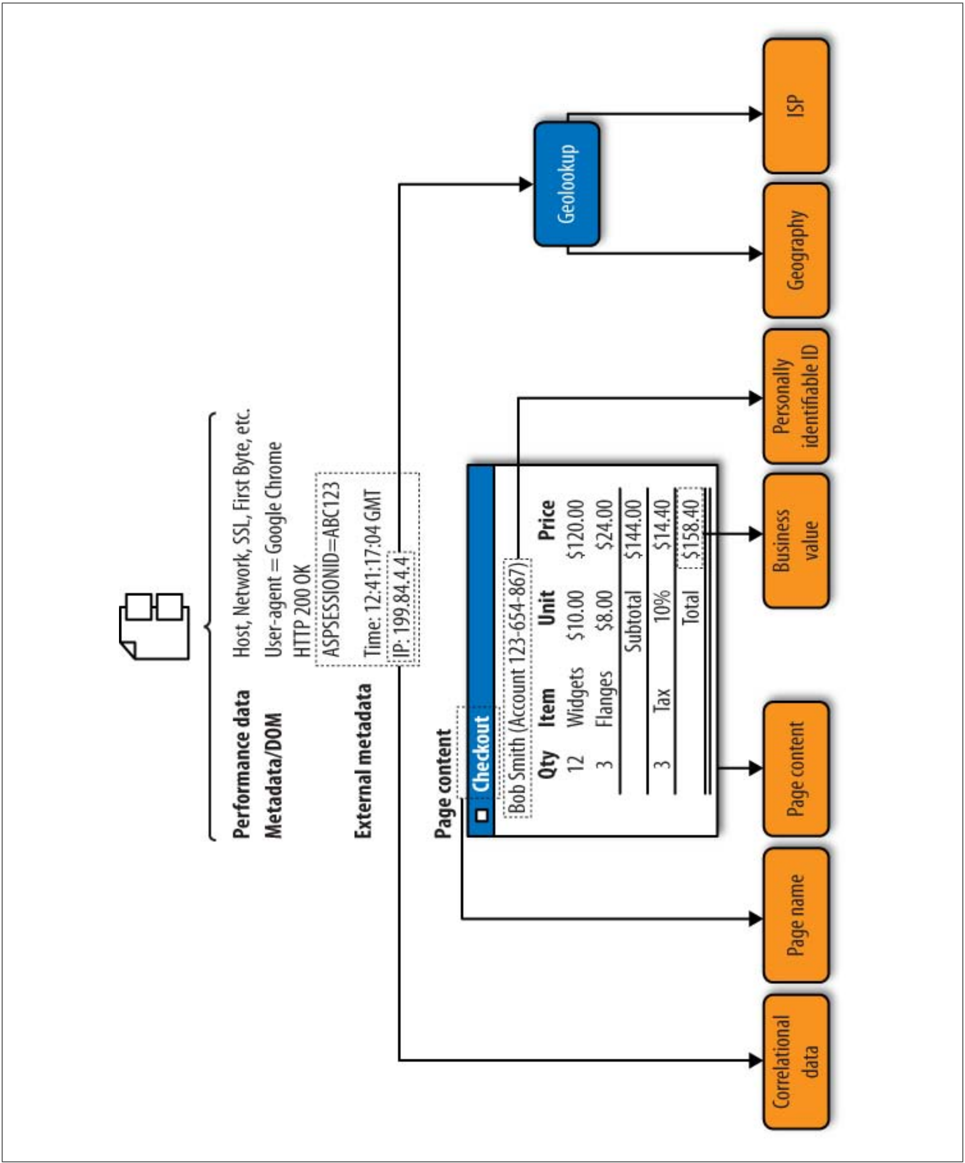
Some RUM packages look up the visitor's IP address in a database and include the municipality, country, and carrier in the session log, as shown in Figure 10-6. While not always accurate, this gives you some insight into where users are coming from.

Many of the databases used for geographic lookup also return the owner of the IP address, which will either be a company or a service provider. Service provider information helps to diagnose networking issues such as peering point congestion, since segmenting by service provider can reveal a problem that's happening in one carrier but not others.

Figure 10-7 shows how the various elements we've just seen can be extracted from a page in a visit.

In this figure, the delivery of a page (and its components) results in a record of the page request that includes:

- Performance data, such as host time, network time, and SSL latency based on the timing of specific events.
- Metadata from within the browser's DOM about the visitor's environment and the HTTP request/response.
- External metadata, like IP address and world time.
- Specific data that can be used to correlate the request with others, such as timing, address, and session cookies.
- Geographic and carrier information based on the IP address of the request.
- Data extracted from the content of the page, including the name of the page ("Checkout"), the identity of the visitor ("Bob Smith"), and the value of the page ("$158.40").



Qty	Item	Unit	Price
12	Widgets	$10.00	$120.00
3	Flanges	$8.00	$24.00
		Subtotal	$144.00
3	Tax	10%	$14.40
		Total	$158.40

Figure 10-7. Extracting performance information and metadata from a page request

Deciding How to Collect RUM Data

Having information on every user's visit is tremendously useful, both for troubleshooting individual incidents and for determining whether your website is living up to its promises.

When deploying a RUM tool, your first decision is how to collect all this data. Your approach to collection significantly affects what you can do with your RUM solution, and there are advantages and disadvantages to each approach.

We've seen that there are two major ways of collecting data: on the server side and on the client side. Server-side collection approaches include server logging, reverse proxies, and inline sniffers or passive analyzers. Client-side collection approaches include desktop agents and JavaScript instrumentation in the browser.

Much of this collection technology resembles the monitoring approaches we looked at for a web analytics implementation, but it's focused more on performance measurement. Consequently, passive analysis approaches are more common in RUM than they are in web analytics because they can collect network timing information and detect failed requests.

Server Logging

Web server logs give you only basic information about visitor performance. You'll have the timestamps at which each object was requested, and basic data like what was requested.

You may have more advanced logging, either through specialized software or within the application server. This can tell you about key milestones of an object request, such as when the request was received, when the server responded, and when the server was finished sending the object.

