Gauging Performance Absence of Measures

By Alexander Podelko

Defining performance requirements is an important part of system design and development. If there are no written performance requirements, it means that they exist in the heads of stakeholders, but nobody bothered to write them down and make sure that everybody agrees on them.

Exactly what is specified may vary significantly depending on the system and environment, but all requirements should be quantitative and measurable. Performance requirements are the main input for performance testing (where they are verified), as well as capacity planning and production monitoring.

There are several classes of performance requirements. Most traditional are response time (how fast the system can handle individual requests) and throughput (how many requests the system can handle). All classes are vital: Good throughput with a long response time often is unacceptable, as is good response time with low throughput.

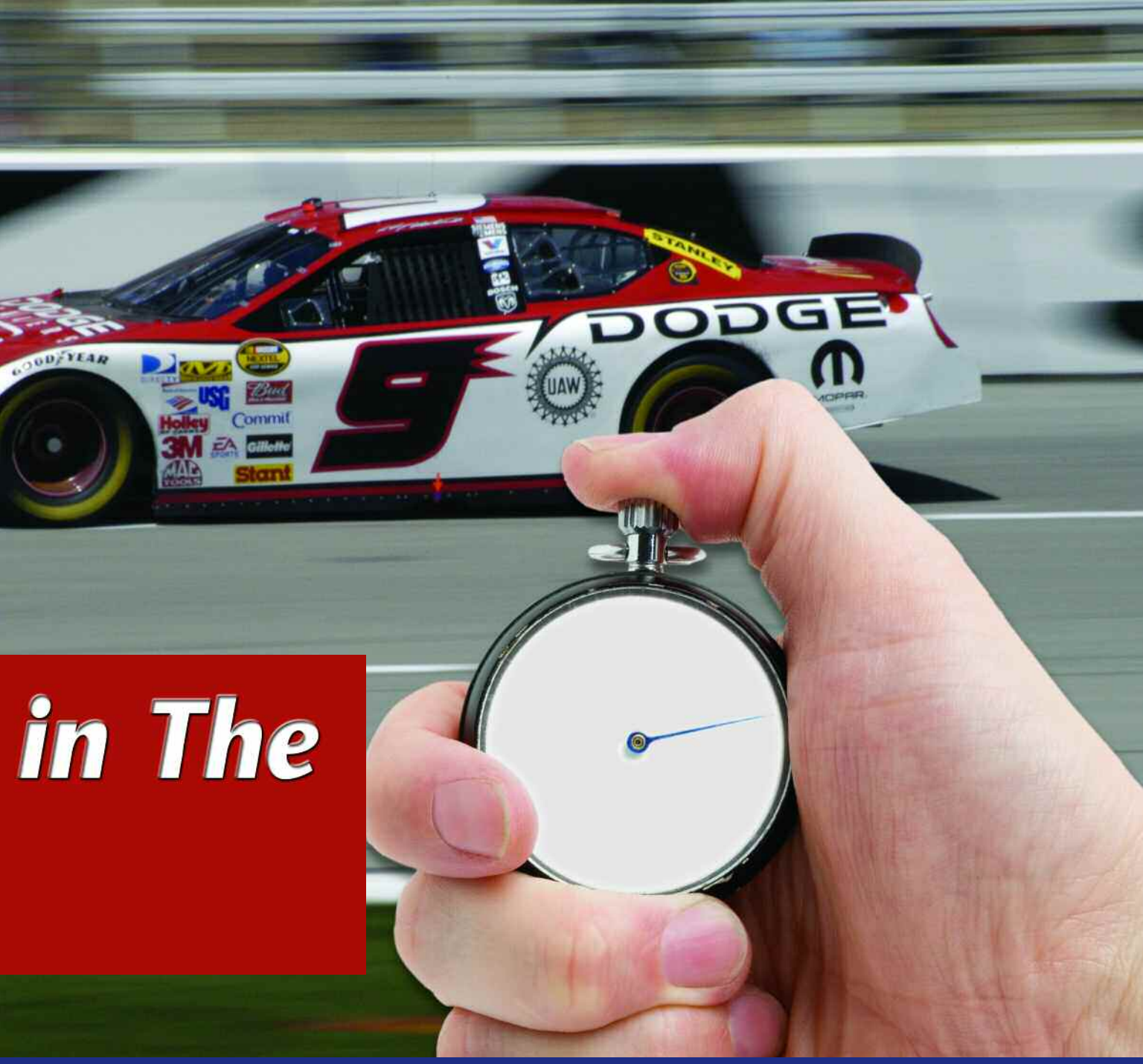
Response time (in the case of interactive work) or ***processing time*** (in the case of batch jobs or scheduled activities) defines how fast requests should be processed. Acceptable response times should be defined in each particular case. A time of 30 minutes could be excellent for a big batch job, but absolutely unacceptable for accessing a Web page in a customer portal. Response time depends on workload, so you must define conditions under which specific response times should be achieved; for example, a single user, average load or peak load.

