

A Web-based Distributed Network Simulation Framework using CORBA IDL-based APIs

Arjun Cholkar
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213.

ABSTRACT

Web-based network simulation frameworks are becoming highly portable and extensible. However, they still lack the degree of language and platform independence required for large-scale deployment on the World Wide Web. Our approach to enabling large-scale deployment uses a set of standard CORBA-IDL based programming interfaces, a publisher-subscriber model for communication, and dynamic composition of all simulation entities (simulated network hosts and links). A prototype application for testing distributed computing policies has demonstrated that the CORBA components not only provide language and platform-independence, but also provide the ability for developers to write their own objects and connect them to a third-party network simulation. By using a uniform messaging approach to all simulation events, objects can be reassigned to different simulation entities without requiring source-code modifications or re-compilation. The support for dynamic loading and unloading of objects during a simulation run facilitates fault simulation, simulation entity polymorphism, and generation of dynamic topologies. A link scheduling example application demonstrates our approach, its extensibility and flexibility.

KEYWORDS: General Applications; CORBA-based simulation, network simulation; Web-based tools, applications and environments.

1. INTRODUCTION

The World Wide Web (WWW) and the underlying Internet provide a potentially huge distributed computing infrastructure, which can be extended to host distributed simulation services [5]. With recent developments in web-based simulation technology, the portability and versatility of simulation tools has increased dramatically [13]. The ubiquity of web access has also created a promising medium for large-scale deployment of such simulation tools.

Achieving large-scale deployability on the web poses some interesting challenges. Web-based simulation tools need to be portable across a variety of computing platforms. These simulation tools also need to be as flexible as traditional tools, where users typically have the ability to extend the tool as desired by writing their own objects. Furthermore, this flexibility should not compromise on user-convenience issues such as using a familiar implementation language. Therefore, web-based simulation tools need to provide a portable and extensible infrastructure with interoperability support for distributed simulation objects written in a variety of languages.

The current trend towards building web-based network simulation frameworks is to use platform-independent object-oriented (OO) technologies such as Java [1] and RMI/Enterprise Java Beans [18]. These frameworks are portable and extensible but require that all simulation objects be developed in one language, Java. This approach limits the ability to re-use existing code and limits the deployability of the framework to only those users who are comfortable with the Java language.

This paper proposes a web-based network simulation framework that effectively uses the Common Object Request Broker Architecture (CORBA) [3] technology to provide a flexible, extensible, platform-independent and language-independent simulation environment that is suitable for large-scale deployment. The framework uses standard CORBA Interface Definition Language (IDL) [3] - based Application Programming Interfaces (APIs) and a CORBA Object Request Broker (ORB) [3] to provide the necessary location transparency and language independence. These CORBA components enable developers to write their own objects on their own platforms, and have their

objects participate from their platforms in a remote-simulation being hosted by a third party.

All simulation entities (simulated hosts and links) in the framework are composed using stock and user-written objects. To facilitate communication among these objects, a mechanism based on the publisher-subscriber [14] model is used. All interactions within the framework are message-based and have a uniform message format. This enables loosely coupled interaction between objects in a simulation entity and facilitates relocation of objects to different simulation entities with ease.

The IDL-based APIs in the framework also include support for dynamic loading and unloading of modules (user-written objects and stock objects). This enables modules to load or unload other modules dynamically during the simulation run. This facility is useful for simulating faults, changing the behavioral aspects of simulation entities (simulation entity polymorphism) and generating dynamic topologies. This adds tremendous flexibility to the simulation framework and makes it well suited to simulate real-world scenarios found in wired and wireless networks.

A prototype application for network simulation frameworks was also implemented using this framework. The application includes networking-specific tools for visualization, debugging and post-simulation data analysis. These tools aid in rapid composition and analysis of a simulated network.

The efforts undertaken by researchers so far and our approach to the problem are discussed in Section 2. Based on that discussion, Section 3 presents our architecture and discusses the CORBA IDL-based APIs. Section 4 describes the prototype application. We present a link scheduling example application in Section 5 and demonstrate that our APIs are applicable and sufficient to implement Resource Priority Multiplexing (RPM) [7] modules. We conclude our discussion in Section 6 and present some future possibilities for similar frameworks.

2. PREVIOUS WORK

Simulation tools can be primarily divided into two classes, *generic simulators* and *simulation frameworks*. Generic simulators are relatively simple and can be used as building blocks for more complex and specialized simulators. Simulation frameworks, on the other hand, include a specialized set of tools that aid in time-efficient and cost-effective development of domain-specific simulations. For example, various routing mechanisms can be developed and evaluated more efficiently and effectively by using a network simulation framework rather than a generic discrete event simulator. Therefore, simulation frameworks may be preferable to generic simulators.

The CORBA-based simulation facility developed at Bellcore [17] for generic discrete-event simulation provides a location-transparent and language-independent mechanism for generic simulations, and is suitable for remote simulation. However, this work is targeted solely for generic simulations and would have to be extended to be applicable to domain-specific contexts.

Mature network simulation frameworks such as REAL [9], ns-2 [19], INSANE [8] and x-Sim [2] provide rich APIs and tools and have been extensively used by researchers. However, even with support for web-based simulation in REAL, their flexibility is limited because they do not provide simulation support for user-written objects in their web-based mode. This limits their utility for large-scale deployment over the Web.

With the recent demand for web-based network simulations, frameworks such as NetSim^Q [11] have been developed. NetSim^Q provides an extensible simulation environment and a rich graphical user-interface. Additionally, it is highly portable because it is written in Java, but this comes at the price of a single-language implementation approach.

The network simulation framework discussed in this paper builds upon the work done in both the simulation and distributed computing communities to create a result that combines the best of both worlds. It draws ideas from the CORBA-based approach for generic simulators, and formulates a CORBA-based approach for the network simulation context. This allows remote user-written objects

to cooperate in a simulation, distributed on several machines, in a language-independent and platform-independent fashion. To our knowledge, this degree of flexibility is novel for a network simulation framework. Additionally, our framework offers a set of networking specific tools for composition and analysis of the simulation. By allowing objects to cooperate in a simulation in a language-independent fashion, our framework does not impose the single-language environment of NetSim^Q.

