MULTIPLE DIMENSIONS OF PERFORMANCE REQUIREMENTS

Alexander Podelko
Oracle

From the first look, the subject looks simple enough. Almost every book or paper about performance has a few pages about defining performance requirements. Quite often a performance requirements section can be found in project documentation. But the more you examine the area of performance requirements, the more questions and issues arise. The goal of this paper is not answer all the questions, but rather discuss and evaluate the many aspects of this topic and to provide different views and examples.

Over the past two years I spent a lot of time thinking about performance requirements. From the first glance, the subject looks simple enough. But the more you look into the subject of performance requirements, the more issues you see. There is even no full agreement on when performance requirements should be collected. Quite often that happens at the end of development just before performance testing or deployment.

The goal of this paper is not to provide answers to all performance requirements questions, but rather to list these questions in one place and illustrate them by different views and examples when possible. Some of these questions may not have a single answer; the answer may depend on the context or even be an agreement between performance practitioners.

Traditional Performance Requirements

Defining performance requirements is an important part of system design and development. If there are no written performance requirements, it just means that they exist in the heads of stakeholders, but nobody bothered to write them down and make sure that everybody agrees with them. What exactly is specified may vary significantly depending on the system and environment, but the 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 times (how fast the system can handle individual requests) and throughput (how many requests the system can handle). All classes are vital: good throughput with long response times often is unacceptable as well as is good response times with low throughput.

Response times (in the case of interactive work) or processing times (in the case of batch jobs or scheduled activities) define 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 times depend on workload, so it is necessary to define conditions under which specific response times should be achieved: for example, a single user, average load or peak load.

A lot of research has been done to define what 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 task like entering orders) and what response time is necessary to avoid web site abandonment (for the Internet). Most researchers agreed that in most cases for interactive applications there is no point making response time faster than 1-2 sec and it is good to provide an indicator (like a progress bar) if it takes more than 8-10 sec.

The service / stored procedure response time requirements should be determined by its share in the end-to-end performance “budget” so the worst combination of all required services, middleware and presentation layer overheads will provide the required time. For example, if there is a web page with 10 drop-down boxes calling 10 separate services, the response time objective for each service may be 0.2 sec to get 3 sec average response time (leaving 1 sec for network, presentation, and rendering).

Response times for each individual transaction vary, so we need to use some aggregate values when specifying performance requirements like averages or percentiles (for example, 90% of response times are less than X). Maximum / timeout times should be provided also if necessary.

For batch jobs, it is also important to specify all schedule-related information like 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 change in one of the jobs may impact others).

Throughput is the rate at which incoming requests are completed. Throughput defines 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 like processing orders or printing reports (in some cases additional metrics may be necessary such as the number of items in an order or the size of a report). It may be more difficult for systems with complex workloads: the ratio of different types of requests can change with time and season.

It is also important to see 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, it is important 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 there are 500 users 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 on "Response Time in Man-Computer Conversational Transactions" described three threshold levels of human attention [Miller68]. 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 [Nielsen94]. 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 sec [Miller68]. According to Bickford 0.2 second forms the mental boundary between events that seem to happen together and those that appear as echoes of each other [Bickford97].

While it is a quite important threshold, it is often beyond the reach of application developers. That kind of interaction is provided by operating system, browser, or interface libraries and usually happens the client side, without interaction with servers (except 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 [Miller68]. Using the research that was available to them, several authors recommended that the computer should respond to users within two seconds [Miller68], [Bailey82], [Shneiderman84]. Another researcher reported that with most data entry tasks there was no advantage of having response times that were faster than one second, and found a linear decrease in productivity with slower response times (from one to five seconds) [Martin86]. With problem solving tasks, which are more like Web interaction tasks, they found no reliable effect on user productivity up to a 5-second delay.

The complexity of the user interface and the number of elements on the screen both impact thresholds. Back in 1960s - 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 1-2 seconds is 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 Barber, even users who are accustomed to a sub-second response time on a client/server system are happy with a 3-second response time from a Web-based application [Barber04].

