Architecture-Based Performance Analysis[*]

Bridget Spitznagel and David Garlan

School of Computer Science
Carnegie Mellon University
E-mail: sprite@cs.cmu.edu, garlan@cs.cmu.edu

Abstract

A software architecture should expose important system properties for consideration and analysis. Performance-related properties are frequently of interest in determining the acceptability of a given software design. In this paper we show how queueing network modeling can be adapted to support performance analysis of software architectures. We also describe a tool for transforming a software architecture in a particular style into a queueing network and analyzing its performance.

1. Introduction

An important issue for the engineering of complex software systems is determining overall system performance. Currently estimating performance for a system, in advance of actually building it, is something of a black art, relying on previous experience, local knowledge, and ad hoc techniques. This is unfortunate, since for many systems, it is often possible to make fairly good predictions of the expected performance of individual parts. At the very least, it would be good to be able to try out various "what-if" scenarios, contrasting the adequacy of different designs under different assumptions.

One emerging approach for dealing with such problems is to take an architectural view of the software. The software architecture of a system determines its overall structure as a collection of interacting components. By operating at this level of abstraction, one would hope to be able to reason in a straightforward way about overall system properties such as performance, reliability, etc.

To take a simple example, consider the mini-architecture illustrated in Figure 1, which contains three interacting components: a web server, a web client, and a database. Assume that the client makes requests of the server and receives responses asynchronously. The server may make a request of the database in the process of filling the client's request.

The acceptability of this design will likely depend on several unanswered questions concerning the overall performance of the system, such as:

- How well can this web site handle the anticipated demand? What will the average response time be? How large should buffers be?
- Given a maximum acceptable response time, what is the highest demand the web site can handle?
- Suppose that the demand is expected to peak for brief periods, and degraded performance during this time is acceptable. How much will performance degrade?
- Which component is the bottleneck? Should it be upgraded or replicated? How does the transmission rate of the data affect performance?

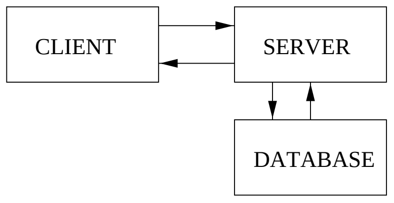

Figure 1. An internal web site

Unfortunately, most architectural designs are characterized informally and provide weak support for system-level analyses. As a result, it may be difficult to answer questions such as these with any degree of precision.

One of the stumbling blocks is that architects have a limited arsenal of concepts and tools to carry out such analyses at an architectural level of design. Even if good estimates of

[*]This research was sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031, and by the National Science Foundation under Grant No. CCR-9357792. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government.

performance can be determined for a system's components, it may be very difficult to derive overall system behavior.

Ideally what is needed is a way to exploit system designers' knowledge to derive the expected performance of the system from performance-related attributes of its parts. Fortunately, a mathematical model already exists in a similar domain. Queueing network theory is used in computer systems performance analysis to predict attributes of a system from attributes of its parts. A queueing network processes jobs. The elements in a queueing network are hardware devices, each of which has a queue. Jobs require service from a set of devices and wait in a queue when a desired device is busy. Each job exists in only one device or queue at a time.

Adaptation of this technique to software architecture performance analysis might seem straightforward at first glance. Hardware devices are replaced by software components. A job is replaced by a sequence of requests for service. Each component receives and processes requests, and when a request is completed, the component may send a new request for service to another component. The path of the sequence of requests is determined by converting the connections between components described in the architecture to an acyclic directional graph.

However this adaptation is overly simplistic and incomplete, as becomes evident when it is applied to examples even as simple as the web site example. In particular, the adaptation implicitly makes several assumptions, some of which are inappropriate in this context. For example, it assumes that all jobs have the same service requirements, and that connectors do not significantly affect performance. In order to apply queueing network analysis to a software architecture, we must examine and resolve such problems.

In this paper we show how to adapt queuing network theory so that it *can* be applied to a significant class of architectural designs. By doing this we show how to bring system building and analysis knowledge to bear on the design process by harnessing an existing analysis technique for a new domain. We begin by briefly reviewing relevant elements of queueing network theory. Next we consider the straightforward application of this theory. Though powerful when it can be applied, its limitations make it less than ideal. Then we show how the basic ideas can be extended to handle a much broader class of system, including those with cycles, autonomous clients, replicated services, and connector delays. Finally, we briefly describe the implementation of a tool that carries out architecture-based performance analysis, and outline future directions in this line of research.