Our framework stresses providing flexibility by dynamically composing simulation entities. The publisher-subscriber model used for communication in the framework along with a uniform message-format aids us in achieving this goal. This communication model is similar to the anonymous communication model [12] used in the distributed computing community and facilitates relocation of objects to different simulation entities during a simulation run.

The Defense Modeling and Simulation Office's (DMSO) High Level Architecture (HLA) [4] also supports such language-independent and platform-independent composition of a simulation. However, the HLA differs from our framework in that it is generic rather than networking-specific. Hence, it does not include any tools for synthesizing and analyzing network simulations. The HLA does provide reasonable underlying mechanisms for future versions of our framework, but was not ready in time to be used on the current project described in this paper.

3. NETWORK SIMULATION FRAMEWORK

The architecture of the network simulation framework is derived from a top-down Object Oriented (OO) view of a network. To produce an extensible foundation for the framework, we identified the core components present in all communication networks and built their software representations. These core components are modeled as abstract classes, where each abstract class heads a hierarchy of a particular type of components. Components get more specialized and embody additional features and properties with each successive level in the hierarchy. In addition, all components were built using well-established OO design patterns [6] to make them highly re-usable.

3.1. OBJECT HIERARCHY

All hardware components are derived from an abstract base class called *Entity*. This abstract class encapsulates method definitions that are common to all hardware components (Figure 1).

Two classes were derived from *Entity*, called *Node* and *Channel* to model the fundamental concept of a communication network in which a set of nodes exchange messages over some communication medium. Another level of class inheritance was used to construct the *Host* and *Link* classes. A *Host*, which is derived from a *Node*, has a Central Processing Unit (CPU) and input/output (I/O) ports, while a *Link*, which is derived from a *Channel*, could be a simplex, half-duplex or full duplex link.

Each entity consists of a *module manager*, which encapsulates an API, known as the ‘Module Plug-in API’ (MPA). The *ModuleManager* class is an abstract base class that heads a hierarchy of more sophisticated module managers. Module managers are also refined using multiple levels of inheritance, and there is always one and only one module manager associated with each entity. These inheritance hierarchies are shown in the class diagram of Figure 1. (All notations in figures are per the Object Modeling Technique (OMT) [15, 16])

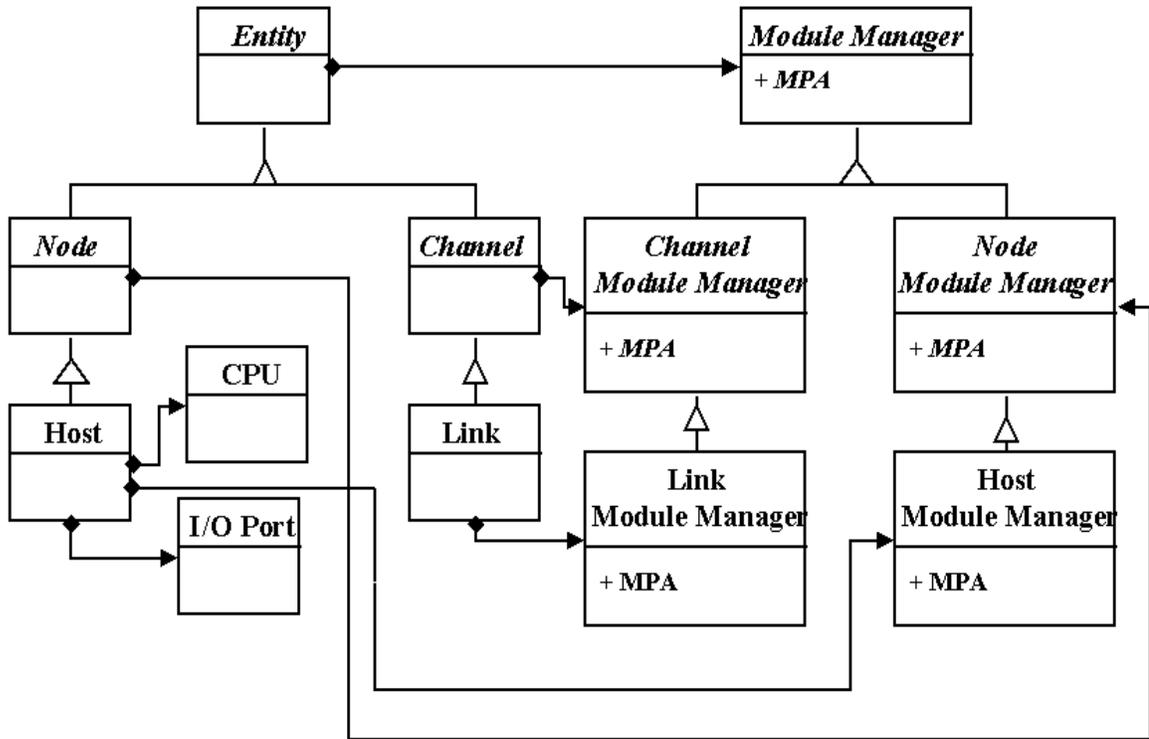


Figure 1. Class diagram depicting the inheritance hierarchy of simulation entities.

Even with additional components, all entities have a well-defined and finite set of behavioral properties. Therefore, one or more intelligent *modules* are assigned to each entity to enhance behavior. For example, a system might consist of a host, which functions as a hub because a forwarding module has been assigned to it. This approach of assigning modules to existing entities, as opposed to creating more sophisticated entities, favors object composition over class inheritance. This preserves encapsulation and is likely to result in fewer implementation dependencies [6]. Additionally, because object composition is defined at run-time, it can provide a significantly higher degree of flexibility than class inheritance, especially when entities need to exhibit polymorphic behaviors.

To support flexible object composition, entities need a standard way to reference all modules without having to know the actual class of a particular module. Hence, all modules are derived from

an abstract base class, *Module*, which exports an interface known as the Module Callback Interface (MCI). The MCI contains methods for module initialization, message processing and module shutdown, which the entity invokes as needed. The inheritance hierarchy for a module is shown in the class diagram of Figure 2 where *ModuleX* and *ModuleY* are example concrete classes that are derived from the abstract base class *Module*.

