Tool Integration for Flexible Simulation of Distributed Algorithms

Shashank Khanvilkar[1,†] and Sol M. Shatz[2,*,‡]

Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, IL.

[1]Shashank Khanvilkar, Univ. of Illinois at Chicago, Department of Electrical Engineering and Computer Science, 851 S. Morgan (M/C 154), Room 1007 Science and Engineering Offices Bldg., Chicago, IL, 60607-7053.

[†]Email: shashank@evl.uic.edu

[2,*]Correspondence to: Prof. Sol M. Shatz, Univ. of Illinois at Chicago, Department of Electrical Engineering and Computer Science, 851 S. Morgan (M/C 154), Room 1120 Science and Engineering Offices Bldg., Chicago, IL, 60607-7053.

[‡]Email: shatz@eecs.uic.edu

SUMMARY

Over the last two decades, considerable research has been done in Distributed Operating Systems, which can be attributed to faster processors and better communication technologies. A distributed operating system requires distributed algorithms to provide basic operating system functionality like *mutual exclusion*, *deadlock detection,* etc. A number of such algorithms have been proposed in the literature. Traditionally, these distributed algorithms have been presented in a theoretical way, with limited attempts to simulate actual working models. This paper discusses our experience in simulating distributed algorithms with the aid of some existing tools, including OPNET and Xplot. We discuss our efforts to define a basic model-based framework for rapid simulation and visualization, and illustrate how we used this framework to evaluate some classic algorithms. We have also shown how the performance of different algorithms can be compared based on some collected statistics. To keep the focus of this paper on the approach itself, and our experience with tool integration, we only discuss some relatively simple models. Yet, the approach can be applied to more complex algorithm specifications.

KEY WORDS

Distributed algorithms, modeling, simulation, and tool integration.

1. INTRODUCTION

Over the last two decades, considerable research has been done in Distributed Operating Systems (DOS), which can be attributed to faster processors and better communication technologies [1, 2]. A distributed operating system requires distributed algorithms to provide basic operating system functionality like *mutual exclusion*, *deadlock detection,* etc. [3-5] are surveys for algorithms on distributed deadlock detection, distributed mutual exclusion and distributed termination detection. Study of these algorithms is a key topic for students in courses on Advanced Operating Systems and Distributed Computing. Because such algorithms exhibit subtle and non-deterministic behavior, simulation can be a valuable tool. It can be particularly helpful to students that are beginning to learn about distributed algorithms since simulation provides an opportunity for hands-on experience with the specific operations and process interactions associated with the algorithm or protocol under study. In addition, the student can use simulation as a means to compare the efficiency of one algorithm over another. Finally, simulation can allow the student to test how an algorithm will react to different fault tolerance issues, such as node/link failures, which are typically not discussed in detail in existing textbooks. However, it is a difficult proposition to simulate such distributed algorithms as they communicate through message passing between two or more processes on the same or different machines. This paper presents our experience in integrating some existing tools to provide for a simple, yet flexible and effective, method for simulation and animation (by graphical display of messages) of such algorithms. This method is driven by models for the various system components. The discussion of examples shows how this method/tool can be used. We found that our approach encourages re-use of models in terms of using simpler models as a basis for building more complex models – this can make the development of distributed algorithms faster and simpler. As such the tool described may also be useful to developers/researchers of new distributed algorithms. Note that this work concentrates on the simulation of distributed algorithms. Issues on distributed simulation, which deals with making an event driven simulation faster and efficient by distributing it across different machines [6], are beyond the scope of this work.

Our study began with one of the classic algorithms, Lamport's Logical Clocks (LLC) [1]. This is a simple algorithm that is used to establish causal ordering of events occurring in different processes. To implement this algorithm, we first considered the conventional process of using C or Java. However, when we actually started to design the code, we realized that it was much more complicated. For example, in C we tried using BSD socket system calls. But, even the normal Client-Server model implemented using sockets could not be used for this algorithm, since the calls to send() (for sending data) and receive() (for receiving data) are blocking, which means that if the process is sending or receiving messages, it gets blocked and disallows simultaneous *send* and *receive* of data. We considered using the fork() and select() system calls to work around this problem, but the solution was becoming complicated and difficult to debug.

The second language that we considered using was Java. We even found some references [7] [8] that discuss how Java can be used to develop distributed algorithms. Although the solutions provided in these papers can be effectively used to simulate distributed algorithms for class projects, we felt that they were not flexible enough to allow students to experiment more with different scenarios or with different fault tolerance issues, or even compare different algorithms. So these approaches really did not meet our requirements.

This motivated us to seek for a more general framework that would allow us to easily simulate different distributed algorithms, study their behaviors and compare their performances. What we needed was an environment in which we could rapidly and flexibly test various algorithms and graphically view their fundamental message-passing behavior. We were able to find (via the Internet) some toolkits that can be used to simulate distributed algorithms. The Lydian [9] system is the closest, in goal, to our work.

