# Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architectures

**Hassan Gomaa**
**Dept. of Information and Software Engineering**

**Daniel A. Menascé**
**Dept.of Computer Science**

**School of Information Technology and Engineering**
**George Mason University**
**4400 University Drive, Fairfax, VA 220300-4444, USA**

**hgomaa@gmu.edu**
**+1-703-993-1652**

**menasce@cs.gmu.edu**
**+1-703-993-1537**

## ABSTRACT

The ability to estimate the future performance of a large and complex distributed software system at design time, and iteratively refine these estimates at development time, can significantly reduce overall software cost and risk. This paper investigates component interconnection in client/server systems, in particular the design and performance modeling of component interconnection patterns, which define and encapsulate the way client and server components communicate with each other. We start with UML design models of the component interconnection patterns. These designs are performance annotated using an XML-type notation. The performance-annotated UML design model is mapped to a performance model, which allows us to analyze the performance of the software architecture executing on various system configurations.

Keywords: component interconnection patterns, software architecture, XML, UML, performance model, queuing networks.

## 1. INTRODUCTION

When designing the software architecture of a complex distributed software system, an important consideration is the performance of the resulting system. The ability to estimate the future performance of a large and complex distributed software system at design time, and iteratively refine these estimates at development time, can significantly reduce overall software cost and risk. In previous papers, the authors have described the design and performance analysis of client/server systems [7,18,19,20].

This paper investigates component interconnection in client/server systems, in particular the design and performance modeling of component interconnection patterns, which define and encapsulate the way client and server components communicate with each other. This paper describes the component interconnection patterns of synchronous and asynchronous communication with a multi-threaded server.

The approach we have taken is to start with UML design models of the component interconnection patterns. We then provide performance annotations of the UML design models using an XML-based notation. The performance-annotated UML design model is mapped to a performance model, which allows us to analyze the performance of the software architecture executing on various system configurations.

The rest of the paper is organized as follows. Section 2 provides the basic background on software architecture and describes our approach. Section 3 describes the UML interconnection patterns. Section 4 describes the performance annotations of the UML-described architectures. Section 5 describes the performance models of the component interconnection patterns and section 6 provides concluding remarks and points to future work.

## 2. SOFTWARE ARCHITECTURE AND PERFORMANCE

### 2.1 Software Architecture

The software architecture of a system is the structure of the system, which consists of software components, the externally visible properties of those components, and the interconnections among them [3,27]. Software components and their interactions can be formally specified using an architecture description language [1,2,12,21,26,27,28] to describe the software architecture. Architecture description

WOSP 2000, Ontario, Canada
© ACM 2000 1-58113-195-X/00/09 ...$5.00

languages (ADLs) separate the description of the overall system structure in terms of components and their interconnections from the description of the internal details of the individual components.

A *component* is defined in terms of its interface, including the operations it provides and requires. A *connector* encapsulates the interconnection protocol between two or more components.

At this stage, most ADLs are research languages. Althought there is a general agreement about the basic concepts of an ADL, there are significant differences among the various ADLs. Some ADLs, such as C2 [29,31], UniCon [28] and Wright [2], support connectors explicitly, while others, e.g., Rapide [12] and Darwin [21], specify them as connections (functions provided or required) between components.

The Unified Modeling Language (UML) is a graphically-based object-oriented notation developed by the Object Management Group as a standard means of describing software designs [4,25], which is gaining widespread acceptance in the software industry. There are efforts to provide a bridge from UML to software architectures [9]. There have been some attempts to integrate ADLs with the Unified Modeling Language (UML) [24]. There have also been investigations on how software architectures can be described using the UML notation [10]. Currently, UML lacks formal semantics for architecture description and for performance modeling.

Analyzing the performance of a software architecture allows a quantifiable tradeoff analysis with the objective of minimizing risks in software designs. An example of performing architecture tradeoff analysis is given in [11]. Other approaches based on analytic performance models are described in [7,18,19,20].

### 2.2 Component Interconnection Patterns

A *design pattern* describes a recurring design problem to be solved, a solution to the problem, and the context in which that solution works [5,6]. The description is in terms of communicating objects and classes that are customized to solve a general design problem in a particular context. In this paper, a *component interconnection pattern* defines and encapsulates the way client and server components communicate with each other via connectors. In distributed applications, *component interconnection patterns* are needed to support the different types inter-component communication, including synchronous, asynchronous, brokered, and group communication [7,8,9,22,23,30].

In the UML notation, the class diagram is used to depict the static view of interconnection patterns, whereas the collaboration diagram depicts the dynamic view. This paper describes component interconnection patterns for synchronous and asynchronous communication [9].

### 2.3 Approach

The approach described in this paper is illustrated in Figure 1. A distributed software architecture is composed of a set of components and a set of connectors that can be used to connect the components. In this paper, components are considered black boxes that interact with other components through connectors. Components send and receive messages to and from other components using the protocols specified by the connectors.

The approach described in this paper is to start with a UML design model of the component interconnection pattern. We then provide a performance annotation of the UML design model using an XML type notation. We then map the performance annotated UML design model to a performance model to analyze the performance of the architecture.