Significant research has been done to define what the response time should be for interactive systems, mainly from two points of view: what response time is necessary to achieve optimal user's performance (for tasks like entering orders) and what response time is necessary to avoid Web site abandonment (for the Internet). Most researchers agreed that for most interactive applications, there is no point in making the response time faster than one to two seconds, and it's helpful to provide an indicator (like a progress bar) if it takes more than eight to 10 seconds.

The service/stored procedure response-time requirement should be determined by its share in the end-to-end performance budget. In this way, the worst-possible combination of all required services, middleware and presentation layer overheads will provide the required time. For example, with a Web page with 10 drop-down boxes calling 10 separate services, the response time objective for each service may be 0.2 seconds to get three seconds average response time (leaving one second for network, presentation and rendering).

Response times for each individual transaction vary, so use some aggregate values when specify-

Alexander Podelko is a software consultant currently engaged by Oracle.



in The

ing performance requirements, such as averages or percentiles (for example, 90 percent of response times are less than *X*). Maximum/timeout times should be provided also, as necessary.

For batch jobs, remember to specify all schedule-related information, including frequency (how often the job will be run), time window, dependency on other jobs and dependent jobs (and their respective time windows to see how changes in one job may impact others).

Throughput is the rate at which incoming requests are completed. Throughput defines the load on the system and is measured in operations per time period. It may be the number of transactions per second or the number of adjudicated claims per hour.

Defining throughput may be pretty straightforward for a system doing the same type of business operations all the time, processing orders or printing reports. It may be more difficult for systems with complex workloads: The ratio of different types of requests can change with the time and season.

It's also important to observe how throughput varies with

time. For example, throughput can be defined for a typical hour, peak hour and non-peak hour for each particular kind of load. In some cases, you'll need to further detail what the load is hour-by-hour.

The number of users doesn't, by itself, define throughput. Without defining what each user is doing and how intensely (i.e., throughput for one user), the number of users doesn't make much sense as a measure of load. For example, if 500 users are each running one short query each minute, we have throughput of 30,000 queries per hour. If the same 500 users are running the same queries, but only one query per hour, the throughput is 500 queries per hour. So there may be the same 500 users, but a 60X difference between loads (and at least the same difference in hardware requirements for the application—probably more, considering that not many systems achieve linear scalability).

Response Times: Review of Research

As long ago as 1968, Robert B. Miller's paper "Response Time in Man-Computer Conversational Transactions" described three threshold levels of human attention[1]. J. Nielsen believes that Miller's guidelines are fundamental for human-computer interaction, so they are still valid and not likely to change with whatever technology comes next [2]. These three thresholds are:

- ***Users view response time as instantaneous (0.1-0.2 second):*** They feel that they directly manipulate objects in the user interface; for example, the time from the moment the user selects a column in a table until that column highlights or the time between typing a symbol and its appearance on the screen. Miller reported that threshold as 0.1 seconds. According to P. Bickford, 0.2 second forms the mental boundary between events that seem to happen together and those that appear as echoes of each other [3].

Although it's a quite important threshold, it's often beyond the reach of application developers. That kind of interaction is provided by operating system, browser or interface libraries, and usually happens on the client side without interaction with servers (except for dumb terminals, that is rather an exception for business systems today).

- ***Users feel they are interacting freely with the information (1-5 seconds):*** They notice the delay, but feel the computer is "working" on the command. The user's flow of thought stays uninterrupted.

Miller reported this threshold as one second. Using the research that was available to them, several authors recommended that the computer should respond to users within two seconds [1, 4, 5]. Another research team reported that with most data entry tasks, there was no advantage of having response times faster than one second, and found a linear decrease in productivity with slower response times (from one to five seconds)[6]. With problem-solving tasks, which are more like Web interaction tasks, they found no reliable effect on user productivity up to a five-second delay.

An animated watch cursor was good for more than a minute, and a progress bar kept users waiting until the end.

