Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.

A WIDELY DEPLOYABLE WEB-BASED NETWORK SIMULATION FRAMEWORK USING CORBA IDL-BASED APIs

Arjun Cholkar

GTE Data Services
Irving, TX 75038, U.S.A.

Philip Koopman

Department of Electrical and Computer Engineering &
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

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 demonstrates that the CORBA components not only provide language and platform-independence, but also provide the ability for simulationists to connect objects to a third party distributed simulation. By using a uniform messaging approach to all simulation events, objects can be reassigned to different simulation entities without requiring code modifications. Dynamic loading and unloading of objects during a simulation run supports fault simulation, simulation entity polymorphism, and generation of dynamic topologies. A link-scheduling example has demonstrated that our language and platform-independent network simulation framework attains extensibility and flexibility.

1 INTRODUCTION

The World Wide Web (WWW) and the underlying Internet provide a potentially huge distributed computing infra-structure that can be extended to host distributed simula-tion services (Fishwick 1996). With recent developments in web-based simulation technology, the portability and versatility of simulation tools has increased dramatically (Page, Griffin, and Rother 1998). The ubiquity of web access has also created a promising medium for large-scale deployment.

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, and in particular need to be extensible by end users. Furthermore, this flexibility should not compromise on user-comfort 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 toward building web-based network simulation frameworks is to use platform independent object-oriented (OO) technologies such as Java (Arnold and Gosling 1996) and RMI/Enterprise Java Beans (Thomas 1998). These frameworks are portable and extensible, but require that all simulation objects be developed in the single language of 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 uses the Common Object Request Broker Architecture (CORBA) (OMG 1998) 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) (OMG 1998) based Application Programming Interfaces (APIs) and a CORBA Object Request Broker (ORB) (OMG 1998) to provide the necessary location transparency and language independence. These CORBA components enable simulationists to write their own objects on their own platforms, and have their objects participate from their platforms in a remote-simulation hosted by a third party.

All simulation entities (simulated hosts and links) in the framework are composed using stock and user-written objects. A mechanism based on the publisher-subscriber model (Rajkumar, Gagliardi, and Sha 1995) facilitates communication among these objects. All interactions within the framework are message-based and have a uniform mess-

age 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 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 wired and wireless networks.

A prototype application for network simulation frameworks was also implemented as a proof-of-concept exercise. 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.

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, specialized simulators. Simulation frameworks, on the other hand, include specialized tools to simplify the development of domain-specific simulations. Therefore, simulation frameworks may be preferable to generic simulators.

The CORBA-based simulation facility developed at Bellcore (Shen 1996) 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 (Keshav 1988), ns-2 (Fall and Varadhan 1998), INSANE (Mah 1998) and x-Sim (Brakmo, Bavier, Peterson, and Raghavan 1997) provide rich APIs and tools, and have been extensively used by researchers. However, even with support for web-based simulation in REAL, its flexibility is limited because it does not provide simulation support for user-written objects in its web-based mode. This limits its utility for large-scale deployment over the Web.

With the recent demand for web-based network simulations, frameworks such as NetSim^{Q} (Hou, Han and Jain 1998) have been developed. NetSim^{Q} provides an extensible simulation environment and a rich graphical user-interface. A Java implementation gives NetSim^{Q} high portability, but 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 combine the best of both worlds. It builds upon ideas drawn from CORBA-based generic simulators to create a language-independent, distributed network simulation framework. 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.

Our framework stresses providing flexibility by dynamically composing simulation entities. The publisher-subscriber communication model along with a uniform message format aids us in achieving this goal. This communication model is similar to the anonymous communication model (Oki et. al. 1993) 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) (Dahmann, Fujimoto, and Weatherly 1997) 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 counterparts. 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 (Gamma et al. 1995) 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.

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) (Rumbaugh et al. 1991, Rumbaugh 1994).

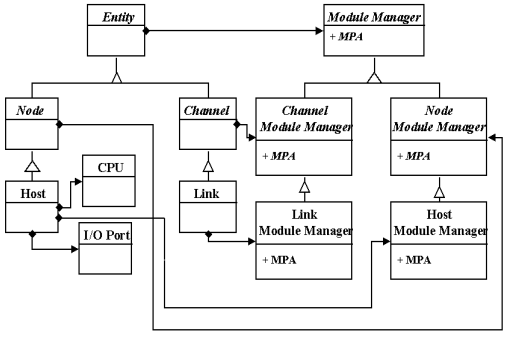

Figure 1: Class Diagram for Simulation Entities

Even with additional components, all entities have a limited set of behavioral properties. Therefore, one or more intelligent *modules* are assigned to each entity to enhance behavior.