Lydian is specifically designed for an educational environment. It comes with a large database of common distributed algorithms that are covered in graduate level courses across different universities. It has built in support for animating the working of different algorithms over different network scenarios, which can help students to better understand their working. Lydian allows newer algorithms to be developed on the fly, by asking the user some specific questions. However, in order to build animations for such new algorithms, one has to learn a separate tool called POLKA. In addition, Lydian is a custom tool, whereas the focus of our approach is the integration of existing component tools. Another difference is that Lydian is not oriented for experiments that compare the performance of different distributed algorithms, because it cannot easily collect statistical results for varying parameters of interest. Since our approach is focused on being more flexible and less tutorial, it provides a complementary alternative to the Lydian system.

Other toolkits that we identified on the Internet are DAJ [10], ViSiDiA [11], and Distributed BACI [12]. However, these tools tend to be somewhat special-purpose. Like Lydian, they do not exploit the features of other existing tools that can easily facilitate flexible simulation and graphical display of captured message sequences. Thus, to pursue a model-based approach (in contrast to a code-based approach) to provide for rapid algorithm specification, we decided to try OPNET^{tm} [13]. OPNET is a commercially available modeling and simulation tool that is traditionally used to simulate computer networks. It comes with a number of built-in models for terminals, routers, servers, etc. Using these models, one can easily simulate almost any kind of network, and analyze any protocol. Most importantly, new protocols can easily be developed using the familiar finite-state modeling concept, as provided by OPNET. In fact, this was the key feature that motivated us to adopt the OPNET tool as the front-end of our simulation approach. Another tool providing similar functionalities is Network Simulator (NS) [14], which is freeware. It appears that all the basic models that we have developed using OPNET can also be developed, with equal ease, in NS.

For any distributed algorithm, visualization of the interaction between peer processes aids in understanding the algorithm [7]. Visualization is based on suitable graphic display of process events that are gathered during some experimentally driven runs of the algorithm under consideration. While OPNET does provide a visualization tool, as a separate module called ACE (Application Characterization Environment), the capability is primarily used to plot packet traces, collected using third party tools like tcpdump or Sniffer, in networks based on Internet Protocol (IP). Since our simulation was not based on IP, we did not use ACE. Also we wanted to keep the visualization part as a separate entity, so that it can also be used in a real (not simulation) environment without any major changes. So, our approach is based on the simple idea of using OPNET to generate simulation-trace log files of different events (sending messages, receiving messages), and then use some standard plotting tool to interpret the log files and display the message sequences.

For the graphical display function, we could have used Matlab or Microsoft Excel, but we settled on a tool called Xplot [15], mainly because of its flexibility and simplicity of use. Xplot is a general-purpose tool developed by Tim Shepard of MIT. It is very efficient in plotting graphs; any type of graph can be plotted and any portion of the graph can be expanded. Interfacing to Xplot is facilitated by the fact that Xplot requires input in the form of an IDL (Interface Definition Language). A sample IDL is described later.

