Architecture-Based Performance Analysis

Bridget Spitznagel
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213 USA
(412) 268-8101
sprite+@cs.cmu.edu

David Garlan
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213 USA
(412) 268-5056
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.

KEYWORDS

Software architecture, software performance, queueing networks, design analysis, architecture analysis tools

1 INTRODUCTION

The software architecture of a system determines its overall structure as a collection of interacting components. Architectural designs are critical to the success of most large-scale system development efforts, because they provide a high-level view of the system that permits engineers to reason about how the key requirements of the system will be satisfied. Among these requirements, performance-related issues are often central.

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? Will the system overload or remain stable?
- Which component is the bottleneck? If the average demand increases, will it be better to upgrade or replicate the database or the web server? How does the transmission rate of the data affect performance?

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. In particular, even if good estimates of performance can be determined for the components of a system, it may be very difficult to derive overall system behavior.

Ideally what is needed is a calculus for deriving 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

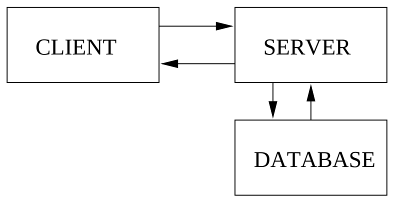

Figure 1: An internal web site

Draft. Submitted for publication September 1997

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 seems straightforward at first. 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.

That this adaptation is overly simplistic and incomplete becomes evident when it is applied to the web site example. The adaptation implicitly makes several assumptions, some of which are inappropriate for this example. 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 mismatches.

In this paper we show how to resolve these mismatches. Specifically, we demonstrate how to adapt queuing-theoretic analyses to the software architecture domain so that it can be applied to a significant class of architectural designs. We begin by briefly reviewing the elements of queueing network theory that are relevant to architectural analysis. Next we consider the straightforward application of this theory. Though powerful when it can be applied, there are a number of limitations that 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 our 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, which can have any one of the usual queueing disciplines (first come first served, round robin, last come first served preemptive resume). A service center provides some necessary service. Examples include a bank teller, hardware device, or database. Each service center has an associated queue containing jobs to be processed by that service center. A service center may also be "replicated" with degree m.

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

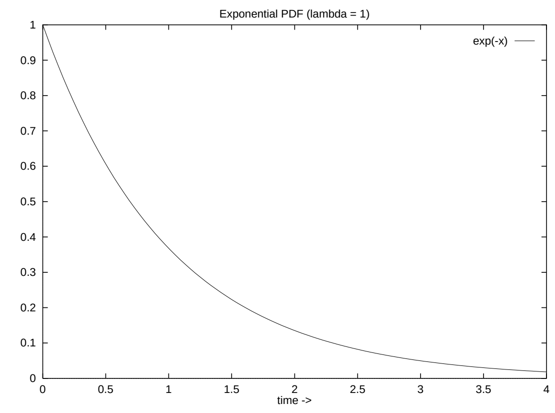

Figure 2: Exponential PDF, $\lambda = 1$

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 may have one line of customers who are processed in order by four ticket agents. An infinitely replicated service center ($m = \infty$) is also called a "delay center." These are sometimes used to model transmission delays.

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 new jobs arrive (arrival rate). The service time and the time between arrivals of new jobs 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, regardless of how much time has already passed, the expected time left to wait, $1/\lambda$, remains the same. This renders the history of the system unimportant, greatly simplifying the analysis. We will be assuming 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:

- The fraction of time the service center is occupied

[2]Sometimes the distribution is known to be non-exponential, but close enough. It should not have a larger variance, because the analysis will then overestimate performance.

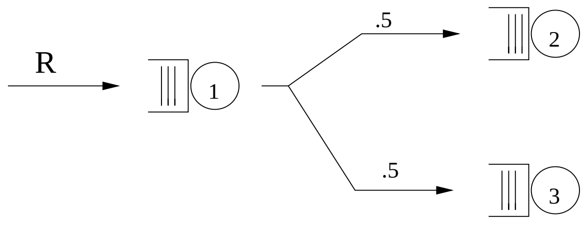

Figure 3: Queueing network

(utilization).

