Identifying Derived Performance Requirements of System Components from Explicit Customer- and Application-Facing Performance Requirements

André B. Bondi
Software Performance and Scalability Consulting LLC
Red Bank, New Jersey 07701 USA
andrebbondi@gmail.com

ABSTRACT

Explicitly stated response time, throughput, and other performance requirements of an application implicitly impose other performance requirements on the system components that implement it. We call these *derived* performance requirements. The explicit performance requirements cannot be met if the derived performance requirements are not met. Explicit performance requirements naturally give rise to corresponding derived performance requirements expressed in terms of the same metrics. Derived performance requirements may also be identified that specify the sizes of object pools or the amount of memory needed to meet explicit and other derived requirements. Moreover, derived requirements may be identified that depend on the implementation of the components. We explore how derived requirements arise and present a methodology for identifying and specifying them.

Categories and Subject Descriptors

• General and reference~Performance • General and reference~Measurement • **Software and its engineering~Software performance • Software and its engineering~Requirements analysis**

Keywords

Performance requirements engineering, software engineering, software life cycle, performance analysis.

1. INTRODUCTION

The specification of an average response time requirement for a transaction inherently imposes upper limits on the acceptable values for the times taken to support the various actions that must occur to complete it .Similarly, the specification of a transaction completion rate inherently imposes throughput requirements on those actions and on those system components that must be traversed to carry those actions out. In addition to derived throughput and response time requirements, we must consider space/time requirements that arise indirectly, such as those for object pools needed to avoid queueing and buffer sizes needed to sustain low probabilities of message loss.

ICPE'17, April 22 - 26, 2017, L'Aquila, Italy.

ACM 978-1-4503-4404-3/17/04□$15.00
DOI: http://dx.doi.org/10.1145/3030207.3030246

We shall examine the performance requirements that must be derived from the end-to-end performance requirements to determine the suitability of these components for inclusion in the system. A performance model would be used to determine whether a component is suitable from a performance standpoint. The outputs of the model would be the parameters of the derived performance requirements. Examining, testing, and validating the performance requirements of components in the architectural phase of the lifecycle reduce the risk of rework close to the intended delivery date. Omission of these steps increases the risk of performance failure on delivery [9]. The identification of derived requirements cannot occur until the components have been specified in the architectural or design phases of the software life cycle. The derived performance requirements must conform to the same guidelines as the external performance requirements [3], [4].

Space/time considerations engender performance requirements on discrete software resources such as page frames, object pools, locks, and logical connectors like JDBCs. Their pool sizes must be large enough to keep the probability of transaction failure due to pool exhaustion below a very low value. The sizes of the resource pools needed to support the performance of the hosted applications are performance requirements that must be derived from the frequency with which their members are requested and the probability distribution of how long they are held.

Here, we explore ways to derive implied performance requirements about system components from specifications about performance requirements as seen from the standpoint of the sources of one or more requests for specified actions, or, in the case of a computer-controlled system, from the standpoints of components that are receiving commands from the control system and issuing stimuli or sending data to it. Examples of specified actions include initiating batch jobs, responding to transaction requests, triggering an activity within a system in response to a browser click, activating an alarm, delivering a message, or failing over to a standby machine. If the machine of interest is a communications switch or some other mission-critical component, there may be a requirement that a boot or failover be completed within a short amount of time. If several machines need to synchronously shake hands with a central server while booting, there will of necessity be both throughput and delay requirements on it that must be formulated and met to support the timing requirements of the booting procedure. The minimum achievable handshaking delay to the application of interest will be a lower bound on the achievable boot time. If the consequent lower bound is greater than the desired boot time, improvements must be made.

The architecture of a large-scale data analytics and storage system may be influenced by the volume of historical and recent data used as input to an analysis algorithm. The tradeoff between the cost of moving and storing that data and the location and cost of computing power may determine whether the computation of the analysis is done where the data are stored or if the data are moved to where the computation is done. That decision induces requirements for bandwidth and storage at each of the nodes involved.

