

# JAVA ENTERPRISE PERFORMANCE

ALOIS REITBAUER  
KLAUS ENZENHOFER  
ANDREAS GRABNER  
MICHAEL KOPP  
STEPHEN PIERZCHALA  
STEVE WILSON

# Java Enterprise Performance

## Chapter 1 Application Performance Concepts

By

**Alois Reitbauer**

© 2012 Compuware Corporation

All rights reserved under the Copyright Laws of the United States.



This work is licensed under a  
Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.

## Chapter 1

# Application Performance Concepts

Application performance is an area of increasing importance. We are building bigger and bigger applications. The functionality of today's applications is getting more and more powerful. At the same time we use highly distributed, large scale architectures which also integrate external services as central pieces of the application landscape. In order to optimize these applications we need profound knowledge of how to measure and optimize the performance of Enterprise applications. Before we delve into the technical details how to optimize performance we look at the important underlying concepts.

A solid understanding of the topic helps to manage the performance of an application more efficiently. It will help to make performance management appear less complex. As you understand the technical background, many things suddenly get much simpler and you will also see your efficiency in solving performance problems increase.

While we discuss fundamental principles of performance management, experienced readers will appreciate a detailed discussion of important performance concepts. Especially a solid understanding of different measurement techniques and their advantages and drawbacks as well as measurement overhead and measurement precision is key in interpreting measurement results. A deep understanding of data collection techniques and data representation also helps us in selecting the proper tools for our analysis tasks.

## Differentiating Performance from Scalability

Throughout this book, we'll be discussing performance and scalability: how to measure them, identify problems, and then optimize. And yet, many people use these terms as synonyms. People are likely to say that "the performance of the application is bad", but does this mean that response times are too high or that the application cannot scale up to more than some large number of concurrent users? The symptoms might be the same, but these are two quite different problems that describe different characteristics of an application.

When discussing performance we must carefully differentiate these two points. It's not always an easy task, because at the same time, they are interrelated and the one can, and often does, affect the other.

## Defining and Measuring Performance

Application state determines the way in which requests are processed. The current load, the complexity of the request, and other application and system factors (like CPU or network usage) all impact application responsiveness. It is the characteristics of response that define application performance. More specifically, there are three basic measures of performance:

- Response time: This is the most widely used metric of performance and it is simply a direct measure of how long it takes to process a request.
- Throughput: A straightforward count of the number of requests that the application can process within a defined time interval. For Web applications, a count of page impressions or requests per second is often used as a measure of throughput.
- System availability: Usually expressed as a percentage of application running time minus the time the application can't be accessed by users. This is an indispensable metric, because both response time nor throughput are zero when the system is unavailable.

Performance can also be defined by resource requests and by measuring resource requests in relation to throughput. This becomes a key metric in the context of resource planning. For instance, you may need to know which resources are needed to achieve full application scaling, which is a matter of knowing the frequency of requests for each resource type.

Resource consumption is becoming even more important for applications deployed in “elastic” environments, such as the cloud. With these larger scale and geographically-dispersed applications, scalability and performance become inextricably linked that can most easily be described from a resource-centric point of view.

Remember, even the most amply-supplied and equipped applications have limited resources. At the same time, increasing system load will make unequal demands on whatever resources are required. This becomes the ultimate, real-world, measure of performance.

The more requests users send to the application, the higher the load. The results can be plotted as simple as basic mechanics. So much as friction or gravity will ultimately bring a moving body to rest, as load increases, performance decreases.

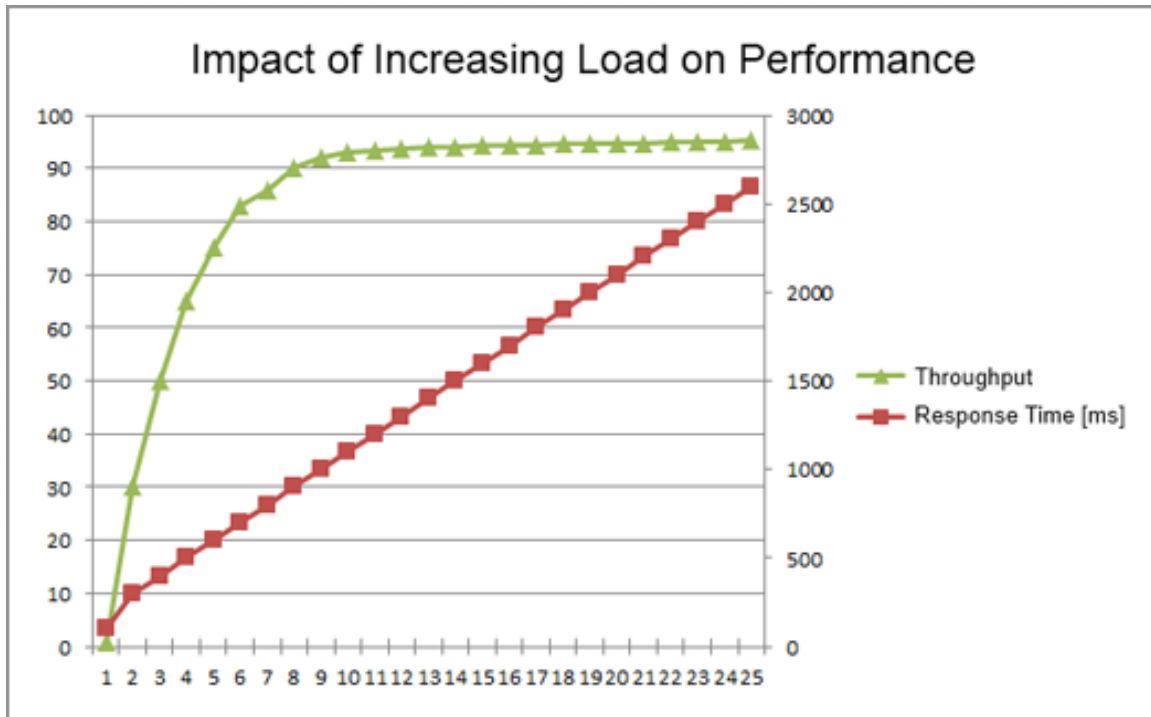


Figure 1.1: Increasing load affects the response time and throughput of an application

In the same way, application performance will always be affected by resource constraints. Your users may experience these constraints as application instability, but the underlying cause can be any of the metrics that define application performance.

It is therefore vitally important to describe performance as a ratio of some measurable quantity—response, throughput, availability, or requests—over time. One is easily tempted to use a kind of “black-box” shorthand, such as:

*Response time is 2 seconds.*

Don’t do it! Without a properly descriptive context, this sort of statement is meaningless and ultimately useless. Instead, be verbose:

*System response time is 2 seconds at 500 concurrent requests, with a CPU load of 50%, and a memory utilization of 92%.*

We’ll discuss tools for measuring performance as well as methods for dealing with these performance issues in the following chapters.

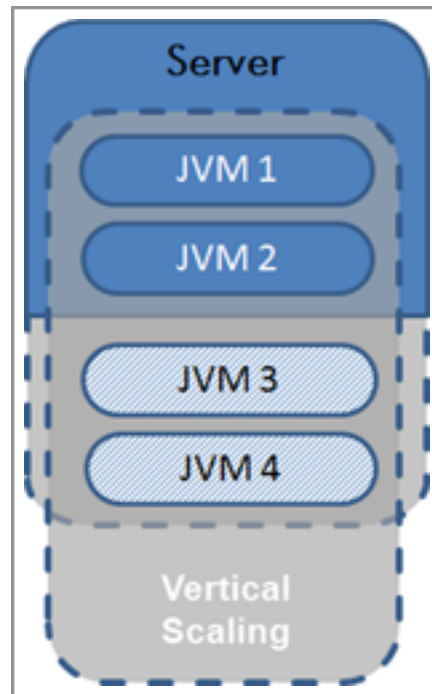
## How to Enable Application Scalability

The ability to overcome performance limits by adding resources is defined as scalability. No matter how much hardware we have at a certain point we will see decreasing performance. This means increasing response times or a limit in throughput. Will adding additional hardware solve the problem? If yes, then we can scale. If not, we have a scalability problem.



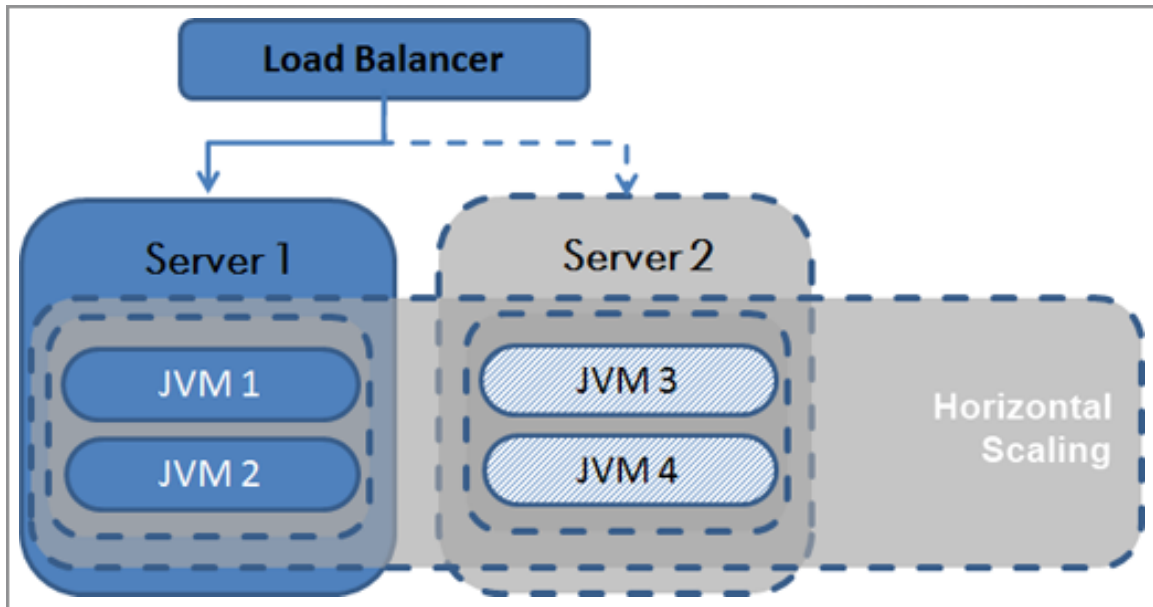
Depending on the changes to the application topology we differentiate the following two scalability approaches:

Vertical Scaling or scaling up a single node: Adding hardware resources to an existing node. This is often used in virtualized environments where it's possible to expand system resources dynamically. It has the clear advantage that it requires no changes to the application architecture. At the same time the level of scalability will always be hardware constrained.