The complexity of the user interface and the number of elements on the screen both impact thresholds. Back in 1960s through 1980s, the terminal interface was rather simple, and a typical task was data entry, often one element at a time. Most earlier researchers reported that one to two seconds was the threshold to keep maximal productivity. Modern complex user interfaces with many elements may have higher response times without adversely impacting user productivity. According to Scott Barber, even users who are accustomed to a sub-second response time on a client/server system are happy with a three-second response time from a Web-based application[7].

P. Sevcik identified two key factors impacting this threshold[8]: the number of elements viewed and the repetitiveness of the task. The number of elements viewed is the number of items, fields, paragraphs etc. that the user looks at. The amount of time the user is willing to wait appears to be a function of the perceived complexity of the request.

Users also interact with applications at a certain pace depending on how repetitive each task is. Some are highly repetitive; others require the user to think and make choices before proceeding to the next screen. The more repetitive the task, the better the expected response time.

That is the threshold that gives us response-time usability goals for most user-interactive applications. Response times above this threshold degrade productivity. Exact numbers depend on many difficult-to-formalize factors, such as the number and types of elements viewed or repetitiveness of the task, but a goal of three to five seconds is reasonable for most typical business applications.

- ***Users are focused on the dialog (8+ seconds):*** They keep their attention on the task. Miller reported this threshold as 10 seconds. Anything slower needs a proper user interface (for example, a percent-done indicator as well as a clear way for the user to interrupt the operation). Users will probably need to reorient themselves when they return to the task after a delay above this threshold, so productivity suffers.

A Closer Look At User Reactions

Peter Bickford investigated user reactions when, after 27 almost instantaneous responses, there was a two-minute wait loop for the 28th time for the same operation. It took only 8.5 seconds for half the subjects to either walk out or hit the reboot. Switching to a watch cursor during the wait delayed the subject's departure for about 20 seconds. An animated watch cursor was good for more than a minute, and a progress bar kept users waiting until the end.

Bickford's results were widely used for setting response times requirements for Web applications. C. Loosley, for example, wrote, "In 1997, Peter Bickford's landmark paper, 'Worth the Wait?' reported research in which half the users abandoned Web pages after a wait of 8.5 seconds. Bickford's paper was quoted whenever Web site performance was discussed, and the 'eight-second rule' soon took on a life of its own as a universal rule of Web site design."

A. Bouch attempted to identify how long users would wait for pages to load [10]. Users were presented with Web pages that had predetermined delays ranging from two to 73 seconds. While performing the task, users rated the latency (delay) for each page they accessed as high, average or poor. Latency was defined as the delay between a request for a Web page and the moment when the page was fully rendered. The Bouch team reported the following ratings:

Good	***Up to 5 seconds***
Average	***From 6 to 10 seconds***
Poor	***More than 10 seconds***

In a second study, when users experienced a page-loading delay that was unacceptable, they pressed a button labeled "Increase Quality." The overall average time before pressing the "Increase Quality" button was 8.6 seconds.

In a third study, the Web pages loaded incrementally with the banner first, text next and graphics last. Under these conditions, users were much more tolerant of longer latencies. The test subjects rated the delay as "good" with latencies up to 39 seconds, and "poor" for those more than 56 seconds.

This is the threshold that gives us response-time usability requirements for most user-interactive applications. Response times above this threshold cause users to lose focus and lead to frustration. Exact numbers vary significantly depending on the interface used, but it looks like response time should not be more than eight to 10 seconds in most cases. Still, the threshold shouldn't be applied blindly; in many cases, significantly higher response times may be acceptable when appropriate user interface is implemented to alleviate the problem.

Not-So-Traditional Performance Requirements

While they're considered traditional and absolutely necessary for some kind of systems and environments, some requirements are often missed or not elaborated enough for interactive distributed systems.

When resource requirements are measured as resource utilization, it's related to a particular hardware configuration.

Concurrency is the number of simultaneous users or threads. It's important: Connected but inactive users still hold some resources. For example, the requirement may be to support up to 300 active users, but the terminology used to describe the number of users is somewhat vague. Typically, three metrics are used:

- ***Total or named users.*** All registered or potential users. This is a metric of data the system works with. It also indicates the upper potential limit of concurrency.
- ***Active or concurrent users.*** Users logged in at a specific moment of time. This is the real measure of concurrency in the sense it's used here.
- ***Really concurrent.*** Users actually running requests at the same time. While that metric looks appealing and is used quite often, it's almost impossible to measure and rather confusing: the number of "really concurrent" requests depends on the processing time for this request. For example, let's assume that we got a requirement to support up to 20 "concurrent" users. If one request takes 10 seconds, 20 "concurrent" requests mean throughput of 120 requests per minute. But here we get an absurd situation that if we improve processing time from 10 to one second and keep the same throughput, we miss our requirement because we have only two "concurrent" users.