Derived requirements can impact the configuration in a system that already exists or in an architecture that has already been chosen. For instance, in a multitier web system, the number of Java database connections (JDBCs) needed to prevent queuing for them is determined by the rate at which transactions occur between the application server and the backend database and the duration of those transactions. While one can observe whether packet loss, call loss, or queueing for a JDBC occurs orders of magnitude more often than it should, e.g., with relative frequency 10^{-3} instead of 10^{-9}, a long test time and a large number of transactions are needed to determine whether the relative frequencies of these events are 10^{-7} instead of 10^{-9}. If we can collect data to estimate throughputs and the means and variances of resource holding times, and buffer or object pool sizes, we can combine approximate modeling with tests that verify the configured object sizes to help us ensure that the frequencies with which undesired events occur are likely to be in the required range. Instrumentation of the object pool is needed to ensure that performance needs are met and to enable the measurement of object holding times and throughputs.

As with performance requirements that are formulated from the perspective of an end user or from the perspective of a software-controlled industrial system, derived performance requirements on software and hardware components provide a basis for defining the parameters of performance test plans. These test plans may be executed whenever the components are available. Testing of components should be conducted as early as possible so that one can identify unsatisfactory performance characteristics and choose alternative components well before release.

The remainder of this paper is organized as follows. After discussing related work, we consider how derived requirements arise, how they might be identified, and the metrics in which they might be expressed. These metrics may differ from those of the external performance requirements. For example, while a buffer pool exhaustion probability is not explicitly stated in end-to-end throughput and response time requirements, it may be necessary to specify one to sustain them. We then go on to describe a process for obtaining derived requirements, and possible impacts of explicit and derived requirements. Finally, we discuss the relationship between derived requirements and performance testing.

2. RELATED WORK

Hierarchical models can sometimes be formulated to determine the drivers of performance requirements, even if this was not the original intent [6], [13], [8].

In [14], Smith uses queueing network models and execution graphs to predict the effect of component throughputs and response times on the performance of an overall system. Smith illustrates heuristics for determining the extent to which components must be sped up to provide the required performance characteristics of components under a given workload. A response time requirement is more easily achieved when a component has a lower anticipated offered load. The Processing vs. Frequency Tradeoff Principle [14] states that one should minimize the processing times frequency product of each component. The implication is that a higher response time requirement might be tolerated if the frequency of execution of the component is smaller. The Centering Principle states that the dominant workload functions should be identified and their processing minimized. In the context of performance requirements, this means that the most frequently visited components should have more stringent response time requirements than those that are visited less often. In [15], it is suggested that performance budgets be allocated to collaborating components to ensure that an end-to-end response time requirement is met. Argent-Katwala *et al* [1] use process algebra models to show how end-to-end performance might be predicted for complex interactions of concurrently executing processes and threads, but they do not describe the derivation of performance requirements based on these predictions. Grassi and Mirandola describe how performance models of components can be composed to yield performance models of the system as a whole [7], but they do not attempt to use their framework to derive performance requirements on the components. Bondi briefly discusses and gives examples of derived performance requirements in [3] and [4]. He explains how performance requirements could be derived from other requirements or from the values of performance measures prescribed in domain-related specifications such as fire codes, but he does not show how they might be derived from queueing network models or execution graphs.

3. OCCURRENCE OF DERIVED PERFORMANCE REQUIREMENTS

From the standpoint of the stakeholders of a system component, such as architects, developers, and owners, the response time and throughput requirements appear to be externally driven. These requirements may have arisen from the needs of the system components that use the component of interest, or they may be domain-specific. For instance, the routing of parcels in a conveyor system may be determined by queries to a database [5]. The query and the response must be delivered before a parcel arrives at the next junction in the system. The time allowed for the response to be delivered depends on the conveyor speed. This imposes a response time requirement on the combined database query response time and the latency of the network and programmable logic controllers that run the belt. The frequency with which queries occur depends on the rate at which parcels pass a bar code scanner. That rate in turn depends on routing patterns, the speed of the belt, and the distance between parcels on each segment of the belt. A model is needed to determine the peak database query rate as a function of the number of parcel movements and of the parcel volume combined. The calculated query rate constitutes a throughput requirement of the parcel routing database.