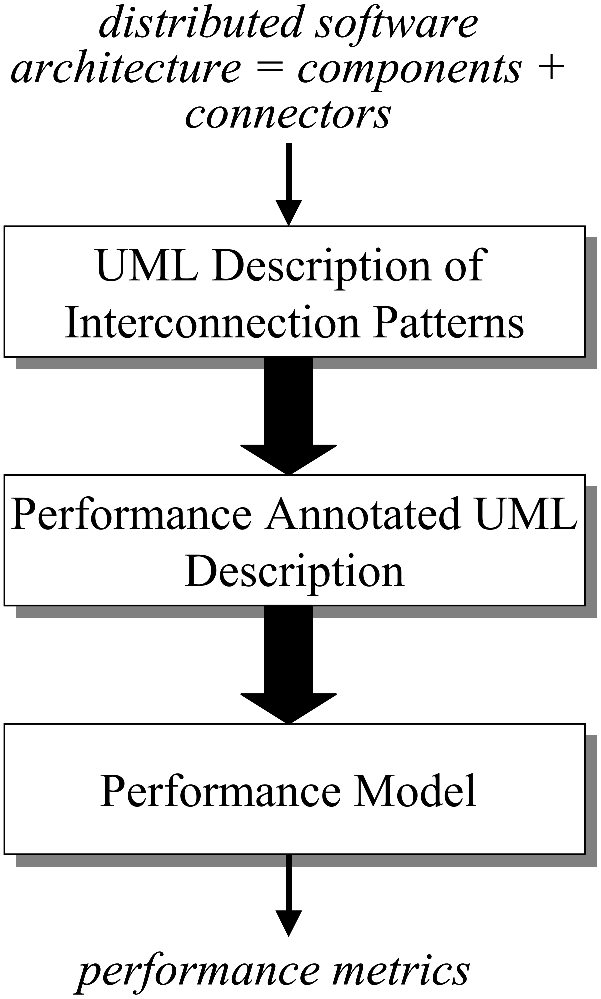
distributed software architecture = components + connectors
UML Description of Interconnection Patterns
Performance Annotated UML Description
Performance Model
performance metrics

**Figure 1 – Design and Performance Modeling Approach**

## 3. SPECIFYING COMPONENT INTERCONNECTION PATTERNS WITH UML

### 3.1 Static Modeling of Component Interconnection Patterns

The UML class diagrams, which depict the static modeling of the component interconnection patterns, use stereotypes to differentiate among the different kinds of component and connector classes. In UML, a **stereotype** is a subclass of an existing modeling element, which is used to represent a usage distinction [25], in this case the kind of component or connector class. A stereotype is depicted using guillemets, e.g., <<Component>>. For this work, the stereotypes introduced are <<Component>>, <<Connector>>, and <<Resource>>.

Figure 2(a) depicts a class diagram for a client/server system. A **component** is an application class that is defined in terms of its interface, which is visible to other components, and its implementation, which is hidden from other components. In this example, the client and server classes are depicted using the stereotype <<component>>. A **connector** hides the details of the interaction between components. To model the performance of a distributed software architecture, it is useful to consider a connector in terms of a client connector that communicates with a server connector via a resource. A **resource** is used to depict a part of the communication infrastructure that supports the interconnection among components or connectors and is introduced to facilitate the mapping to performance models as the presence of a resource needs to be explicitly modeled. In these patterns, the Network is a <<resource>>. Each server component has a logical 1-many association with client components. Physically, however, each client component has a 1-1 association with a client connector and each server component has a 1-1 association with a server connector. The network resource has a 1-many association with the client and server connectors. Optionally, there may also be one or more brokers. A Broker is modeled as a component class. There may be 0,1, or more brokers in a client/server system.

Figure 2(b) shows the Client Connector, which is specialized into a Synchronous Client Connector, an Asynchronous Client Connector (which is a composite class, composed of a Client Message Input Buffer class, a Client Call Stub class and a Client Return Stub class) and a Brokered Client Connector. In Figure 2(c), the Server Connector is specialized into a Single Threaded Server Connector or Multi-Threaded Server Connector. The former is composed of a Single Threaded Server Message Buffer class and a Single Threaded Server Stub class. The Multi-Threaded Server Connector is composed of a Multi-Threaded Server Message Buffer class, a Multi-Threaded Server Stub class, a Dispatcher Buffer (all 1-1 relationships), and a Server Return Stub (1-n relationship). Components are also classified, a Client component is Synchronous or Asynchronous and a Server component is Single-Threaded or Multi-Threaded. A Multi-Threaded Server component class is composed of a Dispatcher class (1-1 relationship) and a Worker Server class (1-n relationship).

### 3.2 Dynamic Modeling of Component Interconnection Patterns

UML collaboration diagrams are used to depict the dynamic interactions between the component and connector objects, i.e., instances of the classes depicted on the class diagrams. An active object has its own thread of control and executes concurrently with other objects. This is in contrast to a passive object, which does not have a thread of control. In these patterns, all components are active apart from the message buffers, which are passive monitor objects [30]. Messages are numbered on the collaboration diagrams to show the sequence of occurrence.

Figure 3 depicts a collaboration diagram for the simplest form of client/server communication, which is a Synchronous Client component communicating with a Single Threaded Server component using static binding. The Synchronous Client component has a Synchronous Client Connector, which hides from it the details of the interconnection with the Server. The Single Threaded Server component has a Single Threaded Server Connector, which hides from it the details of the interconnection with the client.

