Performance Prediction of Component- and Pattern-based Middleware for Distributed Systems

[Extended Abstract]

Shruti Gorappa
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA
sgorappa@uci.edu

ABSTRACT

Design patterns, components, and frameworks have been successfully used to build various distributed real-time, and embedded (DRE) systems such as high-performance servers, telecommunication systems, and control systems. An application developer may choose from several approaches to implement a distributed application and the choice of design patterns and their configuration can impact the overall performance of the system. Unlike components, design patterns are often descriptions of a programming approach and need to be reified for each application. However, some core pattern implementations can often be reused across applications, if they can be correctly configured to meet application requirements. Currently, there is no general way to quantify the performance of components and design pattern implementations across various dimensions such as throughput, response time, and scalability. The overall performance of applications that are composed from pre-coded components and patterns can be inferred using analytical techniques by modeling the behavior of the individual components. In particular, while we know from experience that patterns exhibit various tradeoffs in terms of performance and complexity; we would like to explore these tradeoffs in a formal way. Toward this goal, we make the following contributions in this thesis: 1) we develop analytical models for various design patterns using queuing models, 2) we present a technique to analyze the performance of combinations of design patterns as observed in real-world applications, and 3) we validate the models using empirical measurements.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Distributed systems, Performance

Keywords

Queuing models, design patterns, component middleware

1. INTRODUCTION

Component-based development is fast becoming the preferred approach to assembling systems rather than hand-coding them because of the savings in cost and time in both the enterprise [18] and DRE [28, 6] domains. While components are pre-coded software modules that provide a specific function, design patterns [2, 26] are abstract representations for software are often need to be reified with each application. The real-time Java CORBA middleware, ZEN [3], Compadres [9], and the ACE-based web server, JAWS [10, 25], are examples of frameworks that are built using middleware design patterns. From our experience building DRE middleware, we have found that component- and pattern-based middleware improves the quality and maintainability of software and reduces application development time.

DRE middleware has stringent performance requirements: web servers need to meet requirements such as utilization, throughput and response times, and control systems for industrial machinery need to meet requirements for predictability and latency. Core middleware patterns and components[1] greatly contribute towards the scalability and performance of distributed applications. Frequently, performance tests for these metrics are only performed after the application has been built, when it may be expensive and time-consuming to change bad design decisions. As component and framework-based software development becomes more popular, so does the need for a-priori or design-time performance modeling of software.

Performance modeling of computer systems software has been examined for many years now. In general, there are two main approaches to gauging the performance of software– simulation and analysis. Both approaches involve developing models of the software. However, simulation-based approaches can be expensive in terms of time and resources. On the other hand, analytical approaches have been shown to produce results close to the simulation results [11] and do not consume as much time and resources as the simulations.

[1]In our paper, "components" refer to pre-coded software modules that may be assembled and reused in general, and not to specific component frameworks.

In this thesis, we present an analytical model for the design-time performance prediction of custom middleware composed using patterns and components. This model will allow developers to predict and visualize the performance of their code at design-time, thus allowing them to eliminate software bottlenecks. Design-time performance prediction can allow developers to model the performance of their software for different usage loads, therefore allowing them to provision the framework for the scale of the application. In this work, we use UML diagrams, source code, and sequence diagrams to develop the queuing network models (QNMs) for server-side design patterns and components (modules). We introduce the concept of a *performance profile* for components, which is a visual representation of the performance metrics of the component and will allow users to choose the appropriate component for their application from a set.