To support 20 "concurrent" users with a one-second response time, you really need to increase throughput 10 times to 1,200 requests per minute.

It's important to understand what users you're discussing: The difference between each of these three metrics for some systems may be drastic. Of course, it depends heavily on the nature of the system.

Performance and Resource Utilization

The number of online users (the number of parallel session) looks like the best metric for concurrency (complementing throughput and response time requirements). Finding the number of concurrent users for a new system can be tricky, but information about real usage of similar systems can help to make the first estimate.

Resources. The amount of available hardware resources is usually a variable at the beginning of the design process. The main groups of resources are CPU, I/O, memory and network.

When resource requirements are measured as resource utilization, it's related to a particular hardware configuration. It's a good metric when the hardware the system will run on is known. Often such requirements are a part of a generic policy; for example, that CPU utilization should be below 70 percent. Such requirements won't be very useful if the system deploys on different hardware configurations, and especially for "off-the-shelf" software.

When specified in absolute values, like the number of instructions to execute or the number of I/O per transaction (as sometimes used, for example, for modeling), it may be considered as a performance metric of the software itself, without binding it to a particular hardware configuration.

In the mainframe world, MIPS was often used as a metric for CPU consumption, but I'm not aware of such a

widely used metric in the distributed systems world.

The importance of resource-related requirements will increase again with the trends of virtualization and service-oriented architectures. When you depart from the "server(s) per application" model, it becomes difficult to specify requirements as resource utilization, as each application will add only incrementally to resource utilization for each service used.

Scalability is a system's ability to meet the performance requirements as the demand increases (usually by adding hardware). Scalability requirements may include demand projections such as an increasing number of users, transaction volumes, data sizes or adding new workloads.

From a performance requirements perspective, scalability means that you should specify performance requirements not only for one configuration point, but as a function, for example, of load or data.

For example, the requirement may be to support throughput increase from five to 10 transactions per second over the next two years, with response time degradation not more than 10 percent. Most scalability requirements I've seen look like "to support throughput increase from five to 10 transactions per second over next two years without response time degradation"—that's possible only with addition of hardware resources.

Other contexts. It's very difficult to consider performance (and, therefore, performance requirements) without context. It depends, for example, on hardware resources provided, the volume of data operated on and the functionality included in the system. So if any of that information is known, it should be specified in the requirements.

While the hardware configuration may be determined during the design stage, the volume of data to keep is usually determined by the business and should be specified.

Using multiple performance metrics that only together provide the full picture can complicate your process.

The Difference Between Goals And Requirements

One issue, as Barber notes, is goals versus requirements[11]. Most response time "requirements" (and sometimes other kinds of performance requirements) are goals (and sometimes even dreams), not requirements: something that we want to achieve, but missing them won't necessarily prevent deploying the system.

You may have both goals and requirements for each of the performance metrics, but for some metrics/systems ,they are so close that from the practical point of view, you can use one. Still, in many cases, especially for response times, there's a big difference between goals and requirements (the point when stakeholders agree that the system can't go into production with such performance).

For many interactive Web applications, response time goals are two to five seconds, and requirements may be somewhere between eight seconds and one minute.

One approach may be to define both goals and requirements. The problem? Requirements are very difficult to get. Even if stakeholders can define performance requirements, quite often go/no-go decisions are based not on the real requirements, but rather on second-tier goals.

In addition, using multiple performance metrics that only together provide the full picture can complicate your process. For example, you may state that you have a 10-second requirement and you took 15 seconds under full load. But what if you know that this full load is the high load on the busiest day of year, that the max load for other days falls below 10 seconds, and you see that it is CPU-constrained and may be fixed by a hardware upgrade?

Real response time requirements are so environment- and business-dependent that for many applications, it's cruel to force people to make hard decisions in advance for each possible combination of circumstances. Instead, specify goals (making sure that they make sense) and only then, if they're not met, make the decision about what to do with all the information available.