How server logging captures user sessions

A logging agent sits between the server's operating system and the application container. Some open source RUM tools, such as Glassbox, track each call to a Java Virtual Machine (JVM) and can provide a detailed hierarchical analysis of an application call to see exactly which function had problems, or to determine which database table caused a slowdown.

Server-side RUM tools like Symphoniq, which also rely on client-side JavaScript, can correlate end user experience with platform health, which allows them to span the gap between IT operations and end user experience.

How server logging captures timing information

Logging agents on application servers can time requests to the application from the network, as well as database requests or other backend transactions. Most server agents are more focused on the performance of the application tier (breaking down delay by component or query) than they are on reassembling a user's session and overall experience.

Server logging pros and cons

Logging is essential for application administrators, and forensic analysts may require logging to detect fraud or reproduce incidents. But it's not a popular indicator of end user experience unless it's combined with a client-side monitoring approach.

Here are some of the advantages of using server logging:

- It runs on the server.
- It can capture server health data (CPU, number of threads, memory, storage, etc.).
- It can get granular information on transactions within the application container/ JVM.
- It can include backend transaction latency (such as database calls).

However, server logging has some important limitations:

- It consumes server resources.
- Aggregating logfiles across servers is always problematic.
- A single visitor may hit multiple servers, making records incomplete.
- Servers have limited visibility into WAN health since they're behind load balancers.
- You can't see the ultimate rendering of the page that the end user sees.
- It doesn't see CDN performance.
- It can't measure mashups.
- When the server's down, so are the logfiles that could otherwise tell you what broke.

Reverse Proxies

Reverse proxy servers are located between the web server and the client and can be used to monitor end user experience. While this approach has fallen out of favor in recent years because it adds a point of failure to infrastructure, many load balancers behave much like reverse proxies and may have a role to play in performance monitoring.

How reverse proxies capture user sessions

A reverse proxy server terminates client requests and forwards them on to servers. Similarly, it terminates server responses and sends them to the client. It may respond to some requests, such as those for static images, on its own to offload work from the servers. Because it's terminating connections, it is also the endpoint for SSL encryption, so it may have access to data in plain text that is encrypted on the wire.

The result of reverse proxy data collection is a log of HTTP requests that resembles that of a web server, although some proxy servers offer more granular information that yields better visualization and analysis.

How reverse proxies capture timing information

Reverse proxy servers that record timings track milestones in a connection. The incoming request from the client, the status code response from the server, the first byte of the object the server sends, and the end of object delivery are used to calculate the performance of a page.

Because a reverse proxy is between the client and the server, it can measure the network health and performance of *both* ends of a connection. In other words, it may have two sets of TCP round-trip time information, one representing the Internet connection to the client and one representing the LAN connection to the server.

Reverse proxy pros and cons

Reverse proxies are servers in the middle of a connection. Unless they have to be there, they're probably another point of failure and delay for you to worry about. If you have a load balancer with logging capabilities, however, this may be an option you can use.

Reverse proxy collection provides the following advantages:

- It sits in the middle of the connection, so it sees both perspectives.
- If the proxy is already terminating SSL, it may simplify the monitoring of encrypted traffic.
- It may already be in place as a load balancer.
- It may be able to inject JavaScript, simplifying client-side instrumentation.

Some of the disadvantages of using a reverse proxy include:

- It introduces an additional single point of failure.
- It may introduce delay.
- It can't see the ultimate rendering of the page to the end user.
- It doesn't see CDN performance.
- It can't measure mashups.
- It may be a point of attack or vulnerability, and represents one more server to worry about.
- It's difficult to diagnose problems when the proxy is the cause.

Inline (Sniffers and Passive Analysis)

While reverse proxies actually intercept and retransmit packets across a network, there's another way to sit between the browser and the web server that doesn't interfere with the packets themselves: sniffing. This approach uses either a dedicated device (a *tap*, shown in Figure 10-8) or a spare port on a network switch (known as a *SPAN port* or *mirror port*) that makes a copy of every packet that passes through it.

Collectively, these approaches are known as *inline capture* or *passive analysis*.

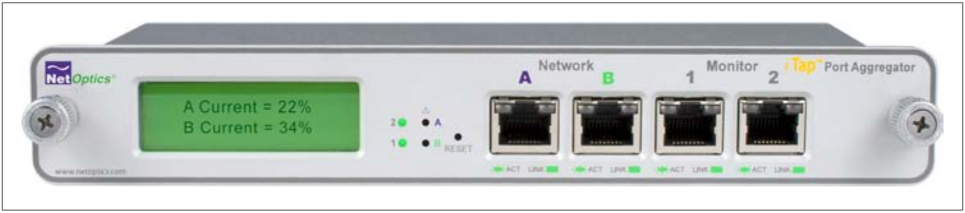


Figure 10-8. A network tap that makes a copy of all traffic flowing through it.

Should I Use Taps or Switches to Capture?

While network taps and mirror/SPAN ports both work well, there are times when you'll want to use one versus the other.

Connecting a tap requires that you disconnect the network cable. This results in downtime, even if only for a moment, which means you'll need to install monitoring during a maintenance window. Taps can also be expensive, consume a power socket on your rack, and in some cases may require configuration. However, if a network tap fails, it simply turns into an expensive piece of wire along which traffic can still flow. Once implemented, taps require little or no maintenance.

On the other hand, mirror ports use up a port on a switch that might be put to better use. If you paid a lot for your switch, or if it's full, you may not be able to use a network port for monitoring. You can implement a mirror port without disconnecting anything, so this approach is easier to set up without disrupting operations, and mirror ports don't use up another power outlet. Under heavy load, older switches won't copy all traffic to the mirror port, however; fortunately, most modern switches can make a copy of traffic at wire speeds without breaking a sweat.

The biggest issue with mirror ports is that they often change or go down due to networking teams and upgrades. It's hard for a team of analysts that doesn't own the routers to know what's going on when a mirror port suddenly disappears. Consequently, we recommend using mirror ports if you need to test something for a few weeks, and using network taps if you're implementing a more permanent solution. It's always a good idea to have a network tap in front of your load balancers just in case.

Sniffing traffic is a common practice for networking professionals. They rely on sniffers to detect virus traffic, even when there's no trace of an infection on a server. They use them to pinpoint Ethernet issues or to figure out which applications are running on a LAN. In fact, they use them anywhere they need to know what's really going on across a network connection.