To support flexible object composition, entities need a standard way to reference all modules without having to know particular module classes. 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. Figure 2 shows the inheritance hierarchy for a module, where ModuleX and ModuleY are example concrete classes derived from the abstract base class *Module*. A typical host or link consists of a number of assigned modules and an appropriate module manager. Figure 3 shows typical host and link configurations.

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 concrete class details. This is helpful when simulation entities need to be composed dynamically.

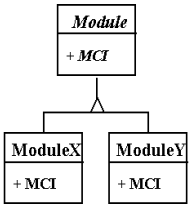

Figure 2: Class Diagram for Modules

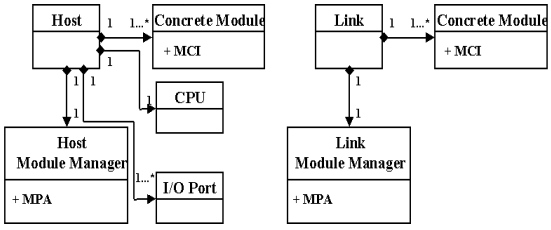

Figure 3: A Typical Host and Link Configuration

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. Modules written in a variety of languages and distributed on a variety of platforms can be dynamically reassigned to different simulation entities while cooperating in a single simulation. CORBA thus provides location transparency and language independence, eliminating the need for complicated inter-process communication code. In addition, it also makes our API highly portable and our framework widely deployable.

3.2.1 The 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.

```
module AmModuleManager
{
  interface AmModulePluginAPI
  {
    long amRegisterPublisher(in AmSimulationDataTypes::AmMessage inMessage);
    long amRegisterSubscriber(in AmSimulationDataTypes::AmMessage inMessage);
    long amSendMessage(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

There are two things worth noting about 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 simulating faults dynamically, changing entity behaviors dynamically, and changing the topology of the simulated network on-the-fly. For example, loading a routing and forwarding module on a host might change its behavior from an end-host to a router. Similarly, a router failure can be simulated by unloading all the modules on a router, causing the simulated network to re-route. This provides a dynamically customizable simulation environment that can simulate a large number of scenarios.

3.2.2 The Module Callback Interface

The CORBA-IDL based Module Callback Interface is shown in Figure 5. This interface has three methods, *amInit*, *amProcessMessage*, and *amDestroy*, which are invoked for module-initialization, message-delivery and module-shutdown respectively. These have a uniform single-argument invocation format, enabling module relocation to different simulation entities with ease.

```
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

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 with scopes limited to each entity. A database of names for existing message-channels is kept by each entity and can be queried by a module to decide on a unique name for its new message-channel. Entities handle message distribution transparently using message-channel names. Thus, an entity acts only as a message-distribution agent for all inter-module messages bearing no knowledge of their contents. This complements our design goals 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. In 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*'.

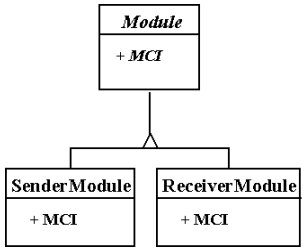

Figure 6: Class Diagrams for Modules SenderModule and ReceiverModule

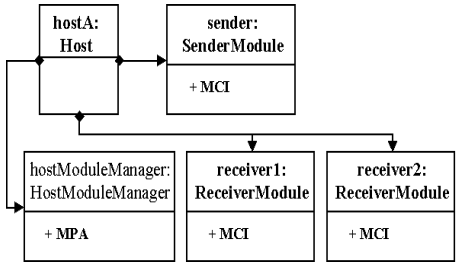

Figure 7: Object Diagram for Host 'hostA'

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.

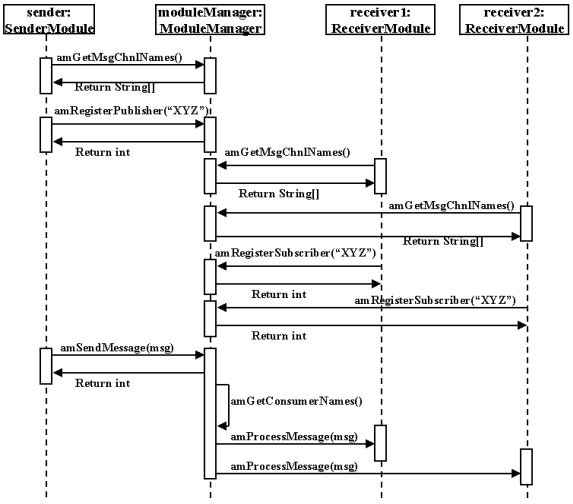

Figure 8: Sequence Diagram for an Example Scenario of Inter-Module Communication

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.

4 PROTOTYPE OVERVIEW

A 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. Additionally, our approach reduces startup costs such as download latencies.

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.