*Figure 1.2: Vertical scaling by adding hardware resources on to existing servers*

Horizontal Scaling or scaling out by adding nodes: Dispatching requests among additional nodes is the preferred scaling solution when it is easier to add additional nodes than increasing the resources of a single node. This is, especially true in cloud-based environments (see figure below). Eventually we have to scale horizontally anyways as a single machine always has limited capacity. An additional advantage is that inexpensive nodes provide both increased failover capacity and improved application availability. The obvious disadvantage is that clusters must be designed so that nodes can be added and removed with ease.



*Figure 1.3: Horizontal Scaling by adding new servers, scaling out*

Comparing these two scaling methods, Vertical Scaling is the simpler of the two and therefore more easily implemented. Horizontal Scaling requires the redistribution and synchronization of data across the new node configuration. Without proper planning, it can be expensive and disruptive to retrofit clusters for scaling out.

### **Will Scaling Solve Our Performance Problems?**

Scaling an application by adding resources is the ideal case, but at some point, this will become too expensive. We must consider adding hardware, scaling out, or changing our architecture.

However frequently the scalability problem is not hardware related and adding hardware will not help. If we observe that resources are not overloaded then this usually indicates a synchronization-related problem, which is often related to serialized access to shared data rather than a performance problem. Adding resources will not help because this is not where the bottleneck is.

Let's take the example of updating an inventory. If we want to keep it consistent at all times, then only one process may update it at any given time. The duration of this processing indicates where our scalability limits are. If processing takes 200 milliseconds, the scalability threshold of our application is five requests per second—regardless of the available hardware.

If we want to achieve higher throughput we have to improve the overall performance rather than improving scalability. This could mean that we decide to give up consistency. We would place our inventory changes in a queue that are then updated at some later point in time. This is called "eventual consistency". With this approach it is however no longer possible to check whether a certain amount of an item is currently in stock.

As this example shows we really need good performance measures to guide us in making scalability decisions.

## **Calculating Performance Data**

Performance analysis is always based on numbers. In many cases we are not working with "raw values" but rather aggregated data. If each measurement were viewed individually, you couldn't see the forest for the trees. At the same time certain measurement approaches, like JMX (which we will discuss later), only provide us with aggregated data.

Understanding how a value is calculated and what this means is essential for drawing the right conclusions. Toward this end, we must examine the statistical methods used to calculate and aggregate performance data.

### **Working with Averages**

The average is the most basic and also most widely used data representation and is built into every performance tool. It is likely the most overused measure, because it can only provide a "first impression" of performance. For instance, the average of a series of volatile measurements, where some values are very low and others very high, can easily be skewed by outliers. So while the average might look good, it can fail to reveal an actual performance problem.

The aggregation interval can skew things even further. When measurement data is subject to time fluctuations over longer durations, the average can't reflect this information and loses its meaning. Similarly, over a very short period in which only a small number of measurements is taken, average is simply statistically imprecise.

For example, if we examine the response time of an application over a period of 24 hours, the peak values will be hidden by the average. Measurements taken when the system was under low load having good response times will "average out" times of peak system load.

### **Interpreting Minimum and Maximum Values**

Minimum and maximum values give us a measure of both extremes and the spread of these extremes. Since there are always outliers in any measurement, they are not necessarily very meaningful. In practice this data is rarely used, as it does not provide a basis for determining how often the corresponding value has actually occurred. It might have occurred one time or a hundred times.

The main use of these values is to verify how high the quality of the calculated average value is. If these values are very close, it can be assumed that the average is representative for the data.



In applications with demanding performance requirements, the maximum is still often used instead of, or in addition to, the average in order to check whether the response times lie below a certain level. This is especially true for applications which must never exceed certain thresholds.

### **Replacing the Average with the Median**

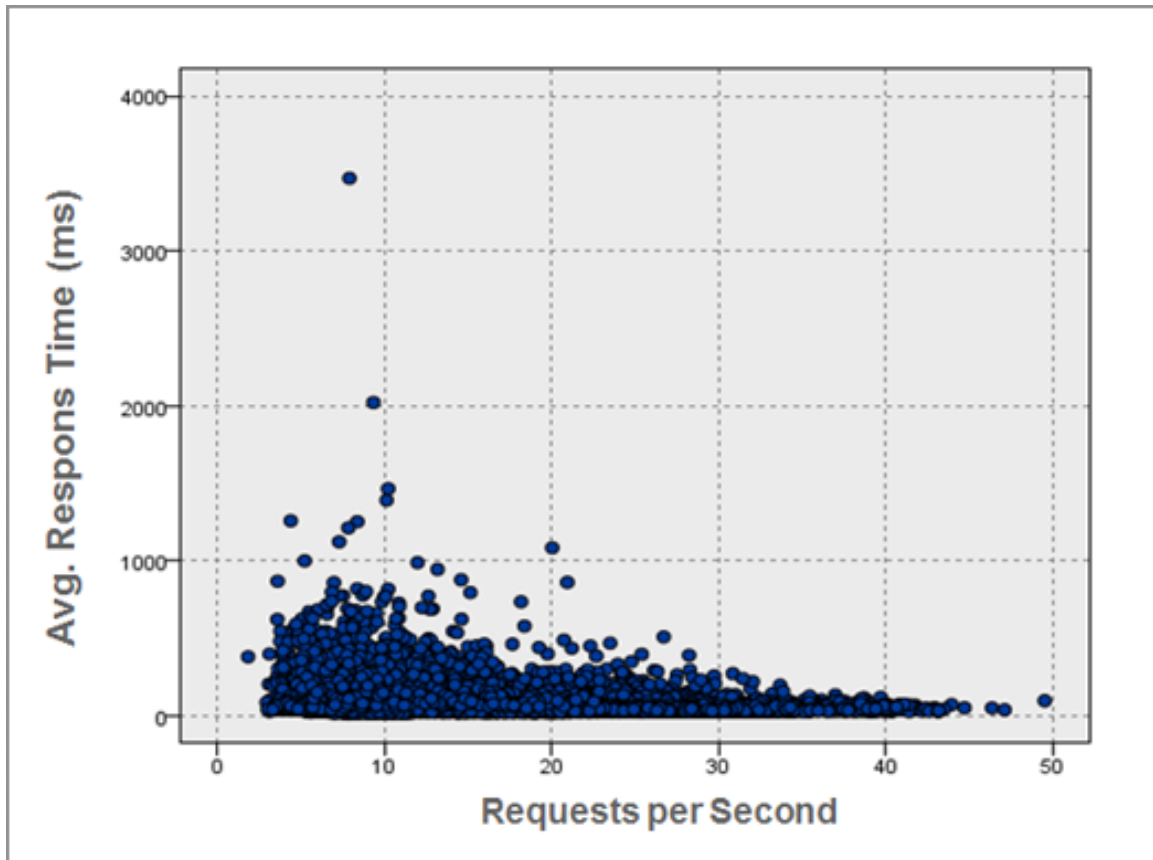
The median or middle value of a series of numbers is another widely used representation of performance data. It is also called the 50th percentile; In a series of 89 measurements, the median would be the 45th measurement. The advantage of the median is that it is closer to real world data and not an artificially calculated value like the average which is strongly influenced by outliers. The impact of outliers on the median is much lower than on the average.

### **Using the Average in Combination with Standard Deviation**

Taken by itself, average has only limited significance. However, it's considerably more meaningful when use with [standard deviation](#). The standard deviation measures the spread of actual values. The greater the standard deviation, the larger is the difference in the measurement data. Assuming that our values are distributed normally, at least two thirds of all values will fall within the range covered by the average plus/minus one standard deviation.

When working with the standard deviation, we assuming the data is normally distributed. If our data does not follow this distribution, then the standard deviation is meaningless. This means that we have to understand the characteristics of the underlying data before we rely on statistically derived metrics.

The figure below shows an example of the response time distribution of an application. We can clearly see that these values are not normally distributed. In this case, using average with standard deviation will give us incorrect wrong conclusions.



*Figure 1.4: Response Time Data not showing a Normal Distribution*

There are many causes for this kind of variation in measurement data. For instance, genuinely volatile response times probably indicate a problem with the application. Another possibility is that the individual requests differ in what they are actually doing. We might have also combined data points into single measurements which measure different things.