In recent years, they're using enhanced versions of sniffing technology to measure end user experience. This approach, called *inline RUM*, is sometimes referred to as passive analysis because the monitoring doesn't generate any additional traffic on the network (as opposed to a synthetic monitoring approach, which is "active" because it generates traffic).

How inline devices capture user sessions

A tap or a mirror port copies traffic indiscriminately—every packet on the active network that's tapped is copied onto the monitoring connection. This means any inline RUM solution has to be good at blocking out nonweb traffic and at reassembling individual packets into web pages at wire speeds.

How inline devices capture timing information

To capture per-session timing, the device watches for important milestones—the start of a page request, the delivery of the last object, and so on—and uses timers and TCP/IP sequence number information to calculate latency. Because the device can see when packets arrive at and leave the server, it can measure extremely precisely. In fact, some inline RUM tools aimed at the financial industry such as SeaNet (*www.seanet-tech.com*) can report trade notification timings to the microsecond.

Inline device pros and cons

Inline RUM is precise and powerful, and sees problems even when the servers are broken or the pages aren't loaded in a browser. It can also be expensive—we're talking about network equipment, after all—and you probably won't be able to use it if you don't own your own hardware, because you're hosted by someone else or running your application in a cloud environment. It also doesn't see the end user's ultimate experience, because it's not on the browser itself.

Inline devices provide the following benefits:

- They don't lie: what you see on the wire is what happened.
- They are transparent, so there is no load on clients or servers.
- They do not present a point of failure in the network.
- You can upgrade and modify them without a maintenance window.
- They see performance even when the web page doesn't load.
- They work for any HTTP request, even when JavaScript isn't executing (mobile devices, RSS feeds, RESTful APIs, etc.).

Of course, inline capture devices have some important shortcomings:

- They are more expensive than other options.
- They require physical deployment in a network you control.
- They can't see the ultimate rendering of the page to the end user.
- They don't see CDN performance.
- They can't measure mashups.
- They have a hard time reassembling pages when the page contains RIA components such as AJAX or Flash, which may make additional HTTP requests at any time.

- They capture huge amounts of data, so storage may be an issue.
- They require a copy of the SSL key when sniffing encrypted traffic.
- You must ensure that security and compliance officers are OK with deployment because you're collecting data that is potentially sensitive.

Agent-Based Capture

One way to collect end user experience data is to put an agent on the user's desktop. This agent can see every aspect of application use, not just for the web application, but also for other applications. Want to know if the user's playing Minesweeper while she's on your site? Client agents will tell you. They've got access to the client's operating system, too, so agents know how healthy the network is and how much CPU resources are being used.

Unfortunately, you probably can't use them.

How agents capture user sessions

Agents are software applications installed on client desktops. They're used almost exclusively in enterprise applications, where they're part of company-wide management platforms that handle everything from antivirus updates to backup systems. They sit between the operating system and the applications, watching traffic between those applications and the operating system's resources.

Aternity, for example, makes desktop agents that track traffic flows to and from applications and that summarize the data and look for exceptions before sending performance metrics back to a management console.

How agents capture timing information

Agents see messages flowing in and out of applications. They can watch for specific strings or for operating system events (such as a window opening or a mouse click). They can also watch for network events like a new DNS lookup or an outbound HTTP request. Agents keep track of the timing of these events, as well as key operating system metrics.

Agent pros and cons

Agents see everything, but you need to own the desktop to install them. If you are able to make use of agents, you can take advantage of the following:

- They provide the best visibility into what the user is really doing.
- They can see system health information (CPU, memory).
- Much of the instrumentation work is done by the client, so this approach scales well as the number of users grows.

Agents have the following disadvantages:

- To use them, you will require access to the end user's desktop, so they are a non-starter for most Internet-facing web applications.
- They cannot see the network outside the end user LAN segment, so IP addressing, packet loss, etc., may be incorrect.
- They require different software for different operating systems (Linux, Windows, OS X, etc.).
- They slow down the client.
- Agents must be maintained by IT.

JavaScript

JavaScript changed the web analytics industry, and now it's transforming RUM. JavaScript-based monitoring sees what the user sees. This means it has a better view than any other web monitoring technology into the final assembly of the page, which may include client-side logic, plug-ins, and so on. It's the only way to capture the performance of mashups and third-party content.

What's more, JavaScript code can access everything the browser knows about the session and the user. This includes data such as cookies stored from previous visits or data on the number and size of browser windows. You can use this information to augment user performance with business and visitor context.

How JavaScript captures user sessions

JavaScript RUM begins with page instrumentation, just as web analytics does. You insert a snippet of JavaScript into your web pages or use an inline device like a load balancer to inject the snippet into pages as they're served. Either way, the visitor downloads a monitoring script that runs on the client.

The script records milestones of page arrival, and then sends performance metrics to a collector—a third-party service, a server, or the inline device that injected the script initially. To do this, the script requests a small image and appends the message it wants to send to the collector as a series of parameters to the URL. This is similar to JavaScript used for web analytics; in this case, however, the message's parameters contain performance and availability information.

Imagine that you're using performance monitoring service Example.com. Your JavaScript watches the page load, and at the end it determines that there were eight objects on the page and that it took 3.5 seconds (3,500 milliseconds) to load.

It then sends a request similar to the following:

```
http://www.example.com/beacon.gif?loadtime=3500&objectcount=8
```

The monitoring script doesn't care about a response—the use of a tiny image is intended to make the response as small as possible. The RUM system now knows that a page loaded, that it had eight objects, and that it and took 3.5 seconds.

The rest of the work, such as reporting, aggregation, and data storage, happens on the RUM service or appliance that received the request for the small object.

How JavaScript captures timing information

Recall from our earlier discussion in Chapter 5 that JavaScript is an event-driven language. To instrument a page as it loads, a monitoring script starts a timer and marks off the moments when important events occur. The first important event is the moment the page loads, and to capture this, the first part of the script appears right at the top of the page. This is the "first byte" time.