A driver of the required object pool size is the number of discrete objects needed to sustain the throughput of the units of work that need them to proceed through the system. For an analogy, consider a physical system such as an airport security checkpoint. The mechanism by which objects such as trays for passengers' belongings are returned to the free pool plays a crucial role in determining the necessary object pool size, i.e., the number of trays, to sustain throughput, because it affects the object holding time. To see this, let us compare the mechanisms by which recently emptied trays are moved from the secured side of the checkpoint to the unsecured side so that they can be reused. At Newark Airport, the released trays are stacked up on dollies and

returned to the unsecured side of the checkpoint according to rules that are unknown to this author. At Gatwick Airport, released trays are placed one at a time on a conveyor positioned under the roller table used to move the trays and luggage towards the X ray machine, and are thus continuously returned to unsecured side. Little's Law tells us that the number of trays (and in the case of Newark, the number of dollies) needed to sustain a given level of passenger throughput depends in part on the tray return procedure, and that the achievable passenger throughput will be degraded if trays are only returned when very large stacks of them accumulate before they are made available. At Gatwick, the throughput and latency of the returning belt are factors that only limit passenger throughput and increase the holding times for trays to the extent that the latency exceeds the time for a passenger to traverse a check point once the trays have been filled. The Gatwick system is analogous to bytewise acknowledgement of messages in a sliding window protocol, while the Newark system is somewhat analogous to intermittently acknowledging the delivery of large packets before more data can be sent. The architecture of each tray return system engenders its own set of derived requirements. At Newark, the use of dollies to return stacks of trays engenders requirements on the number of trays needed at each checkpoint, the number of dollies needed at the passenger entry and exit points at each baggage X ray machine, and the frequency with which the dollies are returned to the passenger entry point at the approach to each baggage X ray machine. At Gatwick, the use of a conveyor mounted under the table of rollers at the approach to each X ray machine engenders requirements for tray throughput, the number trays needed to retain that throughput, the travel time of each tray on return, and the numbers of dollies to hold stacks of trays at the conveyor entry and exit points, even if the trays are not moved on them. The average time to inspect each piece of luggage and each tray as they pass through the X ray machine is independent of tray movements from the exit to the entry point. It depends on the capabilities of the human inspector and perhaps on any automated intelligence that is used to interpret the X ray images.

The airport security example is a metaphor for what might happen in a control system. If each tray is equipped with an RFID, there must be a corresponding record for it in the associated computerized tracking system. This illustrates how domain-specific requirements can arise from physical situations. Domain-specific requirements such as those for computer-controlled systems induce performance requirements on the software systems that provide the control and on the services that support them. Similarly, Software as a Service (SaaS) and Service-Oriented Architectures (SOAs) have performance requirements imposed upon them by the demands and transaction rates of the applications.

4. OBTAINING DERIVED PERFORMANCE REQUIREMENTS

In our experience, derived requirements arise while identifying a top-down hierarchy of non-functional requirements or quality attributes, including performance requirements, starting with external, explicitly specified performance requirements. These correspond to a top down view of both the architecture and of the implementation. There is a need to iterate between architecture choices and performance requirements on the architectural components to eliminate bottlenecks, foci of overload. Then one must respond to tradeoffs between architectures due to the cost of processing, the cost of replication, and the cost of data movement, as well as to constraints imposed by the need to incorporate legacy components or components from a designated supplier. This is especially important if modeling to derive requirements shows that a component cannot meet performance requirements. The predicted or measured inability for a component to meet performance requirements may necessitate its replacement with something else, a performance improvement, or even a change to the envisioned deployment scenario or other architectural aspect.

Our proposed process of deriving performance requirements is closely related to the execution graph methodology proposed in [14] for predicting the demands on various components and modeling their response times. Performance modeling and deriving performance requirements both require that information flows be captured. The required sizes of discrete pools should be explored based on the derived understanding of throughputs and delays. Traversing the execution graphs enables one to identify where performance requirements need to be derived and where the sizing of discrete object pools is needed. A single tool and data representation of the system could be used for both activities. The process is top-down recursive in the sense that each component may in turn consist of other components for which derived performance requirements are needed..

We can use a process to derive performance requirements that is analogous to modeling the performance of complex systems via hierarchical decomposition. This entails decomposing the system into different parts, modeling them separately (perhaps approximately), and then iteratively plugging the outputs of component models into a model of the entire system. This process of hierarchical decomposition allows the modeler to approximately capture the interactions between the components. The global or higher level system model predicts the loads that will be offered to the components. The models of the components are used to predict response times and maximum attainable throughputs. These predictions are fed into the higher-level model. The process is repeated until convergence is achieved.