2. Queueing Network Theory

To set the stage we begin with a brief introduction to queueing theory. We will cover only the essentials of product form networks.[1]

The basic units of a queueing network are "service centers" and "queues." A queue is a buffer with some queueing discipline (FIFO, round robin, LCFS preemptive resume). A service center provides some necessary service. Examples include a bank teller, hardware device, or database. Each service center has a queue containing jobs to process.

A replicated service center represents m identical providers of service, which draw their jobs from a single queue. For example, an airline's baggage check counter has one line of customers but many ticket agents. An infinitely replicated service center ($m = \infty$) is called a "delay center," and may be used to model a transmission delay.

Queueing network analysis can produce results both for individual queues (associated with service centers), as well as for the network as a whole.

To derive performance characteristics for *individual* service centers, two important pieces of information must be known: the average time the service center takes to process one job (service time), and the average rate at which jobs arrive (arrival rate). The service time and the time between job arrivals are usually taken to have exponential distributions.[2]

An exponential probability density function with expected value $1/\lambda$ is given by

$$f(t) = \lambda e^{-\lambda t}$$

It is sometimes called "memoryless" because the expected time left to wait is always $1/\lambda$, regardless of how much time has passed. This renders the history of the system unimportant, greatly simplifying the analysis. We assume exponential distributions, but will return to this issue in section 7.

From this information, results in queueing theory make it possible to calculate for a single queue:

- Utilization (how often the service center is occupied.)
- Average time a job spends waiting in the queue.
- Average queue length.
- The probability that the queue length is n.
- Whether the system is stable or overloaded. In an overloaded system, the queue grows faster than jobs can be processed; the server cannot keep up.
- For a queue implemented as a buffer of length B, the rate at which incoming jobs are discarded due to buffer overflow (drop rate.)

A *queueing network* is an interconnected group of these queues. Jobs enter the network, receive service at service centers, and leave. The average rate at which jobs enter the network (system arrival rate) must be known. For each service center, the service time must be known, as well as the

[1] For a more detailed treatment see Lazowska [8] or Sauer [9].

[2] Sometimes the distribution is known to be non-exponential, but close enough. If it has a larger variance, the analysis will be too optimistic.

rate at which jobs arrive at its queue relative to the system arrival rate (relative arrival rate). In addition to the above results it is possible to calculate expected values for

- Latency, the time for a job to be completely processed
- Throughput, the rate at which jobs are processed
- Number of outstanding jobs in the system
- Most-utilized service center, a possible bottleneck.

The system arrival rate and a queue's relative arrival rate determine the actual arrival rate at that queue. The relative arrival rate may be specified if known; otherwise it must be derived from the probabilistic path of a job through the network. For example, jobs leaving the web server may have a 60% chance of proceeding to the database. These transition probabilities are expressed as a set of linear equations and solved for the relative arrival rates.

Some systems process several kinds of jobs. To model jobs with different behavior, each queue is divided into one or more "job classes." For the purposes of this paper, job class will affect only the next destination of the job, not service times or processing order. Transition probabilities are now specified between job classes, instead of queues.

For example, in the context of our example architecture, it may be that a job which was class c_1 at the web server is likely to proceed to the database, while a job that was class c_2 at the server will always proceed to the client.

3. Application To Software Architecture

We begin with the simplest possible translation of queueing analysis into architectural terms. Though this simple analysis is sufficient for some systems, it is too weak for others. We will resolve these problems by extending the translation in the next section.

This translation is based on a "distributed message passing" architectural style, in which a design is fairly close to its queueing network equivalent. The components represent distributed processes. Connectors in this style are directional and represent asynchronous message streams; the messages are queued for processing by the components.

We assume that each component has a single queue, and that messages are processed in FIFO order.[3] When a component processes a message, it may produce 0 or 1 new messages as a result. A message entering the system thus corresponds to a queueing network job; when the first component finishes processing it and sends a new message to another component, then the new message represents that same job. The job exists as a sequence in time of individual messages, and is completed when the sequence terminates.

To illustrate, consider the following simplification of the example in Figure 1. Jobs (or messages) arrive from outside; each visits the client C, the server S, and the database D, and is finished. R jobs arrive in the system per second, so R jobs/s arrive in each component's queue. The components' service times are S_C, S_S, and S_D. For the automated analysis, we add a service time property to each component, and an arrival rate property to the system.