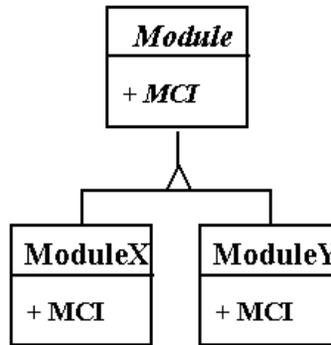


Figure 2. Class Diagram for depicting the inheritance hierarchy of modules.

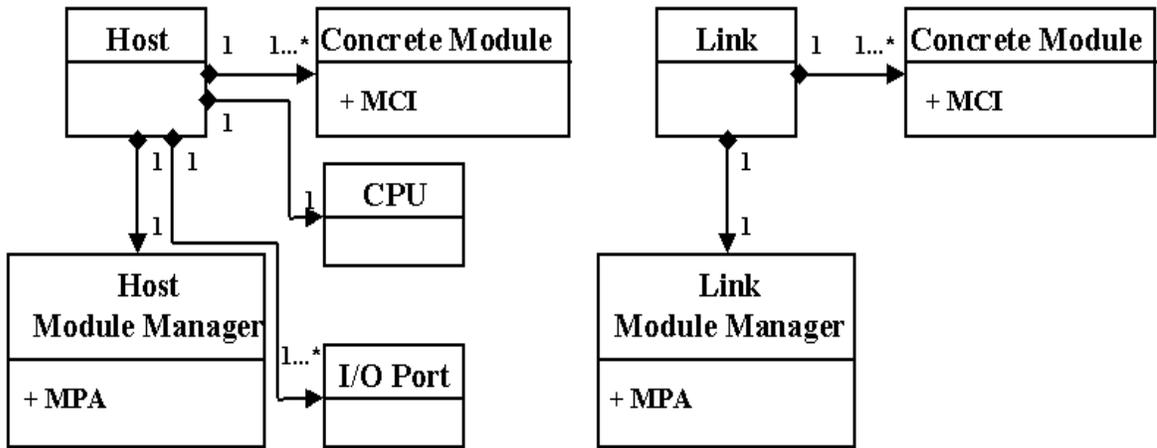


Figure 3. Class diagram for a typical host and link configuration.

A typical host or link consists of a number of modules assigned to it and an appropriate module manager. As mentioned earlier, a host also contains a CPU and a number of I/O Ports, while a link

may or may not possess any additional properties over a channel. These configurations are shown in Figure 3.

3.2. CORBA-IDL BASED APIs

The Module Plug-in API (MPA) and the Module Callback Interface (MCI) provide standard interfaces and semantics for modules and the simulation entities to refer to each other without having to know the actual concrete classes of each other. This is helpful when simulation entities need to be composed dynamically.

The flexibility provided by the MPA and the MCI is extended further by defining them in CORBA IDL and using a CORBA ORB to provide necessary middleware services. This allows modules written in a variety of languages and distributed on a variety of platforms to be dynamically reassigned to different simulation entities while cooperating in a single simulation. This location transparency and language independence provided by CORBA eliminates writing complicated inter-process communication code in a potentially unfamiliar language. In addition, it also makes our API highly portable and our framework widely deployable.

3.2.1. THE MODULE PLUG-IN API

```
module AmModuleModuleManager
{
  interface AmModulePluginAPI
  {
    long amRegisterPublisher(in AmSimulationDataTypes::AmMessage inMessage);
    long amRegisterSubscriber(in AmSimulationDataTypes::AmMessage inMessage);
    long amUnRegisterPublisher(in AmSimulationDataTypes::AmMessage inMessage);
    long amUnRegisterSubscriber(in AmSimulationDataTypes::AmMessage inMessage);
    long amSendMessage(in AmSimulationDataTypes::AmMessage inMessage);
    long amTrigger(in AmSimulationDataTypes::AmMessage inMessage);
    long amLoadModule(in AmSimulationDataTypes::AmMessage inMessage);
    long amUnloadModule(in AmSimulationDataTypes::AmMessage inMessage);

    long amTransmitMessage(in AmSimulationDataTypes::AmMessage inMessage,
                          in string inInterfaceId);
  };
};
```

Figure 4. The core Module Plug-in API.

The CORBA-IDL based Module Plug-in API (MPA) is shown in Figure 4. Note that only the core API calls have been shown. An illustration of how the API would be used is presented later in the paper.

There are a couple of things worth noting in the MPA. First, the MPA uses a uniform message format, *AmMessage*, for all calls. This feature, along with the publisher-subscriber communication model, facilitates relocation of modules to different simulation entities.

Secondly, the MPA includes methods for dynamic loading and unloading of modules. These requests are generated by other cooperating modules desiring to dynamically alter the behavior of the simulation entity without user intervention. This feature enables a developer to simulate faults dynamically, change entity behaviors dynamically, and change the topology of the simulated network on-the-fly. For example, loading a routing and forwarding module on a host would change its behavior from an end-host to a router. Similarly, by having a module unload all other modules on a host acting as a router, a router failure can be simulated, which would then cause routes to change on the neighboring routers. This provides a dynamically customizable simulation environment with an added advantage of simulating a large number of scenarios with only a handful of modules. However, this advantage comes at the price of more complexity in the modules. Such modules have to include code that can react to different stimuli provided in different scenarios. Therefore, the size vs. complexity tradeoff should be carefully evaluated before implementing such modules.

The MPA also provides methods for accessing certain information and output services hosted by the framework. These services provided by the MPA can be classified into the *information service*, the *time service*, the *logging service* and the *data display service*. These services provide informative and output facilities to a module.

3.2.2. THE MODULE CALLBACK INTERFACE

```
module AmModule
{
    interface AmModuleCallbackInterface
    {
        void amInit(in AmSimulationDataTypes::AmMessage inMessage);
        void amProcessMessage(in AmSimulationDataTypes::AmMessage inMessage);
        void amDestroy(in AmSimulationDataTypes::AmMessage inMessage);
    };
};
```

Figure 5. The Module Callback Interface.

The CORBA-IDL based Module Callback Interface is shown in Figure 5. It can be seen that it has three methods *amInit*, *amProcessMessage*, and *amDestroy*, which are invoked for module-initialization, message-delivery and module-shutdown respectively. In addition, as seen from Figure 5, all methods accept only one argument, which has a uniform format. As mentioned earlier, this enables module relocation to different simulation entities with ease.