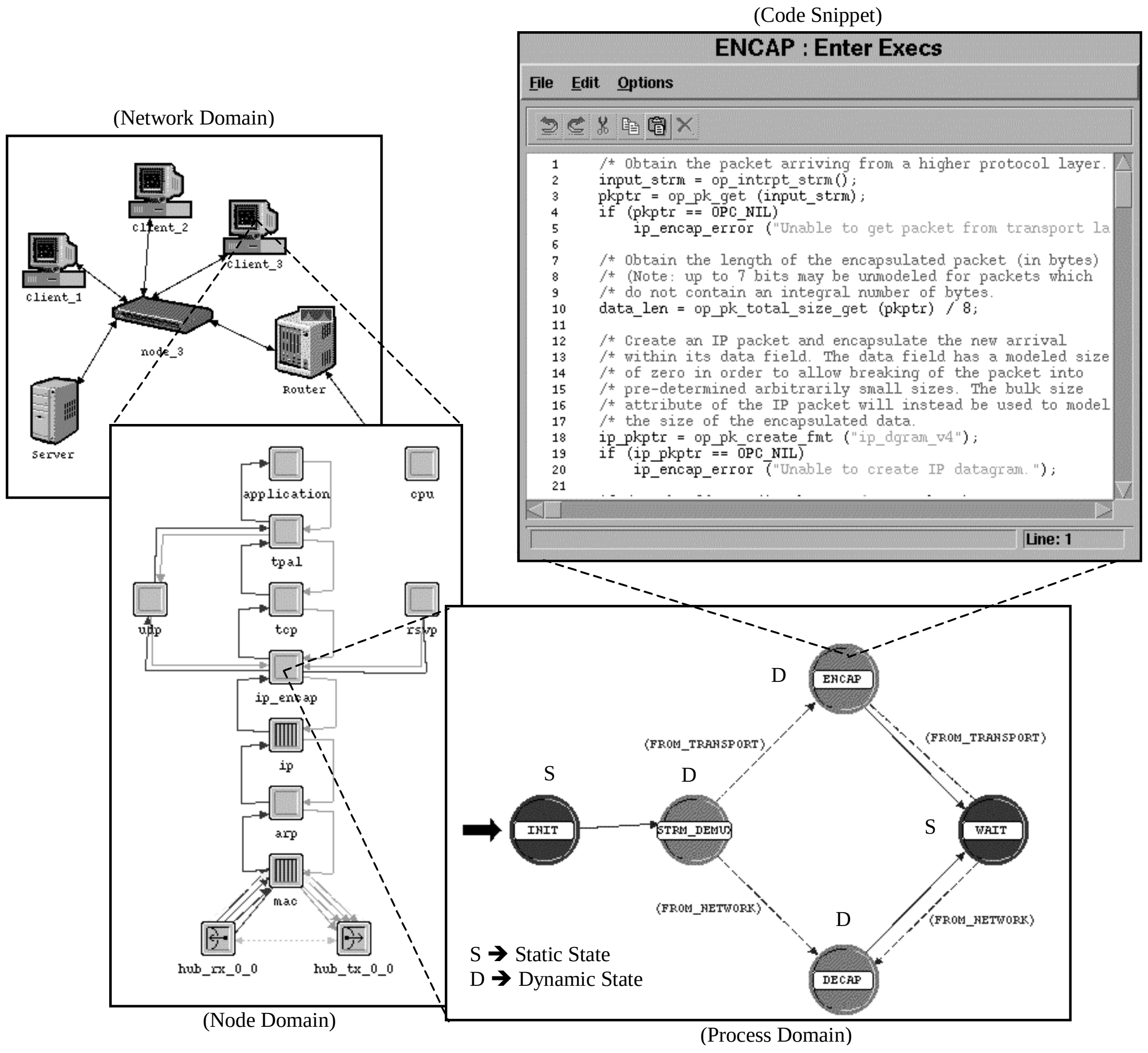
Using the identified tools, we can demonstrate how we crafted a basic distributed architecture model and used the model to develop simulations and animations for the LLC algorithm, along with other distributed algorithms. Our main objective is to introduce the readers to a new way of simulating distributed algorithms, which we feel is quite effective since it exploits to a great degree the capabilities of some already available tools and thus avoids the common tendency to reinvent the wheel.

Every model that we have used in this paper required only a few hours to develop. In order to apply our approach, the user must have OPNET modeler 7.0.B and Xplot installed on a system. The user should also have some basic knowledge of OPNET, concerning how to design a Finite State Machine model for a process and how to run a simulation on OPNET. This information can be easily obtained by referring to the tutorial information that is provided with the OPNET system. No preliminary knowledge about Xplot is necessary.

The remainder of this paper is organized as follows: Section 2 begins with a general overview of the basic framework that we have built in OPNET that is common for all simulations. Only a brief explanation is given, without delving too deep into the actual language constructs. Section 3 gives an introduction to Xplot, with an example on how to provide an input file to Xplot using IDL, and our custom program (Log2Xpl), which is a translator utility that transforms log file data generated by the OPNET models into an appropriate Xplot IDL format. Section 4 discusses a simulation of the LLC algorithm along with other basic distributed algorithms, like Vector Clocks and some mutual exclusion algorithms. We have also given an example describing how performance of different algorithms can be compared at the end of section 4. Finally Section 5 provides a conclusion and discusses future research objectives.

