Performance Requirements: AN ATTEMPT at a Systematic View

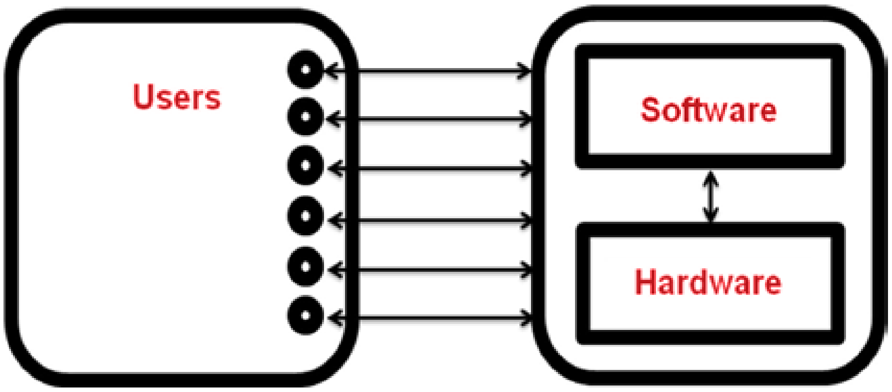

Fig. 1. A high-level view of a system

by AlexPODELKO

Performance Requirements: What is the Problem?

At first glance, the subject of performance requirements looks simple enough. Almost every book about performance has a few pages about 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.

Performance requirements are supposed to be tracked from the system inception through the whole system lifecycle including design, development, testing, operations, and maintenance. However different groups of people are involved in each stage using their own vision, terminology, metrics, and tools that makes the subject confusing when going into details.

For instance, business analysts use business terms. The architects' community uses its own languages and tools (mostly created for documenting functionality so performance doesn't fit them well).

Developers often think about performance through the profiler view. The virtual user notion is central for performance testers. Capacity planners use some mathematical terminology when they come up with queuing models. Production people have their own tools and metrics; and executives are more interested in high-level, aggregated metrics. These views are looking into the same subject – system performance – but through different lenses and quite often these views are not synchronized and differ noticeably. All of these views should be synchronized to allow tracing performance through all lifecycle stages and easy information exchange between stakeholders. Many existing approaches to describing performance requirements try to put these multi-dimensional and cross-dependent performance views into a set of simple flat templates designed for functional requirements.

IEEE Software Engineering Book of Knowledge (SWEBOK, http://www.computer.org/portal/web/swebok) defines four stages for the requirements process:

- **Elicitation:** gathering requirements
- **Analysis:** elaboration and negotiation requirements
- **Specification:** documenting requirements
- **Validation:** making sure that requirements are correct

Before diving into specific stages of performance requirements process, let's discuss the most important performance metrics (sometimes referred as Key Performance Indicators, KPIs). It is a challenge to get all stakeholders to agree on specific metrics and ensure that they can be measured in a compatible way at every stage of the lifecycle (which may require specific monitoring tools and application instrumentation).

Let's take a high-level view of a system (Fig. 1). On one side we have users who use the system to satisfy their needs. On another side we have the system, a combination of hardware and software, created (or to be created) to satisfy user's needs.

Business Performance Requirements

Users are not interested in what is inside the system and how it functions as soon as their requests get processed in a timely manner (leaving aside personal curiosity and subjective opinions). So business requirements should state how many requests of each kind go through the system (throughput) and how quickly they need to be processed (response times). Both parts are vital: good throughput with long response times usually is as unacceptable as are good response times with low throughput. Throughput is a business requirement whereas response times have two components which include usability requirements as well as business requirements. 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 processed orders per hour. In most cases we are interested in a steady mode when the number of incoming requests would be equal to the number of processed requests.

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 when they are homogenous. Clustering requests into a few groups, such as small, medium and large reports, may be needed if requests differ significantly. It may be more difficult for systems with complex workloads because the ratio of different types of requests can change with the time and season.

PERFORMANCE REQUIREMENTS ARE SUPPOSED TO BE TRACKED FROM THE SYSTEM INCEPTION THROUGH THE WHOLE SYSTEM LIFECYCLE INCLUDING DESIGN, DEVELOPMENT, TESTING, OPERATIONS, AND MAINTENANCE. AlexPODELKO

Throughput usually 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 environments with fixed hardware configuration the system should be able to handle peak load, but in virtualized or cloud environments it may be helpful to further detail what the load is hour-by-hour to ensure better hardware utilization.

"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."

AlexPODELKO

Homogenous throughput with randomly arriving requests (sometimes assumed in modeling and requirements analysis) is a simplification in most cases. In addition to different kinds of requests, most systems use a kind of session; some system resources are associated with the user (source of requests). So the number of parallel users (sessions) would be an important requirement further qualifying throughput. In a more generic way this metric may be named concurrency: the number of simultaneous users or threads. It is important, because connected but inactive users still hold some resources.

Quite often, however, the load on the system is characterized by the number of users. Partially it is coming from the business (in many cases the number of users is easier to find out). Partially it is coming from performance tests. Unfortunately, quite often performance requirements get defined during performance testing and the number of users is the main lever to manage load in load generation tools.

But 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 all systems achieve linear scalability).

