

# DISCRETE-EVENT SIMULATION ON THE WEB

Chien-Chung Shen

Bellcore

Network Control and Management Systems Department  
Red Bank, New Jersey

## Abstract

In this paper, we describe a mechanism of providing discrete-event simulation facility on the Web. The mechanism calls for integrating the CORBA and Java technologies – simulation applications (models) written in Java access discrete-event simulation capability through a CORBA interface. The discrete-event simulation facility is specified in CORBA IDL which defines operations for object definition, inter-object communication, and event scheduling. The IDL interface definition may be implemented in Java by using different simulation algorithms, over different operating systems or hardware platforms. Simulation applications, developed as stand-alone Java applications or Java applets running within Web browsers, access the facility via the Internet Inter-ORB Protocol (IIOP) to obtain runtime support for model execution. The paper presents the discrete-event simulation facility IDL interface, describes its prototype implementation using Java and OrbixWeb, and illustrates its application via a producer-consumer example written in Java.

## 1 INTRODUCTION

Since its introduction six years ago, the World Wide Web has become the world's largest infrastructure providing ubiquitous information access. With its encompassment of the Java technology, the Web may well be the largest computing infrastructure ever. However, building distributed applications is cumbersome and difficult with the current Web technologies. For example, technology like HTTP was originally developed to reduce the inefficiencies of the FTP protocol. It focuses on interaction with the 'user' rather than on 'application' interaction, which imposes fundamental limitations on the nature of distributed services accessible from applications running within Web browsers. Also, the limitations of CGI-based solutions are soon reached once applications move away from simple query to complex, distributed computing. In addition, mechanisms like Remote Method Invocation (RMI) for Java objects come with the limitation of a single-language environment.

In this paper, we describe a mechanism of providing discrete-event simulation facility on the Web. The mechanism calls for integrating the CORBA [1] and the Java [2] technologies – simulation applications (models) written in Java access discrete-event simulation capability through a

CORBA interface. The discrete-event simulation facility is specified in CORBA IDL which defines operations for object definition, inter-object communication, and event scheduling. The IDL interface definition may be implemented in Java by using different simulation algorithms, over different operating systems or hardware platforms. Simulation applications, developed as stand-alone Java applications or Java applets running within Web browsers, access the facility via the Internet Inter-ORB Protocol (IIOP) to obtain runtime support for model execution. The proposed approach has the following advantages:

- Simulation models written in Java are target platform independence, which may be dynamically downloaded and executed on demand. This ensures the widest possible development of simulation applications and frees the application developers from target operating system and hardware platform considerations.
- The CORBA-based discrete-event simulation facility provides a standardized interaction paradigm for simulation applications which may be viewed as collections of interacting objects. In addition, implementation details of the simulation facility will be encapsulated by the interface, and not be constrained by simulation applications.

The remainder of the paper is organized as follows. In Section 2, message-based discrete-event simulation is first introduced. Based on that, we describe a discrete-event simulation facility and present its CORBA IDL interface definition. Section 3 describes its Java implementation. Application of the facility using a bounded-buffer producer-consumer example is described in Section 4, and Section 5 is the conclusion.

## 2 SIMULATION FACILITY

We adopt the message-based approach to discrete-event simulation [3]. The approach provides a more natural paradigm for simulating distributed systems, and therefore better serves as the foundation for the simulation facility.

In message-based simulation, each physical entity is abstracted by an *logical object* (*lo*), and interactions among the entities, called *events*, are represented by message communications among the corresponding *lo*<sup>1</sup>. A message-based sim-

<sup>1</sup>Symbol *lo* represents both singular and plural forms.