3.3 INTER-MODULE COMMUNICATION

All inter-module communication in the framework is message-based and is built upon a publisher-subscriber model that is functionally similar to the anonymous communication model.

Messages are published on named software buses called *message-channels*. These message-channels are dynamically created by an entity in response to requests from modules to either publish on, or listen to, non-existent message-channels. Message-channel names serve as unique identifiers and their scope is limited to each entity. A database of names for existing message-channels is kept by the entity and can be queried by a module to decide on a unique name for its new message-channel. Message distribution is handled by the entity transparently using message-channel names. Thus, an entity acts only as a message-distribution agent for all inter-module messages and essentially has no knowledge of their contents. This complements our design goal of “simple entities” and “intelligent modules”.

To illustrate the communication mechanism, consider an example in which the two classes, *SenderModule* and *ReceiverModule*, shown in Figure 6, are derived from *Module*. Both these classes inherit the MCI and provide the necessary implementations for the methods in the MCI. Also, consider the object diagram of Figure 7. The instance *hostA* of class *Host* comprises of an instance *hostModuleManager* of class *HostModuleManager*, which provides the necessary implementations for methods in the MPA. The instance *hostA*, also has three modules assigned to it, namely *sender*, which is an instance of class *SenderModule* and *receiver1* and *receiver2*, which are both instances of class *ReceiverModule*. As their names suggest, *sender* acts as a publisher and desires to publish messages on a new message-channel 'XYZ'. *receiver1* and *receiver2* on the other hand act as subscribers and desire to receive all messages published on message-channel 'XYZ'. Thus, all messages published by *sender* would be received by modules *receiver1* and *receiver2*.

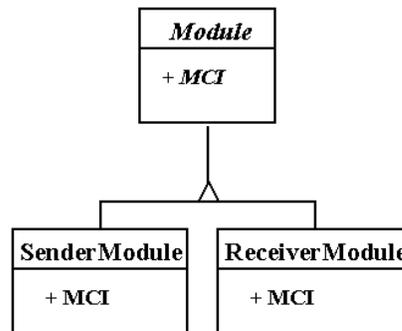


Figure 6. Class diagram for modules *SenderModule* and *ReceiverModule*.

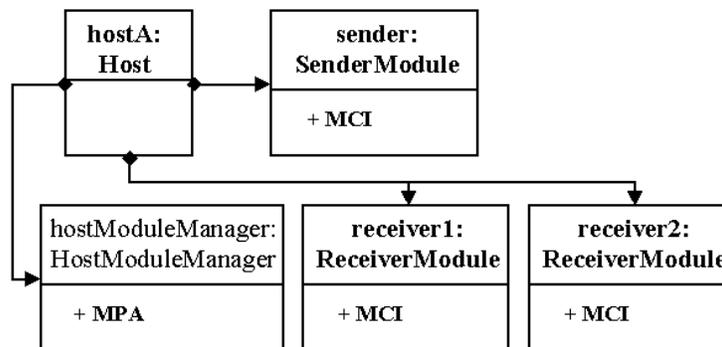


Figure 7. Object diagram for *Host hostA*.

The sequence of events using this communication model for the example configuration is shown in the sequence diagram of Figure 8. For clarity, API calls have been simplified and arguments have not been encapsulated via the *AmMessage* message format.

Three features provided by this model are noteworthy. First, if no subscribers had subscribed to message-channel ‘XYZ’, publisher *sender* could have still published messages. Second, the message distribution service in *hostA* does not care about the contents of a message, and just delivers the message to all the registered subscribers. Finally, all publishers and subscribers can subscribe and unsubscribe whenever they choose. Thus, these features provide a loosely coupled communication mechanism that is useful for dynamic inter-module communication and module reassignment.

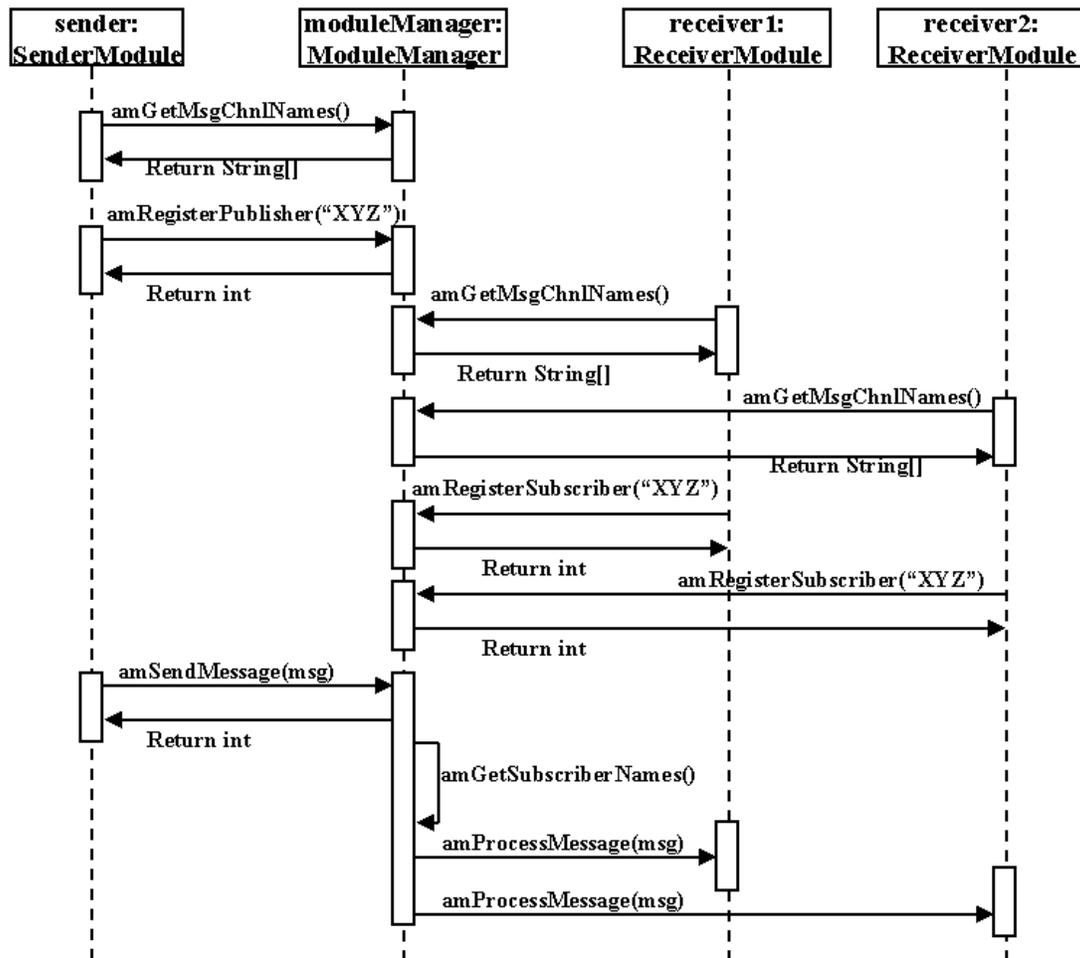


Figure 8. Sequence diagram for an example scenario of inter-module communication.