2. RELATED WORK

Analytical methods for performance prediction of software systems has been an active area of research but we only summarize a few approaches related to distributed systems. Ramani et. al. [20] present the use of Stochastic Reward Nets (SRNs) to study and configure the CORBA Event Service. The Layered Queuing Network model [12] has been developed to analyze the scalability of distributed systems. A layered queuing model with CPU sharing is presented in [21] and is used to analyze multi-level client/server systems for synchronous and asynchronous messages. A performance analysis of the Reactor pattern using an SRN model is presented in [13]. One problem with this approach is dealing with the increase in the number of variations of the model with the increase in options such as threads and queue size. In later work, a model-driven tool has been proposed to overcome this limitation [14]. Analytical approaches have been used to predict the performance of JMS [17] and J2EE applications [16]. A queuing network model for ASP.NET is presented in [1]. A model to analyze the performance of a web service under varying client workloads and protocol and server configurations is presented in [27]. An analysis of e-commerce services by mapping a consumer behavior model graph to a discrete time markov model is presented in [7]. While these works focus on modeling the performance of web servers in general, our research focuses on modeling the performance of core communication framework patterns and components.

3. RESEARCH GOALS

The goal of this research is to model the performance of individual QNMs so that they may be used to predict the performance of composite distributed system software in terms of throughput, latency, and processor utilization. Throughput allows the users to predict the scalability in terms of number of users and rate of requests that the software can safely handle. Latency is of interest in soft real-time applications where response time needs to be within a certain value. Utilization is of interest when the designer needs to identify the bottlenecks in the system. Performance prediction of component- and pattern-based middleware provides the following advantages:

- The analytical performance prediction approach will help us predict the performance of software that is composed from individual components and patterns. DRE systems are increasingly built using component frameworks; a tool that models the performance and scalability of these systems can prove to be extremely useful.
- Multiple implementations of a functional component may be available for use in a particular situation. Comparing the performance of different pattern implementations and components will allow developers to choose the right components for the application.
- The design-time performance evaluation will help prevent costly mistakes by identifying performance issues at the early stage of design rather than during testing.
- Analyzing the utilization at various modules will help identify software bottlenecks and sources of delay so that they may be provisioned with the appropriate computing power and threading models.

The following research challenges are addressed in order to achieve the goals:

- **Identifying the queuing model of the individual components:** By modeling patterns and components using queuing models, we can allow users to generate performance profile of component implementations at design time to visually compare the performance of two or more implementations.
- **Defining a mechanism for modeling the performance of a system composed from components:** Composed systems often contain multiple components and will need to be modeled as a network of queues. We use a layered queuing network approach to model and solve the distributed system.
- **Providing design-time user information:** It will be valuable to present insights obtained from the compositional analysis to the user so that they may use this information to optimize their systems.

4. ANALYTICAL MODELING APPROACH

This component and pattern (henceforth module) profiling methodology is illustrated in Fig. 1 and the steps are described in this section. The process involves two stages: 1) Identifying individual modules that affect the performance of middleware and developing their performance models, and 2) Analyzing how these modules combine architecturally and modeling the performance of the resulting software. This section explains the steps involved in the analytical modeling approach.

1. **Developing Queuing Network Models:** As one of the goals of this research is to enable the performance comparison of pre-coded components and patterns, we first identify, model, and validate the individual pattern implementations. On examining the code and sequence diagrams of some building block components and pattern implementations, we have found that they can be modeled analytically. Analytical modeling allows us to predict the performance metrics such

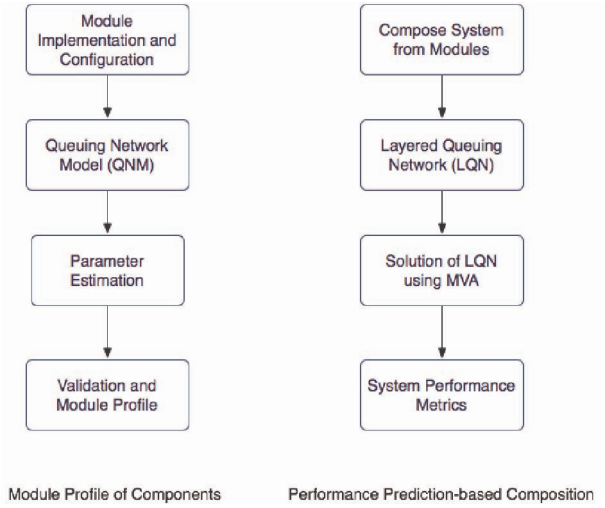

Figure 1: The Performance Profiling Methodology

as throughput. latency, and processor utilization analytically. We make the following assumptions in our models: [2]

- The execution rate for tasks is assumed to have a Poisson distribution with a mean of μ.
- The arrival rates for tasks is assumed to be Poisson with a mean of λ.