For example, when examining the response times of a portal start page, our data will vary greatly depending on how many portlets are open in each instance. For some users, there may be only one or two, while for others, there can be twenty or more per page. In this case, it is not possible to draw reasonable conclusions about the response times of the start page, because there is no specific "start page." The only way to get meaningful measurements in this case is to aggregate the measurements differently. For instance, we could group measurement data by the number of portlets that are shown on a page.

### **Working with Percentiles**

Percentiles are probably the most precise, but also the most difficult to calculate representation of data. Percentiles define the maximum value for a percentage of the overall

measurements. If the 95th percentile for the application response time is two seconds, this means that 95% of all response times were less than or equal to two seconds.

Higher percentile values, meaning the 95th to 98th, are often used instead of the average. This can eliminate the impact of outliers and provide a good representation of the underlying raw data.

The concept for the calculation is very simple and similar to the median. The difference is that we not only divide the measurements at 50 percent, but according to the percentile ranges. So the 25th percentile means that we take the lowest 25 percent of all measured values.

The difficulty lies in calculating percentiles in real time for a large number of values: more data is necessary for this calculation compared to the average, which requires only the sum and the number of values. If it is possible to work with percentiles, they should definitely be used. Their meaning is easier to understand than the average combined with standard deviation resulting in faster and better analysis

## Collecting Performance Data

The first step in performance management is to collect proper measurements. As there is a variety of different measurement approaches we will look at the most widely used ones to understand when to use them and how they are interpreted.

### JMX

[Java Management Extension \(JMX\)](#) is a standard technology that has been part of the JVM since Java 5. It makes it possible to manage and monitor applications and system resources. Resources are represented as so-called MBeans (Managed Beans).

The JMX architecture consists of three layers.

- The instrumentation level consists of the actual MBeans. This is the only layer you normally get in touch with.
- The agent level or MBean server, which represents the interaction between the MBeans and the application.
- The distributed service level, which enables access to the MBeans via connectors and adapters.

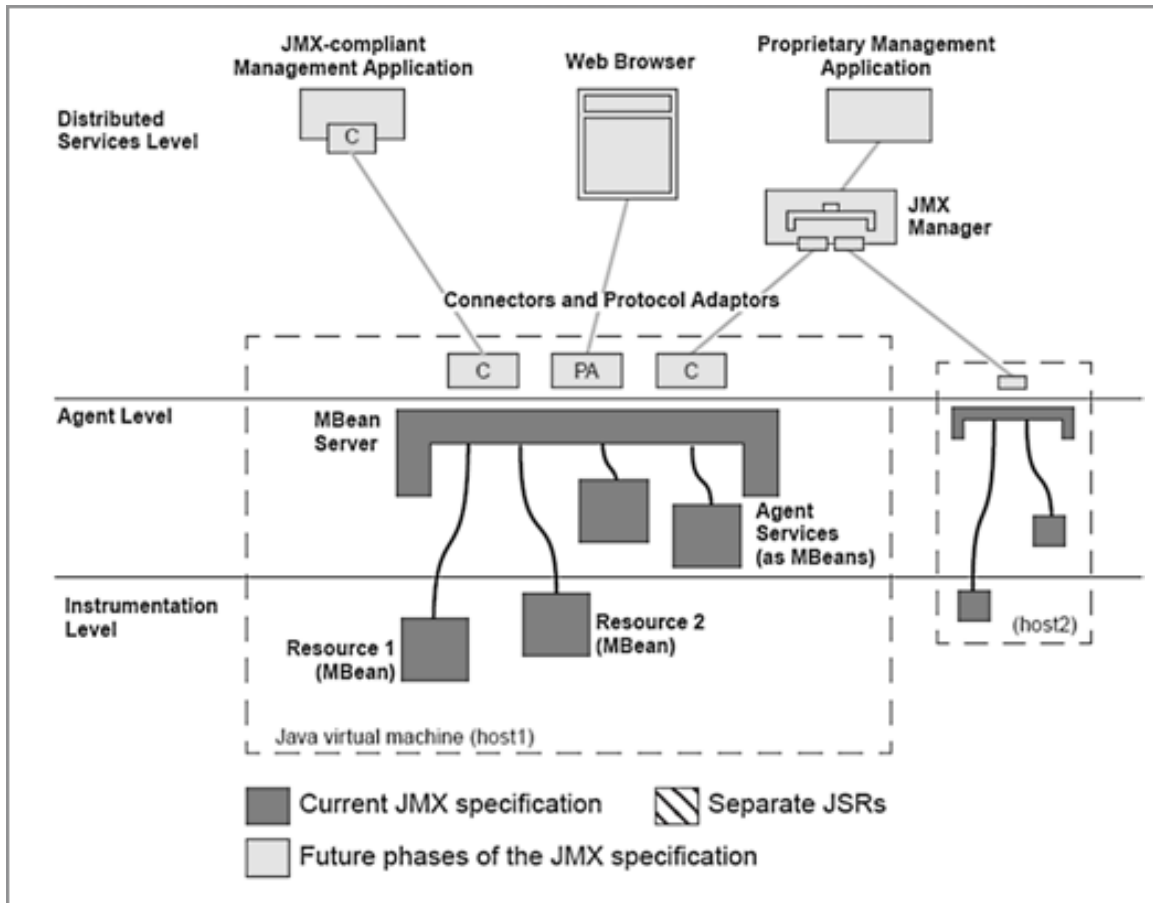


Figure 1.5: Overview of JMX: Illustration of the JMX Management Specification, v1.0

JMX management tools provide access to the MBeans—meaning the actual monitoring data—of a local or remote JVM. JConsole and VisualVM, parts of the JDK, provide a simple way to query a process's performance values, which are accessible via the MBeans.

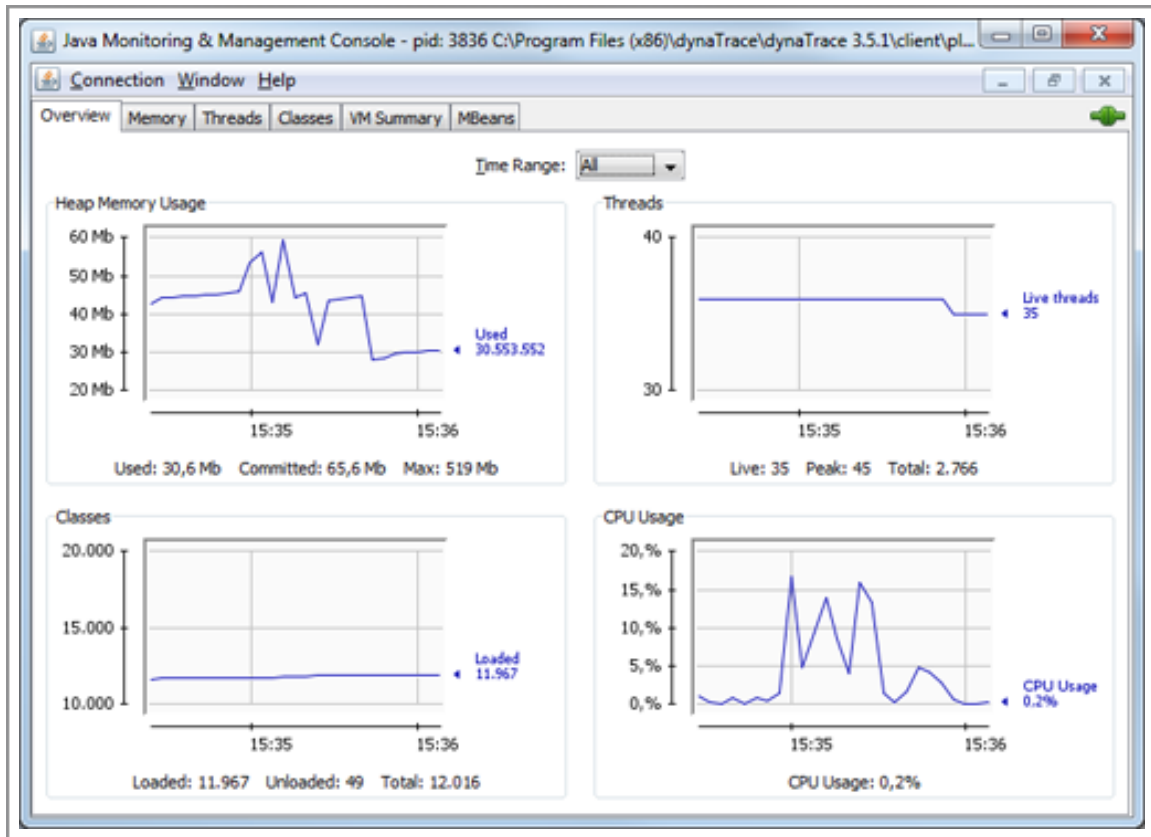


Figure 1.6: JConsole uses JMX to access the MBeans of a Java application

## Using JVMPI/JVMTI to Collect Performance Data

The [Java Virtual Machine Tooling Interface \(JVMTI\)](#) was introduced with Java 5 and replaces [Java Virtual Machine Profiling Interface \(JVMPI\)](#), which was available in previous JVM versions. JVMTI is a native interface and allows C/C++ agents to access the internal JVM state, register for events, and modify bytecode. Tool providers use this interface for debugging, profiling and monitoring as it provides universal access to JVM internals. As the interface executes native code, the overhead for retrieving performance data can be kept very low depending on how data is retrieved.

When the JVM is started, the native JVMTI agent which is specified via a JVM command line option, is loaded into the Java process. The agent then registers for all events it is interested and then receives callbacks while the application is running.