We can use a similar process to derive performance requirements. The higher-level model predicts the throughput requirements of the subcomponents. We can also use the higher-level model to approximately predict whether the achievable response times of the subcomponents at the offered loads are low enough to achieve the required response time of the entire system, and, if not, to determine how low the response times of the subcomponents must be to achieve the overall response time. Sufficiently low values for the response times of the subcomponents at the predicted offered throughputs constitute the performance requirements for the respective measures of the components of interest. Performance improvements must be made to subcomponents that cannot meet those requirements, or else alternative components or even a different architecture must be used instead.

We may identify the following steps to derive performance requirements for components.

The first step is to ensure that a baseline set of end-to-end performance requirements exists. If the system is a platform for service-oriented architectures (SOA) [11] and for software as a service (SaaS), a baseline set of performance requirements may be obtained by first identifying current use cases and then computing reference workloads from them. The reference workloads will constitute the baseline for performance requirements. Like all performance requirements, the baseline requirements must be traceable, testable, verifiable, and unambiguous. They must be expressed in measurable terms [4]. It is important that the baseline performance requirements include specifications regarding peak traffic and the anticipated durations of the peaks, because the

onset of peaks will tax discrete object pools most heavily. If the workload has not yet been defined or is in dispute, a reference workload and a set of corresponding performance requirements should be devised as a basis for system design and testing [12].

The second step is a review of the information flow through the system for frequently occurring and computationally intensive use cases. This step should be carried out whether the system already exists or whether it is in the early stages of a software development life cycle. It is also the first step in a performance-oriented architecture review. A welcome side benefit of this step is that it may uncover performance bottlenecks that had not been thought of before, and thus trigger a modification of the architecture before any implementation or procurement is done. The links between functional requirements, end-to-end performance requirements, security, and other non-functional requirements can be explored at this time. The development of execution graphs to identify performance bottlenecks could be part of this step [14].

The third step is to develop a top-down recursive hierarchy of throughput, response time, and supporting requirements such as buffer sizes and object pool sizes for the various model components. These can be mapped to external transactions. The method is top-down because we are using end-to-end requirements to identify derived requirements. The method is recursive in the sense that we are identifying derived requirements as we proceed from one layer of components to the next. Performance requirements and resource usage demands for background activity such as database maintenance, archiving, batch jobs, or backups should be considered in this step as well. A framework for doing so could be developed using the principles in [14] as starting points. For systems in which two or more activities are launched simultaneously that must all be concluded before work can proceed to the next processing step, one must use the time to complete the longest activity to compute the holding times of any resources that are used.

5. ARCHITECTURAL IMPACTS

An architecture may have to be changed if one or more of its components cannot meet the derived performance requirements imposed upon it by traffic or by architectural characteristics. This can occur in different contexts. For example, domain-specific requirements such as those for computer-controlled systems induce performance requirements on the software systems that provide the control and on the services that support them. The combined effects of stringent response time requirements and bandwidth requirements for status information may necessitate a decentralized architecture in which control logic is deployed close to the system being controlled, while voluminous status data is moved upstream for storage and subsequent analysis. This is one of the concerns related to Fog Computing and the architecture of the Internet of Things [2]. As an architecture is formulated and the data flows between its components are modeled, a hierarchy of functional and nonfunctional requirements, including performance and security requirements, will emerge to reveal that saturation of some components is unavoidable unless the deployment scenario is changed [10]. This means that it will be necessary to iterate between (a) architectural specifications, (b) modeling to understand the performance requirements on system components, and (c) the formulation of models and new performance requirements as proposed changes the architecture are evaluated.

6. IMPACTS ON PERFORMANCE TESTING

End-to-end performance requirements are useful for planning performance tests over suitable ranges of load parameters and configurations [4]. The same holds for the derived performance requirements of software and hardware components, such as networks, object pools, and even of third party services whose function and performance are outside the control of the system owner. A third-party service might be essential to a transaction when a functional requirement or regulation demands that it be invoked before a transaction can be completed. Yet, the attainable throughputs and response times of these services might be outside the control of the owners of primary application web site, while affecting its end-to-end performance.