Sevcik identified two key factors impacting this threshold [Sevcik03]: 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 a 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 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 depends on many difficult-to-formalize factors as the number and types of elements viewed or repetitiveness of the task, but it looks like it is about 3-5 seconds for most typical business applications.

Users are focused on the dialog (8+ seconds): they keep their attention on the task. Miller reported that threshold as 10 seconds [Miller68]. It is recommended that 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.

Bickford investigated user reactions when after 27 almost instantaneous responses there was a 2 min wait loop for 28th time for the same operation. It took only 8.5 seconds for half the subjects to either walk out or hit the reboot [Bickford97]. Switching to a watch cursor during the wait delayed the subject's departure for about 20 seconds. An animated watch cursor was good for over a minute, and a progress bar kept users waiting until the end.

Bickford results were widely used for setting response times requirements for Web applications. 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 "8-second rule" soon took on a life of its own as a universal rule of Web site design” [Loosley05].

Bouch attempted to identify how long users would wait for pages to load [Bouch00]. Users were presented with Web pages that had predetermined delays ranging from 2 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. They reported the following ratings:

Good	Up to 5 seconds
Average	From 6 to 10 seconds
Poor	Over 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, they had the Web pages load 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 over 56 seconds.

That 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 times should not be more than 8-10 seconds in most cases. Still the threshold shouldn't be applied blindly; there are many cases when significantly higher response times may be acceptable when appropriate user interface is implemented to alleviate the problem.

Not So Traditional Performance Requirements

While used for long time and are considered as traditional and absolutely necessary for some kind of systems and environments, some requirements are often missed or not elaborated enough for interactive distributed systems.

Concurrency is the number of simultaneous users or threads. It is important too: connected, but inactive users still hold some resources. For example, the requirement may be to support up to 300 active users.

When we speak about the number of users, the terminology is somewhat vague. Usually three metrics are used:

• Total or named users: all registered or potential users. That 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. That one is the real measure of concurrency in the sense it is used here.

• Really concurrent: users actually running requests at the same time. While that metric looks appealing and is used quite often, it is 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 sec, 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 1 second and keep the same throughput, we miss our requirement because we have only 2 “concurrent” users. To support 20 “concurrent” users with 1 second response time we really need to increase throughput 10 times to 1,200 requests per minute.

It is important to understand what users you are speaking about: the difference between each of these three metrics for some systems may be drastic. Of course, it heavily depends on the nature of the system.

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

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 is related to a particular hardware configuration. It is 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%. Such requirements won't be very useful if the system deploys on different hardware configurations (and especially for “Off-the-Shelf” software).

When it is 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 often was used as a metric for CPU consumption, but I am not aware about such a widely used metric in the distributed systems world.

It looks like the importance of resource-related requirements will increase again with the trends of virtualization and service-oriented architectures. When you go away 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 the ability of a system 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.

If you think about scalability from a performance requirements perspective, it 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 5 to 10 transactions per second over next two years with response time degradation not more than 10%. Most scalability requirements I have seen looked like "to support throughput increase from 5 to 10 transactions per second over next two years without response time degradation" that is possible only with addition of hardware resources

Other Context. It is very difficult to consider performance (and, therefore, performance requirements) without context. It depends, for example, on hardware resources provided, volume of data it operates on, and 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.

Goals vs. Requirements

One issue is goals versus requirements [Barber2007]. Most of 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.

We can probably say that we have both goals and requirements for each of performance metrics, but for some metrics / systems they are so close that from the practical point of view we can use one. Still in many cases, especially for response times, there is 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 times goals are 2-5 seconds and requirements may be somewhere between 8 seconds and 1 minute.

One approach may be to define both goals and requirements. The problem is that requirements are very difficult to get. Even if stakeholders can define performance requirements, quite often when it comes to "go"-"no go" decisions it becomes clear that it was not the real requirements but rather second-tier goals.