In Figure 3, the Client Connector acts as a client stub performing the marshalling of messages and unmarshalling of responses [22]. Thus it receives a client message, packs the message, and sends the packed message to the Server Connector (single threaded or multi-threaded). The Client Connector also unpacks the server responses for delivery to the Client component.

Figure 4 shows the more complex case of a Multi-threaded Server. Both the Multi-Threaded Server component and the Multi-Threaded Server Connector are composite objects. The clients could be either synchronous or asynchronous, as described next, and the binding could be static or dynamic.

The Server Connector receives the packed message and stores it in a message buffer. From there, the Server Stub (Single Threaded or Multi-threaded) unpacks the message. In the case of the Single Threaded Server Stub (Fig. 3), it delivers the message to the Single Threaded Server component and waits for the response. In the case of the Multi-threaded Server Stub (Fig. 4), it places the message in a Dispatcher Buffer and proceeds to unpack the next incoming message. The Dispatcher receives the message from the Dispatcher Buffer and dispatches the message to an available Worker Server. Alternatively, the Dispatcher instantiates a Worker Server to handle the request. The Worker Server services the request and sends the response to a Server Return Stub, which packs the response and forwards it to the Client Connector.

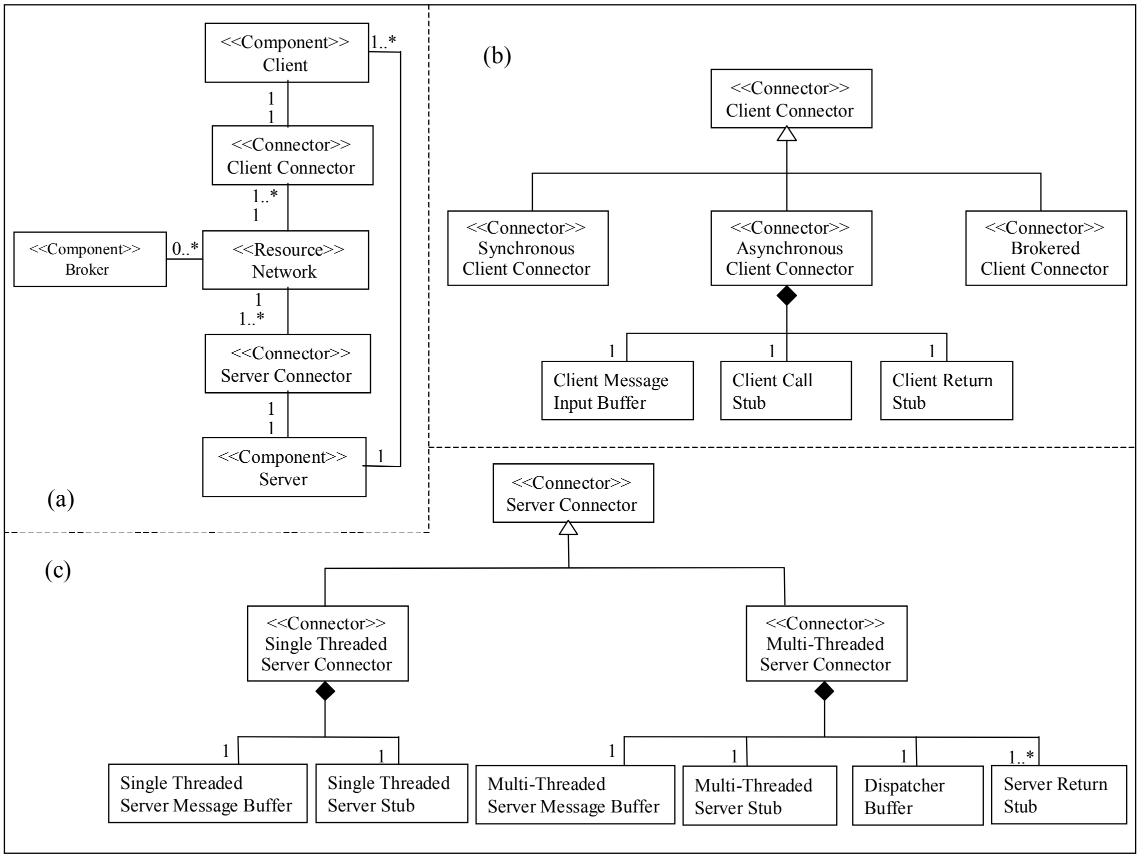

**Figure 2 - Class Diagram for Client/Server System Connectors**

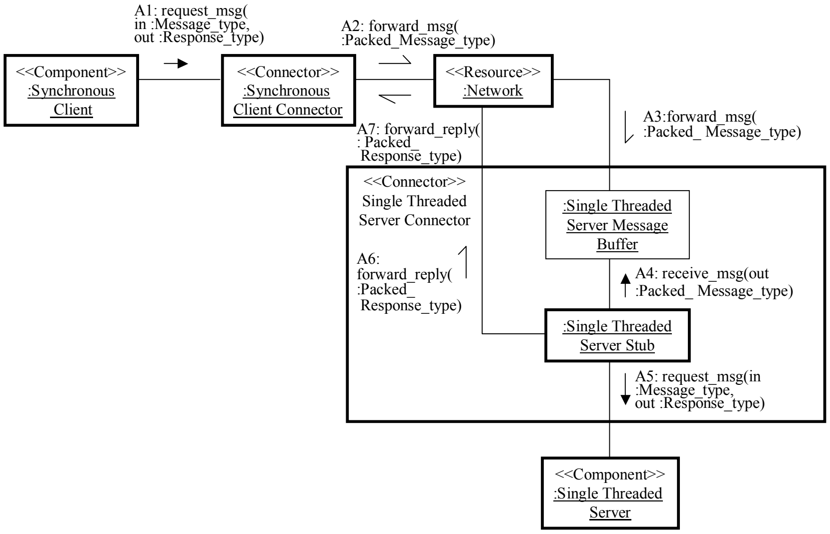

**Figure 3 – Collaboration Diagram for Synchronous Client and Single Threaded Server**

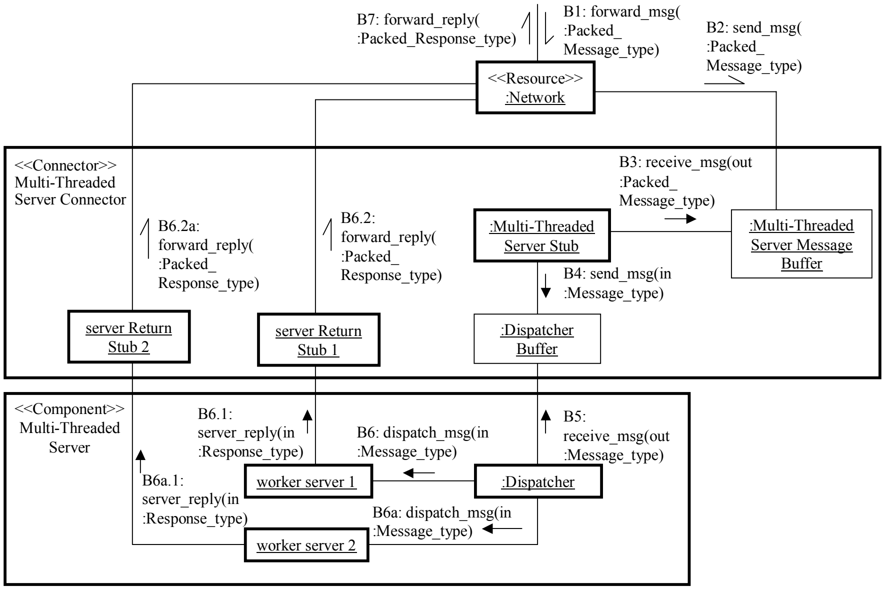

**Figure 4 – Collaboration Diagram for Multi Threaded Server Connector**

Fig. 5 shows the case of an Asynchronous Client component, which needs an Asynchronous Client Connector. The Asynchronous Client Connector is more complex than the Synchronous Client Connector as it has to support a call-back capability, whereby the Asynchronous Client component can send a