ulation algorithm uses two data structures [4]: a *simulation clock* and an *event-list*. The simulation clock gives the time up to which the physical system has been simulated. The event-list is a partial order of tuples; a tuple is represented by  $(m, s, d, t)$ , where  $m$  represents a message,  $s$  and  $d$  are the source and destination *lo* for  $m$ , and  $t$  is a timestamp. The partial order is typically based on the timestamp and ensures that events are simulated in the order of their dependencies. In general, at every step of the simulation, the algorithm selects the tuple with the *smallest* timestamp, say  $(m_i, s_i, d_i, t_i)$ , removes it from the event-list, and delivers  $m_i$  to  $d_i$ . Multiple tuples with the same timestamp may be handled in an arbitrary order, or be ordered deterministically using transparent sequence numbers to reflect their dependencies. However, an object may delay acceptance of a message based on its state such that messages are not necessarily delivered in the partial order specified by the event-list. Each *lo* may specify a *wait-condition* which restricts the messages that it is willing to receive; a message is delivered to the destination *lo* only if it satisfies its wait-condition. The simulation of  $m_i$  by *lo*  $d_i$  may generate additional messages which are added to the event-list.

```

clock = 0;
Initialize event-list;
while (execution not terminated) do
{  fetch next tuple  $(m_i, s_i, d_i, t_i)$  from event-list;
   if  $(m_i$  is not accepted by  $d_i)$  then
       store  $m_i$  in temp-queue;
   else
       {  if  $(m_i$  is a timeout message) then  $clock = t_i$ ;
          deliver  $m_i$  to  $d_i$  for simulation;
          merge temp-queue with event-list;
        }
}

```

Figure 1: Message-Based Simulation Algorithm

During the execution of a simulation program, the simulation clock advances in a *monotonic non-decreasing* manner through the timestamps associated with each tuple. Note that the simulation clock is completely decoupled from the physical processor clock. The physical time required to simulate a message does not have any effect on the simulation clock. How is the timestamp assigned to a message? When a message is generated, it is timestamped with the current value of the simulation clock — with one exception. A special *timeout* message is defined. The timeout message is scheduled by an *lo* for delivery to *itself* at a *future* time and is typically used to simulate the time of a simulation step that would be required by the physical entity to execute the corresponding operational step. An *operational step* refers to the statements executed by a physical entity to process a message received by it, and a *simulation step* models the activities that would be executed by the corresponding operational step. For example, consider a file-handler entity. On receiving a *read* request for the file, a physical (opera-

tional) file-handler will read the appropriate record from the file and return it to the requesting entity. If the file-handler is abstracted by an *lo*, on receiving a *read* request, the *lo* estimates  $t$ , the time required for the corresponding physical entity to read the file, and schedules a timeout message to itself  $t$  time units later. As the timestamp on all messages other than the timeout message refers to the current value of the simulation clock, the simulation time advances only when a timeout message is delivered to an *lo*. Figure 1 shows the described simulation algorithm.

In light of the above description, we define the DESFacility interface in CORBA IDL, as shown in Figure 2, for message-based discrete-event simulation. The facility consists of operations for object definition, inter-object communication and event scheduling.

```

typedef string  ObjName;
typedef string  MsgType;
typedef any     MsgContents;
typedef long    Time;
typedef boolean BGuard;

struct WCItem {
    MsgType  msg_type;
    BGuard   bguard;
    Time     duration;
};
typedef sequence<WCItem> WaitCondition;

struct Message {
    ObjName  source;
    ObjName  sink;
    MsgType  msg_type;
    MsgContents contents;
};

interface DESFacility {
    void role(in ObjName myself);
    void enroll(in ObjName myself);
    void resign(in ObjName myself);
    void send(in Message msg);
    void receive(out Message msg,
                 in ObjName myself);
    void waituntil(out Message msg,
                   in ObjName myself,
                   in WaitCondition wc);
    void hold(in Time tm, in ObjName myself);
    Time now();
};

```

Figure 2: Interface Definition (DESFacility.idl)

**Object Definition.** Operations *role*, *enroll*, and *resign* are used to define participating simulation objects, and to demarcate their beginning and ending of execution.