The tasks in the systems can be events or method calls and are modeled as jobs and are executed in server modules, which may be single or multi threaded.

As an example to illustrate our process, we choose the Reactor pattern, which is used to decouple event I/O from event handling. Events arrive at the server node, and the server objects provide handlers that may be invoked in order to process the incoming events. These event handlers are registered with the Reactor. When an event arrives, the Reactor thread reads and demultiplexes the event to the appropriate event handler. This may be implemented by using the `select()` method in C++, or the `Selector` class in Java. The interaction diagram of the Reactor pattern is illustrated in Figure 2 (from [22]). The Reactor thread can only process one event at a time. In large applications, however, multiple Reactors may be implemented to handle events from different sockets, so that these events may be processed in parallel [19]. In order to model the Reactor pattern, we make the following assumptions: the packet arrivals are markovian with mean λ; the processing times of the packets follow a Poisson distribution with mean μ. There is only one server (Reactor thread) processing the events, and the queue size is K. Hence, the Reactor may be modeled as an

[2]These assumptions are reasonable and are frequently used to model distributed systems since they form the basis for many queuing network results. These assumptions are further justified in our study since the models are used for performance comparison, with all models being based on the same assumptions.

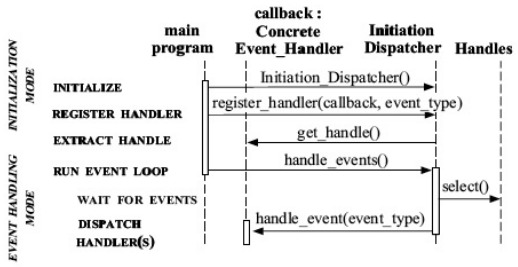

Figure 2: The Interaction Diagram of the Reactor Pattern.

M/M/1/K queue. Once the number of events in the system reaches K, events are dropped, or blocked. The model is illustrated in Fig. 5. The metrics we modeled are:

- Throughput (γ): the number of requests serviced per unit time,
- Latency ($E[T]$): the time taken to service the request, and

For the model of the Reactor pattern, we can obtain the equations for the three metric from literature [15]. The throughput γ is given by

$$\gamma = \lambda(1 - P_k) = \frac{\lambda(1 - \rho^K)}{1 - \rho^{K+1}}$$

The average number of customers in the system is given by

$$E[N] = \begin{cases} \frac{\rho}{1-\rho} - \frac{(K+1)\rho^{K+1}}{1-\rho^{K+1}}, & \text{for } \rho \neq 1 \\ \frac{K}{2}, & \text{for } \rho = 1 \end{cases}$$

Using Little's law, we can find the average waiting time or latency for an event in the system using

$$E[T] = \frac{E[N]}{\lambda(1 - P_k)}$$

We have also developed the QNM for the Leader/Follower pattern [24], which is an efficient multi-threaded event handling pattern. Details of the model may be found in [8].

Since our methodology is based on models, it is important that the implementations of the design pattern be consistent with its sequence diagram. Although design patterns are abstract representations, we analyse software implementations of design patterns so that we model the performance of the actual code. The implementations of components and design patterns may use different technologies or languages; empirical measurements are used to validate the implementations and obtain their parameter values. These parameter values are used as inputs to the model of composite systems that are built using the module.