- 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 with the demand placed on it.
- For a queue implemented as a finite buffer with 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 from some set of the 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 rate at which jobs arrive at its queue relative to the system arrival rate (relative arrival rate). It is then possible to calculate the previous list of results for each queue in the system, and to determine 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
- The most heavily utilized service center, likely to become a bottleneck.

The system arrival rate and a queue's relative arrival rate determine the actual arrival rate seen by that queue. If known, the relative arrival rate may simply be specified; for example, in a simple pipeline all relative arrival rates are 1. Otherwise it must be derived from the probabilistic path of a job through the network. For example, jobs leaving service center S_1 may be equally likely to go on to S_2 or S_3, and so on (Figure 3). These transition probabilities must be independent of the job's history beyond the most recently visited service center. They are transformed into a set of linear equations and then solved to produce the relative arrival rates for each queue.

Some systems process more than one kind of job. To model jobs that have different behavior, each queue is divided into one or more "job classes." For the purposes of this paper, we will not be considering cases in which job class affects service time or the order in which jobs are processed; job class will affect only the subsequent path of the job. Instead of specifying transition probabilities from queue to queue, transition probabilities are now specified from job class to job class. (If each queue has only one class, this reduces to the former simpler case.)

For example, in the context of our example architecture (Figure 1), it may be that a job which was class c_1 at the web server is likely to proceed to some class in the database's queue, while a job that was class c_2 at the server will always proceed to a class in the client's queue.

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.

The simple translation is based on a "distributed message passing" architectural style. A design in this style will already be fairly close to its queueing network equivalent. The components in this style represent distributed processes, and will correspond to service centers. The connectors in this style are directional and represent asynchronous message streams; 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; as observed in section 2, other queueing disciplines would be permissible. When a component processes a message, it is assumed to produce 0 or 1 new messages as a result. A message entering the system thus corresponds to a queueing network job, until the first component finishes processing it and sends a new message to another component; then the new message corresponds to that same job. The job exists as a sequence in time of individual messages, and is completed when the sequence terminates, i.e. when the component currently processing it produces 0 new messages.

To illustrate how this can be applied, consider the following simplification of the original example architecture (Figure 1). Jobs (or messages) arrive at the client from outside the system. Each job visits the client, the server, and the database, and is finished. R jobs arrive in the system per second, so R jobs arrive in each component's queue per second. The three components have service times of S_{client}, S_{server}, and S_{db}, respectively. In the interest of automating the performance analysis, we associate a service time property with each component,

$$\begin{aligned}
u_{client} &= 9.5\,\text{jobs/s} \times 10^{-3} \times 65\,\text{ms/job} = 0.6175\\
u_{server} &= 9.5 \times 10^{-3} \times 20 = 0.19\\
u_{db} &= 9.5 \times 10^{-3} \times 103 = 0.9785\\
p_{client} &= \frac{u_{client}}{1-u_{client}} = 1.61\,\text{jobs}\\
p_{server} &= \frac{u_{server}}{1-u_{server}} = 0.235\,\text{jobs}\\
p_{db} &= \frac{u_{db}}{1-u_{db}} = 45.5\,\text{jobs}\\
P &= p_{client} + p_{server} + p_{db} = 47.3\,\text{jobs}\\
\text{Response} &= \frac{47.3\,\text{jobs}}{9.5\,\text{jobs/s}} = 4.98\,\text{s}
\end{aligned}$$

Table 1: Calculating response time

and an arrival rate property with the system.

The utilization of each component 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 the component is $p_i = u_i/(1-u_i)$, comprising jobs in the queue and jobs receiving service. The probability that $p_i \geq n$ is u_i^n. The population of the system is the total population of the components $P = p_a + p_b + p_c$, and the response time of the system is P/R.

Suppose that $R = 9.5$/second, $S_{client} = 65, S_{server} = 20, S_{db} = 103$ ms. Then we expect an average system response time of 5 seconds (Table 1), which may be acceptable to the users. The utilization of the database component is 98%. This component is close to overloaded, which means it will tend to have a long queue and therefore 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_{db} turns out to be slightly larger, the database will be unable to keep up with its expected load. We conclude that 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.