There is also the option of implementing agents in Java. These can be used easily across platforms. Java agents however lack some of the features only available to the native interface. The most important function used by performance tools—bytecode instrumentation—is possible using both native and Java agents.

## Bytecode Instrumentation

Bytecode instrumentation refers to changing the Java bytecode of a Java class before or while it is executed by the Java Runtime. This approach is widely used by profiling, diagnostic, and monitoring tools. In order to measure the execution time for a particular method, the method's bytecode is changed by adding Java code at both the start and the end of the method. The instrumented code is executed along with the method and the execution time recorded.

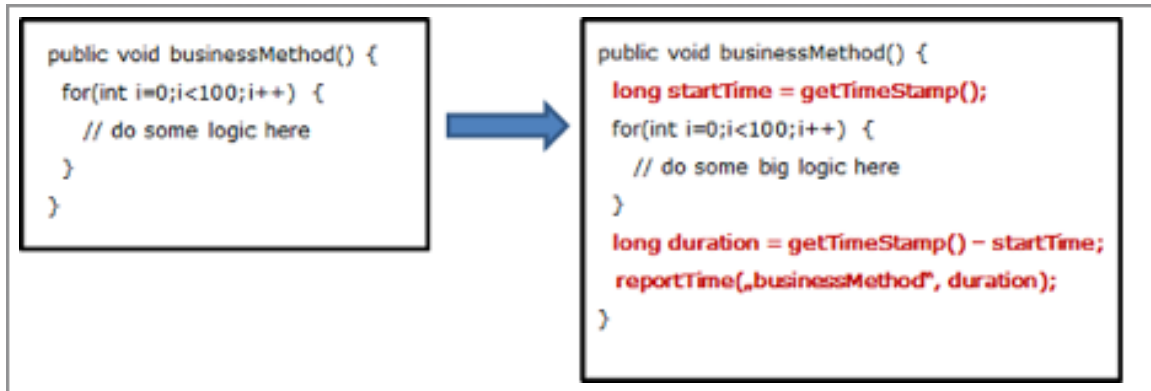


Figure 1.7: Bytecode instrumentation makes it possible to insert special monitoring code

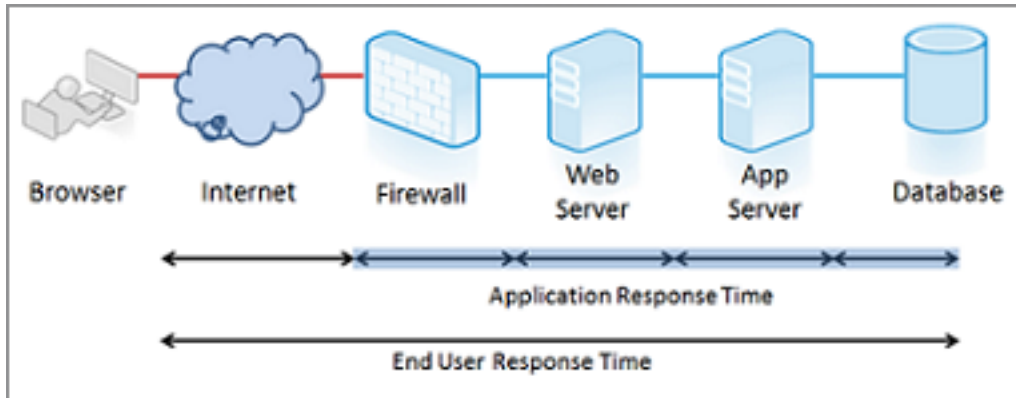
The bytecode of an application can be changed at different stages of the application life-cycle. Let us look at three approaches:

- We can statically change the bytecode as part of the build process. This is the least flexible approach. It also requires access to the sources which have to be rebuilt. Every change of the instrumentation also requires a redeployment of the application, which is often not easy to do and can be time consuming.
- The second approach is to modify the bytecode when the class is loaded by the JVM: JVMTI makes it possible to modify the bytecode before the class is actually used without having to change the actual class file. This approach is more convenient as there are no required changes to the build process and the required deployment artifacts like jar files. Nearly all modern tools follow this approach.
- The third approach is to dynamically instrument classes which have already been loaded and are currently in use by the JVM. This allows us to change the bytecode at runtime without having to restart the application, which is especially useful when an application must be debugged in production. And since debug-level diagnostic information can be turned on and off at runtime, application restarts become unnecessary.



## Measurement from the End-User Perspective

Modern application performance management goes beyond measuring performance solely within the application by including measurements from the user's perspective. If we consider response time, the most important metric for performance from a user's perspective, we see only a fraction of the overall response if we measure only on the server side—typically something between 20 and 50 percent. For instance, we miss the network transmission time and resource load times.



*Figure 1.8: Response time is the time elapsed from sending the request to the complete receipt of the response.*

Timing details, such as how long it takes the server to return an HTML document, are of no concern to end users. Instead, they're much more aware of the total time it takes to load a page, all the content, including all images and other dynamic content.

In the following figure, we see all activities associated with loading a page. This includes loading the document, images, as well as executing JavaScript, and rendering the content. In this example, server optimization will not prevent end users waiting a long time for the execution of the JavaScript before the page is completely loaded for them.

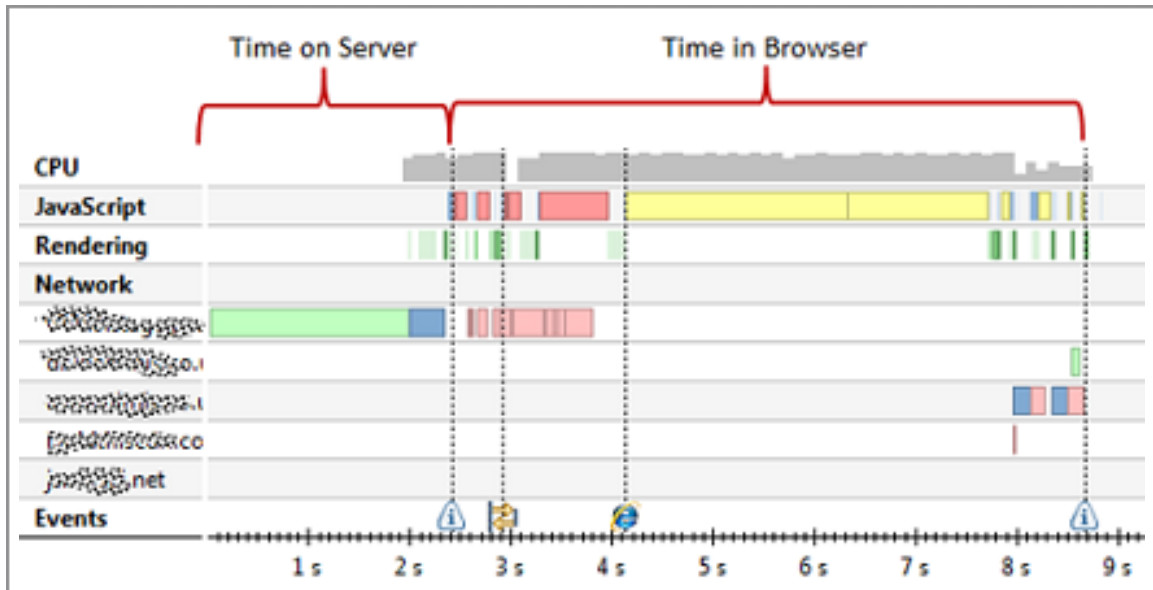


Figure 1.9: Both browser and server activities impact response time of a page.

Technologies vary greatly, but are all grouped within the terms "end user monitoring", "real user monitoring," or "user experience monitoring". In short, they are all used to measure performance closer to the end user. This can be accomplished by monitoring network traffic, capturing performance data in the users browsers, or by sampling data from a number of remote locations.

Before we look at the different methods in detail, we have to define which questions these tools must be able to answer:

- How long does it take to load a page?
- Were there any errors or problems in loading the page?
- How long has it taken to complete certain user interactions on the page?

This data can be collected in a number of different ways, and we need to look at the advantages and disadvantages of each, especially since they are complementary to a large extent.

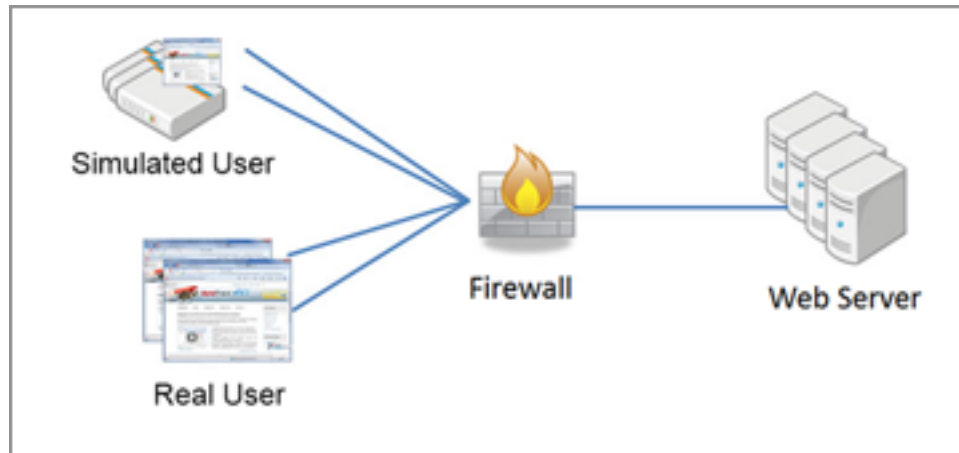
## Synthetic Monitoring

Synthetic monitoring involves executing scripts that mimic typical user transactions from locations around the globe. The performance of these transactions is measured and used to draw conclusions on how the sites behaves for real users. We can derive trends and make statements about the general performance and availability of the application.