**Inter-Object Communication.** Objects communicate with each other using buffered message-passing (asyn-

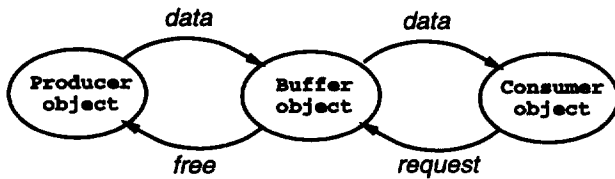


Figure 3: Producer-Consumer Example

chronous communication), where every object has a unique message-buffer. An object sends a message to another by invoking a `send` operation. Each message, with structure defined by `Message`, contains sender, receiver, message type, and message contents information. A message is deposited in the message-buffer of its destination object on the invocation of a `send` operation, and carries a timestamp which corresponds to the simulation time at which the corresponding `send` operation is invoked. An object accepts messages from its message buffer by invoking a `waituntil` operation. The `waituntil` operation takes a sequence of wait-condition items as an input argument to determine which message will be accepted by the object. Each wait-condition item specifies a message-type, say  $m_i$  and a boolean value, say  $b_i$ , which is said to be *enabled* if the message buffer contains a message of type  $m_i$  and  $b_i$  is true, and  $m_i$  is called an *enabling* message. The enabling message with the earliest timestamp is removed and delivered to the object. If all wait-condition items are disabled, the object is suspended for a *maximum* duration of *simulation* time equal to *duration* of a wait-condition item with `msg_type` equal to `timeout`. If omitted, a default wait-condition item with `msg_type` equal to `timeout` is set with an arbitrarily large *duration* value. If no enabling message is received in the *duration* interval, the object is sent a `timeout` message. An object must accept a `timeout` message that is sent to it. A separate `receive` operation is also provided to accept the message with the earliest timestamp from the message-buffer, regardless of its message type. It will block the invoking object until a message is available.

**Event Scheduling.** The `hold` operation enables the invoking object to schedule a timeout message for delivery to itself `tm` time units from now. It is used to specify the simulation time used by a simulation step. The `now` operation lets the invoking object to read the current value of the simulation clock.

### 3 IMPLEMENTATION

In this section, we describe a Java implementation of the simulation facility using the commercial Java ORB, OrbixWeb<sup>2</sup> [5].

In the Java implementation, the `DESFacility` interface is implemented as a Java class which implements the defined operations. Upon receiving an invocation for operation `send`, a thread is created implicitly to insert the message into the event list according to increasing timestamp order. On receiving an invocation for operations `waituntil` and `receive`,

```
package DESpackage;
```

```
public class java_producer {
    public static _DESFacilityRef p = null;
    public static void main(String args[]) {
        Message msg_s, msg_r;
        int i, n, now, t_wait;
        p = DESFacility._bind("sim:simSrv");
        p.enroll("producer");
        now = 0; t_wait = 0; n = Q_LENGTH;
        msg_s.source = "producer";
        msg_s.sink = "buffer";
        msg_s.msg_type = "data";
        for (i = 0; i < MAX_ITEM; i++) {
            if (n == 0) {
                now = p.now();
                p.receive(msg_r, "producer");
                n++; t_wait += p.now() - now;
            }
            p.hold(my_rand(), "producer");
            n--; p.send(msg_s);
        }
        now = p.now();
        System.out.println("Prod utilization = " +
            ((float)(now - t_wait)/(float) now));
        p.resign("producer");
    }
}
```

Figure 4: The Producer Code

a thread is created implicitly to check if there is any enabling message. It will block the invoking object until an enabling message becomes available. Finally, upon receiving an invocation for operation `hold`, a thread is created implicitly to first deposit a timeout message in the event list, and then block until the scheduled timeout message becomes deliverable.

As simulation facility is defined in CORBA IDL, it does not dictate its implementation. The simulation facility could very well be implemented using a parallel simulation algorithm to take advantage of parallel processing environment. With respect to its clients, the implementation is totally transparent, except for potential performance improvement.

### 4 AN EXAMPLE

As an example, we consider the simulation of a *bounded-buffer producer-consumer* system [6], as shown in Figure 3. The consumer object repeatedly requests a data item from the buffer using a `request` message, waits to receive a `data` message, and invokes a `hold` operation to simulate data consumption. The producer object waits to receive a `free` message from the buffer which indicates that the buffer has a free slot, invokes a `hold` operation to simulate the generation of a data item, and sends the item to the buffer via a `data`

<sup>2</sup>OrbixWeb is a Registered Trademark of IONA Technologies Ltd.

```

package DESpackage;

public class java_consumer {
    public static _DESFacilityRef c = null;
    public static void main(String args[]) {
        Message msg_s, msg_r;
        int i, t, t_used, now;
        c = DESFacility._bind("sim:simSrv");
        c.enroll("consumer");
        now = 0; t_used = 0;
        msg_s.source = "consumer";
        msg_s.sink = "buffer";
        msg_s.msg_type = "request";
        for (i = 0; i < MAX_ITEM; i++) {
            c.send(msg_s);
            c.receive(msg_r, "consumer");
            t = my_rand();
            t_used += t;
            c.hold(t, "consumer");
        }
        now = c.now();
        System.out.println("Cons utilization == " +
            ((float)(t_used) / (float)now));
        c.resign("consumer");
    }
}

```

Figure 5: The Consumer Code

message. The buffer object repeatedly invokes a `waituntil` operation that accepts a `request` message only when it is not empty and a `data` if it is not full.