- In some designs, the set of services required by a job may actually be implicit in the structure of an architecture. However, this is not always the case. The simple translation does not attempt to handle architectures with cycles, in which a job can visit the same service center more than once. Even in an acyclic architecture, some jobs may need to visit only a subset of the components.
- There is no notion of autonomous clients. Jobs are assumed to arrive from outside at a known rate, not to be generated by one or more components of the system at rates specific to the components, 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, since the mathematical model assumes that the instances of the replicated device are not distinguishable. In addition, we must determine what it means to replicate an autonomous client.
- 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; for example, the degree to which one service center is loaded should not affect the service time of another service center, so connectors must be asynchronous.

4 EXTENDING THE MODEL

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

4.1 Cycles

The original architecture processed two kinds of jobs. The first kind ("fetch" jobs) visit the client, the server, and return to the client. The second kind ("query" jobs) visit 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. As indicated earlier, these properties are used to create a set of linear equations. Once the equations are solved for the components' relative arrival rates, the analysis proceeds as before.

Here is one way the example system might be annotated with job class information. The server can receive three kinds of messages: fetch-msg and query-msg (from the client) and query-result-msg (from the database). The client can receive start-job-msg (from outside) and end-job-msg (from the server). Suppose that 40% of jobs are fetch and the rest are query. When the client finishes processing a start-job-msg, there is a 40% probability that it sends the server fetch-msg; otherwise, it sends query-msg. When the server finishes fetch-msg or query-result-msg, it always sends the client end-job-msg. When the server finishes query-msg, it sends a message to the database; when the database finishes that message, it will send the server a query-result-msg.

4.2 Autonomous Clients

In the original architecture, jobs were initiated by the web client instead of arriving from outside the system. This system can easily be transformed to one in which jobs arrive from the outside. In a more complex system in which several components initiate jobs of various classes, this is still possible but becomes tedious to do by hand. It is preferable to associate the generation of jobs with the approriate components and add this transformation 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 generation rates of the clients, and the transition probabilities of the arriving jobs proportionally correspond to their generation.

One open question is whether generating a request should take the usual service time, or a negligible amount of time[3]. In this example, the client spends most of its time processing and displaying the information received from the server, and takes essentially no time to generate requests.

4.3 Replication

There are several obvious ways to deal with a component predicted to be a bottleneck: reduce the demand on the system, speed up the component, or replicate the component. These options are likely to vary in expense and difficulty, and may not all be reasonable for a given system. When considering these tradeoffs it is helpful to calculate what degrees of demand reduction, speedup, and replication will result in comparable or acceptable performance improvement. The benefit resulting from the first two can be readily calculated using the techniques already discussed.

For the purposes of analysis, the instances of the replicated component should be identical. It should not matter which instance services a particular job. This creates a problem: jobs processed by some systems do distinguish between instances of replicated components on subsequent visits. In the ongoing example, a job initiated by the web client returns to that client for final processing and display. If this client is to be replicated, the model will attribute the final processing to whichever client is not busy. This could result in a slightly better predicted response time for the system, but would not affect the individual performance characteristics of other components.

We must also consider the rate at which this replicated client generates requests. One possibility is to have the specified generation rate r represent the effective rate. The better option is to have the specified rate represent the rate for a single instance. Then for a component replicated m times, the effective rate is mr; if m is changed, r still remains the same.

The individual analysis for a replicated component is somewhat more complicated than for an ordinary component. It would be substantially more difficult if we did not restrict the effect of job classes to transitions only, so that within a component the analysis may be carried on as though all jobs are identical. In brief, the formulas must now include the probability that m or more jobs are already occupying the component, which is the point at which the queue begins to grow.

4.4 Delay in Connectors

In our example, the connectors are an abstraction for creating, sending, and enqueueing messages. So far this has been assumed to have no significant effect on the performance of the distributed system. In reality, transmission delays may increase the response time of the system, and should be represented.

Service centers could conceivably be used to model connectors. However, this assumes that only one message can be present in a connector at a time, and may produce inflated response times when this is not the case. Instead we choose to model a connector as a delay center. The arrival rate at this delay center is the same as the arrival rate at the service center corresponding to 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. The connector does not have a queue.

A connector's transmission delay affects the system response time: the delay time of each connector traversed by a job is added to the response time of the job. It does not affect the performance of individual components or the presence of bottlenecks. If there are multiple differing connectors between two components, it is necessary to specify in the job transition probabilities which connector will be taken so that the correct delay time is used.