message, continue processing, and later receive a server response. Thus whereas, the Synchronous Client Connector (Fig. 3) sends the message to the server and waits for the response, the Asynchronous Client Connector (Fig. 4) is capable of sending multiple client messages before a server response is received. To handle this, the Asynchronous Client Connector has two concurrent stub objects, a Client Call Stub for handling outgoing messages and a Client Return Stub for handling incoming responses. Responses are buffered in the Client Message Input Buffer, where they can be received as desired by the Asynchronous Client component.

## 4. PERFORMANCE ANNOTATIONS OF UML-DESCRIBED ARCHITECTURES

This section describes the performance annotation of software architectures depicted in UML notation. The performance parameters needed to characterize the performance of the objects in UML-described architectures are given. It includes the performance annotations for components, connectors, and resources. The UML models are mapped to XML, which allows us to capture both the architecture and performance parameters in one notation.

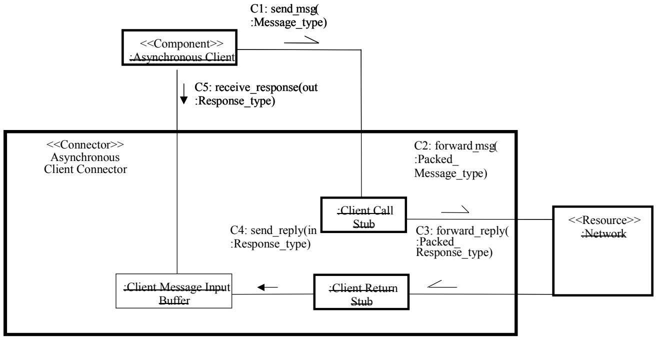

**Figure 5 – Collaboration Diagram for Asynchronous Client Connector**

### 4.1. Performance Parameters

The performance of distributed systems can be analyzed through analytic [14] or simulation models. The parameters required for these models fall into two categories:

a) Workload intensity parameters. This type of parameter measures the load placed on the various system resources. This can be measured in terms of arrival rates (e.g., requests/sec) or by the number of concurrent entities (e.g., clients) requesting work along with the rate at which each client requests work.

b) Service demand parameters. These parameters indicate the amount of each resource used on the average by each type of request. Service demands can be measured in time units or in other units from which time can be derived. For example, the service demand on a communication link can be expressed by the size of a message in bits, provided we know the link capacity in bps.