As the page loads, the browser generates other events, such as the `onLoad` event, which signifies that all objects have loaded. Simply by knowing the time the page started and ended, we can determine a useful performance measurement—how long the page took to deliver, otherwise known as *network time*.

Most JavaScript measurement happens in a similar fashion. Using the system's time (known as *epochtime*), measurements are determined by calculating the elapsed time between two events.

There's a problem, however. JavaScript is page-specific. When you load a new page, you load new JavaScript. There's no way to start a timer on page A (when the user clicks a link) and then stop the timer on page B (when the page loads), because everything related to page A ends when page B is loaded in its place.

There are good security reasons for this. If JavaScript didn't work this way, someone who'd instrumented site A with analytics could watch everything users did for the rest of their online time, even after leaving site A. This feature of JavaScript provides security and privacy to web users at the expense of being able to monitor their page performance.

Fortunately, developers have a way around the problem that doesn't undermine security. When a user is about to leave page A in a visit, the browser fires an event (`onBeforeUnload`) telling JavaScript that it's about to get rid of the current page and load a new one. JavaScript stores the current epochtime in a cookie, which is then available for the newly loaded JavaScript on page B.

JavaScript uses a cookie to store the time at which the user clicked the link. The script on page A effectively passes that start time to the script on page B, where it can be used to calculate the elapsed time—how long it took the server to receive and respond to the click that launched page B.

Despite its appeal, JavaScript still has many problems. Timing through JavaScript is more complex than for other collection models that time things independently of page loads, because *the JavaScript that monitors performance is itself part of the page being monitored.*

A recent initiative, called Episodes, addresses several of these problems.

JavaScript pros and cons

JavaScript sees everything from a user's perspective—when it's loaded properly—including third-party content and mashups. However, implementing it is usually vendor-specific, making switching services difficult. Furthermore, JavaScript can't see outside the browser's sandbox.

The following are advantages to using JavaScript:

- It sees all objects from all locations, so it's good for mashups and sites coming from CDNs.
- It sees client-side delay, so it knows when scripts or plug-ins are causing problems, and it measures "perceived render time."
- It knows exactly what the components of a page are.
- It can instrument user actions (clicking play on a video, for example) and make them part of the timing.
- It works in cloud computing and managed hosting environments because there's no need for access to servers and no hardware to install.

JavaScript still has some key limitations, however:

- If the JavaScript isn't loaded, you don't get any data, so it's not good for diagnosing problems.
- Power users may skip a page before JavaScript can run, resulting in gaps in monitoring.
- Using JavaScript increases page size and delay.
- It doesn't work for documents (PDF), RSS feeds, some mobile devices, or anywhere that there's no JavaScript being executed.
- Additional coding is required to instrument events beyond those in the DOM.
- It can't see anything outside the browser sandbox (TCP round-trip time, out-of-order segments, public IP address, etc.).
- It can't measure the server delay on the very first page of a visit because it lacks a timer from the previous page—there is no "previous page."
- It must be maintained along with other software within the web page, and is subject to release cycles and QA.
- It may introduce some privacy concerns, similar to web analytics, causing users to block third-party scripts.

JavaScript and Episodes

An effort by Steve Souders of Google (and the author of YSlow while at Yahoo!) may address the just described issues and give us an industry-wide approach to performance monitoring for rich Internet applications.

To understand Episodes, let's first look at the limitations of JavaScript monitoring today.

onLoad is not a reliable indicator of page load time
: Many applications aren't ready for the user at precisely the moment the browser's onLoad event occurs. Some are ready before the onLoad event because they've carefully loaded what users need first. Others have additional code to execute before pages are truly complete and ready for users. In both of these cases, we require a way to report a "truly ready" event.

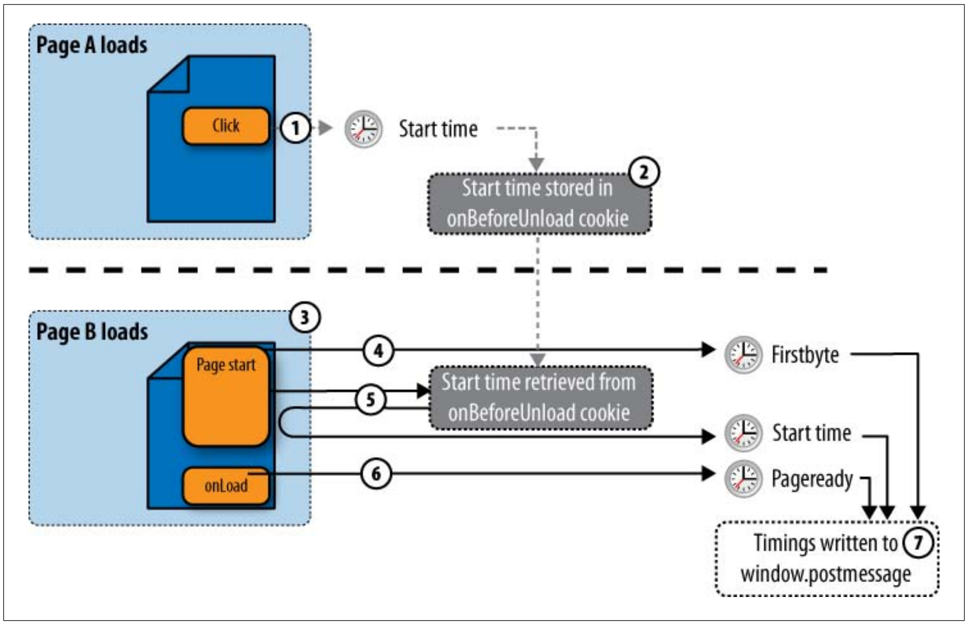
No standardization of important milestones
: Browsers support a limited number of timing milestones. Modern websites have unique milestones, such as the moment when video starts playing. Coders must write code to generate their own timing data to mark these milestones, so there isn't an easy way to compare the performance of two sites. The result is many proprietary definitions of timing. There's also no consistent way to track timing of the user's click on the preceding page, forcing coders to resort to cookies to store click time.

What is monitored and how it's collected are intertwined
: This is by far the biggest problem with JavaScript-based RUM today, and it's the one Episodes fixes most cleanly.