2. SYSTEM MODEL OVERVIEW

OPNET provides a three layer modeling hierarchy. Fig. 1 illustrates the basic idea, but the reader should not be concerned yet with any of the details in this figure. The highest layer, referred to as the *network domain*, allows definition of network topologies, like Ethernet LAN and STAR network, using standard models. The second layer, referred to as the *node domain*, allows definition of node architectures. This defines all the processes active in a node and the interactions between them. It should be noted that processes within the same node run *asynchronously* (i.e., they do not run in lock-step) but synchronize based on the exchange of messages (or packets), through “packet streams” defined between them. “Packet streams” are abstractions of Pipes, commonly used in C. (In Fig. 1, under “Node Domain”, processes are denoted by gray square boxes, while packet streams are denoted by solid arrows). The third layer is called the *process domain*. This layer specifies a Finite State Machine (FSM) for every process within the node. The FSM consists of dynamic and static states. Every state has some associated procedures, which get executed every time the process enters (“Enter Executives”) or exits (“Exit Executives”) the state. A transition from one state to another state takes place in response to certain events (in the form of an interrupt) and these transitions can be conditional (denoted by dotted arrows) or unconditional (denoted by solid arrows). A process can remain in a static state, waiting for an event to occur. But, a process cannot remain in a dynamic state; the process must transition to some static state after executing the corresponding “Enter” and/or “Exit” procedures. This is a fairly standard FSM characteristic. We will now describe the basic models that we have designed for our simulations starting with the network topology defined in the network domain.



```
/* Obtain the packet arriving from a higher protocol layer.
input_strm = op_intrpt_strm();
pkptr = op_pk_get (input_strm);
if (pkptr == OPC_NIL)
    ip_encap_error ("Unable to get packet from transport la

/* Obtain the length of the encapsulated packet (in bytes)
/* (Note: up to 7 bits may be unmodeled for packets which
/* do not contain an integral number of bytes.
data_len = op_pk_total_size_get (pkptr) / 8;

/* Create an IP packet and encapsulate the new arrival
/* within its data field. The data field has a modeled size
/* of zero in order to allow breaking of the packet into
/* pre-determined arbitrarily small sizes. The bulk size
/* attribute of the IP packet will instead be used to model
/* the size of the encapsulated data.
ip_pkptr = op_pk_create_fmt ("ip_dgram_v4");
if (ip_pkptr == OPC_NIL)
    ip_encap_error ("Unable to create IP datagram.");

```

Fig. 1: Three layer modeling hierarchy provided by OPNET.

2.1. The Network topology

Fig. 2 shows the network topology that we used throughout our study. Although only four terminal nodes are shown, we developed our model to support a maximum of 16 terminals. The limitation is imposed by the design of the hub, which can be modified to accommodate more terminals if desired. Each terminal (Terminal-0, Terminal-1, etc.) is connected to the hub, using full duplex, point-to-point links in a star like configuration. The hub can unicast as well as broadcast packets, by the setting of a switch. Every terminal is assigned an address. For example, “Terminal-0” is assigned an address of 0; “Terminal-1” is assigned an address of 1, and so on. This topology represents the most common configuration that is in use today, and can be changed with some minor modifications.

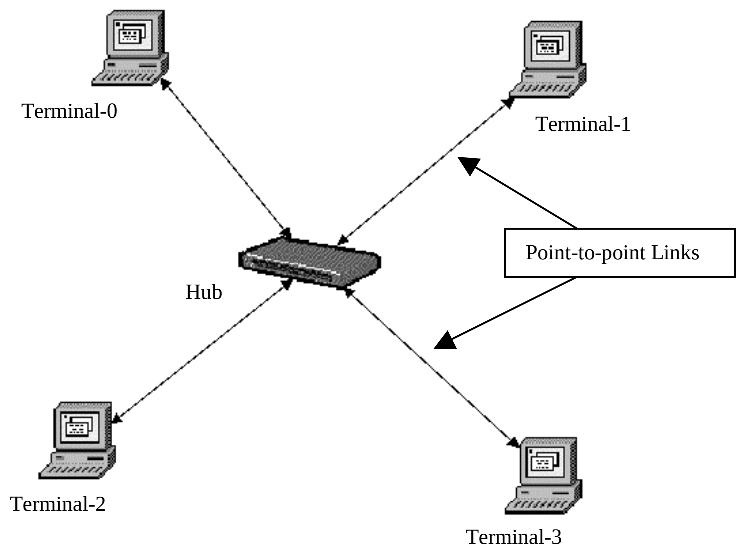


Fig. 2. General Network Topology.

2.2. Packet format of the messages passed between peer processes.

The processes, in different terminals, participating in a distributed algorithm communicate with each other using messages. Each message contains the following fields:

- *Destination Node Address (dest_address)*: Specifies address of the terminal for which the message is intended.
- *Source Node Address (src_address)*: Specifies address of the terminal originating the message.
- Sequence Number: Maintains a count of the number of messages generated by a particular terminal.

Some other optional fields are present, which are used depending on the specific distributed algorithm under simulation. The message format is defined using the *Packet Format Editor* provided by OPNET. OPNET provides simple API’s that can be used to perform different operation on each field in the message.

2.3.Link model.

As mentioned earlier, in Section 2.1, the four terminals are connected to the hub using full duplex point-to-point links. These links are designed using OPNET’s Link Model Editor, which allows the user to vary different attributes of the link including data rate, bit error rate, propagation delay, etc. By varying these attributes, the effect of different fault tolerance issues (like excessive message delays) can be studied.

2.4. Node and FSM models

We now define the node and FSM models that we designed for the hub and the terminals. The FSM model is described for only the relevant processes.

2.4.1. Hub

The hub acts as a packet switch, responsible for routing packets to the appropriate destination terminal, based on the *dest_address* field in the packet. *(Refer to "Node Model for Hub" in Fig. 3)*. Whenever a packet enters the hub, a process in the hub (Hub Process) examines the packet and then sends it to the appropriate destination.