The performance parameters for a component deal with two types of interface, the **port** and the **entry**. A **port** supports one-way communication. A message is sent on an output port and received on an input port. Message communication may be asynchronous or synchronous without response, as shown in Figs. 3-5. An entry supports two-way message communication, i.e., synchronous message communication with response, as shown in Figs. 3-4. Fig. 6 shows the UML objects refined to explicitly depict the input and output ports, using the Darwin notation [21], although the ROOM [26] notation could also be used. The connector shown in Fig. 6 corresponds to the client connector – resource – server connector shown in Figs. 3-5.

Figure 6 illustrates two interconnected components or objects with their input ports (i.e., places where messages arrive) and their output ports (i.e., places where messages are sent to input ports of other components). Messages can arrive at an input port from another component via a connector or from outside the system. In this case, one must specify the message arrival rate. In the example of Fig. 6, port *a* of Object A receives messages from the outside world at a rate of 5 messages/sec.

The figure also illustrates what happens when a message is received at an input port. For example, when a message arrives at port *b* in Object A, there is a 0.3 probability that a message will be generated at port *d* and a 0.7 probability that a message will be generated at output port *e*. In the former case, it takes an average of 1 sec to process the incoming message and generate the outgoing message. In the latter case, it takes 3 sec. The figure also shows the message sizes in bytes.

The type of connector used to interconnect components influences the performance of component based systems.

The following performance parameters are important when describing connectors:

- Client and server stub times. This is the time needed by the client and server stubs to carry out tasks such as parameter marshaling, parameter conversion, and message generation.
- Number of concurrent server threads.
- Maximum number of connections to be accepted by the server.

The following parameters are important when specifying a network resource:

- Bandwidth: defines the rate at which bits are transmitted over the network.
- Latency: time it takes a bit to propagate across the network.

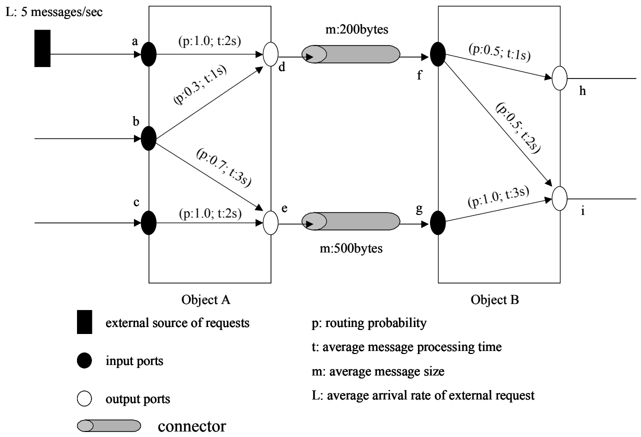

**Figure 6 – Interconnected Components**

### 4.2 Performance Annotations of Components

From a performance perspective, an input port interface is classified into two parts: an incoming message part and an outgoing message part, which is sent via an output port. Each incoming message will require some CPU time to process it and will optionally result in an output message of a given size being generated, which is then sent over a specified output port. The arrival of a message on a given input port needs *c* milliseconds of CPU time to process, and results in a message of size *m* bytes being generated and sent on an output port *p*.

It is also possible that the input port is an external port, which receives messages directly from the external environment. In this case, it is necessary to specify the characteristics of the incoming message load in terms of the average incoming message size and arrival rate.

In the case of an entry interface, which supports two-way communication, the incoming message results in an outgoing message of a certain size being generated and sent out on the same interface. It is assumed that entries, which represent server interfaces, do not receive messages directly from the external environment.

#### *4.2.1 XML Notation for Performance Annotations of Components*

The specification, in an XML-type notation, for a component's performance parameters is given below. We use italics to indicate built-in types such as string or real and use boldface to indicate keywords Optional specifications appear within square brackets. Specifications that may appear at least once are indicated within [ ]+.

Specifications that may appear zero or more times are indicated within [ ]*.

```
<component>
  <ComponentTypeName> string
    </ComponentTypeName>
  [<port> <InputPortName> string </InputPortName>
   <PortType> external | internal </PortType>
     [<AvgIncomingMsgSize> real bytes
        </AvgIncomingMsgSize>
      <IncomingMsgArrRate> real msg/sec
        </IncomingMsgArrRate>]
   <case>
    [<probability> real </probability>
     <ProcessingTime> real sec </ProcessingTime>
     <OutgoingMsgSize> real  bytes
        </OutgoingMsgSize>
      <OutputPortName> string </OutputPortName>]+
   </case>
   </port>]*
  [<entry> <EntryName> string </EntryName>
   <ProcessingTime> real sec </ProcessingTime>
   <OutgoingMsgSize> real bytes </OutgoingMsgSize>
   </entry>]*
</component>
```

The details are as follows: There is one component declaration for each component. There is one port declaration for each input port of the given component. The **input port name** is specified. The **port type** is also specified: **external** or **internal**. An external port receives messages from the external environment. An internal port receives messages from another component. In the case of an external port, it is also necessary to specify the **average incoming message size** and the **message arrival rate**.

Performance parameters for an outgoing message are the processing time, ProcessingTime, of the incoming message, the size, OutgoingMsgSize, of the outgoing message and the port, OutputPortName, where it is sent to. If there is the possibility of more than one message being generated and sent out of a different port, then it is necessary to specify the probability of each case.

For each entry interface, the entry name EntryName is specified. The performance parameters are the processing time, ProcessingTime, of the incoming message and the outgoing message size, OutgoingMsgSize. The outgoing message is sent out on the same interface.