4. PROTOTYPE OVERVIEW

The prototype framework was implemented using a traditional two-tier client server model with a portable user interface that requests simulation services from a remote, compiled simulation kernel. This system partitioning provides portability to the framework without compromising on model execution speed.¹ Additionally, our approach reduces startup costs such as download latencies.

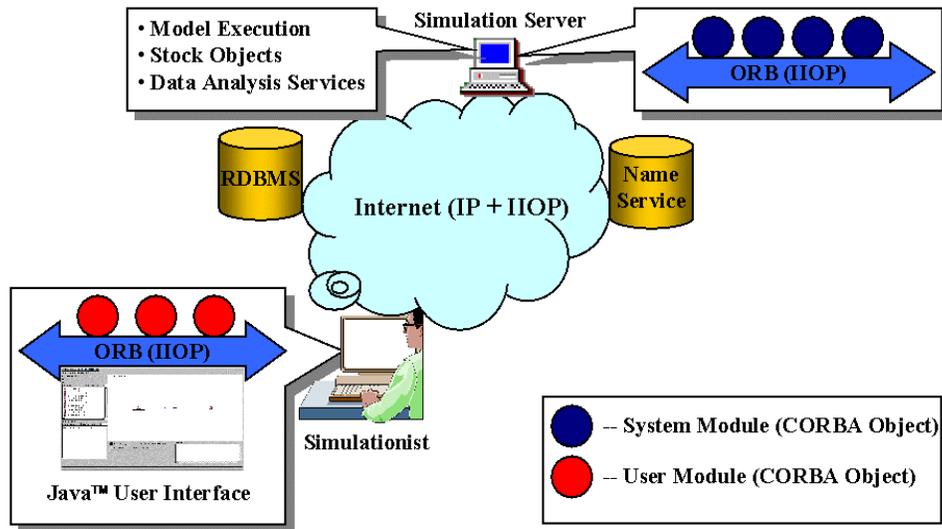


Figure 9. System Architecture of the prototype.

The prototype consists of a simulation kernel implemented in C++ and a user-interface written in Java. A CORBA ORB is used to provide remote object invocations and facilitates communication between the user-interface and the simulation kernel. A CORBA name service is used for providing the appropriate object references at run-time. In addition, a Relational Database Management System (RDBMS) is used to provide the necessary logging facilities. This system architecture is illustrated in Figure 9.

¹ This simulation framework was developed to test strategies for implementing network management policies. Because network management messages are infrequent compared to regular network traffic, this efficiency argument is based on an underlying assumption that the communication traffic between the user-interface and the simulation kernel is not a dominant factor in the overall workload.

Access to the simulation kernel is via a simulation server, which spends most of its time waiting for simulation requests. Simulation requests are generated by user interfaces when a user decides to start a new simulation. Upon receiving a simulation request, the simulation server spawns a process called a *Simulation Executive*. The simulation executive takes over the task of managing the simulation and coordinating interactions between different entities in the simulation. Each user interface has exactly one simulation executive assigned to it. The simulation executive handles the scheduling events and checking for breakpoints. Each host or link is encapsulated in a *HostExecutive* or *LinkExecutive* class respectively. These classes act as wrappers for the corresponding entities and present an API that other components in the system use for startup and shutdown of the encapsulated entity, event delivery, instrumentation and injecting asynchronous inputs. These executives are derived from a base class called *EntityExecutive* that includes all the common functionality.