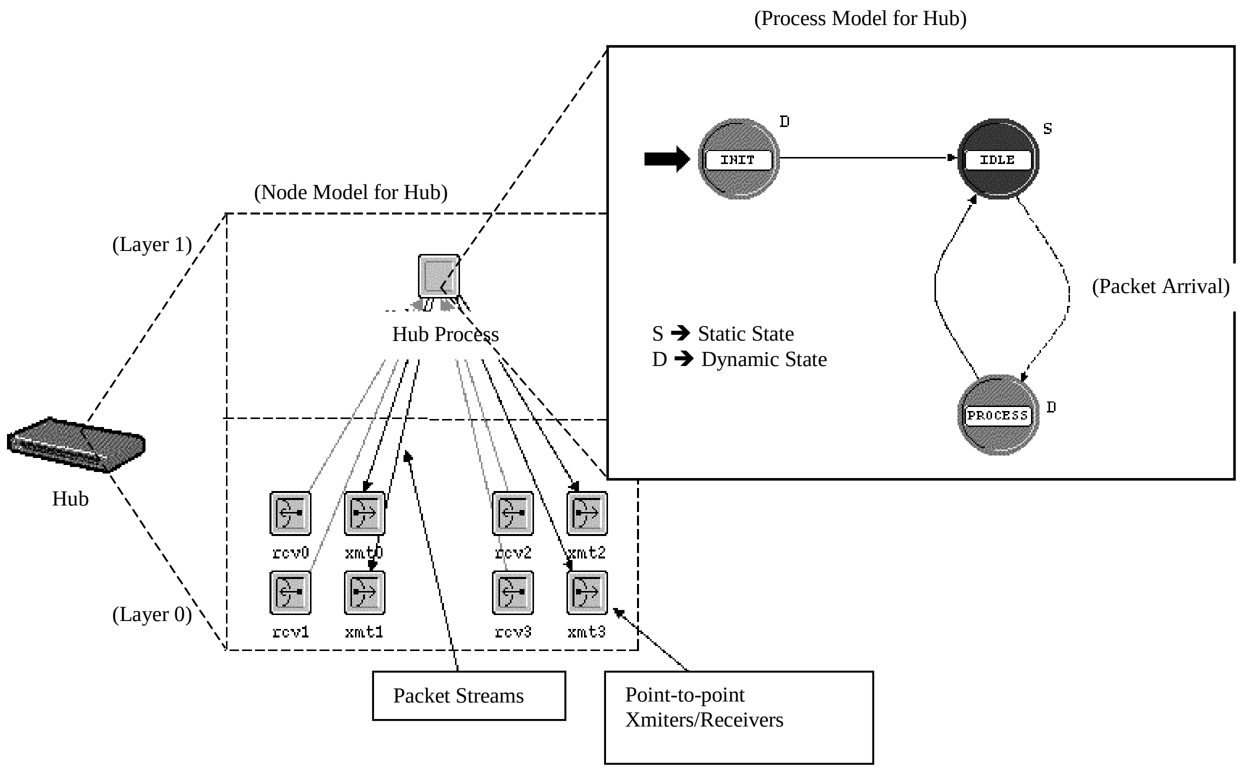


Fig. 3. Node and FSM models for Hub.

The *Hub Process* communicates with 16 pairs of point-to-point transmitters (xmt0-15) and receivers (rcv0-15) using packet streams. Only four such pairs are shown for simplicity. The point-to-point receivers (transmitters) must be used to get (send) packets from (to) the physical links. After receiving a packet on a particular link, the point-to-point receiver for that link interrupts the *Hub Process*, and delivers the packets using the packet streams.

This is analogous to the way in which an NIC (Network Interface Card) delivers packets to the upper layers in a computer. The Hub Process then examines this packet and sends it to the appropriate point-to-point transmitter, to be forwarded on the link connected to the destination. As noted earlier, the hub can be programmed to broadcast the received packets on all the outgoing links.

The Finite State Machine (FSM) model for the hub process is also shown in Fig. 3. When a simulation is started, the process first enters the "INIT" state, initializes the different variables and immediately goes to the "IDLE" state, where it waits for a "packet arrival" type of interrupt. When such an interrupt occurs, due to a packet being received by one of the point-to-point receivers, the process goes from the "IDLE" state to the "PROCESS" state where it examines the destination address in the packet and forwards it to the appropriate point-to-point transmitter. The process then returns to the "IDLE" state and waits for the next "packet arrival" type of interrupt. The enter executives for the "PROCESS" state are shown in Fig. 4. The op_* functions are system functions provided by OPNET.

```
pkptr = op_pk_get(op_intrpt_strm());
            //get packet from the appropriate pt-to-pt receiver
op_pk_nfd_get(pkptr, "dest_address", &dest_address);
           //get dest_address field from the packet
op_pk_send(pkptr, dest_address);
           //send the packet to the appropriate pt-to-pt transmitter
```

Fig. 4. Enter executives for PROCESS state.