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 com-

[3] Specifying a different service time for request generation is a third option, but would violate an assumption needed to keep the mathematics simple. For the purposes of this paper we will not consider this option.

ponent exactly once: each job returns to the client for final processing, and some jobs never visit the database. Replication is a permissible option for heavily utilized components. To illustrate these extensions in terms of a model, assume that the following is true for this system.

Service times are estimated as $S_{client} = 65$ ms, $S_{server} = 20$ ms, $S_{db} = 103$ ms. The delay time D_{long} of the connectors between the client and server is 1.5 seconds, and the delay time D_{short} of the connectors between the server and database is 50 ms. The client generates two kinds of jobs, "fetch" and "query." A fetch visits the client, the server, and the client again; a query visits the client, server, database, server, and client. As noted in section 4.1, at this point the architecture should be annotated with this text specification rewritten as transition probabilities; we will omit the rewrite for this example. The client generates 3 fetches/second and 7 queries/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. Here we would also use the generation rates to calculate transition probabilities for the system arrivals. Because we are assuming that generation does not require the client's usual 75 ms of service, the system arrivals will all be sent to the server; otherwise, they would be sent to the client first to receive service there. (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. The average queue length of the database 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: $17 \times 1.5 \times 2 = 51$ in transit between the client and server, one between the server and database, and the remainder waiting in queues or receiving service. (Section 4.4)

The system response time for the average job is $58/10 = 5.8$ seconds. (3 seconds of this, roughly half, are due to the delay between client and server).

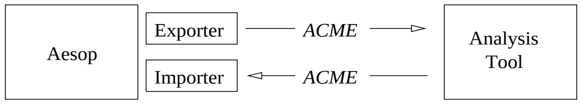

Figure 4: Performance analysis via Aesop

Now let us suppose that the database component is under consideration for upgrade or replication. 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. In order to make an informed decision, 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 in the queue and receiving service, and an average queue length of .58 messages. The second option will have 3.8 in the queue and receiving service, and a queue length of 2.2. The utilizations are 53% and 39%. The second option will be better able to handle periods of above-average load. The response times are 160 ms and 141 ms. From a performance standpoint, the second option appears to be the better choice; other factors such as expense may also be important in making 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" (or property lists), and run analysis tools on the architecture. Certain analysis results such as the expected queue length are displayed in the workshops. Other results are indicated graphically. In particular, overloaded components (i.e., components that will be unable to keep up with the anticipated demand) are highlighted.

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. An example with three Processes (a client, one overloaded server and one server replicated four times) and one MessageStream is shown in Figure 5.

The analysis tool automatically performs the transformations described above. It reads in a text file containing an architecture described in Acme [4], and outputs