  In Chapter 6 we saw how stylesheets separate web design from content, making it easier for a designer to change the color of a heading across an entire site with just a single stylesheet change. Stylesheets are an example of specialization in web design: developers can code the application and make it visually appealing, while authors can focus on content.

  A similar problem exists with proprietary RUM approaches. The person who builds the application is not the person who's in charge of monitoring it. The developer knows which important milestones exist in the page—the rendering of a table, the loading of a video, or small messages back to the server. At the same time, the person monitoring the application knows what he wants to watch.

  Unfortunately, to monitor an application with JavaScript today, many developers are forced to design not only what is monitored, but also how it's reported back to a service for analysis. The timing of the page, the metrics to report, and the mechanism for reporting them are all intertwined in much the way content and formatting were with HTML in the early years of the Web. As Steve Souders says (*http://stevesouders.com/episodes/paper.php*), "There are drawbacks to the programmatic scripting approach. It needs to be implemented.... The switching cost is high. Actually embedding the framework may increase the page size to the point that it has a detrimental effect on performance. And programmatic scripting isn't a viable solution for measuring competitors."



Page A loads
Click
1
Start time
2
Start time stored in onBeforeUnload cookie
Page B loads
3
Page start
4
5
Start time retrieved from onBeforeUnload cookie
onLoad
6
Firstbyte
Start time
Pageready
Timings written to window.postmessage
7

Figure 10-9. How Episodes captures page timings with JavaScript

Episodes does for EUEM what stylesheets did for web design: it provides a model in which the developer defines milestones and measurements, but one in which those measurements can be collected independently by someone in charge of operations and monitoring.

How Episodes works

Figure 10-9 shows how Episodes works, particularly Steve Souders' *episodes.js* reference application.

Monitoring of a page's performance begins when the visitor leaves page A.

1. The monitoring script records the current time (Starttime) when the visitor clicks a link.
2. Starttime is stored in a cookie on the browser.
3. When page B loads, it includes a script near the start of the page.
4. That script records the current time as soon as it runs (which approximates the first byte of the page) in the DOM (in `window.postmessage`), calling it "Firstbyte."

5. The script also retrieves the Starttime left by the previous page from the locally stored cookie.
6. At the end of the page (the `onLoad` event) it records the Pageready timing. It may also record custom events the application developer wants to track (such as the start of a video). By measuring the elapsed time between these milestones, other timings (such as server time and network time) can also be calculated.
7. All of this information is stored in `window.postmessage`, where any other tool can receive it.

A browser plug-in could read the contents of that space and display information on timings. A synthetic testing site could grab those timings through browser puppetry and include them in a report. And a JavaScript-based RUM solution could extract the data as a string and send it back to a RUM service.

Where Episodes really shines, however, is in operational efficiency. So far, the developer has simply recorded important milestones about the page's loading in a common area. If the page changes, the developer can just move the snippets of code that generate Episodes milestones accordingly. If new functions (such as the loading of a video) need to be measured, the developer can publish these new milestones to the common area.

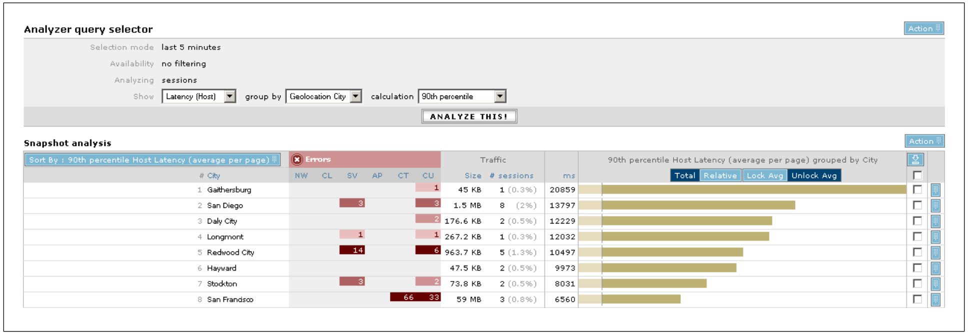
As a result, switching RUM service providers is trivial—just change the script that assembles the milestones and sends them to the RUM service. There's no need to change the way developers mark up the events on the pages. In the same way CSS separates the page's meaning from its formatting, Episodes changes the page's functional timings from the way in which they are collected and reported.

Episodes proposes several standard names and timings, as shown in Table 10-1.

Table 10-1. Episodes names and timings

Metric	What it is	How it's calculated
Starttime	The moment the previous page unloads (approximates a user's click)	Stored in the onBeforeUnload cookie by the preceding page's JavaScript and retrieved by the current page's script when loaded
Firstbyte	The moment the content is received from the server	Measured when the browser executes an Episodes message near the top of the page
Frontend	The time it takes for the browser to get the page prepared once a server response has been received	Pageready – firstbyte
Backend	The time it takes the server to prepare and send the content	Firstbyte – starttime
Pageready	The moment at which the page is ready for the user	Browser's onLoad event
Totaltime	The total time it takes a page to load	Pageready – starttime

You can calculate custom timings from new milestones and these default ones.



Analyzer query selector

Selection mode: last 5 minutes
Availability: no filtering
Analyzing: sessions
Show: Latency (Host) group by: Geolocation City calculation: 90th percentile

ANALYZE THIS!

Snapshot analysis

Sort By : 90th percentile Host Latency (average per page)

		Errors						Traffic			90th percentile Host Latency (average per page) grouped by City
#	City	NW	CL	SV	AP	CT	CU	Size	# sessions	ms	
1	Gaithersburg						1	45 KB	1 (0.3%)	20859	
2	San Diego			3			3	1.5 MB	8 (2%)	13797	
3	Daly City						2	176.6 KB	2 (0.5%)	12229	
4	Longmont			1			1	267.2 KB	1 (0.3%)	12032	
5	Redwood City			14			6	963.7 KB	5 (1.3%)	10497	
6	Hayward							47.5 KB	2 (0.5%)	9973	
7	Stockton			3			2	73.8 KB	2 (0.5%)	8031	
8	San Francisco					66	33	59 MB	3 (0.8%)	6560	

Figure 10-10. An aggregate report of host latency by city across visitor sessions in Coradiant TrueSight

The right answer: Hybrid collection

So what's the right approach for collection?