2.4.2. Terminals

For the Terminals, the node model and FSM model are shown in Fig. 5. The node model consists of a packet generator (shown as "*Packet Generating Source*" in Fig. 5) and a packet processor (shown as "*Packet Process*" in Fig. 5). The packet generator generates packets according to some pre-selected distribution. These packets are interpreted differently, depending on the algorithm under simulation. For example, in the mutual exclusion algorithms that we have simulated, the packets generated by the packet generator indicate to the packet processor that the upper layers need to enter the Critical Section (CS). The distributed algorithm, under consideration, itself is implemented in the packet processor. *Thus, only this part has to be modified to simulate* *different algorithms*. Here, it is shown only as a single process at layer 1, but in many models that we have developed it consists of two different processes.

The FSM model for the packet processor, shown in Fig. 5, is very general. Most of the algorithms that we simulated used this model for the packet processor. This is a simple model in which a process waits for the arrival of packets. The packets can arrive from the Packet generator (Layer 2) or from the point-to-point receivers (Layer 0). Depending on the provider of a packet, appropriate actions are taken. More specific code for the packet processor, related to a particular distributed algorithm, is described later.

The process model for the packet generator is standard in OPNET, so we will not discuss this model any further. For more information, refer to the appropriate OPNET manuals [13].

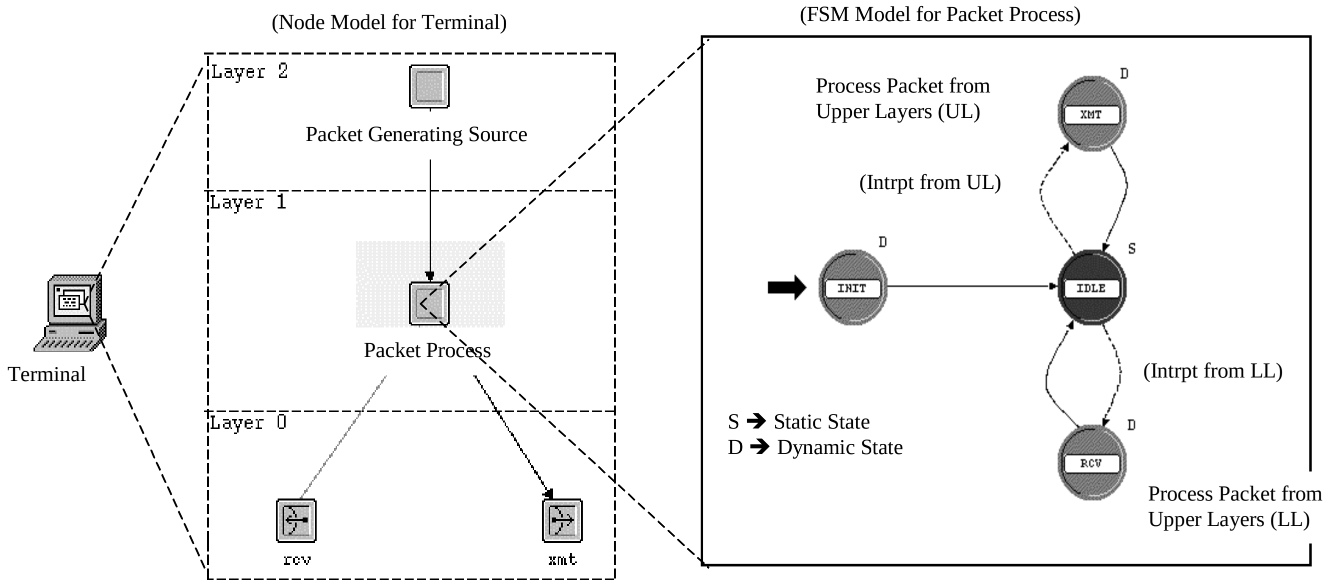


Fig. 5. Node model for Terminal.

3. INTRODUCTION TO XPLOT AND LOG2XPL

The second tool that we used is Xplot [15], a general-purpose graph generation tool. This program is freeware and can also be downloaded from our site [16]. Many different versions of Xplot are available on the Internet. We used a Unix version running on X-Windows. The main advantage of Xplot is that a programmer can use this tool without being concerned about any internal aspects of the tool, or even any set of APIs. All that is required is knowledge of the different command keywords of the Interface Definition Language (IDL) that drives Xplot. Fig. 6 gives an example IDL file. Words in bold are command keywords, which tell Xplot what to display.

```
unsigned unsigned         //plot unsigned integers on X and Y axis
title    //keyword indicates next line is the title of the graph
A demo xplot input file   //actual title displayed on the graph
xlabel   //keyword indicates next line is a label for X-axis
x axis in cm
ylabel   //keyword indicates next line is a label for Y-axis
y axis in inches
xunits   //keyword indicates next line is a scale for X-axis
cm
yunits   //keyword indicates next line is a scale for Y-axis
in
red      //everything after this line to be displayed in Red
line 0 1 15 1             //Red line from (0,1) to (15, 1)
diamond 4 2               //Diamond at (4,2)
blue     //everything after this line to be displayed in blue
. . .
```

Fig. 6. A demo Xplot input file.