Access to the simulation kernel is via a simulation server, which waits for simulation requests. Simulation requests are generated by user interfaces when a user

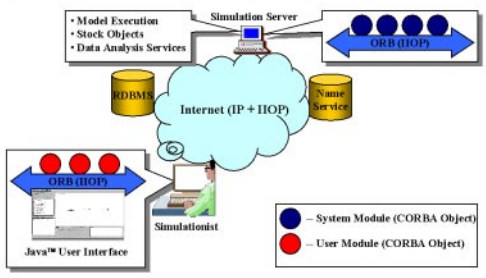

Figure 9: System Architecture of the Prototype

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 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/shutdown of the encapsulated entity, event delivery, instrumentation and injecting asynchronous inputs. These executives are derived from a base class *EntityExecutive* that supports common functionality.

Typical sequences of interactions that take place among 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 *SimulationExecutive*. This specification is then parsed and the appropriate *HostExecutive*s and *LinkExecutive*s 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 need to 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 typically consists of many 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 might be modules written by the user or could be third-party modules conforming to the MCI and the MPA. In order to bootstrap these modules into a simulation, an activation daemon is included in the code shipped with the user interface.

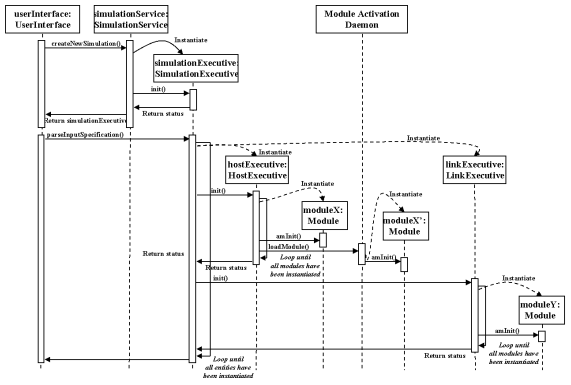

Figure 10: Sequence Diagram for a Typical Startup Scenario

5 EXAMPLE APPLICATION

To exercise the MPA and the MCI, we implemented the behavior of Resource Priority Multiplexing (RPM) (Hansen 1999) modules for scheduling network traffic. We found it straightforward to implement RPM schedulers as modules in our framework, and obtained simulation results matching 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. Due to space limits, we only discuss the RPM scheduler, although the equation solver was also implemented. The RPM scheduler is made up of a *marker* that assigns priorities to packets and a *scheduling mechanism* that transmits these packets. The marker has a built-in state machine having one state per competing flow. The holding times for each state are decided by the RPM equation solver. These holding 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; 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 user-specified via a configuration file, or can be set 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.

(a) Initialization

```
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;
}
```

(b) Message Processing

```
void AmRPMScheduler::amProcessMessage(AmMessage* inMessage)
{
 if(strncmp(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;
 }
}
```

(c) Shutdown

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

Figure 11: MCI Method Implementations for the RPM Scheduler

Figure 11(b) shows the implementation of the *amProcessMessage* method, containing the decision process of the RPM module. The *amClassifyPacket* method classifies and marks packets with an appropriate priority that depends on the state of the state-machine in the *marker*. 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. As 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 and is thus 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 used to generate traffic bound from the sender to the receiver. The traffic profiles for the three modules (or 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. A RPM scheduler was also assigned to 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-levels required by each task.

Table 1: Traffic Profiles for the Modules A, B, and C

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

The results obtained for the network simulation problem are shown in Figure 12 and 13. Figure 12 shows the observed traffic generated by the three modules. It can be seen that this traffic confirms to the traffic profiles of Table 1. The delivered traffic at the receiver and the assurance-levels for the tasks are shown in Figure 13. It is apparent that all three modules get equal to or more than the assurance-levels that they desire. The values actually obtained were 99.16%, 99.28% and 99.58% for tasks A, B and C respectively. Additionally, it can be seen that guaranteeing a service-level of 99%, 99% and 98% for the three modules, which have the potential of generating more traffic than the link can handle, is a non-trivial task.

The RPM module implementation delivered the expected results with a minimum of implementation effort. Thus, we feel that our MPA and MCI are sufficient and useful for implementing even relatively complex modules.

6 CONCLUSIONS AND 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 plus a good design. Previous web-based network simulation frameworks fall short of providing complete language and platform-independence, limiting deployability.

Our approach to enabling large-scale simulation framework 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 simulationists 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. A networking example demonstrated the applicability of this approach to realistic modeling projects.

Despite the flexibility and suitability of our framework for large-scale deployment, an implementation based on it still 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 the future use of SSL-based (Freier, Karlton, and Kocher 1996) CORBA ORBs.