In addition, things are complicated by having multiple performance metrics which only together provide the full picture. For example, you may state that you have a 10 sec requirement and you got 15 sec under full load. But what if we know that this full load is the high load on the busiest day of year, max load for other days gets below 10 sec, and we see that it is CPU-constrained and may be fixed by hardware upgrade? The real response time requirements are so environment and business dependent, that for many applications it is somewhat cruel to force people to make hard decision in advance for each possible combination of circumstances. Maybe it is okay to specify goals (making sure that they make sense) and then, if they are not met, make the decision what to do with all the information available.

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, median can be used. Percentiles are more typical in SLAs (service-level agreements). For example, "99.5% of all transactions should have a response time less than 5 sec". While it is sufficient for most systems, it doesn't answer all questions. What happens with the rest 0.5%? Does these 0.5% of transactions finish in 6-7 sec or all of them timeout? We may need to specify a combination of requirements: for example, 80% below 4 sec, 99.5% below 6 sec, 99.99% below 15 sec (especially if we know that the difference in performance is defined by distribution of underlying data). Other examples may be average 4 sec and max 12 sec or average 4 sec and 99% below 10 sec. The things get more complicated when there are many different types of transactions. Still 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 it makes things overcomplicated and difficult to analyze.

There are efforts to make an objective user satisfaction metric. One is Application Performance Index (Apdex). 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 between tolerating and frustrated users F [Apdex, Sevcik06]. There probably is a relationship between T and the response time goal and between F and the response time requirement.

Where do performance requirements come from?

If we look at performance requirements from another point of view, we probably can classify them into business, usability, and technological requirements. Business requirements come directly from the business and they may be captured very early in the project lifecycle, before design starts. Something like "a customer representative should enter 20 requests per hour and the system should support up to 1000 customer representatives". So here we have that requests should be processed in 5 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 requirements and thus limit the number of available design choices. If we speak about a web system, for example, it is 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.

Another aspect is usability requirements (mainly related to response times). Many researchers agree that users lose focus if response times are more than 8-10 sec and that response times should be 2-5 seconds for maximum productivity. That is an important thing to remember and usability considerations may influence design choices (like using several web pages instead of one). In some cases, usability requirements are linked closely to business requirements: for example, make sure that response times are not worse than response times of similar or competitor systems.

The third category, technological requirements, comes from chosen design and used technology. Some of technological requirements may be known from the beginning if some parts of design are given, but other are derived from business and usability requirements throughout the design process and depends on the chosen design. For example, if we need to call ten web services sequentially to show the web page with a 3 second response time, the sum of response times of each web service, the time to create the web page, transfer it through network and render it in a browser should be below 3 second. That may be translated into response time requirements of 200-250 ms 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 is the resource consumption requirements. In its simplest form it may be that CPU and memory utilization should be below 70% 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 why 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 and may be changed if necessary.

Determining what specific performance requirements should be is another large topic that is difficult to formalize. An example how the task may be addressed is 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 ten different methods [Sevcik06]:

- default value (the Apdex methodology suggest 4 sec)
- empirical data
- user behavior model (number of elements viewed / task repetitiveness)
- outside references
- observing the user
- controlled performance experiment

- best time multiple
- find frustration threshold F first and calculate T from F (the Apdex methodology assumes that F = 4T)
- interview stakeholders
- mathematical inflection point.

Each method is discussed in details in [Sevcik06]. 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 strong correlations 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 4 seconds for default or F=4T relationship may be argued about, the approach itself conveys the important message that there are many ways to determine a specific performance requirement and it would be better to get it from several sources to validate it. 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.

Requirements Verification: Performance vs. Bugs

Requirement verification brings another subtle issue: how to differentiate performance issues from functional bugs exposed under load. Quite 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 you have 99% of response times 3-5 sec (with the requirement 5 sec) and 1% 5-8 sec it usually is not a problem. But it probably should be investigated if you get that 1% failed or had strangely high response times (for example, more than 30 sec with 99% 3-5sec) in an unrestricted isolated test environment. This is not because it is above some kind of artificial requirement, but because it is an indication of an anomaly in system behavior or test configuration. My concern is that 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 similarly looking situations are completely different: 1) the system is missing a requirement, but results are consistent: that is rather a business decision like a cost vs. response time trade off; 2) results are not consistent (while it even can meet requirements): that may be an indicator of a problem (and its scale isn't clear until investigated).