Figures 4, 5, and 6 list the Java source code for the producer, consumer, and buffer object, respectively, and Figure 7 depicts their interaction with the `DESFacility` object to obtain the discrete-event simulation service.

## 5 CONCLUSION

The astonishing growth of the Web may turn itself into the largest computing infrastructure. However, building distributed applications is cumbersome and difficult with the current Web technologies. In this paper, we describe a mechanism of providing discrete-event simulation facility on the Web. The mechanism calls for integrating the CORBA and Java technologies – simulation applications written in Java access discrete-event simulation capability through a CORBA interface. The Web-based simulation facility demonstrates the versatility of CORBA and Java integration. It also serves as an example for other Web-based distributed applications to be developed in the near future.

## References

[1] *The Common Object Request Broker: Architecture and*

*Specification*. Object Management Group and X/Open, 1993.

- [2] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.
- [3] R. L. Bagrodia, K. M. Chandy, and J. Misra, "A message-based approach to discrete-event simulation," *IEEE Transactions on Software Engineering*, 13(6):654–665, 1987.
- [4] J. Misra, "Distributed discrete-event simulation," *Computing Survey*, 18(1):39–65, 1986.
- [5] *OrbixWeb: Programming Guide*. IONA Technologies Ltd., 1996.
- [6] R. L. Bagrodia and C.-C. Shen, "MIDAS: Integrated design and simulation of distributed systems," *IEEE Transactions on Software Engineering*, 17(10):1042–1058, 1991.
- [7] C.-C. Shen, "A CORBA facility for network simulation," *Winter Simulation Conference*, Coronado, 1996.

```

package DESpackage;

public class java_buffer {
    public static _DESFacilityRef b = null;
    public static void main(String args[]) {
        Message msg_s_p, msg_s_c, msg_r;
        WaitCondition wc(2);
        int q, i, v, tstart, tend;
        b = DESFacility._bind("sim:simSrv");
        b.enroll("buffer"); tstart = b.now();
        msg_s_p.source = "buffer";
        msg_s_p.sink = "producer";
        msg_s_p.msg_type = "free";
        msg_s_c.source = "buffer";
        msg_s_c.sink = "consumer";
        msg_s_c.msg_type = "data";
        wc.buffer[0].msg_type = "data";
        wc.buffer[1].msg_type = "request";
        i = 0; q = 0; v = 0;
        while (true) {
            wc.buffer[0].bguard = (q < Q_LENGTH);
            wc.buffer[1].bguard = (q > 0);
            b.waituntil(msg_r, "buffer", wc);
            if (msg_r.msg_type == "data") {
                tend = b.now(); v += q*(tend-tstart);
                tstart = tend; q++;
            } else {
                b.send(msg_s_c); tend = b.now();
                v += q*(tend-tstart); tstart=tend; q--;
                if ((++i) < MAX_ITEM)
                    b.send(msg_s_p);
                else
                    break;
            }
        }
        System.out.println("avg item # in buf: " +
            (float) v / (float) b.now());
        b.resign("buffer");
    }
}

```

Figure 6: The Buffer Code

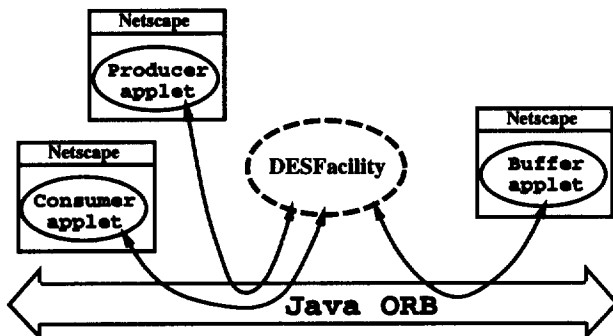


Figure 7: Implementation