### *4.2.2 XML Notation for Performance Annotations of Connectors*

There are two types of connector: Client Connector and Server Connector.

The specification in XML notation for a client connector's performance parameters is:

```
<ClientConnector>
  <ClientConnectorName> string
    </ClientConnectorName>
  <ClientConnectorType> asynchronous | synchronous
    | brokered </ClientConnectorType>
  <ClientStubTime> real sec </ClientStubTime>
</ClientConnector>
```

The Client Connector has a name as well as a ClientConnectorType, which can be asynchronous, synchronous, or brokered. The ClientStubTime is specified, which is the CPU time required to process a message and/or response.

The specification in XML notation for a server connector's performance parameters is as follows:

```
<ServerConnector>
  <ServerConnectorName> string
    </ServerConnectorName>
  <ServerStubTime> real sec </ServerStubTime>
  <NumberServerThreads> integer
    </NumberServerThreads>
  <MaxNumServerConnections> integer
    </MaxNumServerConnections>
</ServerConnector>
```

The Server Connector has a name. The server stub time, ServerStubTime, is specified, which is the CPU time required to process a message and/or response. The number of server threads is specified. For a sequential server, this value is equal to 1. For a concurrent server, this value is greater than 1. In addition, the maximum number of server connections is specified.

### *4.2.3 XML Notation for Performance Annotations of Resources*

The specification in XML notation for a resource's performance parameters is given next. At this time, there is only one resource specified, namely the network that connects components and connectors.

The network parameters are the latency and the bandwidth.

```
<resource>
  <ResourceName> string </ResourceName>
  <latency> real msec </latency>
  <bandwidth>  real Mbps </bandwidth>
</resource>
```

## 5. PERFORMANCE MODELS OF COMPONENT INTERCONNECTION PATTERNS

In this section, we describe the performance models of the Component Interconnection Patterns for synchronous and asynchronous communication. These models are based on previous work in performance modeling of software systems [13,14,15,16,17,32].

### 5.1 Synchronous Communication – Fixed Number of Threads

Here we assume that the client sends a request to a multi-threaded server. Each request is handled by a single thread. The number of threads is fixed. All threads are assumed to be created at server initialization time. A queue of requests to be processed by the server arises. It is assumed that the server uses both CPU and I/O in order to process the request.

Figure 7 shows the performance model for a synchronous Client/Server (C/S) interaction. The number of requests within the dashed box cannot exceed the number of threads *m.* If there are more requests than available threads, the additional requests have to wait in the queue of threads.

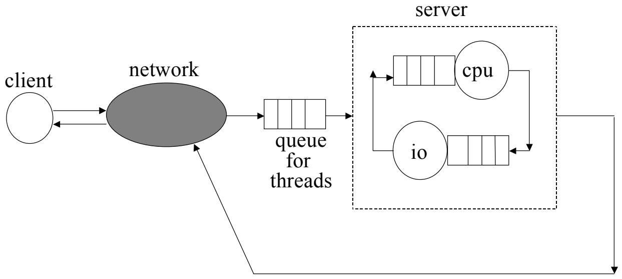

**Figure 7 – Performance Model for Synchronous C/S Interaction with Fixed Number of Threads**

The queuing model for the server consists of a hierarchical model in which the server throughput $X(n)$ for *n* = 0, …, *m* is obtained by solving the queuing network that represents the server (CPU and IO) using Mean Value Analysis. Then, using the decomposition principle [14], the entire server including the queue for threads is modeled using a Markov Chain model.

In a closed system, the number of clients, *N,* is fixed and the average think time, *Z*, per client has to be known. Figure 8 shows a graph of response time versus request arrival rate for different values of the number of threads. The curves show that as the number of threads increases, the response time decreases and the maximum achievable throughput increases.

### 5.2 - Synchronous Communication – Dynamic Thread Pool

In the case of dynamic thread pool, each time a request is created, a new thread is created to serve the request. When the request is complete, the thread is destroyed. In this case, there is no queue for threads. The corresponding queuing model is shown in Fig. 9.

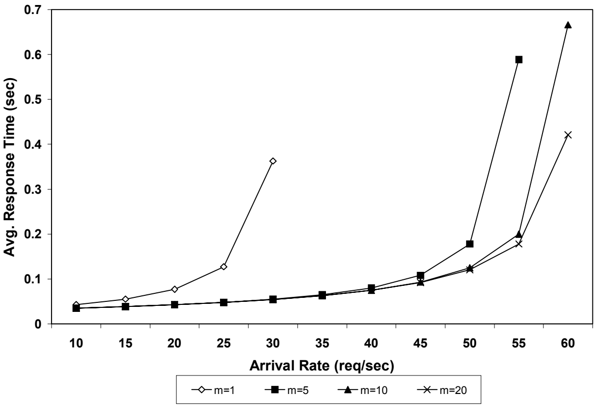

**Figure 8 - Response Time vs. Arrival Rate for Different Values of the Number of Threads for Synchronous Communication and Fixed Number of Threads.**

The model of Fig. 9 can be solved using a non-hierarchical queuing network model given that there is no limit in the number of threads in the system. The tradeoff in this case is between the time to wait for an existing thread and the time to create and destroy threads. Fig. 10 shows a graph that illustrates this tradeoff. The graph shows the average response time versus the number of threads in the fixed number of threads case under the assumption that the time to create and destroy a thread is one third of the time to execute the thread. As it can be seen, there is a cross over point for six threads. Below this limit, the dynamic thread case provides a better response time because of the excessive time in the thread queue. As more threads become available in the thread pool, the thread waiting time decreases in a more than linear fashion. For more than six threads, the thread creation and destruction time exceeds the thread waiting time.