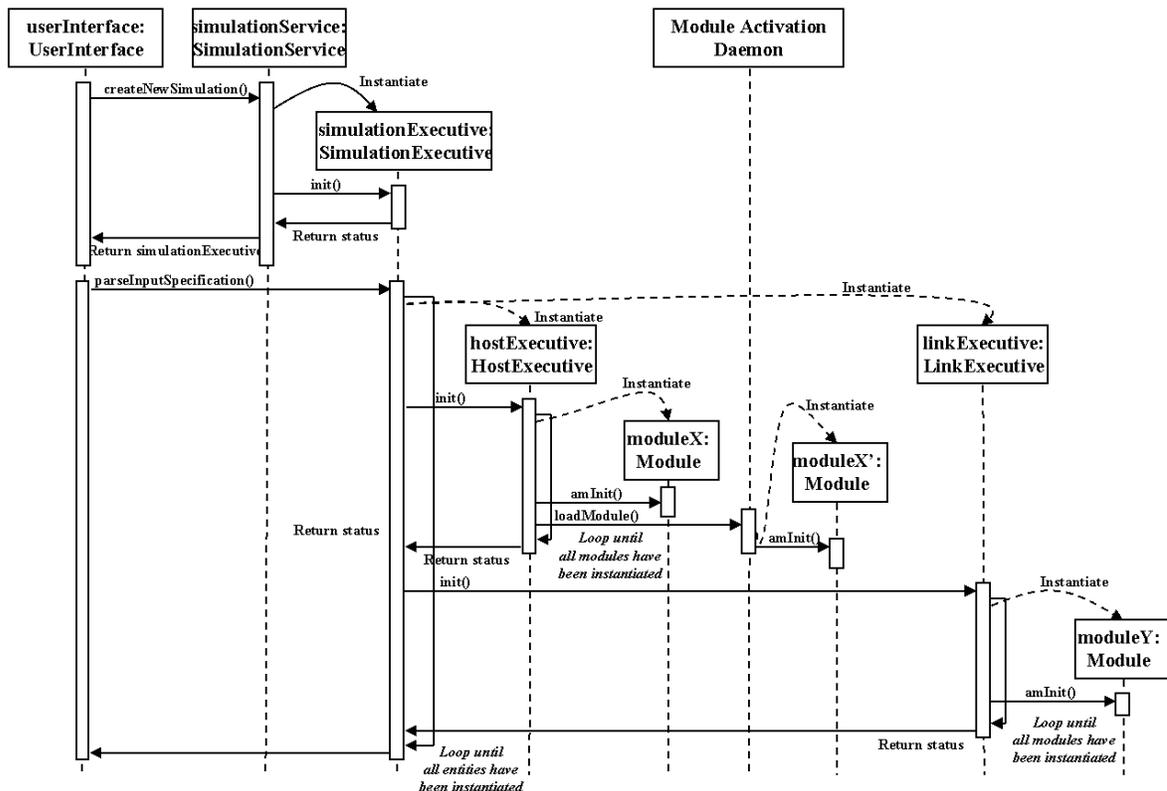


Figure 10. Sequence Diagram for a typical startup scenario.

Typical sequences of interactions that take place between the various components in the system are shown in the sequence diagram in Figure 10. On startup, the user interface first contacts the `SimulationService` and requests the creation of a new `SimulationExecutive`. Next, an input specification provided by the user is sent to the `Simulation-Executive`. This specification is then parsed and the appropriate `HostExecutives` and `LinkExecutives` are instantiated. The specification also contains identifiers for modules that need to be loaded (recall that for a host or link to exhibit a desired set of behavioral properties, modules must be assigned to it). Thus, each `HostExecutive` or `LinkExecutive` instantiates its modules. These modules then interact with the entity encapsulated in the executive through the MPA and drive the simulation.

A simulation normally consists of a large number of stock modules and a few user-written modules. Stock modules are pre-defined to help provide a variety of facilities such as routing abilities for a `Host` or statistical message corruption for a `Link`. User-written modules could be modules written by the user or could be third-party modules conforming to the MPA. In order to bootstrap these modules into a simulation, an activation daemon is included in the code shipped with the user interface.

5. EXAMPLE APPLICATION

To test the applicability of the MPA and the MCI, it was necessary to implement and test the behavior of modules that were reasonably complex. Hence, we decided to implement and test Resource Priority Multiplexing (RPM) [7] modules for scheduling network traffic. We found that implementing RPM schedulers as modules in our framework was readily done, and that the simulation results matched theoretical behavior predictions.

RPM is a probabilistic method for sharing resources among competing tasks, and is normally used for scheduling network traffic from flows that compete for access to a specific network link. RPM involves two components – the RPM scheduler and the RPM equation solver. To keep the discussion simple, we only talk about the RPM scheduler. The RPM scheduler is made up of a *marker* that assigns priorities to packets and the actual *scheduling mechanism* that transmits these packets. The marker has a built-in state machine that has one state per competing flow. The time spent in each state is decided by the RPM equation solver. These times depend on the task mix and the service levels required by each task. (The terms task and flow are used interchangeably throughout this section, because the RPM scheduler sees the manifestation of a task as a flow of packets that the task produces.)

The implementation for the methods in the MCI for the RPM scheduler is shown in Figure 11(a) through 11(c). Only the necessary code to explain the concepts and illustrate the use of the MCI and MPA has been included.

Figure 11(a) shows the initialization sequence for the RPM scheduler module. Initially, the module parses the input parameters, which can be specified using a configuration file by the user or can be specified by a module dynamically at run-time. The input parameters contain information about queue sizes and other configuration options. On parsing the input parameters, the module registers as a subscriber to a message-channel named “XMIT_REQUEST”. This message-channel is used to receive messages from other modules on the entity that want to send packets out on the network.

```

void AmRPMScheduler::amInit(AmMessage* inMessage)
{
    char* initString = strdup((char*) inMessage -> getMessage());
    amParseInput(initString);
    free(initString);

    AmMessage* registrationMessage = new AmMessage(name, "XMIT_REQUEST");
    moduleManager -> amRegisterSubscriber(registrationMessage);
    delete registrationMessage;
}

```

(a) Initialization

```

void AmRPMScheduler::amProcessMessage(AmMessage* inMessage)
{
    if(strcmp(inMessage -> getMessageType(), "XMIT_REQUEST", AM_MAX_NAME_SIZE) == 0)
    {
        AmPacket* packet = AmPacket::deserialize(inMessage -> getMessage());

        if(packet == NULL)
        { cerr << "Could not de-serialize the packet." << endl; return; }

        amClassifyPacket(packet);
        amSchedulePacket(packet);

        delete packet;
    }
}

```

(b) Message Processing

```

void AmRPMScheduler::amDestroy(AmMessage* inMessage)
{}

```

(c) Shutdown.

Figure 11. Implementations of methods in the MCI for the RPM scheduler.

Figure 11(b) shows the implementation of the *amProcessMessage* method, containing the decision process of the RPM module. Packets are first classified and marked with the appropriate priority, which depends on the state of the state-machine in the *marker*. This takes place in the *amClassifyPacket* method. After marking, the packets are enqueued and scheduled for transmission. If the queues get full, the scheduler in the *amSchedulePacket* method starts dropping packets with the lowest priorities until there is enough space in the queues to enqueue the new packet.

Figure 11(c) shows the *amDestroy* method of the module. Because there is neither any shared state between the RPM module and other modules nor any persistent state in the RPM module, the module

does not have any cleanup work to perform. Therefore, the method is empty.

A simple two-host network was simulated to test the RPM modules, in which one host was the sender and the other the receiver. Three modules were written to model three tasks that generate traffic bound from the sender to the receiver. The traffic profiles for the tasks are shown in Table 1.