The global distribution of points being measured also makes it possible to identify regional problems. So the quality of measurement strongly depends on the number of locations The more and the closer to the end user the better.

The biggest shortcoming of synthetic monitoring is that it can only be used for a subset of user transactions. Transactions which have side effects like triggering an order cannot be monitored. It is obvious that you do not want your monitoring to trigger actual shipments in an eCommerce application.

The other downside of this approach is the effort required to maintain the scripts. Whenever the application logic is changed the corresponding scripts have to be modified as well. While a modular script definition approach will make this task easier, synthetic monitoring still requires constant maintenance.



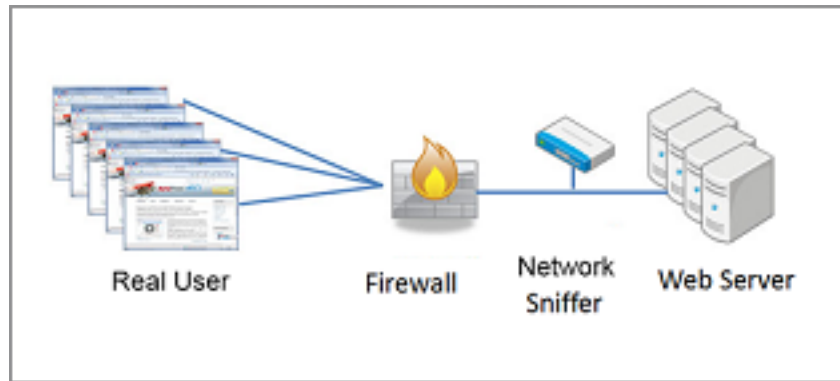
*Figure 1.10: Synthetic monitoring with simulated users*

## Network-Centric Monitoring

In contrast to synthetic monitoring, network-centric monitoring measures actual response times for real users by analyzing the network traffic between browsers and back-end servers. Monitoring the response times and also network-related errors for each individual request enables us to measure network transmission times and also detect application errors sent back as responses to user requests.

Imagine that a customer gets an error message when purchasing an item on a web site. If the user contacts customer support for the site, it is difficult to determine what happened for this specific user. With passive monitoring, the optimal case, the problem can be found automatically by looking at the actual network traffic.

Special appliances, hardware and virtual, are used for to collect performance data on the network level. These "network sniffers" log the network communication and thus reconstruct the click path for a user. Some solutions make it possible to combine this data with instrumentation in the browser, which is especially useful for single-page AJAX applications where user behavior either cannot be analyzed based on network communication, or only with difficulty.



*Figure 1.11: Network-centric monitoring using a network sniffer*

The biggest shortcoming of network-centric monitoring is that re-sponse times are still measured within the datacenter. Performance problems coming from connectivity issues on the user's networks are hard to analyze. Another issues with this approach is the missing insight into the impact of Third-Party content—like ads or social widgets—on application performance.

## Browser Instrumentation

This approach is becoming more popular, because it can collect performance data from directly inside the browser via HTML and JavaScript instrumentation. Either the re-quested pages already contain monitoring code or it is inserted into the HTML code during runtime.

The advantage of this approach is that the exact response times can be collected for each user of the application. [The W3C Navigation Timing specification](#) defines a standardized approach for collecting this data. Currently the Web Timing Specification is already implemented in most modern browsers like Chrome, Internet Explorer or Firefox.

There is even a mobile implementation for IE on Windows Mango and the Android implementation is in its early steps. Even if a browser has not implemented the standard, frameworks such as [Boomerang](#) or the information exposed by the Google Toolbar can be used to collect similar data.

The data is sent back to the server either through XHR or via a so-called beacon. A beacon represents a simple URL that is requested from a server. The measurement data is packaged as URL parameters. An example of such a beacon is <http://www.myserver.com/beacon?loadTime=2000>. In this case the load time is sent to the server as a parameter. Browser Instrumentation is the only monitoring approach which collects real user performance data. Therefore it provides the most accurate measurement. It, however, cannot be used to monitor availability as measurements are only collected when pages are delivered to users. When the server however is not reachable, no pages will be requested.

## Collecting and Analyzing Execution Time Data

There are two widely-used techniques for collecting execution time data: Time-Based Measurement and Event-Based Measurement. Both have become standard parts of performance management products and both typically represent data as stacks, so they look superficially similar. We'll examine each in detail, pointing out the fundamental differences in their approaches to data measurement and to what these means in terms of data interpretation.

### Time-Based Measurement

In time-based measurement, the stacks of all executing threads of an application are captured at a constant interval (sample rate). Then the call stack is examined and the top method is set to the state of the thread – executing or waiting. The more often a method appears in a sample, the higher it's execution or wait time.

Each sample gives an indication of which method was actively executing, but there are no measures of execution duration or of when a method completes execution. This tends to miss shorter-running method calls, as if they were not actively executing. At the same time, other methods may be over-weighted because they happen to be active during, but not in between sampling periods. For both reasons, this makes it difficult to collect exact execution time data for an individual method.

The selected time interval determines the granularity and quality of the measurement. A longer measurement time also increases the probability that the data is more representative and that the errors mentioned will have less of an effect. This is simple statistics. The higher our number of samples is the more likely it is that methods that execute more often will appear as the executing methods on the call stack.

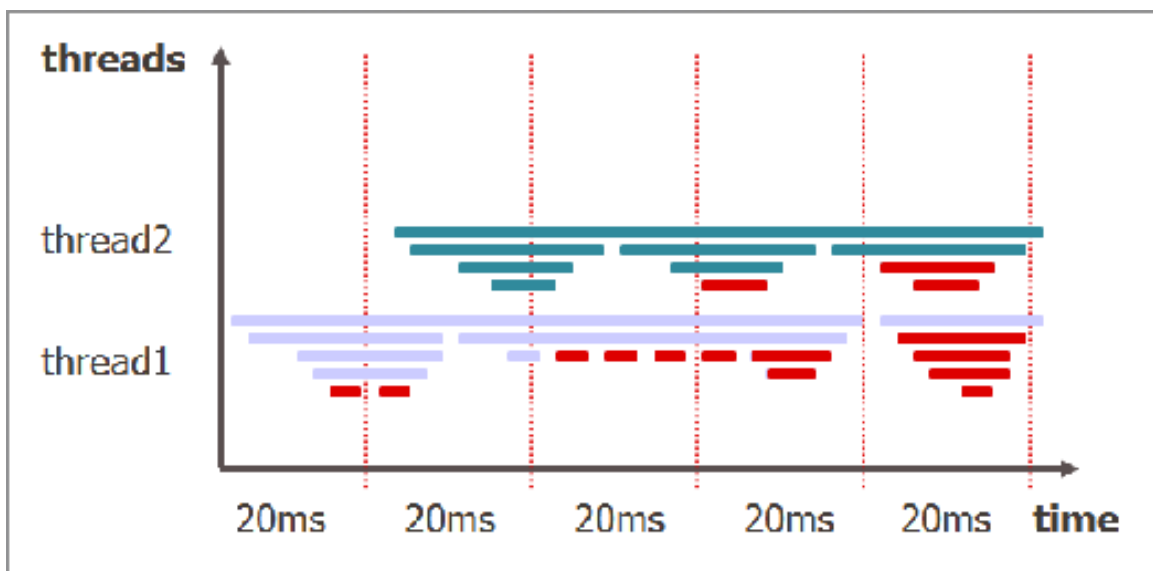


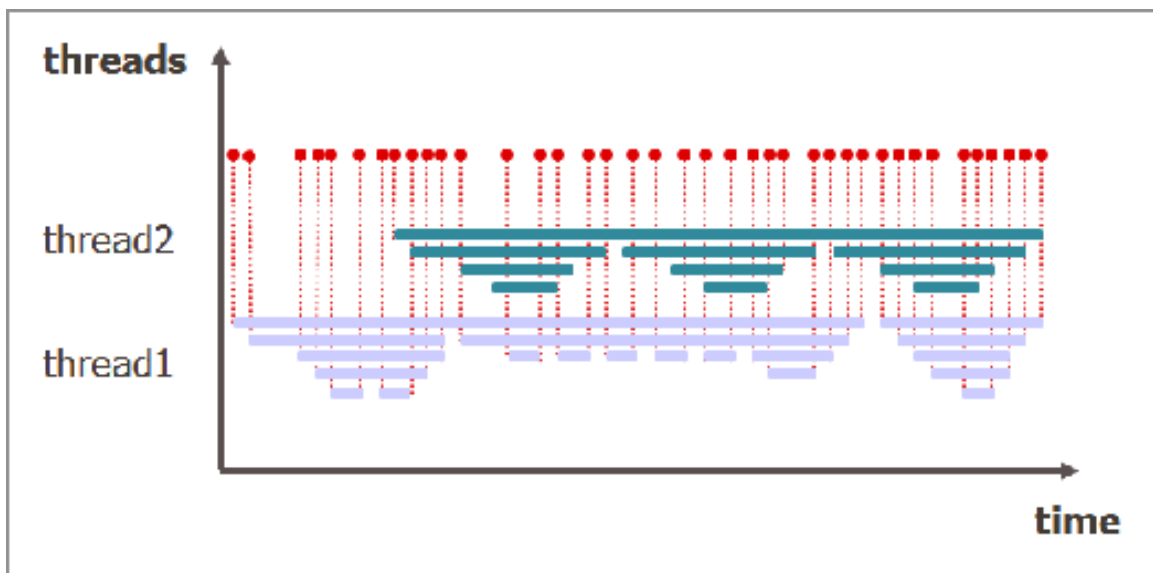
Figure 1.12: The stacks of each thread are analyzed every 20 ms.