2. **Parameter Estimation:** Once the analytical model of the pattern is developed, we identify its parameters,

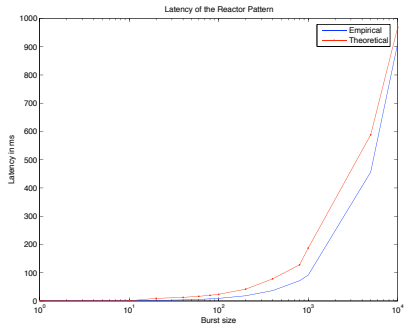

Figure 3: Latency comparison of the Reactor pattern

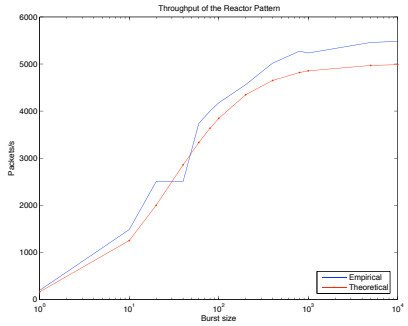

Figure 4: Throughput comparison of the Reactor pattern

the execution time and queue length. In order to compute μ and K, we empirically measure the metrics of throughput and latency and use the method of mean squares to find the analytical parameters. We use ACE as our testbed since it is based on design patterns and is a popular framework for building web applications. This exercise only needs to be performed once and the parameters may be later used in the network models for performance prediction.

3. **Validation of the QNMs:**

   This section presents a comparison of the results obtained by evaluating the patterns implemented in the ACE framework [10] using the analytical model presented in the paper. The experiments were run with the following parameters. The packet size was 1000 bytes. The number of messages sent in each trial of the experiment was 10000. The packets were sent in bursts of varying size with a time interval between packet bursts. The parameter values for the Reactor pattern were found to be $K = 400$ and $\mu = 98$ respectively .

   Fig. 3 and Fig. 4 show the comparison of the latency and throughput of the Reactor pattern obtained from the experiment versus the theoretical values obtained from the queuing model. It can be seen that both the theoretical and experimental values follow a similar trend, and hence the model may be used as a good approximation to describe the performance of the pattern. When the burst size, and hence λ, is low, the latency is also low. As λ increases, the latency increases. This is because as the number of packets that are being queued up is increasing, packets spend more time in the queue waiting to be processed. In the throughput curve, the number of packets exiting the system increases with λ and then levels off. This is because the queue is full and some incoming packets are being blocked.

4. **Composing DRE Frameworks:** One of the goals of our research is to model and analyze the performance of DRE middleware frameworks composed from modules. Toward this goal, we have identified components and patterns from middleware frameworks such as Zen and JAWS as examples to illustrate our methodology. We have classified the modules into the following three groups based on their architectural functionality as illustrated in Fig. 5:

   (a) **Network Management:** This group of patterns belong to the layer responsible for connection establishment and maintenance in distributed systems. Examples are the Acceptor/Connector and Proxy patterns.

   (b) **Protocol Handling and Event Demultiplexing:** These components process the packet headers and classify the packets based on their header or request type. The event dispatcher and protocol pipeline patterns are two examples of patterns that may be used in this layer.

   (c) **Request Handling:** These components handle the request processing at the server and include the Proactor, Leader/Follower, and Reactor patterns. These patterns may in turn be composed of smaller components depending on the architecture at the server.

   Each of these modules can in-turn be modeled as queues and so that the composite pattern is a network of queues.