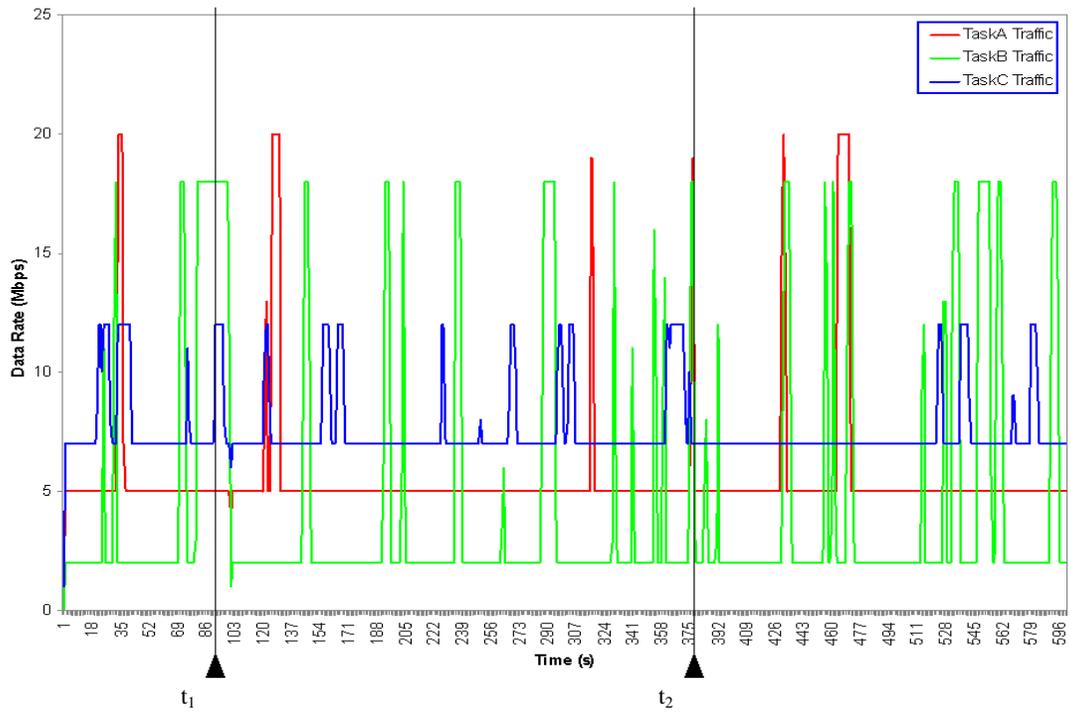
The notation used for the high and low profiles denotes the amount of time the task is in a particular state (high or low) and the traffic it generates. For example, the task A generates 5Mbps of traffic, 95% of the time and 20Mbps of traffic, the other 5% of the time. An RPM scheduler was also assigned to the sender and was used to schedule packets from the three modules. The bandwidth of the link connecting the two hosts was set to 30Mbps.

The last column in Table 1 indicates the assurance level required by each task for the duration of its session. An assurance level for a task is defined as the fraction of the amount of traffic generated by the task, that is successfully delivered to its intended receiver, expressed as a percentage. Therefore, an assurance level of 99% required by task A implies that it cannot tolerate a loss of more than 1% of its packets. The session length for all three tasks was set to be 600 seconds.

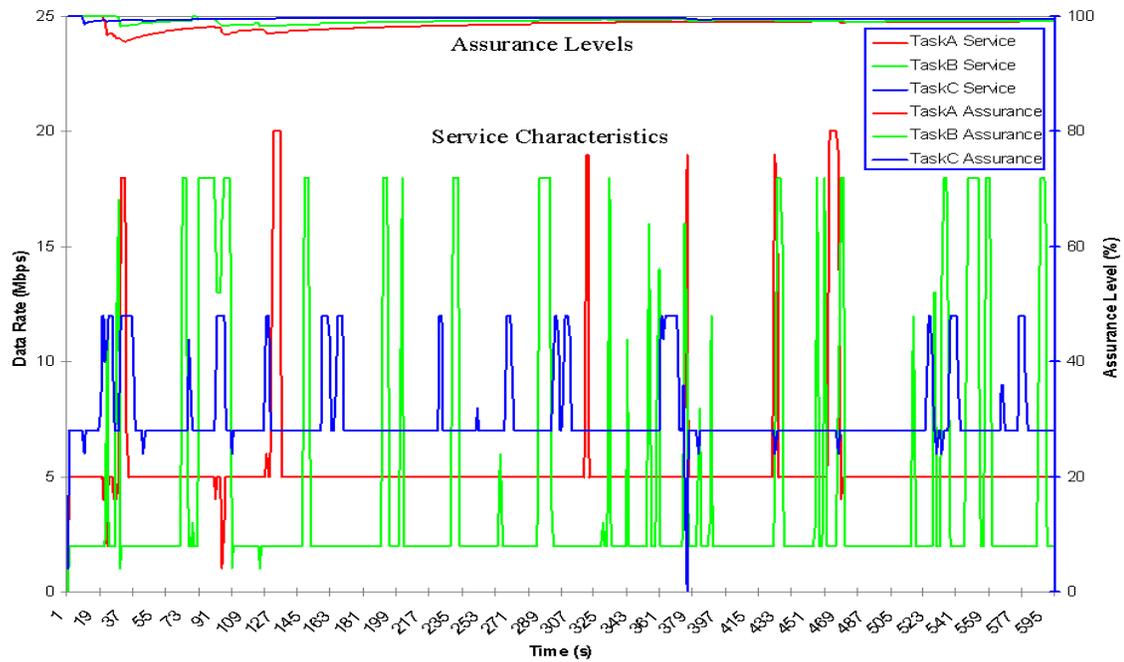
Task ID	Low Profile Specs.	High Profile Specs.	Assurance Level
A	95%@5Mbps	5%@20Mbps	99%
B	85%@2Mbps	15%@18Mbps	99%
C	90%@7Mbps	10%@12Mbps	98%

Table 1. Task Profiles for the RPM scheduler example on a 30Mbps link.

The results obtained for the network simulation problem are shown in Figure 12(a) and 12(b). Figure 12(a) shows the observed traffic generated by the three tasks. It can be seen that this traffic confirms to the traffic profiles of Table 1.



(a) Traffic generated by tasks.



(b) Service characteristics of tasks including assurance levels.

Figure 12. Traffic and service characteristics of tasks in the example simulation.