This approach creates its own measurement errors, but it provides a straightforward way to track down performance problems. Perhaps the most important advantage is that measurement overhead can be controlled by the sampling interval; the longer the interval, the lower the overhead. This low overhead is especially important in production environments under full load. Here's why:

We see that a particular user request is slow, but we're not able to see the input data in use. At the same time, we have no knowledge of actual execution counts for particular methods. Time-based measures can't provide this information. The collected data gives us no way to determine when a single method execution is slow or when there is overly-frequent method execution. A problem analysis of these sorts of details requires the use of event-based measurement.

### Event-based Measurement

Instead of thread samples, event-based measurement analyzes individual method executions. We log a timestamp at the beginning and end of each method call, which makes it possible to count the number of times individual methods are called and determine the exact execution time of every call. It's a straightforward matter of calculating the differences between the collected timestamps.



*Figure 1.13: For event-based measurement, the start and end time for each method call is logged.*

The Java Runtime offers special callbacks via JVMPI/JVMTI, which are called at the beginning and end of method executions. However, this approach carries a high overhead that may have a major impact on the application's runtime behavior. Modern performance measurement methods use bytecode instrumentation, requiring very little over-



head. There's also the added functionality of instrumenting the most relevant methods, only, and the ability to capture context information, like method parameters.

## Transactional Measurement

For complex applications, we make use of more sophisticated forms of event-based measurement. So in addition to storing method execution metrics, transactional measurement stores context, which makes it possible to examine the methods of different transactions individually. This way we can find out that a specific slow SQL statement is executed only for a specific Web request.

Method executions are also stored in their individual chronological order instead of being aggregated based on the call stack, which can reveal dynamic behavior of the processing logic of a transaction – for example, a servlet request. The isolated examination of individual transactions thus makes it possible to analyze data-dependent problems and outliers.

The detailed diagnostics provided by transactional measurement are particularly well-suited for working with distributed applications. There are a number of ways in which cross-tier transactions are correlated:

Transaction type based approaches pass a transaction type ID to the next tier and additionally use time based correlation. This provides more details as not only time but also the transaction type (e.g. URL) is taken into account. Transaction id based correlation passed individual transaction IDs to the next tier. This allows following individual transactions across tiers and is the most precise approach to cross tier correlation.

Elapsed Time [ms]	Method	Argument	Return	Total [ms]
1.24	doFilter(ServletRequest req, ServletResponse res, FilterChain chain)			13923.41
1.25	doFilter(ServletRequest req, ServletResponse res, FilterChain chain)			13923.40
2.11	doFilter(ServletRequest servletRequest, ServletResponse servletR			13922.54
2.12	doGet(HttpServletRequest request, HttpServletResponse resp			13922.51
16.72	getStatus()		6	0.02
16.75	prepareStatement(String sql, int resultSetType, int resultS	SELECT ID, ISSUE, CUSTOMFIELD, P...		0.07
16.84	executeQuery()	SELECT ID, ISSUE, CUSTOMFIELD, P...		0.41
18.01	getStatus()		6	0.01
18.02	prepareStatement(String sql)	select min( act.CREATED) from dbo...		0.05
18.08	executeQuery()	select min( act.CREATED) from dbo...		0.44
19.08	getStatus()		6	0.01
19.09	prepareStatement(String sql)	select distinct(AUTHOR) from dbo.j...		0.04
19.15	executeQuery()	select distinct(AUTHOR) from dbo.j...		0.35
20.82	getStatus()		6	0.01
20.84	prepareStatement(String sql, int resultSetType, int resultS	SELECT ID, issueid, AUTHOR, actio...		0.07
20.92	executeQuery()	SELECT ID, issueid, AUTHOR, actio...		0.29
22.93	getStatus()		6	0.01

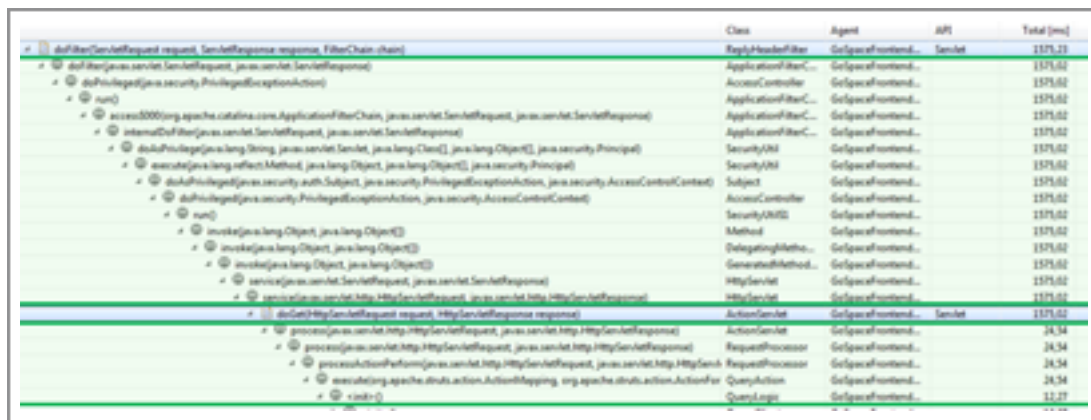
Figure 1.14: Each method is logged using start and end time to calculate total execution time.

## Hybrid Approaches

Hybrid methods combine time-based and event-based performance measurement to exploit the advantages of both and avoid the individual disadvantages. The main advantage of event-based data collection lies in capturing each call and its context – such as method parameters. The primary advantage of the time-based approach lies in the constant and load-independent overhead.

Hybrid measurement combines transactional event-based data with thread snapshots that can be assigned to a specific transaction. Instrumentation ensures we can capture the transactional context as well as method parameters and other details, while time-based data provides a low overhead approach to capture execution time information.

This approach offers the most precise insight into the runtime behavior of an application, and the nearly constant overhead is well suited to production monitoring. The figure shows a transactional trace that combines both data sources with one another. The event-based data is shown in blue and the time-based data in green.



Class	Agent	API	Total (ms)
ReplyHeaderFilter	GoSpace of nontend...	Servlet	1175.23
ApplicationFilter...	GoSpace of nontend...		1175.01
AccessController	GoSpace of nontend...		1175.01
ApplicationFilterC...	GoSpace of nontend...		1175.01
ApplicationFilterC...	GoSpace of nontend...		1175.01
ApplicationFilterC...	GoSpace of nontend...		1175.01
SecurityUtil	GoSpace of nontend...		1175.01
SecurityUtil	GoSpace of nontend...		1175.01
Subject	GoSpace of nontend...		1175.01
AccessController	GoSpace of nontend...		1175.01
SecurityUtil	GoSpace of nontend...		1175.01
Method	GoSpace of nontend...		1175.01
DelegatingMethod...	GoSpace of nontend...		1175.01
GeneratedMethod...	GoSpace of nontend...		1175.01
HttpServlet	GoSpace of nontend...		1175.01
HttpServlet	GoSpace of nontend...		1175.01
ActionServlet	GoSpace of nontend...	Servlet	1175.01
ActionServlet	GoSpace of nontend...		34.34
RequestProcessor	GoSpace of nontend...		34.34
RequestProcessor	GoSpace of nontend...		34.34
QueryAction	GoSpace of nontend...		34.34
QueryLog	GoSpace of nontend...		32.27

Figure 1.15: Hybrid transactional trace (green portions were added via thread snapshots)

## Visualizing Performance Data

In addition to measuring performance data, we also have to visualize it for analysis. Depending on the underlying data structure, we differentiate two types of data for visualization:

- Time-based data, like JMX values, are represented as relations, or tuples, where the data value is paired with a timestamp. So for each measurement of, for example, CPU load, network usage, or transaction count, there can only be a single value or data point. Depending on the use case, we can visualize these data using various chart types: line, meter, traffic lights, whatever it takes to plot this information for a given use case.

- Execution-time data is represented hierarchically as an ordering of methods and associated execution times—a call tree. In fact, all of the execution-time measurements we’ve mentioned can be visualized as trees. Note however, that the semantics for reading these different trees is not always the same.

We will not cover the representation of time-based monitoring data in further detail as it is quite easy to understand. However, the different representations for execution-time data require a bit more of explanation.

## Aggregating Executions in Call Trees

Both time-based and event-based execution data can be represented as call trees. This nicely illustrates the call hierarchy, but at the cost of obscuring the actual sequence of individual calls.

Call trees are primarily used to help identify execution hot spots—those areas where methods are called most frequently and have the greatest execution times.

Practically speaking, problems due to high CPU utilization or long wait times caused by lengthy synchronization are easily diagnosed with call trees. However, the loss of call sequence information makes it impossible to use call trees to find logical control flow issues.

On the one hand, using call trees, we should be able to identify an application hot spot caused by a long-running and complex database call. On the other hand, call trees are not likely to illuminate a problem stemming from a high number of cache misses requiring repeated queries of the database. The cache miss method is likely lost in the hot-spots, since it does not consume much execution time.

Depending on the performance tool used, you may also have additional information available, such as called database statements, specific URLs, or other important diagnostic data. There is also a certain amount of data and report customization available. Using these data can help to distinguish call data and to enable better and simpler analysis. With regard to complex performance problems, however, one quickly reaches the limits of this approach.