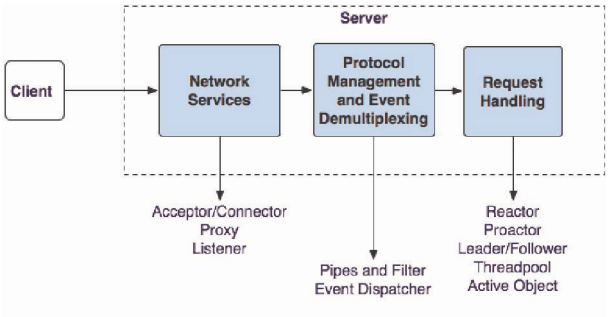

Figure 5: Classification of server patterns

5. **Solving the network of queues using MVA:** In order to compute the throughput or latency of a patterns that contain more than one component, such as the Half Sync/Half Async, we need to analyze it as a network of queues (Jackson network). This is because each event will traverse through multiple queues in series. In these cases, it is often not possible to formulate

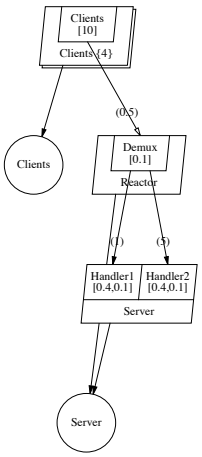

Figure 6: The LQN model of the Half Sync/Half Async pattern

simple analytical expressions to model the patterns. However, solutions for Jackson networks exist in the form of mean-value analysis (MVA) algorithms [11].

We have identified the Layered Queuing Network Solver (LQNS) [5] as a suitable used tool for modeling composable systems. The LQN modeler allows us to model different types of method calls: synchronous, asynchronous, and request forwarding. It allows us to specify service time in *phases*, which is useful while modeling event handlers and callback functions. Implementation specific details such as host CPUs, number of threads, and service times can be input into the model. Further details on the LQN model and semantics may be found in the manual [4] and the discussion is omitted here due to space restrictions.

The Reactor pattern is often used as part of a composite pattern such as the Half Sync/Half Async pattern [23]. The event queue receives requests asynchronously from the network. The Reactor thread is in charge of dequeuing these events and demultiplexing them to the appropriate event handler. The event handlers process the event and return the results, and may be configured as single or multithreaded based on the application requirements. An LQN representation of this pattern is illustrated in Fig. 6. The parameters such as service time and queue length can be obtained from the QNMs in phase 1, and through running profiling tools.

5. SUMMARY AND FUTURE WORK

The design-time performance prediction of software is a valuable tool for developers of DRE applications and frameworks. Toward this goal, we have defined a methodology for performance prediction based on analytical QNM analysis. Currently, we have modeled individual server-side patterns as queuing networks and identified their configuration parameters. Although the main focus of this work is middleware frameworks, the methodology may be extended to model component-based applications in general. We have identified tools and techniques that enable us to model composite patterns as queuing networks and solve them to obtain performance metrics such as end-to-end latency and throughput. The next steps in this research are the following:

1. **Analysis of Tradeoffs:** Modeling the performance of patterns allows users to observe the tradeoffs in different dimensions of performance. The scalability of components may be influenced by factors such as message size, arrival rate, service time, and threading models. For example, although the Reactor pattern has smaller latency than the Leader/Follower pattern, it has smaller throughput and does not scale as well as the latter. However, the Reactor may be configured with a threadpool to improve scalability. It will be valuable to provide this information to the users in the form of an analytical model at design time so that they can visualize the implications of their design decisions on performance and help them make better choices of components based on application constraints and requirements.

2. **Additional Components and Patterns:** We are developing modular implementations of framework components and patterns and developing their analytical models. Although the ACE framework served as a testbed for the validation of QNMs, it may not be the ideal testbed for validating LQNs because 1) it is not strictly modular and 2) the QNM parameters cannot be easily configured. Hence, we are implementing a new framework that is modular and that can be parameterized for the LQN validation.

3. **Composition and Module Reusability:** A key advantage of model-driven development is reusability of software in terms of both design and code. We are building examples of distributed server architectures from modules described in Fig. 5 to demonstrate this feature. For example, web servers that may be built from the following modules: a) Acceptor/Connector, b) Leader/Follower, and c) Threadpool, and publish/subscribe systems from the following modules: a) Proxy, b) Pipes/Filters, and c) Proactor. Further, we are developing the comparitive performance models for these architectures and validating them experimentally.