A combination of inline monitoring and a JavaScript-based programmatic script that's compatible with the Episodes approach is the right choice for RUM. The inline device has the most visibility into what is really happening, even when pages aren't loaded or servers aren't working, and is invaluable for troubleshooting. The JavaScript approach shows client-side activity, as well as mashups, CDN delivery, and third-party content that closely mimics end user experience. Together, they're unbeatable.

RUM Reporting: Individual and Aggregate Views

RUM yields two kinds of data: individual visits and aggregate reports.

Individual visits are great for diagnosing an issue or examining why a particular page was good for a particular user. They're the primary use of RUM in troubleshooting and customer support environments. They usually consist of a list of pages within a visit, along with timing information; in some products, each page can be viewed as a cascade diagram of container and component load times.

While looking at individual visits is useful, however, it's important to recognize that if you worry about a problem affecting only one visitor, you may overlook a more significant, widespread issue.

Aggregate calculations, on the other hand, give you a broad view of the application as a whole, as shown in Figure 10-10. They can, for example, show you a particular metric ("host latency") across a segment of your traffic ("the login page", "users from Boston", or "handled by server 10").

Aggregation also means making Top-N lists, which helps to prioritize your efforts by showing you the slowest, most errored, or busiest elements of your infrastructure. Finally, aggregate data is the basis for baselining, which decides what's normal for a particular region or page, and lets you know when something is unacceptably slow or broken.

Support teams, QA testers, and developers tend to use individual visit views, while aggregate data views are more often used for reporting and defusing SLA disputes.

Analyzing aggregate data has to be done properly. This is the place where statistics matter—you need to look at percentiles and histograms, not just averages, to be sure you're not missing important information or giving yourself a false sense of security.

RUM Concerns and Trends

Watching end user activity does present some concerns and pitfalls to watch out for, from privacy to portability and beyond. We've already considered many of the privacy concerns in the section on WIA, so be sure to check there for details on data collection.

Cookie Encryption and Session Reassembly

Some websites store session attributes in encrypted cookies. Unfortunately, obfuscating personally identifiable information may make it hard to reassemble a user's visit or to identify one user across several visits. Whenever the visitor changes the application state (for example, by adding something to a shopping cart) the entire encrypted cookie changes.

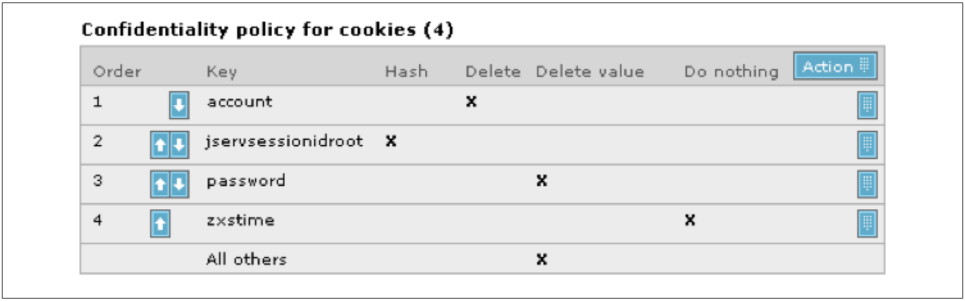
Your development team should separate the things you need to hide (such as an account number) from the things that you don't (such as a session ID). Better yet, store session state on the servers rather than in cookies—it's safer and makes the cookies smaller, improving performance. This is particularly true if your sessionization relies on the information in that cookie.

Privacy

RUM tools may extract content from the page to add business context to a visit record. While this is less risky than collecting an entire page for replay (as we do in some WIA tools), you still need to be careful about what you're capturing.

When you implement your data collection strategy, you should ensure that someone with legal authority has reviewed it. In particular, pay attention to POST parameters, URI parameters, and cookies. You'll need to decide on a basic approach to collection: either capture everything except what's blocked, or block everything that's not explicitly captured.

A permissive capture strategy might, for example, tell the RUM solution to blank out the POST parameter for "password." Unless it's explicitly blocked, it will be stored. Permissive capture means you may accidentally collect data you shouldn't, but it also means that a transcript of the visit will contain everything the visitor submitted, making it easier to understand what went wrong during the visit.


Confidentiality policy for cookies (4)

Order	Key	Hash	Delete	Delete value	Do nothing	Action
1	account		x			
2	jservsessionidroot	x				
3	password			x		
4	zxstime				x	
	All others			x		

Figure 10-11. Configuring confidentiality policies in Coradiant TrueSight

On the other hand, a restrictive capture strategy will capture *only* what you tell it to. So you might, for example, collect the user's account number, the checkout amount, and the number of items in a shopping cart. While this is the more secure approach (you won't accidentally collect things you shouldn't), it means you can't go back and look for something else later on. Figure 10-11 shows an example of a restrictive capture configuration screen in a RUM tool—everything that isn't explicitly captured has its value deleted from the visit record.

RIA Integration

We've looked at programmatic RUM using client-side JavaScript. More and more applications are written in browser plug-ins (like Flash and Silverlight) or even browser/desktop clients (Adobe AIR and Sun's Java FX, for example.)