Name	Class	CPU Time [ms]	Total CPU Time...
get(Object)	HashMap	78847.00	78847.00
triggerFired(SchedulingContext, Trigger)	RAMJobStore	62.00	187.00
get(Object)	SequencedHashMap	0.00	76404.00
get(Object)	IdentityMap	510.00	76914.00
getCollectionEntry(PersistentCollection)	SessionImpl	0.00	52278.00
updateReachableCollection(PersistentCollection, T)	SessionImpl	14383.00	93541.00
processCollection(Object, PersistentCollectionT)	FlushVisitor	8012.00	101554.00
processValue(Object, Type)	AbstractVisitor	18020.00	119575.00
getEntry(Object)	SessionImpl	0.00	24636.00
getCollectionPersister(String)	SessionFactoryImpl	0.00	1968.00
get(String)	ProxyVMContext	0.00	195.00
get(String)	ProcedureCache	0.00	97.00
compile(String)	OgnlUtil	0.00	83.00
get(Object)	ThreadLocalCache	0.00	43.00
getPropertyDescriptor(String)	CachedIntrospectionResults	0.00	52.00
getBeanDefinition(String)	DefaultListableBeanFactory	0.00	50.00
getPersister(Class)	SessionFactoryImpl	0.00	50.00
getBodyContents()	ContentEntityObject	77430.00	77430.00
entryList()	IdentityMap	52485.00	62999.00

Figure 1.16: `HashMap.get` requires the most CPU time; the return call tree shows what the calls were.

## Visualizing Individual Call Traces

Using transactional measurement methods makes it possible to show the call sequence of each transaction or request individually. The visual representation will look very similar to a call tree, but the data is far more precise and therefore the visualization will be more accurate.

Call traces show the timely sequence of method execution, which provides a visual representation of transaction-based logical flow. With this visualization technique, complex problems, like the one described above, are no longer un-diagnosable! Furthermore, as we can see in the example below, this information can also be used to diagnose functional problems, like an exception being thrown in the context of a remote service invocation.

Method	Argument	Total [ms]	Exec [ms]	Agent
ProcessRequest(HttpContext)		11483.13	0.07	dTWebsite[dynaweb]...
ProcessRequest(HttpContext)		11483.07	7.19	dTWebsite[dynaweb]...
login(String, String)		394.15	6.66	dTWebsite[dynaweb]...
Synchronous Path (Web Service)		-	-	eServices@dynaweb20...
doPost(HttpServletRequest request, Http...		387.49	386.61	eServices@dynaweb20...
getLicensesWithPossibleWorkflowsOfUser(Stri		602.64	1.08	dTWebsite[dynaweb]...
Synchronous Path (Web Service)		-	-	eServices@dynaweb20...
doPost(HttpServletRequest request, Http...		601.56	10.44	eServices@dynaweb20...
prepareStatement(String sql)	select u....	0.07	0.07	eServices@dynaweb20...
execute()	select u....	0.23	0.23	eServices@dynaweb20...
invoke(Object[] params)		13.79	9.06	eServices@dynaweb20...
exception	No service named...	-	-	eServices@dynaweb20...
exception	No service named...	-	-	eServices@dynaweb20...
Synchronous Path (HTTP)		-	-	externalUira@dynaweb...
doFilter(ServletRequest servletF		4.73	0.03	externalUira@dynaweb...
invoke(Object[] params)		6.11	3.92	eServices@dynaweb20...
prepareStatement(String sql)	select distinct gr...	0.08	0.08	eServices@dynaweb20...
execute()	select distinct gr...	0.40	0.40	eServices@dynaweb20...
prepareStatement(String sql)	SELECT count(*) f...	0.04	0.04	eServices@dynaweb20...
execute()	SELECT count(*) f...	0.52	0.52	eServices@dynaweb20...

Figure 1.17: A transactional trace visualizes the execution of a request, including context information.

## Showing Complex Transactions as Transaction Flows

Particularly in distributed environments, call trace visualizations provide invaluable insight into the dynamic characteristics of application performance. However, when used in a production environment where there's a constant load of thousands of transactions per second, it is no longer feasible to examine individual transactions.

This brings us into the realm of transaction flow visualizations, which, as the name implies, are able to represent the flow of individual or aggregated transactions through an application. As shown in the figure below, this visualization is used to analyze performance problems in complex, high-volume, distributed environments.

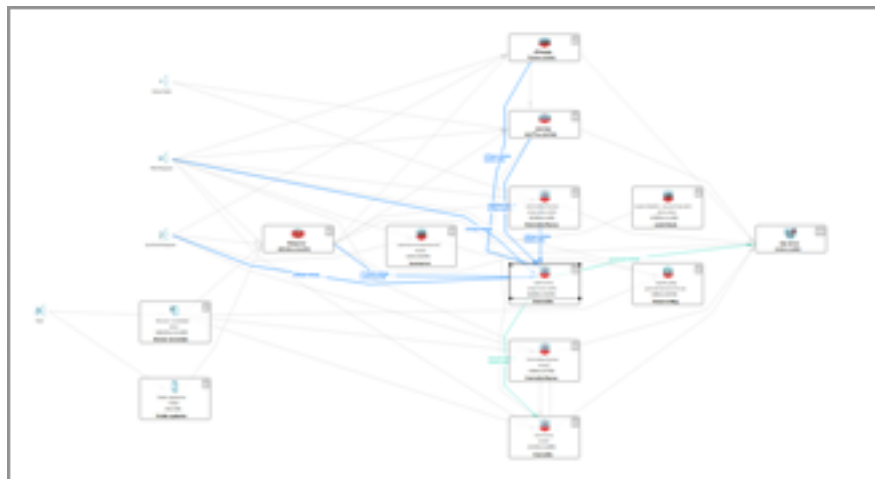


Figure 1.18: Transaction Flow of a Transaction through an Application Landscape

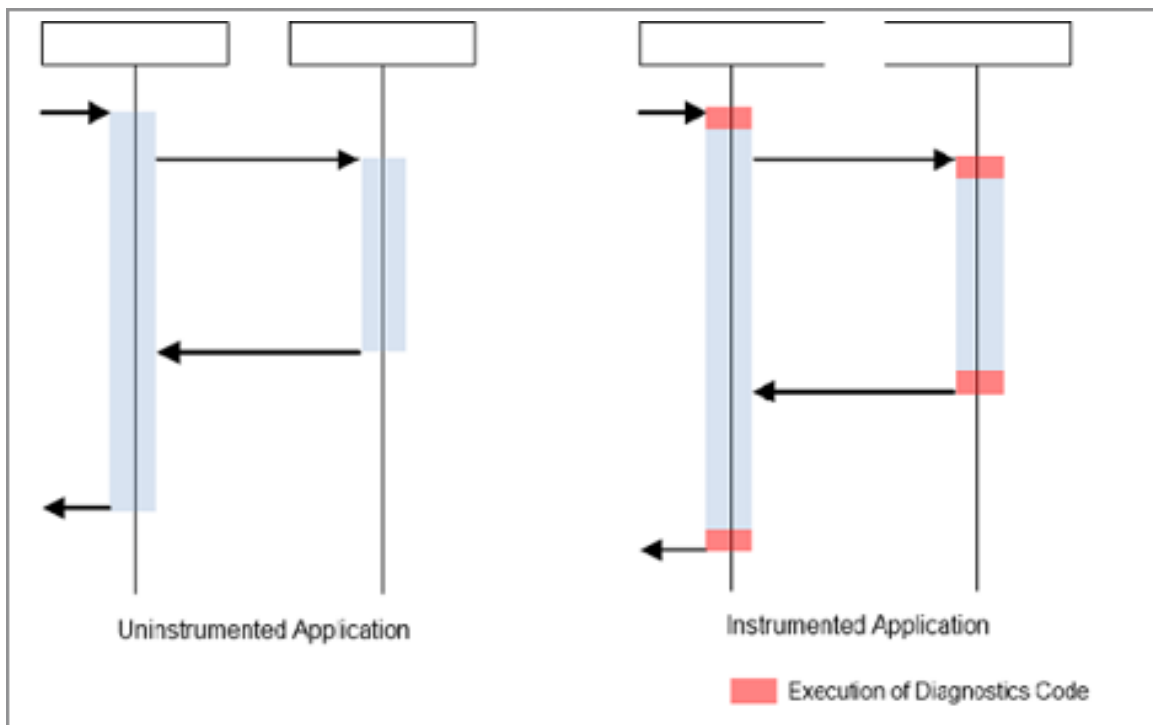
## Controlling Measurement Overhead

Performance measurement comes at a cost, and this will vary depending on the approach employed. For instance, it's impossible to measure the execution time of a method, and then process and store the data without creating overhead. We may perceive this extra load in the form of longer response times, increased CPU utilization, increased memory usage, or as additional network load.

In addition to the performance impact, overhead can result in so-called Heisenbugs, named after the [Heisenberg uncertainty principle](#), in which problems appear or disappear as a result of the measurement. There's an inevitable risk, however small, that the overhead of measuring performance will degrade performance. Let's examine the different types of overhead that can occur and see how performance measurements can affect an application's runtime behavior

### Controlling Response Time Overhead

Increasing response times have the most direct effect on users, so response time overhead is the most critical. Generally, the more we measure, the larger the overhead. This is specifically true for all event-based measurements taken during the execution of application code. When we use an instrumentation-based measurement approach, we should not instrument methods with a very high call frequency and an individually short execution time. Overhead will be biggest in this case.