6. REFERENCES

[1] A. Bogárdi-Mészöly, T. Levendovszky, and H. C. (Hungary). Using queueing model in predicting the response time of asp.net web applications. In *The IASTED Conference on Software Engineering*, February 2006.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York, 1996.

[3] Center for Distributed Object Computing. The ZEN ORB. www.zen.uci.edu, University of California at Irvine.

[4] R. G. Franks. *The Layered Queueing Network Tutorial*.

[5] R. G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Carleton University, Ottawa, Ontario, Canada, December 1999.

[6] T. Gensler and C. Zeidler. Rule-driven component composition for embedded systems. In *4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, 2004.

[7] S. S. Gokhale and J. Lu. Performance and availability analysis of an e-commerce site. In *COMPSAC (1)*, pages 495–502, 2006.

[8] S. Gorappa and R. Klefstad. Modeling the performance of communication framework design patterns using queuing theory. In *Proceedings of the 10th International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, July 2007.

[9] J. Hu, S. Gorappa, J. A. Colmenares, and R. Klefstad. Compadres: A Lightweight Real-time Java Component Middleware Framework for Composing Distributed, Real-time, Embedded Systems. In *Middleware*, November 2007.

[10] S. D. Huston, J. C. E. Johnson, and U. Syyid. *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[11] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[12] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):589–603, 2000.

[13] A. Kogekar and A. Gokhale. Performance evaluation of the reactor pattern using the omnet++ simulator. In *ACM-SE 44: Proceedings of the 44th annual southeast regional conference*, pages 708–713, New York, NY, USA, 2006. ACM Press.

[14] A. Kogekar, D. Kaul, A. Gokhale, P. Vandal, U. Praphamontripong, S. Gokhale, J. Zhang, Y. Lin, and J. Gray. Model-driven generative techniques for scalable performability analysis of distributed systems. In *Parallel and Distributed Processing Symposium*, April 2006.

[15] A. Leon-Garcia. *Probability and Random Processes For EE's (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2007.

[16] Y. Liu and I. Gorton. Performance prediction of j2ee applications using messaging protocols. In *CBSE*, pages 1–16, 2005.

[17] M. Menth and R. Henjes. Analysis of the message waiting time for the fioranomq jms server. *icdcs*, 00:1, 2006.

[18] S. Microsystems. Enterprise JavaBeans specification, v2.1.

[19] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang, V. Kachroo, and A. Gokhale. Applying optimization principle patterns to real-time orbs, 2000.

[20] S. Ramani, K. S. Trivedi, and B. Dasarathy. Performance analysis of the corba event service using stochastic reward nets. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 238, Washington, DC, USA, 2000. IEEE Computer Society.

[21] S. Ramesh and H. G. Perros. A multilayer client-server queueing network model with synchronous and asynchronous messages. *IEEE Transactions on Software Engineering*, 26(11):1086–1100, 2000.

[22] D. C. Schmidt. *Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching*, pages 529–545. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[23] D. C. Schmidt and C. D. Cranor. Half-sync/half-async: an architectural pattern for efficient and well-structured concurrent i/o. In *Pattern languages of program design 2*, pages 437–459. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[24] D. C. Schmidt and et al. Leader/Followers - A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching. In *7th Pattern Languages of Programs Conference*, 2000.

[25] D. C. Schmidt and J. C. Hu. Developing flexible and high-performance Web servers with frameworks and patterns. *ACM Computing Surveys*, 32(1):39–39, 2000.

[26] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[27] R. D. van der Mei, R. Hariharan, and P. Reeser. Web server performance modeling. *Telecommunication Systems*, 16(3-4):361–378, 2001.

[28] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. Rodrigues. Qos-enabled middleware (chapter 6). In *Middleware for Communications*. John Wiley & Sons, 2004.