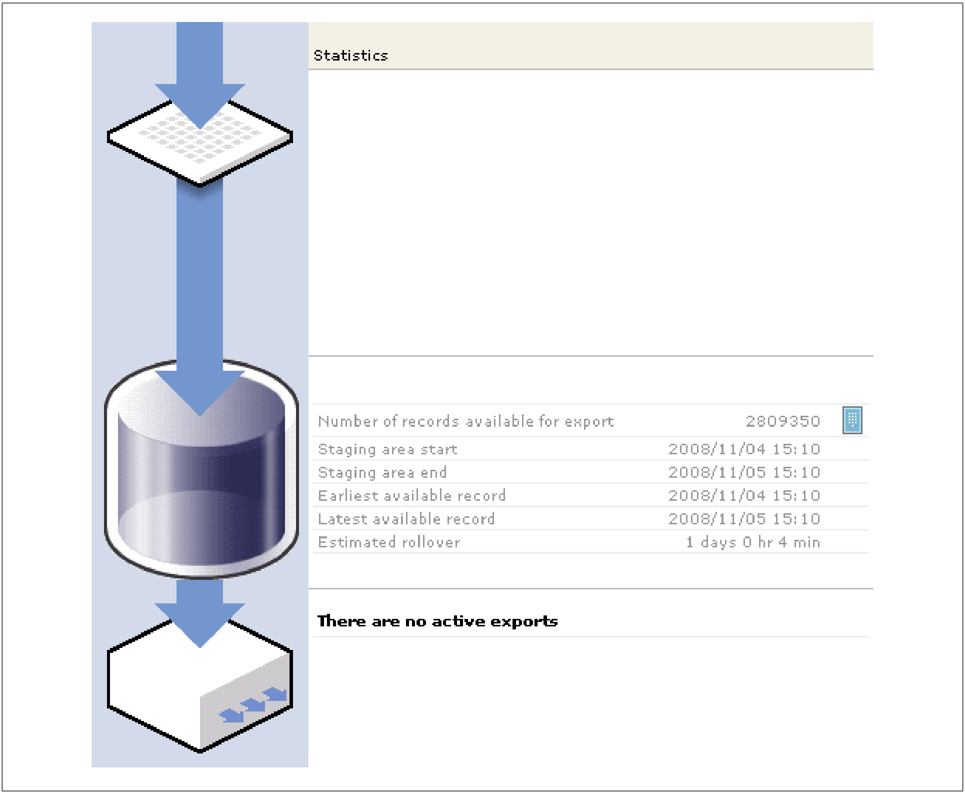
The methods described here for sending messages back to a hosted RUM service work just as well for RIAs. The application developer has to create events within the application that are sent back to the service. Episodes is a good model for this because it's easily extensible. As part of their RUM offerings, some solutions provide JavaScript tags or ActionScript libraries that can also capture multimedia data like startup time, rebuffer count, rebuffer ratio, and so on.

Storage Issues

As we've noted, capturing user sessions generates a tremendous amount of information, particularly if those sessions include all of the content on the page itself. If you're planning on running your own RUM, make sure your budget includes storage.

Many server-side RUM tools allow you to extract session logs so that they can be loaded into a business intelligence (BI) tool for further analysis (Figure 10-12).

With a hosted RUM service, it's important to understand the granularity of the offering, specifically whether it can drill down to an individual page or object, as well as the length of time that the stored information is available. Some systems only store user session information for sessions that had problems or were excessively slow.



Number of records available for export	2809350
Staging area start	2008/11/04 15:10
Staging area end	2008/11/05 15:10
Earliest available record	2008/11/04 15:10
Latest available record	2008/11/05 15:10
Estimated rollover	1 days 0 hr 4 min

There are no active exports

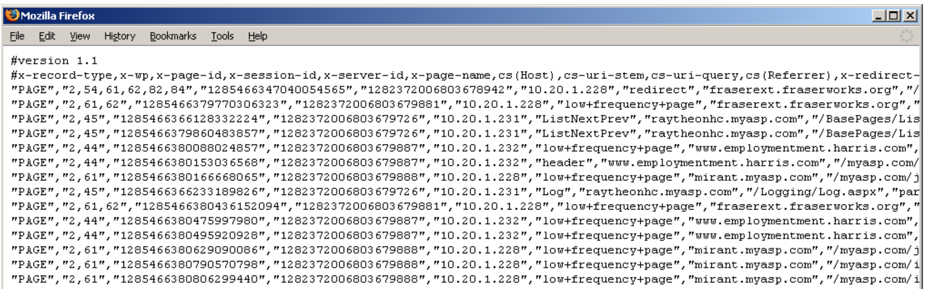
Figure 10-12. Bulk data export in Coradiant's TrueSight

Exportability and Portability

RUM data must be portable. Whatever technology you deploy, you need to be sure you can take your data and move it around. Often, this will be in the form of a flat logfile (for searching) or a data warehouse (for segmentation and sharing with other departments).

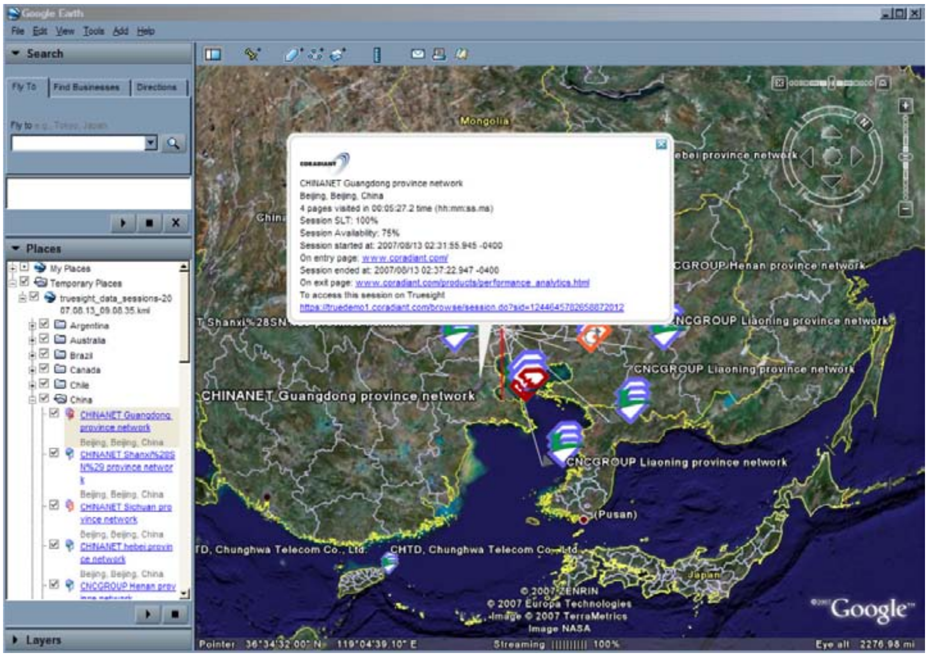
With the advent of new tools for visualization and data exchange, you will often want to provide RUM in real time and in other formats. For example, if you want to stream user events to a dashboard as structured data, you'll want a data feed of some kind, such as the one shown in Figure 10-13.