### 5.3 Asynchronous Communication

The asynchronous communication case includes the case in which the client goes through three phases:

1. Phase 1: local processing, at the end of which a request is sent to the server. The client does not stop while waiting for the request and proceeds to phase 2. Let $t_1$ be the average local processing time at this phase.
2. Phase 2: local processing while waiting for the reply from the server. If the reply from the server does not arrive before this processing, of duration $t_2$, is complete, the client stops and waits for the reply. Otherwise, the client finishes execution of its phase 2.
3. Phase 3: the client does local processing based on the reply received from the server. Let $t_3$ be the duration of this phase.

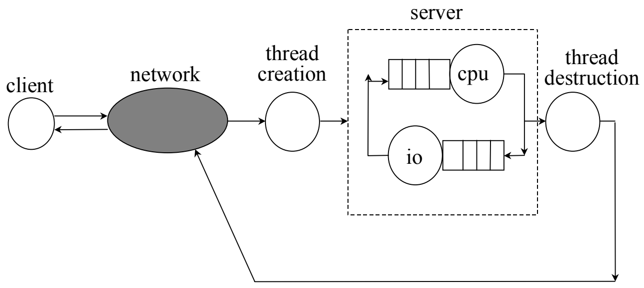

**Figure 9 – Performance Model for Synchronous C/S Interaction with a Dynamic Thread Pool**

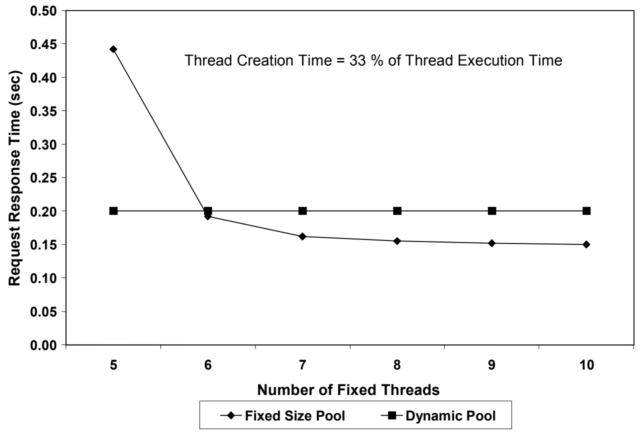

**Figure 10 - Tradeoff Between the Fixed Number of Threads and the Dynamic Thread Pool Cases.**

The type of asynchronous communication discussed here can be implemented with multithreaded clients or with callbacks. To model asynchronous communication, an approximate iterative model is used.

Figure 11 shows the throughput, measured in requests/sec, versus the number of clients for different values of $t_2$. The following observations can be made. After a certain number of clients, 50 in Fig. 11, the throughput is the same for all values of $t_2$ considered. This happens because the server will be congested and the response time will always be received after $t_2$ expires. In a lighter load range, the throughput increases as $t_2$ decreases. This happens because clients are able to send requests at a higher rate.

## 6. CONCLUSIONS AND FUTURE WORK

This paper has described an approach for the design and performance modeling of component interconnection patterns. We start with UML design models of the component interconnection patterns. We then provide performance annotations of the UML design models. We then map the performance annotated UML design model to a performance model, which allows us to analyze the performance of the software architecture executing on various system configurations. This paper has described the case of synchronous and asynchronous communication with a multi-threaded server. We have also modeled component interconnection patterns for brokered communication, although there is insufficient space to describe this.

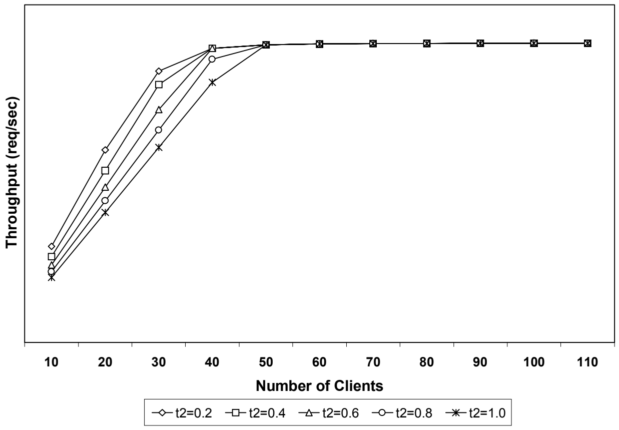

**Figure 11 - Throughput vs. Number of Clients for Asynchronous Communication.**

We plan to extend the work developed in this project by providing a language-based method for the design and performance analysis of component-based systems prior to their implementation. For that purpose, we plan to specify a Component-based System and Performance Description Language (CBSPDL). The goal is to specify both the architecture and performance of a large component-based distributed system in terms of new components, as well as predefined components and inter-component communication patterns that are stored in a reuse library. Thus, our goal is to expand our research in the direction of advancing the methods and software technology needed to reduce the costs and risks in the software development of large and complex distributed software systems.

## ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation through Grant CCR-9804113.

## REFERENCES

[1] G. Abowd, R. Allen, D. Garlan. "Using Style to Understand Descriptions of Software Architecture," Proc. Symposium on Foundations of Software Engineering, December 1993.

[2] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection, " ACM Transactions on Software Engineering and Methodology, 1997.

[3] L. Bass, P. Clements, and Rich Kazman, " Software Architecture in Practice," Addison-Wesley, 1998

[4] G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide," Addison Wesley, Reading MA, 1999.

[5] F. Buschmann et al, "Pattern Oriented Software Architecture: A System of Patterns," Wiley, 1996.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[7] H. Gomaa, D. A. Menascé, and L. Kerschberg, A Software Architectural Design Method for Large-Scale Distributed Data Intensive Information Systems, *Journal of Distributed Systems Eng.*, Vol. 3, 1996, pp. 162-172.

[8] H. Gomaa, "Use Cases for Distributed Real-Time Software Architectures", *Journal of Parallel and Distributed Computing Practices*, June 1998.

[9] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison Wesley, 2000.

[10] C. Hofmeister, R.L. Nord, D. Soni, "Applied Software Architecture", Addison Wesley, 2000.

[11] R. Kazman, M. Barbacci, M. Klein, S. J. Carrière, S. Woods, "Experience with Performing Architecture Tradeoff Analysis", *Proc. IEEE International Conf. on Software Engineering*, IEEE Computer Soc. Press, 1999.

[12] D. Luckham et al, "Specification and Analysis of System Architecture using Rapide", *IEEE Transactions on Software Engineering*, Vol. 21, No.4, April 1995.

[13] Menascé, D. A. and V. A. F. Almeida, Client/Server System Performance Modeling, in Performance Evaluation - Origins and Directions, eds. G. Haring, C. Lindemann, and M. Reiser, Lecture Notes in Computer Science, Springer-Verlag, 2000.

[14] D. A. Menascé, V. A. F. Almeida, and L. Dowdy, *Capacity Planning and Performance Modeling: from mainframes to client-server systems*, Prentice Hall, Upper Saddle River, New Jersey, 1994.

[15] D. A. Menascé and V. A. F. Almeida, Two-level Performance Models of Client-Server Systems, *1994 Computer Measurement Group Conference*, Orlando, Florida, December 1994.

[16] D. A. Menascé, Computer System Performance Evaluation: a key aspect of system design practice, *1998 Yearbook of Science and Technology*, McGraw-Hill, NY.

[17] D. A. Menascé, A Framework for Software Performance Engineering of Client/Server Systems, *Proc. 1997 Computer Measurement Group Conference,* Orlando, December 1997.

[18] Menascé, D. A. and H. Gomaa, "A Method for Design and Performance Modeling of Client/Server Systems," to appear in the IEEE Transactions on Software Engineering.

[19] Menascé, D. A. and H. Gomaa, "On a Language Based Method for Software Performance Engineering of Client/Server Systems," First International Workshop on Software Performance Eng.," New Mexico, Oct., 1998.

[20] D. A. Menascé, H. Gomaa, and L. Kerschberg, A Performance-Oriented Design Methodology for Large-Scale Distributed Data Intensive Information Systems*, Proc. First IEEE International Conf. on Eng. of Complex Computer Systems*, Southern Florida, USA, Nov. 1995.

[21] Magee, J., Dulay, N., and Kramer, J., "Regis: A Constructive Development Environment for Parallel and Distributed Programs", J. Distributed Systems Engineering, 1994, pp. 304-312.

[22] R. Orfali, D Harkey, J. Edwards, "Essential Client/Server Survival Guide", Wiley, Third Ed., 1999.

[23] R. Orfali, D Harkey, J. Edwards, "Essential Distributed Objects Survival Guide", Wiley, 1996.

[24] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. Rosenblum, "Integrating Architectural Description Languages with a Standard Design Method," Proc. Intl. Conf. on Software Engineering, Kyoto, Japan, April 1998.

[25] J. Rumbaugh, G. Booch, I. Jacobson, "The Unified Modeling Language Reference Manual," Addison Wesley, Reading MA, 1999.

[26] B. Selic, G. Gullekson, and P. T. Ward, "Real-Time Object-Oriented Modeling," John Wiley & Sons, 1994

[27] M. Shaw & D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[28] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," IEEE Transactions on Software Engineering, April, 1995, 314-335.

[29] R. N. Taylor, N. Medvidovic, K. Anderson, Jr. E. J. Whitehead, J. E, Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component and message-based architectural style for GUI software. IEEE Trans. Software Engineering, June 1996, Vol.22, no.6, pp.390-406.

[30] J. Magee and J. Kramer, "Concurrency: State Models & Java Programs", John Wiley & Sons, 1999.

[31] Eric M. Dashofy, N. Medvidovic, and R. Taylor, "Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures," *Proc. IEEE Intl. Conf. on Software Engineering*, May 1999.

[32] A. Ferscha, D. Menascé, J. Rolia, B. Sanders, and M Woodside, Performance and Software, in System Performance Evaluation: Origins and Directions, eds. M. Reiser, C. Lindemann, and G. Haring, Schloss Dagstuhl Report no. 9738, September 1997.