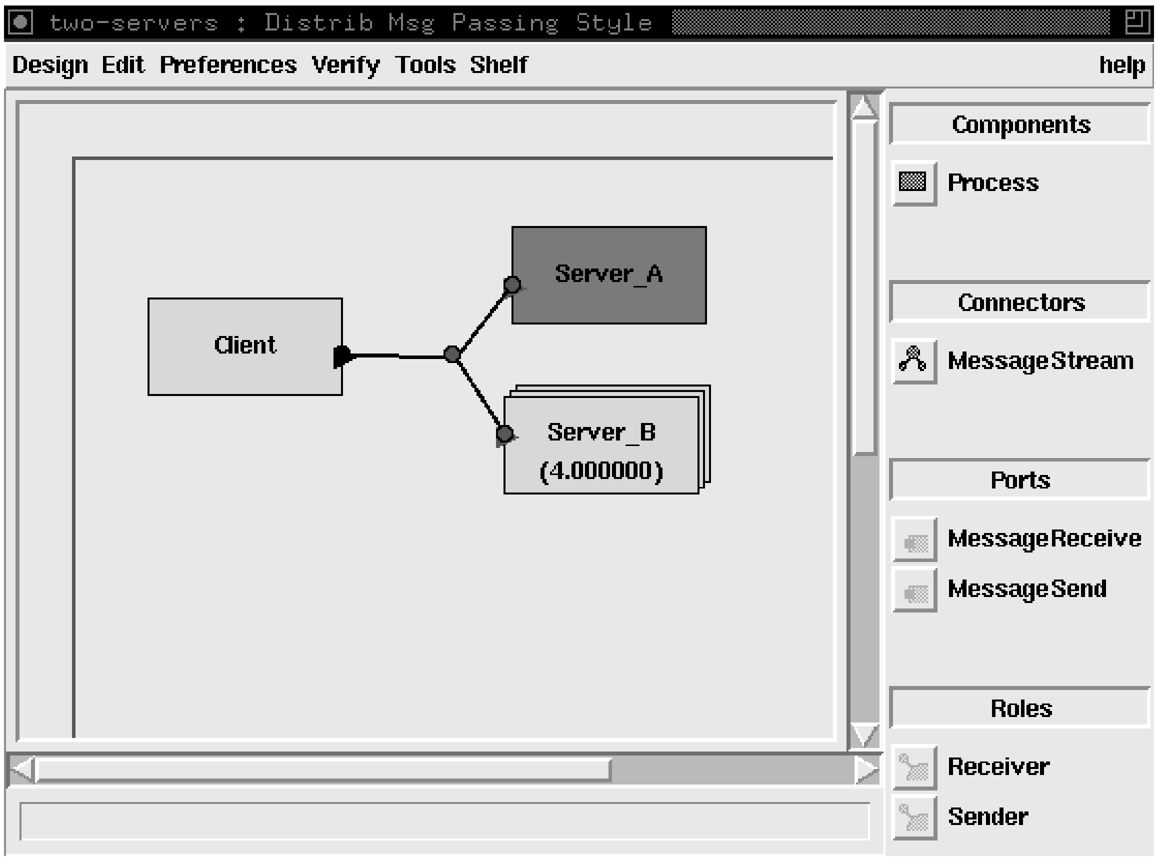

Figure 5: Aesop, distributed message passing style

an Acme description annotated with the results of the performance analysis. Aesop exports and imports these Acme descriptions (Figure 4).

In a typical scenario, the user begins the iterative design process by constructing the top level design. He (or she) 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 and provide estimates of service times at a lower level, or otherwise refine the design. After making informed modifications, the user 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. With the extensions that we have illustrated, the adaptation becomes much more useful.

Three concerns remain. The performance predictions are necessarily based on unreliable data: estimates must be supplied by the user. The application above is restricted to the distributed message passing style. The underlying mathematical assumptions further restrict the systems that can be modeled; for example, the validity of the results obtained in the example is dependent on the service time distributions being roughly exponential. Let us consider each of these in turn.

Dependence on unreliable data is unavoidable. Estimates made early in the design process cannot be completely accurate, whether they are naively guessed, based on the user's experience, or extrapolated from measurements of existing components. The initial performance analysis cannot be more accurate than its inputs. However, there is no reason to take this for the final answer. The performance analysis tool can be rerun both 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 queueing network model. We expect other styles to violate one or more of these assumptions. As a result, further style-specific transformation of the original architecture will be necessary to produce a tractable queueing network; also note that for some styles queueing network analysis may not be the most appropriate means of estimating performance.

In the pipe and filter style, for example, modeling systems with fan-out and fan-in presents a problem. Jobs are generally assumed to exist in exactly one queue or service center at a time, but in a branching pipe and filter system, this is not the case. Fortunately, techniques exist for collapsing a subnetwork into an approximately equivalent single composite queue, repeatedly if necessary, and for allowing a job to occupy a set of service centers simultaneously by declaring all but one "passive resources." Any subnetwork forming a simple pipeline can be collapsed into a composite queue. By treating some of the collapsed branches as passive resources, it would be possible to approximate non-trivial pipe and filter systems.

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, particularly concerning probability distributions of various time intervals. It is usual to assume that the distributions are exponential (i.e., within an interval, tracking the time since its start does not provide more information about how much time remains). If this is not really the case, the accuracy of the analysis described will suffer, and the performance values obtained may be more misleading than helpful. Results in queueing theory that we have not discussed here permit the (more difficult) analysis of queues with known nonexponential distributions. This additional analysis 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.

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.

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.

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.

REFERENCES

[1] A. Abd-Allah. Extending reliability block diagrams to software architectures. Technical Report USC-CSE-97-501, University of Southern California, March? 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, December 1994.

[4] D. Garlan, R. Monroe, and D. Wile. ACME : An architecture description interchange language. In *Proceedings of CASCON' 97*, November 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, November 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, April 1995.