To support generation of graphical views of our simulation results, we created a simple conversion utility program (Log2Xpl), which is written in C and is available for download from [16]. During the OPNET simulation execution, every terminal produces a log file. Our conversion utility parses through all the log files and creates an appropriate IDL file to be displayed using Xplot. Every log file contains a minimum of 4 columns in addition to other optional columns depending on the algorithm under simulation. The first column always contains the words “XMT” or “RCV,” which indicates whether this message was transmitted (XMT) or received (RCV) by this terminal. The second and the third columns indicate the destination and the source address, respectively, associated with the message. The forth column is a sequence number of the message transmitted by this terminal.

The Log2Xpl program operates by reading an XMT line from the first log file, and finding its match (i.e., corresponding RCV line) in the remaining log files. The matching is facilitated by the fact that the source address and the sequence number uniquely identify the message. Once a match is found, commands are written into the output IDL file to display the message. Typical commands that are generated look like the following sequence

```
…
blue                //everything following this is displayed in blue
line 29.68 0 32.71 5  //draws a line from (29.68,0) and (32.71,5)
diamond 32.71 5       //draws a diamond at co-ordinate (32.71,5)
vtick 29.68 0        //puts a vertical tick at co-ordinate (29.68,0)
vtick 32.71 5        //puts a vertical tick at co-ordinate (32.71,5)
…
```

The same procedure is repeated for the remaining log files. Our conversion utility can also be used to support graphically display of message sequences captured in a real environment (as opposed to a simulation). Of course in this case, all the participating terminal processes would need to create log files in the appropriate format, and they would need to be synchronized, perhaps by using Network Timing Protocol (NTP) (so that a positive time is logged when a message traverses from one terminal to another).

4. EXAMPLES OF ALGORITHM SIMULATIONS

We will now describe how we used the basic framework developed in OPNET to simulate the LLC algorithm. This is followed by an example of how the models developed for LLC can be modified to simulate the Vector Clocks. We will not discuss all the changes made, per se, but we will try to introduce the reader to how these changes can be made. Finally we illustrate, using two mutual exclusion algorithms, how the performance of different algorithms can be compared.

4.1. Lamport's Logical Clocks (LLC)

The LLC algorithm [1] establishes a causal ordering among events occurring in different processes. Being able to ascertain the ordering of events is very important for designing, debugging and understanding the sequence of execution in distributed computing [1]. Events within a process can mean either an execution of an instruction or execution of an entire procedure depending on the application. When processes exchange messages the sending of a message constitutes one event and the reception of the message constitutes another event. The LLC algorithm achieves causal ordering by using virtual clocks at every participating process, which assign monotonically increasing timestamps to different events. When processes exchange messages, the sending process also transmits the timestamp of the send event along with the message. The process receiving the message adjusts it virtual clock if necessary – i.e., if its clock is "behind" relative to the timestamp of the received message. This is required to ensure that the timestamp on the receive event itself is always greater than the timestamp of the corresponding send event. Thus the LLC algorithm ensures the *happened-before* relationship, which states that if event E1 *happened-before* event E2 (denoted as E1 ➔ E2), then the timestamp on E1 will always be less than the timestamp on E2.

By modifying the "Packet Process" (layer 1, Fig. 5) we implemented the LLC algorithm. The FSM model used for this process is the same as shown in Fig. 5, while Fig. 7 shows commented code-snippets for the "INIT," "XMT" and the "RCV" states. All the functions starting with op_* are library functions provided in OPNET.