Knowing What Metrics to Use

Another question is how to specify response time requirements or goals. For example, such metrics as average, max, different kinds of percentiles and median can be used. Percentiles are more typical in SLAs (service-level agreements). For example, "99.5 percent of all transactions should have a response time less than five seconds."

While that may be sufficient for most systems, it doesn't answer all questions. What happens with the remaining 0.5 percent? Does this 0.5 percent of transactions finish in six to seven seconds or do all of them time out? You may need to specify a combination of requirements: for example, 80 percent below four seconds, 99.5 percent below six seconds, 99.99 percent below 15 seconds (especially if we know that the difference in performance is defined by distribution of underlying data). Other examples may be average four seconds and max 12 seconds, or average four seconds and 99 percent below 10 seconds.

Things get more complicated when there are many different types of transactions, but a combination of percentile-based performance and availability metrics usually works fine for interactive systems. While more sophisticated metrics may be necessary for some systems, in most cases sophistication can make the process overcomplicated and difficult to analyze.

There are efforts to make an objective user-satisfaction metric. One is Application Performance Index (www.Apdex.org). Apdex is a single metric of user satisfaction with the performance of enterprise applications. The Apdex metric is a number between 0 and 1, where 0 means that no users were satisfied, and 1 means all users were satisfied.

The approach introduces three groups of users: satisfied, tolerating and frustrated. Two major parameters are introduced: threshold response times between satisfied and tolerating users T, and between tolerating and frustrated users F[12]. There probably is a relationship between T and the

TABLE 1: THE SEVCIK METHODS

1. Default value (the Apdex methodology suggests 4 seconds)
2. Empirical data
3. User behavior model (number of elements viewed/task repetitiveness)
4. Outside references
5. Observing the user
6. Controlled performance experiment
7. Best time multiple
8. Find frustration threshold *F* first and calculate *T* from *F* (the Apdex methodology assumes that $F = 4T$)
9. Interview stakeholders
10. Mathematical inflection point

response time goal and between *F* and the response time requirement.

Where Do Performance Requirements Come From?

If you look at performance requirements from another point of view, you can classify them into business, usability and technological requirements. Business requirements come directly from the business and may be captured very early in the project life cycle, before design starts. For a requirement such as "A customer representative should enter 20 requests per hour, and the system should support up to 1,000 customer representatives," requests should be processed in five minutes on average, throughput would be up to 20,000 requests per hour, and there could be up to 1,000 parallel sessions.

The main trap here is to immediately link business requirements to a specific design, technology or usability requirement, thus limiting the number of available design choices. If we consider a Web system, for example, it's probably possible to squeeze all the information into a single page or have a sequence of two dozen screens. All information can be saved at once, or each page of these two dozen can be saved separately. We have the same business requirements, but response times per page and the number of pages per hour would be different.

Usability requirements (mainly related to response times) also figure into the performance equation. Many researchers agree that users lose focus if response times are more than eight to 10 seconds, and that response times should be two to five seconds for maximum productivity. These usability considerations may influence design choices (such as using several Web pages instead of one). In some cases, usability requirements are linked closely to business requirements; for example, make sure that your system's response times aren't worse than response times of similar or competitor systems.

Small anomalies from expected behavior are often signs of bigger problems.

The third category, technological requirements, comes from chosen design and used technology. Some technological requirements may be known from the beginning if some design elements are used, but others are derived from business and usability requirements throughout the design process and depend on the chosen design.

For example, if we need to call 10 Web services sequentially to show the Web page with a three-second response time, the sum of response times of each Web service, the time to create the Web page, transfer it through the network and render it in a browser should be below three seconds. That may be translated into response-time requirements of 200-250 milliseconds for each Web service.

The more we know, the more accurately we can apportion overall response time to Web services. Another example of technological requirements can be found in the resource consumption requirements. In its simplest form, CPU and memory utilization should be below 70 percent for the chosen hardware configuration.

Business requirements should be elaborated during design and development, and merge together with usability and technological requirements into the final performance requirements, which can be verified during testing and monitored in production. The main reason that we separate these categories is to understand where the requirement comes from: Is it a fundamental business requirement or a result of a design decision that may be changed if necessary?

Determining specific performance requirements is another large topic that is difficult to formalize. Consider the approach suggested by Sevcik for finding *T*, the threshold between satisfied and tolerating users. *T* is the main parameter of the Apdex (Application Performance Index) methodology, providing a single metric of user satisfaction with the performance of enterprise applications. Sevcik defined 10 different methods (see Table 1).