The number of online users (the number of parallel sessions) looks like the best metric for concurrency (complementing throughput and response time requirements). However terminology is somewhat vague here, sometimes "the number of users" may have a completely different meaning:

- 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. In some cases it may be used as a way to find out concurrency as a percentage of total user population, but definitely is not a concurrency metric.
- "Really concurrent" users: the number of users running requests at the same time: In most cases it is matching the number of requests in the system. While that metric looks appealing, it is not a load metric: the number of "really concurrent" requests depends on the processing time for this request. The shorter the processing time, the fewer concurrent requests we have in the system. 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 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, we really need to increase throughput 10 times to 1,200 requests per minute.

It is important to understand what users we are discussing. The difference between each of these three "number of users" metrics may be drastic.

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.

Response time is the time in the system (the sum of queuing and processing time). Usually there is always some queuing time because the server is a complex object with sophisticated collaboration, multiple components including processor, memory, disk system, and other connecting parts. That means that response time is larger than service time (to use in modeling) in most cases.

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 website 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 is helpful to provide an indicator (like a progress bar) if it takes more than eight to 10 seconds.

Response times for each individual transaction vary, so we need to use some aggregate values when specifying performance requirements, such as averages or percentiles (for example, 90 percent of response times are less than X). Apdex standard (http://www.apdex.org) uses a single number to measure user satisfaction.

For batch jobs, it is important 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).

It is very difficult to consider performance (and, therefore, performance requirements) without full context. It depends, for example, on the volume of data involved, hardware resources provided, and functionality included in the system. So if any of that information is known, it should be specified in the requirements. Not everything may be specified at the same point. While the volume of data is usually determined by the business and should be documented at the beginning, the hardware configuration is usually determined during the design stage.

Technological Performance Requirements

The performance metrics of the system (the right side of the fig.1) are not important from the business (or user) point of view, but are very important for IT (people who create and operate the system). These internal (technological) requirements are derived from business and usability requirements during design and development and are very important for the later stages of the system lifecycle. Traditionally such metrics were mainly used for monitoring and capacity management because they are easier to measure and only recently tools measuring end-user performance get some traction.

The most wide-spread metric, especially in capacity management and production monitoring, is resource utilization. The main groups of resources are CPU, I/O, memory, and network. However, the available hardware resources are usually a variable in the beginning. It is one of the goals of the design process to specify hardware needed for the system from the business requirements and other inputs like company policies, available expertise, and required interfaces.

When resource requirements are measured as resource utilization, they are related to a particular hardware configuration. They are meaningful metrics when the hardware configuration is known. But these metrics do not make any sense as requirements until the hardware configuration would be decided upon; how can we talk, for example, about processor utilization if we don't know yet how many processors we would have? And such requirements are not useful as requirements for software if it gets deployed to different hardware configurations, and, especially, for Commercial Off-the-Shelf (COTS) software.

Only way we can speak about resource utilization on early phases of the system lifecycle is as a generic policy. For example, corporate policy may be that CPU utilization should be below 70 percent.

When required resources are specified in absolute values, like the number of instructions to execute or the number of I/O operations 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 such metric for CPU consumption, but there is no such widely used metric in the distributed systems world.

The importance of resource-related requirements is increasing again with the trends of virtualization, cloud computing, and service-oriented architectures. When we 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. There are attempts to introduce such metrics. For example, the 'CPU usage in MHz' or 'usagemhz' metric used in the VMware world or the 'Megacycles' metric sometimes used by Microsoft (for example, see Exchange mailbox sizing http://technet.microsoft.com/en-us/library/ee712771.aspx). Another related metric sometimes (but rarely) used is efficiency when it is defined as throughput divided by resources (however the term is often used differently).

In the ideal case (for example, when the system is CPU bound and we can scale the system linearly just adding processors) we can easily find needed hardware configuration if we have an absolute metric of resources required.

For example, if software needs X units of hardware power per request and a processor has Y units of hardware power, we can calculate the number of such processors N needed for processing Z requests as N=Z*X/Y. The reality, of course, is more sophisticated. First of all, we have different kinds of hardware resources: processors, memory, I/O, and network. Usually we concentrate on the most critical one keeping in mind others as restrictions.

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 of the number of users, transaction volumes, data sizes, or adding new workloads. How response times will increase with increasing load or data is important too (load or data sensitivity).

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

Scalability is also a technological (internal IT) requirement. Or perhaps even a "best practice" of systems design. From the business point of view, it is not important how the system is maintained to support growing demand. If we have growth projections, we probably need to keep the future load in mind during the system design and have a plan for adding hardware as needed.

Software Requirements Process

In the next part we plan to discuss all stages of the performance requirements process, which include elicitation, analysis, specification, and validation, according to the IEEE Software Engineering Book of Knowledge (SWEBOK). The article will consider each stage and their connection with other software life cycle processes.

About The Author

Alex Podelko** has specialized in performance engineering for the last fourteen years. Currently he is Consulting Member of Technical Staff at Oracle, responsible for performance testing and tuning of Hyperion products. Alex has more than 20 years of overall IT experience and holds a PhD in Computer Science from Gubkin University and an MBA from Bellevue University. Alex serves as a board director for Computer Measurement Group (CMG), His collection of performance-related links and documents can be found at **http://www.alexanderpodelko.com.