You may also want to overlay visitor information atop third-party visualization tools such as Google Earth, particularly if you're trying to find a geographic pattern. For example, you may want to demonstrate that visitors who are prolific posters are in fact coming from a single region overseas and are polluting your community pages with blog spam, as is the case in Figure 10-14.



```
#version 1.1
#x-record-type,x-wp,x-page-id,x-session-id,x-server-id,x-page-name,cs(Host),cs-uri-stem,cs-uri-query,cs(Referrer),x-redirect-
"PAGE","2,54,61,62,82,84","1285466347040054565","1282372006803678942","10.20.1.228","redirect","fraserext.fraserworks.org","/
"PAGE","2,61,62","1285466379770306323","1282372006803679881","10.20.1.228","low+frequency+page","fraserext.fraserworks.org","
"PAGE","2,45","1285466366128332224","1282372006803679726","10.20.1.231","ListNextPrev","raytheonhc.myasp.com","/BasePages/Lis
"PAGE","2,45","1285466379860483857","1282372006803679726","10.20.1.231","ListNextPrev","raytheonhc.myasp.com","/BasePages/Lis
"PAGE","2,44","1285466380088024857","1282372006803679887","10.20.1.232","low+frequency+page","www.employmentment.harris.com",
"PAGE","2,44","1285466380153036568","1282372006803679887","10.20.1.232","header","www.employmentment.harris.com","/myasp.com/
"PAGE","2,61","1285466380166668065","1282372006803679888","10.20.1.228","low+frequency+page","mirant.myasp.com","/myasp.com/j
"PAGE","2,45","1285466366233189826","1282372006803679726","10.20.1.231","Log","raytheonhc.myasp.com","/Logging/Log.aspx","par
"PAGE","2,61,62","1285466380436152094","1282372006803679881","10.20.1.228","low+frequency+page","fraserext.fraserworks.org","
"PAGE","2,44","1285466380475997980","1282372006803679887","10.20.1.232","low+frequency+page","www.employmentment.harris.com",
"PAGE","2,44","1285466380495920928","1282372006803679887","10.20.1.232","low+frequency+page","www.employmentment.harris.com",
"PAGE","2,61","1285466380629090086","1282372006803679888","10.20.1.228","low+frequency+page","mirant.myasp.com","/myasp.com/j
"PAGE","2,61","1285466380790570798","1282372006803679888","10.20.1.228","low+frequency+page","mirant.myasp.com","/myasp.com/i
"PAGE","2,61","1285466380806299440","1282372006803679888","10.20.1.228","low+frequency+page","mirant.myasp.com","/myasp.com/i
```

Figure 10-13. Raw data of individual object requests from a streaming API


Google Earth
File Edit View Tools Add Help
Search
Fly To
Find Businesses
Directions
Places
My Places
Temporary Places
truesight_data_sessions-20 07.08.13_09.08.35.kml
Argentina
Australia
Brazil
Canada
Chile
China
CHINANET Guangdong province network
CHINANET Shanxi%28S N%29 province network
CHINANET Sichuan province network
CHINANET hebei province network
CNCGROUP Henan province network
Layers
Mongolia
CHINANET Guangdong province network
CNCGROUP Henan province network
CNCGROUP Liaoning province network
CHTD, Chunghwa Telecom Co., Ltd.
(Pusan)
Japan
Google
Pointer 36°34'32.00" N 119°04'39.10" E
Streaming 100%
Eye alt 2276.98 mi

CHINANET Guangdong province network
Beijing, Beijing, China
4 pages visited in 00:05:27.2 time (hh:mm:ss.ms)
Session SLT: 100%
Session Availability: 75%
Session started at: 2007/08/13 02:31:55.945 -0400
On entry page: www.coradiant.com/
Session ended at: 2007/08/13 02:37:22.947 -0400
On exit page: www.coradiant.com/products/performance_analytics.html
To access this session on Truesight
https://truedemo1.coradiant.com/browse/session.do?sid=1244645782658872012

Figure 10-14. User visits showing performance and availability, visualized in Google Earth

These kinds of export and visualization are especially important for gaining executive sponsorship and buy-in, since they present a complex pattern intuitively. When selecting a RUM solution, be sure you have access to real-time and exported data feeds.

Data Warehousing

Since we're on the topic of data warehousing, let's look at some of the characteristics your RUM solution needs to have if it is to work well with other analytical tools.

- It must support regular exports so that the BI tool can extract data from it and put it into the warehouse at regular intervals. The BI tool must also be able to "recover" data it missed because of an outage.
- It must mark session, page, and object records with universally unique identifiers. In this way, the BI tool can tell which objects belong to which pages and which pages belong to which sessions. Without a way of understanding this relationship, the BI tool won't be capable of drilling down from a visit to its pages and components.
- If the data includes custom fields (such as "password" or "shopping cart value"), the exported data must include headers that allow the BI tool to import the data cleanly, even when you create new fields or remove old ones.

We'll look at consolidating many sources of monitoring data at the end of the book, in Chapter 17.

Network Topologies and the Opacity of the Load Balancer

A load balancer terminates the connection with clients and reestablishes its own, more efficient connection to each server. In doing so, it presents a single IP address to the Internet, even though each server has its own address. This means that the server's identity is opaque to monitoring tools that are deployed in front of the load balancer, including inline monitoring devices and client-side monitoring.

To overcome this issue, some load balancers can insert a server identifier into the HTTP header that the RUM tool can read. This allows you to segment traffic by server even though the server's IP address is hidden. We strongly suggest this approach, as it will allow you to narrow a problem down to a specific server much more quickly. You can use a similar technique to have the application server insert a server identifier, further enhancing your ability to troubleshoot problems.

Real User Monitoring Maturity Model

We've seen the maturity model for web analytics; now let's look at the model for web monitoring. There are two parallel types of monitoring: synthetic testing and RUM. As the organization matures, its focus shifts from bottom-up, technical monitoring to top-down, user-centric monitoring. It also moves from simple page analysis to the automatic baselining and alerting of transactions, and to tying the performance and availability of the site back to analytics data about business outcomes.