One element of a process for identifying derived performance requirements is somewhat similar to the process of hierarchical decomposition used to model components and their interactions with one another. It also has much in common with the combination of execution graphs and queueing models used to predict the performance of complex systems [14]. Performance test plans can be derived for system components that can be tested in environments one controls just as they would be for the whole system. Once a set of performance requirements has been derived for a system component, it should be subjected to performance tests structured in the same manner as for end-to-end performance tests. A component whose load is driven by transactions would be tested for a range of offered throughputs above and below the targets predicted by a high-level model described in [14]. Similarly, software platforms such as a messaging environment would also be tested over a variety of loads before it is incorporated into the architecture and before a commitment is made to development decisions that assume its presence [8]. Where it is not possible to verify a performance requirement that an event occur rarely, such as packet loss or the exhaustion of an object pool, tests should be conducted to ensure that the preconditions for the event to occur rarely as predicted by a model are satisfied given assumptions about the nature of the traffic and resource holding times. For example, a test would verify that the pool size is configured to N if, according to the model, N is the number needed to keep the exhaustion probability below a set level.

7. CONCLUSIONS

External performance requirements inherently impose performance requirements upon system components. The components may have structural characteristics, such as object pool sizes, that necessitate the formulation of performance requirements for loss and blocking probabilities. When these requirements cannot easily be validated by testing, because they refer to events that are intended to occur very rarely, it may be useful to verify instead that the configuration parameters correspond to those used to make a performance prediction with a model, such as a pool size or buffer size. In the foregoing, we have outlined how a hierarchical decomposition of the system could be used to model the parameters of performance requirements on components. This method is analogous to that used for approximate performance modeling. Finally, we discussed how performance test plans could be written to verify that the derived performance requirements on the components are met.

8. ACKNOWLEDGMENTS

This paper was completed while the author was a visiting professor at the University of L'Aquila.

9. REFERENCES

[1] Argent-Katwala, A., J. T. Bradley, and N.J. Dingle. Expressing performance requirements using regular expressions to specify stochastic probes over process algebra models. Proc. ACM WOSP 2004Workshop on Software and Performance) Redwood Shores, California, 49-58, 2004.

[2] Bonomi, F., R. Milito, J. Zhu, S. Addeppelli. Fog computing and its role in the internet of things. MCC'12, Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, 13-16, 2012.

[3] Bondi, A. B. "Best practices for writing and managing performance requirements: a tutorial." *Proc. ICPE 2012*, 2012.

[4] Bondi, A. B. Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice. Addison-Wesley, 2014.

[5] Bondi, A.B., C. S. Simon, and K. A. Anderson. Bandwidth usage and network latency in a conveyor system with Ethernet-based communication between controllers. Proc. IEEE PACRIM 2005, 9-12, 2005.

[6] Chandy, K.M., U. Herzog, and L. Woo. Parametric Analysis of Queuing Networks. IBM J. of R.&D. 19 (1), 36-42, 1975.

[7] Grassi, V., and R. Mirandola. Toward automatic compositional performance analysis of component-based systems. Proc. ACM WOSP 2004Workshop on Software and Performance) Redwood Shores, California, 59-63, 2004.

[8] Lazowska, E. J., J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance.* Prentice Hall, 1984. Also available online at www.cs.washington.edu/homes/lazowska/qsp .

[9] Masticola, S., A. B. Bondi, and M. Hettish. "Model-based scalability estimation in inception-phase software architecture." In *ACM/IEEE 8th International Conference on Model-Driven Engineering Languages and Systems, 2005. Lecture Notes in Computer Science* 3713, 355–366. Springer, 2005.

[10] Mylopoulos, J., L. Chung, and B. Nixon. Representing and using nonfunctional requirements: a process-oriented approach. IEEE Transactions on Software Engineering 18 (6), 483 – 497, 1992.

[11] O'Brien P. P. Merson, and L. Bass. Quality Attributes for Service-Oriented Architectures. SDSOA '07 Proceedings of the International Workshop on Systems Development in SOA Environments, 2007.

[12] Rempel, G. Defining Standards for Web Page Performance in Business Applications, Proc ICPE2015 (International Conference on Performance Engineering), 245-252, 2015.

[13] Sauer, C. and K. M. Chandy. Computer Performance Modeling. Prentice Hall, 1981.

[14] Smith, C.U. Independent General Principles for Constructing Responsive Software Systems. ACM TOCS 4(1), 1-31, 1986.

[15] Smith, Connie U., and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.