The utilization of a component i is $u_i = RS_i$, its average queue length is $q_i = u_i^2/(1 - u_i)$, and its average response time is $S_i/(1 - u_i)$. The average population of i is $p_i = u_i/(1 - u_i)$, comprising jobs in its queue and jobs receiving its service. The probability that $p_i \geq n$ is u_i^n. The system population P is the sum of the components' populations, and the system response time is P/R.

Suppose $R = 9.5$ / s, $S_C = 65, S_S = 20, S_D = 103$ ms. Then we expect a system response time of 5 s. The utilization of the database component is 98%. This component is close to overloaded, and is likely to have a long queue and a high latency. On average the queue length will be 44 elements; there is a 27% probability that the length will be 60 or more. If the estimated S_D turns out to be slightly larger, the database will be unable to keep up. The system will be acceptable only if the database is upgraded.

While these results are useful, the technique is of limited applicability for several reasons.

- It assumes that the set of services required by a job is implicit in the architecture, such that each job visits each service center once. This is not always the case.
- There is no notion of autonomous clients. Jobs must arrive from outside at a known rate; components cannot issue new jobs as in the original example.
- When a bottleneck is found in a system, sometimes the component responsible is replicated to distribute its load across more than one device. Modeling this at the software architecture level is desirable but will require careful consideration.
- Connectors between components can add delays, affecting the system's response time. They should certainly be included in the model.
- There are further complications in understanding, at the level of software architecture, the requirements and assumptions which are natural at a mathematical level and may be inherent at a hardware component level; e.g., the degree to which one service center is loaded must not affect the service time of another.

4. Extending The Model

Having observed the inadequacies of the simple translation illustrated above, we now describe several key extensions which make the translation more widely applicable.

[3] Other queueing disciplines would be permissible; see section 2.

4.1. Cycles

The original architecture processed two kinds of jobs. A "fetch" visits the client, the server, and the client again; a "query" visits the client, server, database, server, and client.

To analyze the performance of this system, it is necessary to attach additional information to the architecture: each component will have a list of the kinds of messages (corresponding to job classes) it services and their transition probabilities, and the system will have a list of the incoming job classes and their arrival rates. These properties are used to create a set of linear equations for the components' relative arrival rates, enabling analysis as before.

For the example system, this information might be as follows. The server can receive three kinds of messages: *fetch* and *query* from the client and *answer* from the database. The client can receive *start* from outside and *end* from the server. Suppose that 40% of jobs are "fetch" and the rest are "query". When the client finishes processing a *start*, there is a 40% probability that it sends the server *fetch*; otherwise, it sends *query*. When the server finishes *fetch* or *answer*, it always sends the client *end*. When the server finishes *query*, it sends a message to the database; when the database finishes that message, it will send the server *answer*.

4.2. Autonomous Clients

In the original architecture, jobs were generated by the web client. This system can be easily transformed to one in which jobs arrive from the outside. However, a more complex system, in which many components initiate jobs of various classes, would be tedious to transform by hand. Job generation can remain associated with these components if this transformation is added to the automated analysis.

We add to each client properties specifying the classes and generation rates of initiated jobs. The system properties added above are no longer specified, since they will be calculated from these client properties. The effective system arrival rate R is the sum of the clients' generation rates.

One open question is whether request generation should take the usual service time, or a negligible amount of time[4]. In the example system, the latter is more appropriate.

4.3. Replication

There are several obvious ways to deal with a bottleneck component: replicate it, speed it up, or reduce the demand on the system. These options may vary in expense and difficulty, or even feasability, for a given system. When considering these tradeoffs it is helpful to compare the performance improvement that will result from applying each option at some level of expense. The results of the latter two can be calculated using the techniques already discussed.

For the purposes of analysis, the instances of the replicated component should be identical. This creates a problem: jobs processed by some systems do distinguish between instances of replicated components. In the web example, a job initiated by the client returns to that client for final processing and display. Replicating the client could result in an optimistic prediction, since the model will attribute the final processing to any client that is not busy.

We must also consider the rate at which this replicated client generates requests. The generation rate r will be redefined as the rate of one instance; then the overall rate is mr. The alternative is to simply specify the overall rate, which is inconvenient if the degree of replication m is changed.

The individual analysis for a replicated component is somewhat more complicated than for an ordinary component; the queue does not grow until m jobs are in service.