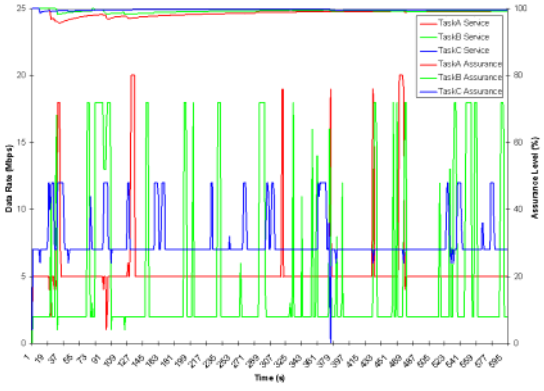

Figure 12: Traffic Generated by Tasks A, B, and C

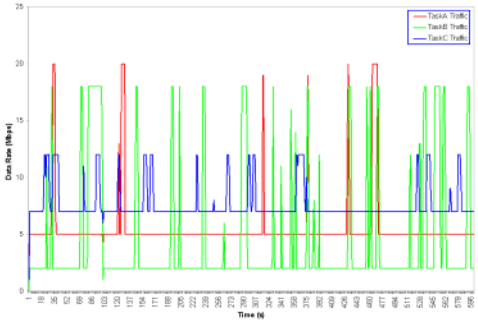

Figure 13: Service Characteristics and Assurance Levels of Tasks A, B, and C

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

Arnold K. and J. Gosling. 1997. *The Java Programming Language*. 2d ed. Addison Wesley Publishing Co.

Brakmo L., A. Bavier, L. Peterson, and V. Raghavan. 1997. `x-Sim` User's Manual [online]. Available from

http://www.cs.arizona.edu/classes/cs525/xsim/xsim.html [accessed 2/12/1999].

Dahmann J., R. Fujimoto, and R. Weatherly. 1997. The Department of Defense High Level Architecture. In *Proceedings of the 1997 Winter Simulation Conference*, 142-149.

Fall K., K. Varadhan. 1998. `NS` [software online]. University of California, Berkeley, CA. Available from http://www-mash.cs.berkeley.edu/ns/ns.html [accessed 03/21/1999].

Fishwick P. A. 1996. Web-Based Simulation: Some Personal Observations. In *Proceedings of the 1996 Winter Simulation Conference*, 772-779.

Freier A., P. Karlton, and P. Kocher. 1996. The SSL Protocol – Version 3.0. Internet Draft.

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

Hansen J. 1999. Resource Priority Multiplexing. ICES, Carnegie Mellon University, Pittsburgh, PA.

Hou C., C. Han, and R. Jain. 1998. `NetSim`^{Q} [online]. Ohio State University. Available from http://eewww.eng.ohio-state.edu/drcl/grants/middleware97/netsimQ.html [accessed 03/21/1999].

Keshav S. 1988. REAL: A Network Simulator. UCB CS Tech Report 88/472. University of California, Berkeley, CA.

Mah B. 1998. `INSANE` [software online]. Sandia National Labs. Available from http://www.ca.sandia.gov/~bmah/Software/Insane/index.html
[accessed 03/21/1999].

Object Management Group (OMG). 1998. *The Common Object Request Broker Architecture: Architecture and Specification – Revision 2.2*.

Oki B., M. Pfluegl, A. Siegel, and D. Skeen. 1993. The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the 1993 ACM Symposium on Operating System Principles*.

Page E., S. Griffin, and S. Rother. 1998. Providing Conceptual Framework Support for Distributed Web-Based Simulation within the High Level Architecture. In *Proceedings of SPIE: Enabling Technologies for Simulation Science II*, 287-292.

Rajkumar R., M. Gagliardi, L. Sha. 1995. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proceedings of the 1995 IEEE Real-time Technology and Applications Symposium*.

Rumbaugh J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. 1991. *Object Oriented Modeling and Design*. Prentice Hall.

Rumbaugh J. 1994. The Life of an Object Model: How the Object Model Changes During Development. *Journal of Object-Oriented Programming*. 7(1):24-32.

Shen C. 1996. A CORBA Facility for Network Simulation. In *Proceedings of the 1996 Winter Simulation Conference*, 613-619.

Thomas A. 1998. Enterprise Java Beans Technology: Server Component Model for the Java Platform. White Paper, Sun Microsystems.

AUTHOR BIOGRAPHIES

ARJUN CHOLKAR is a Systems Architect at GTE Data Services. He holds an MS degree in computer engineering from Carnegie Mellon University and his interests lie in distributed system design and development.

PHILIP KOOPMAN is an Assistant Professor of Electrical and Computer Engineering at Carnegie Mellon University. He holds a Ph.D. in computer engineering, and has had a variety of experiences with distributed embedded computer systems in industry and the military.