The traffic profiles shown in Table 1, indicate that there are modes of operation for the task-set where the sum of the total traffic generated by the three tasks exceeds the 30Mbps capacity of the link. For example, even if just task A or task B enters its 'high-profile' mode the total traffic generated by the three tasks is equal to 32Mbps or 33Mbps respectively. Figure 12(a) shows such an instance, 't₁', where tasks B and C are in their 'high-profile' modes and the total traffic generated at the sender is 35Mbps (5Mbps for task A + 18Mbps for task B + 12Mbps for task C). This additional traffic is first enqueued to smooth out small bursts. However with finite queue sizes large bursts result in queue overflows and eventually dropping of packets. Additionally, if all three tasks enter their respective 'high-profile' modes at the same time, as shown by instance 't₂' in Figure 12(a), the drop rate could be as high as 20Mbps worth of packets. In such situations, guaranteeing an assurance-level of 99%, 99% and 98% for the three tasks becomes a non-trivial exercise.

The delivered traffic at the receiver and the delivered assurance-levels for the tasks are shown in Figure 12(b). It can be seen that initially the assurance level for Task A goes below its probabilistic guarantee and drops down to 95.62%. This is a result of all three tasks switching to their high profile modes in the initial stages of their respective sessions. This surge of traffic causes many packets to be dropped in the system. This impacts the assurance levels of all tasks because the traffic generated in the initial stages of the session accounts for most of the traffic. Although, all tasks loose packets, Task A is penalized the most because during that interval the RPM marker happens to be in a mode where task A has the lowest priority. However, as the session progresses, all three tasks get equal to or greater than the assurance levels they desire. The values that were actually obtained were 99.16%, 99.28% and 99.58% for tasks A, B and C respectively.

Thus, because the RPM module implementation delivered the expected results we could conclude that our MPA and MCI were applicable and sufficient to implement even modules as complex as RPM schedulers.

6. CONCLUSIONS & FUTURE WORK

The concurrent need to provide portability and extensibility to web-based simulation frameworks while providing complete language-independence and platform-independence for large-scale deployment mandates a careful choice of distributed object technologies along with a good design. Current, web-based, network simulation frameworks fall short of providing complete language and platform independence and hence have limited deployability.

Our approach to enabling large-scale deployment uses CORBA-IDL based APIs, a publisher-subscriber communication model, and dynamic composition of all simulation entities. The CORBA components provide language and platform-independence. In addition, they allow developers to write their own objects on their own platforms and have their objects participate from their platforms in a remote-simulation. We used a uniform message format to enable module reassignment to different simulation entities without requiring source-code modifications or recompilation. Our APIs also include support for dynamic loading and unloading of modules, which facilitates simulation of faults, simulation entity polymorphism and dynamic topologies. We also demonstrated the applicability and sufficiency of the APIs to model real world networking mechanisms through an example application.

Despite the flexibility and suitability of our framework for large-scale deployment, an implementation based on it has to address issues such as security, protection and fault-tolerance, if it is to be deployed on the World Wide Web. For example, in order to achieve secure communications, users need to be authenticated and messages may have to be encrypted. We envision using SSL-based [10] CORBA ORBs for this in the future.

The work presented herein demonstrates that it is possible to attain a web-based network simulation framework suitable for large-scale deployment without sacrificing extensibility or flexibility. The prototype described in this paper is being used to evaluate distributed computing policies, and is being itself executed as a simulation on distributed computers. It is envisioned that other such naturally distributable simulations will benefit from a similar approach.

ACKNOWLEDGEMENTS

This research was sponsored by DARPA under contract N66001-97-C-8527 (the Amaranth project), and made use of hardware donated by Intel Corporation.

REFERENCES

- [1] K. Arnold and J. Gosling. 1996. The Java Programming Language. *Addison Wesley*.
- [2] L. Brakmo, A. Bavier, L. Peterson, and V. Raghavan. 1997. *x-Sim User's Manual (V. 1.0)*.
- [3] The Common Object Request Broker Architecture: Architecture and Specification – Revision 2.2. February 1998. *Object Management Group*.
- [4] J. Dahmann, R. Fujimoto, and R. Weatherly. The Department of Defense High Level Architecture. *Proc. of 1997 Winter Simulation Conf.*, pp. 142-149.
- [5] Paul A. Fishwick. December 1996. Web-Based Simulation: Some Personal Observations. *Proc. 1996 Winter Simulation Conf.*, pp. 772-779.
- [6] E. Gamma, R. Helm, R. Johnson, J. Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. *Addison Wesley*, pp. 17-18.
- [7] J. Hansen. 1999. Resource Priority Multiplexing. *To be published in INFOCOMM 2000*.
- [8] INSANE. <http://www.ca.sandia.gov/~bmah/Software/Insane/index.html>, accessed 03/21/1999.
- [9] S. Keshav. 1988. REAL: A Network Simulator. *UCB CS Tech Report 88/472*.
- [10] Netscape Communications Corporation. The SSL Protocol – Version 3.0. March 1996. *IETF - Internet Draft*.
- [11] NetSim^Q. <http://eewww.eng.ohio-state.edu/drcl/grants/middleware97/netsimQ.html>, accessed 03/21/1999.
- [12] B. Oki, M. Pfluegl, A. Siegel, D. Skeen. 1993. The Information Bus - An Architecture for Extensible Distributed Systems. *ACM Symposium on Operating System Principles*.

- [13] E. H. Page, S. P. Griffin, and S. L. Rother. April 1998. Providing Conceptual Framework Support for Distributed Web-Based Simulation within the High Level Architecture. *Proc. of SPIE: Enabling Technologies for Simulation Science II*, pp. 287-292.
- [14] R. Rajkumar, M. Gagliardi, L. Sha. 1995. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. *Proceedings of the 1995 IEEE Real-time Technology and Applications Symposium*.
- [15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. 1991. *Object Oriented Modeling and Design*. Prentice Hall.
- [16] J. Rumbaugh. 1994. The Life of an Object Model: How the Object Model Changes During Development. *Journal of Object-Oriented Programming*. pp. 7(1):24-32.
- [17] C. C. Shen. December 1996. A CORBA Facility for Network Simulation. *Proc. of the 1996 Winter Simulation Conf.*, pp. 613-619.
- [18] A. Thomas. 1998. Enterprise Java Beans Technology: Server Component Model for the Java Platform. *White Paper, Sun Microsystems*.
- [19] UCB/LBNL/VINT Network Simulator. <http://www-mash.cs.berkeley.edu/ns/ns.html>, accessed 03/21/1999.