4.4. Delay in Connectors

In our example, the connectors are an abstraction for creating, sending, and enqueueing messages. So far their effect on system performance has been neglected. In reality, transmission delays can increase the system response time.

Service centers could be used to model connectors; this model assumes that only one message can be in transit at a time, and may produce inflated response times. Instead, we model a connector as a delay center. Its arrival rate is equal to the arrival rate at the component fed by the connector. Each connector has a delay time property, representing the average time it takes a message to traverse the connector.

A connector's transmission delay affects the system response time: the delay time of each connector traversed by a job is added to the job's response time. It does not affect component performance or bottlenecks.

5. Example Revisited

Consider a new version of the example used in section 3. Messages traversing connectors now incur a transmission delay. Jobs are no longer required to visit each component exactly once: each job returns to the client for final processing, and some jobs never visit the database. Replication is a permissible option. To illustrate these extensions, in terms of a model, assume that the following is true for this system.

Estimated service times are $S_C = 65$ ms, $S_S = 20$ ms, $S_D = 103$ ms. The delay D_{CS} for the connectors between the client and server is 1.5 s, and the delay D_{SD} for the connectors between the server and database is 50 ms. The client generates the two kinds of jobs, "fetch" and "query," described in section 4.1. The client generates 3 fetches and

[4]Specifying a different service time for request generation would violate an assumption needed to keep the mathematics simple.

7 queries per second. With this information we can now calculate various performance characteristics of the system.

The effective arrival rate of the system is the sum of all generation rates: 10. Transition probabilities for the system arrivals are calculated from the generation rates. In this system, request generation takes essentially no time, so the system arrivals will all be sent to the server; otherwise, they would be sent to the client for service. (Section 4.2)

Now it is possible to calculate relative arrival rates from transition probabilities. The relative arrival rate at the server is $1 \times \frac{3}{10}$ fetch $+2 \times \frac{7}{10}$ query $= 1.7$. At the database it is 0 fetch + 0.7 query. At the client it is 1, because the system arrivals are sent to the server to mimic a negligible client generation time; otherwise, it would be 2. (Section 4.1)

The relative arrival rates of the connectors are the same as the components whose queues they feed. (Section 4.4)

Now we return to the equations of section 3. The utilization of the database is $7 \times 0.103 = 72\%$. The server's is $17 \times 0.020 = 34\%$. The client's is $10 \times 0.065 = 65\%$. The database, closest to being a bottleneck, is of greatest interest. Its average queue length is 1.85 messages, and its response time is $103/(1 - 0.72) = 370$ ms.

The average population of the system is about 58 messages; of these, $17 \times 1.5 \times 2 = 51$ are in transit between C and S, and one is between S and D. (Section 4.4)

The average system response time is $58/10 = 5.8$ s. (3 s are due to the delay between client and server).

Now let us suppose that the database is being considered for an upgrade. Assume that it will be replaced by either a single instance with a service time of 75 ms, or two identical instances, each with service time of 110 ms. We would like to compare the performance of the two options while assuming the rest of the system remains unchanged.

The first option will have an average of 1.1 messages present in the queue and receiving service, and an average queue length of .58. The second option will have 3.8 present, and a queue length of 2.2. The utilizations are 53% and 39%; the second option will be better able to handle an above-average load. The response times are 160 ms and 141 ms. From a performance standpoint, the second option seems the better choice; other factors such as expense may also factor into the final decision. (cf. Section 4.3)

6. Implementation Status

The distributed message passing style described above has been implemented as a style in Aesop [3]. The basic component type is a Process, and the connector type is a MessageStream. MessageStreams are directional.

The Aesop environment allows a user to graphically construct a software architecture in this style, enter the numbers needed for analysis in component and connector "workshops," and run analysis tools on the architecture. Numeric analysis results such as expected queue length are displayed in the workshops. Other results are indicated graphically; overloaded components, which will be unable to keep up with the anticipated demand, are highlighted.

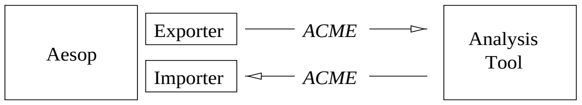

Figure 2. Performance analysis via Aesop

The style also provides an option on components to set the degree of replication. The effect of this option is to alter the component's appearance and change the predicted performance as described in section 4.3.

The analysis tool automatically performs the transformations described above. It reads in a text file containing an architecture described in Acme [4], determines which transformations to apply, analyzes the resulting network, and outputs an Acme description annotated with the results of the analysis. Aesop exports and imports these Acme descriptions (Figure 2), making the results visible to the user.