*Figure 1.19: The shorter the method, the greater the percentage share of the instrumented code*

For example, if the instrumented diagnostics code requires one millisecond, then the overhead for a method execution time of 100 ms is one percent. If the method takes only 10 ms, then we are already at 10 percent overhead. Response time overhead is often described in percentages, for example 3% – 5% for a production system and slightly more for a test environment. You can describe overhead as a percentage of the actual response, but this value can be hard to put into context. Still, in many cases, referring to actual numbers is easiest. For instance, it's more descriptive to say that monitoring makes the application 20 ms slower, than it is to say that an application has a two percent overhead.

### **Controlling CPU and Memory Overhead**

Performance measurement not only impacts response times, but also affects CPU and memory usage. The more measurement logic is executed, the more CPU and memory are taken away from the application.

Monitoring tools use two basic strategies to reduce overhead. One is simply to sample a percentage of transactions. This has the obvious drawback that we lose full application visibility. The more sophisticated approach is to offload CPU- and memory-intensive processing to dedicated servers, leaving only timestamp capturing in the application.

As it turns out, it's most important to avoid CPU overhead, keeping it at less than one per cent. For memory consumption, we must ensure that none of the memory resources required by the application are for monitoring. Also, if the monitoring logic does not manage memory efficiently, this could cause additional garbage collection, which is yet more overhead.

### **Controlling Network Overhead**

Most performance tools send monitoring results over the network to a centralized server. The greater the amount of data measured, the greater the volume of data transferred over the network. Measurement interval and the number of measurements also affect amount of data transferred.

Let's look at the effects of sending event-based measurements of method calls over the network. We have 1000 users working with the application, resulting in 100,000 method executions measured per second. If, in a naive approach, the method name (average of 15 characters) and the execution duration are sent in an 8-byte data type for each method, this results in  $100,000 * (15 + 8) = 2,3 \text{ MB/sec}$ . Such a high usage of the network can, and often will, have serious side effects on the application behavior.

This might not be a problem in a small load or even local desktop environment. However, when monitoring large production environments, we need highly optimized methods in which only a few bytes are transmitted.

To reduce network traffic, some measurement methods aggregate data within the application before sending it to a centralized server. Unfortunately, this approach has the drawback of consuming additional resources that might be needed by the application.

## **Measuring Overhead**

The most straightforward way to measure overhead is to run the same load test twice; once against the non-instrumented applications, and once against the instrumented application. During the test run, measure response times, CPU, memory, and network utilization. Comparing the measurements of these two tests provides a measure of overhead.

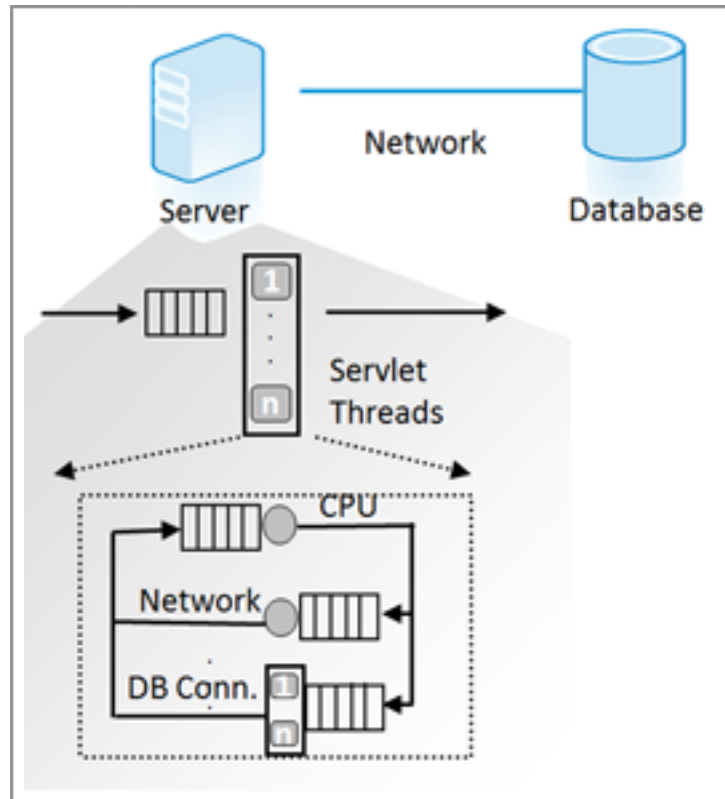
## **Theory Behind Performance**

Finally, we must take the plunge into the theoretical foundations of performance management. It's essential to our everyday work to understand some of the immutable laws of application performance. We promise to limit this discussion to the most important and practical considerations.

### **Performance Modeling using Queuing Theory**

Queuing models are often used to model the characteristics of a software system. Although they present a simplification, they are helpful in understanding the characteristics of software systems.

The basic idea is simple. An application requires resources to fulfill a request, and these resources are available in limited quantity. If a resource is not available, a request must wait for it. The figure below shows a simple queuing model for a single-tier application. The resources include a pool with servlet threads for the database, memory, and the CPU. These are interdependent, so if the thread pool is too small, it would not help if more CPU were available.



*Figure 1.20: In an application, there are different queues that require waiting.*

What can we conclude from this? The performance and scalability of our application depend upon the resources available. When there are fewer resources available, the more heavily what is available will be used. This can quickly lead to poor application scaling and ever longer execution times.

While a resource is in use, it's not available to other requests. These limited resources have the potential to become performance bottlenecks, and this is why they become an important focus for monitoring and application scaling. When application response becomes slower, it is almost always due to a lack of resources.

This is why it's so important to keep an eye on the utilization of individual resources and monitor the transactions using them. Once we have constructed such a model, we can detect and eliminate almost all performance and scalability problems. Queuing models are also the basis for two important performance and scalability limits.

### **Little's Law**

Little's Law states that given a queue system in a stable state, the average number of customers, or requests, is equal to the product of their average arrival rate and the average time spent in the system. Put more simply, a system is stable when the number of new requests is not higher than the maximum load that can be processed.

For example, when it takes half a second to process a request, then a maximum of two requests can be processed per second. If, over a longer time period, more than two requests per second are received, the system will become unstable.

This may seem both trivial and logical, but how does it help us in our daily work? First, we can use it to estimate scalability. We know that we have two cores available for an application, meaning 2,000 ms CPU time per second. If a request requires 200 ms, our maximum stable load is ten requests. We can therefore determine very easily the point at which our resources are saturated. This is already a basic way for capacity planning. Although it is oversimplified it helps as a first guess.

Another interesting use case is for verification of a test setup. We can use Little's Law to determine whether test results are valid and not impacted by an overloaded test driver. If a load test shows that we had a maximum of five requests per second, each one taking 200 ms in a system with ten threads, then we know something is wrong. According to Little's Law, we should have had ten times the throughput as we had 10.000ms of available time across the ten threads. In this case the load generator did not succeed in putting our system under an appropriate load and the test does not represent the limit of our application.

### **Amdahl's Law**

Amdahl's Law defines the effect of parallelization on the response time of a system. Let us look at an example. If we can parallelize 200 ms of a request taking 500 milliseconds and divide the execution to 4 processes, we achieve a response time of 350 ( $300 + 200/4$ ) milliseconds.

This concept is exploited primarily in grid and large batch systems, in which a high degree of parallelization can be achieved. The higher the proportions of parallelization, the greater are the effects.

Amdahl's Law cannot be used 1:1 for the amount of execution time gained. However, analogous considerations help us understand which effects parallelization has on an application. It can also be used for an estimation of effectiveness. At a particular point, parallelization no longer makes sense because the gain in overall performance is very low.

## **How Humans Perceive Performance**

We've talked a lot about the technical aspects of measuring performance. While this is important, it's important to remember that users are the ultimate judge of performance. As users are not clocks, their time perception may be different from what we measure empirically. If you are interested in a detailed discussion of this topic, we recommend the book, *Designing and Engineering Time: The Psychology of Time Perception in Software*, by Steven C. Seow.

When talking about human time perception, we must first be aware that we (humans) always compare perceived time against our expectation of how fast things should be. So time for us is always relative. We judge system responsiveness by comparing expectations against our perceptions of response time.

In judging system responsiveness, we must first know how fast users expect a system to be. Research shows that there are four main categories of response times:

- We expect an instantaneous response, 0.1 to 0.2 milliseconds, for any action similar to a physical interaction. Pressing a button, for example. We expect this button to indicate that it is pressed within this time.
- We expect an immediate response, within a half to one second, indicating our information is received. Even in the case of a simple interaction, we expect an actual response to our request. This is especially true for information we assume is already available, as when paging or scrolling content.
- We expect a reply within 2 to 5 seconds for any other interactive request. If the response takes longer, we begin to feel that the interaction is cumbersome. The more complex we perceive a task to be, the longer we are willing to wait for the response.

When it comes to system performance, users have very precise expectations, and these metrics are useful both for understanding our performance perceptions and for use in SLA definitions. It can be difficult to find proper SLAs or response time thresholds, but having these precise categories gives us a straightforward approach to defining performance thresholds.

### **What It Means to Be Faster**

As with performance, human perception for faster or slower is not as precise as technical measurements. This is very important when judging whether a performance optimization is worth the effort.

In essence, our limited perception of application responsiveness can't detect performance changes of much less than 20%. We can use this as a rule of thumb when deciding which improvements will impact users most noticeably. If we optimize 300 milliseconds a request that took 4 seconds, we can assume that many users will not perceive any difference in response time. It might still make sense to optimize the application, but if our goal is to make things faster for the users, we have to make it faster by at least 0.8 seconds.