The idea is to use several (say, three) of these methods for the same system. If all come to approximately the same number, they give us *T*. While the approach was developed for production monitoring, there is definitely a strong correlation between *T* and the response time

goal (having all users satisfied sounds as a pretty good goal) and between *F* and the response time requirement. So the approach probably can be used for getting response time requirements with minimal modifications.

While some specific assumptions like four seconds for default or the $F = 4T$ relationship may be ip for argument, the approach itself conveys the important message that there are many ways to determine a specific performance requirement, which, for validation purposes, is best derived from several sources. Depending on your system, you can determine which methods from the above list (or maybe some others) are applicable, calculate the metrics and determine your requirements.

Usually you have no idea what caused the observed symptoms or how serious it is.

Requirements Verification: Performance vs. Bugs

Requirement verification presents another subtle issue: how to differentiate performance issues from functional bugs exposed under load. Often, additional investigation is required before you can determine the cause of your observed results. Small anomalies from expected behavior are often signs of bigger problems, and you should at least to figure out why you get them.

When 99 percent of your response times are three to five seconds (with the requirement of five seconds) and 1 percent of your response times are five to eight seconds, it usually isn't a problem. But it probably should be investigated if 1 percent fail or have strangely high response times (for example, more than 30 seconds, with 99% three to five seconds) in an unrestricted, isolated test environment.

This isn't due to some kind of artificial requirement, but is an indication of an anomaly in system behavior or test configuration. This situation often is analyzed from a requirements point of view, but it shouldn't be, at least until the reasons for that behavior become clear.

These two situations look similar, but are completely different in nature: 1.) The system is missing a requirement, but results are consistent: This is a business decision, such as a cost vs. response time tradeoff; and 2.) Results aren't consistent (while requirements can even be met): This may indicate a problem, but its scale isn't clear until investigated.

Unfortunately, this view is rarely shared by development teams too eager to finish the project, move it into production, and move on to the next project. Most developers aren't very excited by the prospect of debugging code for small memory leaks or hunting for a rare error that's difficult to reproduce. So the development team becomes very creative in finding "explanations."

For example, growing memory and periodic long-running transactions in Java are often explained as a garbage collection issue. That's false in most cases. Even in the few instances when it is true, it makes sense to tune garbage collection and prove that the problem is gone.

Teams can also make fatal assumptions, such as thinking all is fine when the requirements stipulate that 99 percent of transactions should be below *X* seconds, and less than 1 percent of transactions fail in testing.

Well, it doesn't look fine to me. It may be acceptable in production over time, considering network and hardware failures, OS crashes, etc. But if the performance test was run in a controlled environment and no hardware/OS failures were observed, it may be a bug. For example, it could be a functional problem for some combination of data.

When some transactions fail under load or have very long response times in the controlled environment and you don't know why, you've got one or more problems.

When you have an unknown problem, why not trace it down and fix it in the controlled environment? What if these few failed transactions are a view page for your largest customer, and you won't be able to create an order until it's fixed?

In functional testing, as soon as you find a problem, you usually can figure out how serious it is. This isn't the case for performance testing: Usually you have no idea what caused the observed symptoms or how serious it is, and quite often the original explanations turn out to be wrong.

Michael Bolton described the situation concisely [13]:

As Richard Feynman said in his appendix to the Rogers Commission Report on the Challenger space shuttle accident, when something is not what the design expected, it's a warning that something is wrong. "The equipment is not operating as expected, and therefore there is a danger that it can operate with even wider deviations in this unexpected and not thoroughly understood way." When a system is in an unpredicted state, it's also in an unpredictable state.

Raising Performance Consciousness

We need to specify performance requirements at the beginning of any project for design and development (and, of course, reuse them during performance testing and production monitoring). While performance requirements are often not perfect, forcing stakeholders just to think about performance increases the chances of project success.

What exactly should be specified—goal vs. requirements (or both), average vs. *X* percentile vs. Apdex, etc.—depends on the system and environment, but all requirements should be both quantitative and measurable. Making requirements too complicated may hurt here. You need to find meaningful goals/requirements, not invent something just to satisfy a bureaucratic process.

If you define a performance goal as a point of reference, you can use it throughout the whole development cycle and testing process, tracking your progress from a performance engineering viewpoint. Tracing this metric in production will give you valuable feedback that can be used for future system releases.

REFERENCES

1. Miller, R. B. Response Time in User-system Conversational Transactions, In Proceedings of the AFIPS Fall Joint Computer Conference, 33, 1968.