In a typical scenario, the user begins the iterative design process by constructing the top level design. He estimates the service times of the components, names the job classes, and fills in their transition probabilities. The user then runs the performance analysis tool. Based on the results he may replicate bottleneck components, decompose some components to provide estimates at a lower level, or otherwise refine the design. After making informed modifications, he repeats the process until an acceptable architecture is found.

7. Discussion And Future Work

We have shown how to apply queueing network modeling to software architectures in a particular style. A naive adaptation is sufficient for a few simple architectures, but proves inadequate for more interesting designs. It becomes much more useful with the extensions we have illustrated.

Three concerns remain. The performance predictions are based on unreliable estimates supplied by the user. The application above is restricted to one style. The underlying mathematical assumptions further restrict the systems that can be modeled. We will consider each of these in turn.

Dependence on unreliable data is unavoidable. Estimates made early in the design process cannot be completely accurate, and the performance analysis relies on these inputs. However, there is no reason to take this for the final answer. The performance analysis tool can be rerun as the design is altered and as more accurate estimates become available, providing the user with incrementally improving feedback.

The application of queueing network analysis can be extended to other architectural styles. The distributed message passing style was intended to mesh with the usual assumptions of the most simple queueing network model. We expect other styles to violate one or more of these; further style-specific transformation of the original architecture will be needed to produce a tractable queueing network.

In the pipe and filter style, for example, modeling systems with fan-out and fan-in presents a problem; they do not meet the assumption that jobs exist in exactly one queue or service center at a time. Fortunately, techniques exist for repeatedly collapsing subnetworks into an approximately equivalent composite queue, and for allowing a job to occupy a set of service centers simultaneously by declaring all but one "passive resources." Such systems could be approximated by applying these techniques. We believe that many other styles would be amenable to similar adaptation of our results. However, this remains an area for future research.

Earlier we made several mathematical assumptions; in particular, probability distributions are assumed to be exponential. Results in queueing theory that we have not discussed here include the analysis of queues with known nonexponential distributions. This could also be adapted to software architecture – following the same approach that we have described here – thereby allowing users to estimate the performance of systems with nonexponential distributions. Working out the details of using nonexponential distributions, however, is an item for future research.

The style in which an architecture is designed constrains the design in particular ways. We have described here only the most basic of queueing analyses; other, more powerful, queueing theory results exist and can be incorporated in the tool. Using the knowledge of a style's constraints to select appropriate queueing theory techniques, automatic transformation and analysis may be made feasible for designs in that style. Conversely, an architect's criteria for selecting a style to use may include its amenability to analysis.

8. Related Work

Two bodies of related work exist.

The first area is classical results in queueing theory. A great deal of work has been done in queueing theory, and many texts are available (e.g., Lazowska [8], Sauer [9], Jain [5]). We build on this work by applying it in a different domain and interpreting the results in the software design world. As we have noted, several issues must be resolved in order to do this.

The second area is architecture-based analysis. Architecture based static analysis is an important and growing area. Types of analyses include real-time systems in Unicon [10] and Aesop [3], component-connector protocol compatibility [2], reliability block diagrams [1], and adaptability in SAAM [6, 7]. Our adaptation of queueing network modeling adds to the repertoire of available static analysis tools, complementing the growing body of architecture-based notations and toolsets.

9. Acknowledgements

We would like to thank the developers of the Acme Library – Bob Monroe and Drew Kompanek – who provided much of the infrastructure on which our analysis tool is based. We also acknowledge the developers of the Aesop Environment, which provided the front end to the tool.

References

[1] A. Abd-Allah. Extending reliability block diagrams to software architectures. Technical Report USC-CSE-97-501, University of Southern California, Mar. 1997.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[3] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, Dec. 1994.

[4] D. Garlan, R. Monroe, and D. Wile. ACME : An architecture description interchange language. In *Proceedings of CASCON' 97*, Nov. 1997.

[5] R. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, New York, NY, 1991.

[6] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, pages 47–55, Nov. 1996.

[7] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM : A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, Sorrento, Italy, May 1994.

[8] E. D. Lazowska et al. *Quantitative system performance : Computer system analysis using queueing network models*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[9] C. H. Sauer and K. M. Chandy. *Computer systems performance modeling*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[10] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314–335, Apr. 1995.