Unfortunately that view is rarely shared by development teams that are eager to finish the project, move it into production, and move on to the next project. Most developers are not very excited by the prospect of debugging code for small memory leaks or hunting for a rare error that is 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 is false in most cases. Even in the few cases when it's true it makes sense to tune garbage collection and prove that the problem went away.

Another typical approach is to say that the requirements are that 99% of transactions should be below X seconds, and for this test we have less than 1% of failed transactions. So everything is fine. Well, it doesn't look this way 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.

I am never comfortable when under load some transactions fail or have very long response times in the controlled environment and we don't know why. That indicates that we have one or more problems. When we have an unknown problem, why not to 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 any order for that customer?

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

Michael Bolton wrote that idea down very concisely [Bolton06]:

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.

Summary

We need to specify performance requirements at the beginning of any project for design and development (and, of course, re-use 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. That should be something quantitative and measurable. Making requirements too complicated may hurt here. We need to find meaningful goals / requirements, not invent something just to satisfy a bureaucratic process.

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

References

[Apdex] Apdex web site
http://www.apdex.org/

[Bailey82] Bailey, R.W. *Human Performance Engineering* (1st Edition), Prentice-Hall: Englewood Cliffs, NJ, 1982.

[Barber04] Barber, S. *Beyond performance testing: How fast is fast enough,* The Rational Developer Network, 2004.
http://www-128.ibm.com/developerworks/rational/library/4249.html

[Barber07] Barber, S. *Get performance requirements right - think like a user*, Compuware white paper, 2007.
http://www.perftestplus.com/resources/requirements_with_compuware.pdf

[Bickford97] Bickford P. *Worth the Wait?* Human Interface Online, View Source, 10/97.
http://web.archive.org/web/20040913083444/http://developer.netscape.com/viewsource/bickford_wait.htm

[Bolton06] Bolton M. *More Stress, Less Distress*, Better Software, November 2006.
http://www.stickyminds.com/sitewide.asp?ObjectId=11536&Function=edetail&ObjectType=ART

[Bouch00] Bouch, A., Kuchinsky, A. and Bhatti, N. *Quality is in the eye of the beholder: Meeting users' requirements for Internet quality of service*, CHI 2000, 297-304.
http://research.hp.com/personal/Nina_Bhatti/papers/chi.pdf

[Miller68] Miller, R. B. *Response time in user-system conversational transactions*, In Proceedings of the AFIPS Fall Joint Computer Conference, 33, 1968, 267-277.

[Loosley05] Loosley C. *When Is Your Web Site Fast Enough?* E-Commerce Times, 2005.
http://www.ecommercetimes.com/story/46627.html

[Martin86] Martin, G.L. and Corl, K.G. *System response time effects on user productivity*, Behavior and Information Technology, 5(1), 1986, 3-13.

[Nielsen94] Nielsen J. *Response Times: The Three Important Limits*, Excerpt from Chapter 5 of Usability Engineering, 1994.
http://www.useit.com/papers/responsetime.html

[Ramsey98] Ramsay, J., Barbesi, A. and Preece, J. *A psychological investigation of long retrieval times on the World Wide Web*, Interacting with Computers, 10, 1998, 77-86.

[Sevcik02] Sevcik, P. *Understanding How Users View Application Performance*, Business Communications Review, July 2002, 8–9.
http://www.bcr.com/architecture/network_forecasts%10sevcik/application_performance_20020719289.htm

[Sevcik03] Sevcik, P. *How Fast Is Fast Enough*, Business Communications Review, March 2003, 8–9.
http://www.bcr.com/architecture/network_forecasts%10sevcik/how_fast_is_fast_enough?_20030315225.htm

[Sevcik06] Sevcik, P. *Applying Apdex to Your Enterprise*, CMG 2006.

[Shneiderman84] Shneiderman, B. *Response time and display rate in human performance with computers*, Computing Surveys, 16, 1984, 265-285.