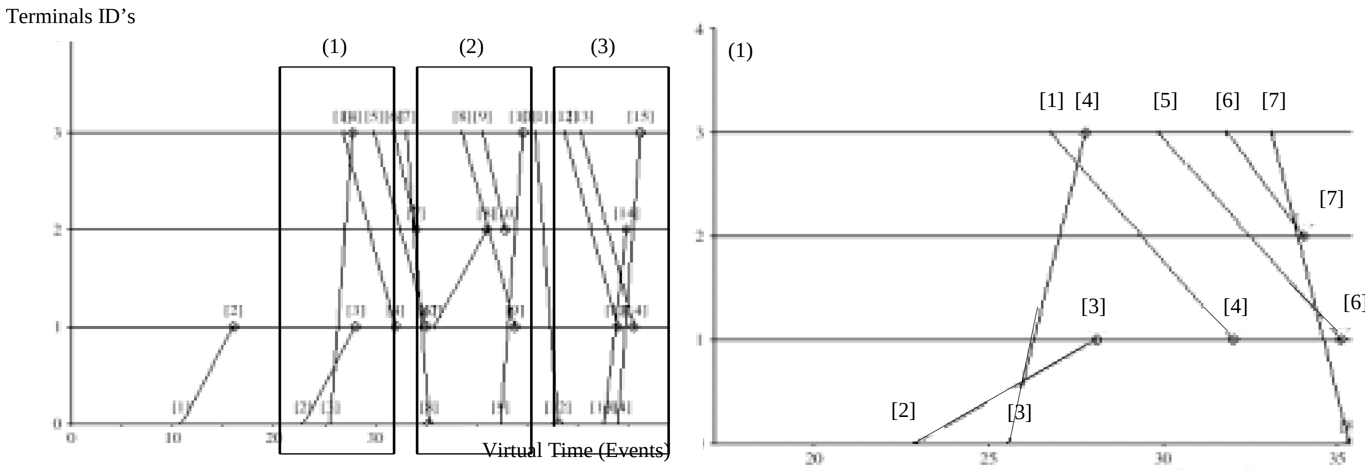
```
ENTER EXECUTIVES FOR INIT:
state_vector=0;  //Initialize the state-vector for LLC
seq_no=0;        //Initialize the Sequence Number for this Terminal

ENTER EXECUTIVES FOR XMT:
pkptr=op_pk_get(SRC_IN_STRM); //Intrpt from Upper Layer, get packet
seq_no++;                     //Increment seq_no for this terminal
state_vector++;               //Increment the LLC clock
                       //Get a random Dest.(Code not Shown)
                       //Populate the message with app. values
op_pk_nfd_set(pkptr,"dest_address",random_generated_destination);
op_pk_nfd_set(pkptr,"src_address",my_address);
op_pk_nfd_set(pkptr,"optional fields",state_vector);
op_pk_nfd_set(pkptr,"Sequence No",seq_no);
print_packet(pkptr,"XMT");    //Record packet in Log as XMT line
op_pk_send_delayed(pkptr, XMT_OUT_STRM);       //Send packet

ENTER EXECUTIVES FOR RCV:
pkptr=op_pk_get(RCV_IN_STRM);//Msg. recvd. from other terminal
state_vector++;              //Incr LLC clock for this terminal
op_pk_nfd_get(pkptr,"optional fields",&temp_value);
                             //get LLC clock of other terminal
//LLC Algorithm
if (state_vector < (temp_value + 1)){
    state_vector = temp_value + 1;
}
print_packet(pkptr,"RCV");  //Record packet in Log as RCV line
op_pk_destroy(pkptr);       //Destroy packet
```

Fig. 7. Code snippets from LLC Packet Processor.

The INIT state is dynamic, and is the first state that the process enters when the simulation is started. Since this is a dynamic state, the process cannot remain in this state, and has to transition to a static state ("IDLE").

However the process does some initializations while in this state. In the IDLE state the process waits for some interrupt to occur. Two types of Interrupts can occur. One, due to a packet generated by the Upper Layers (UL) in the same terminal (“Intrpt from UL”), and the other due to packet generated by other terminals (“Intrpt from LL”). Depending on which interrupt occurs, the process might go from the IDLE state to the RCV or XMT state, and then execute the corresponding “Enter executives” and “Exit executives” (which is Null, in this case). It then returns to the IDLE state to handle further interrupts.

Some attributes need to be set before the simulations can execute: the addresses assigned to the different terminals (this must be unique), and the names of the log files created by each terminal. The “Packet Generating Source” for each terminal is programmed to send a message using an exponential probability distribution function, with a mean value of *X* seconds (for some selected X). In other words a message will be generated on each terminal every *X* seconds. The simulation is then executed for a specific time, and then the log files created are converted to an appropriate IDL file using the Log2Xpl utility. Finally, Xplot is used to display the generated IDL file. An example is as shown in Fig. 8. Because at this time no standard way is available to display arrows in Xplot, we simply use a diamond symbol, instead of an arrow head, at the head of a line to indicate the direction of a message (i.e., toward the diamond). The numbers in square brackets indicate the event timestamps assigned by the LLC algorithm. Fig. 8 represents actual screen snapshots. Since the events and timestamp values are crowded in Fig. 8-a, we have presented in Fig. 8-b a zoomed-in view of the first region. From this expanded view it can be observed that for any pair of events E1 and E2, if E1 ➔ E2 then the timestamp value for E1 will be less than the timestamp value for E2, as is expected for the LLC algorithm. For example consider in Fig. 8-b the 3rd event of process 3, which has a timestamp of “5.” This is a send event. When the corresponding messages is received by process 1, the receive event is assigned a timestamp of value “6.” We can observe that the LLC algorithm properly adjusted this timestamp value, for otherwise the timestamp on this receive event would simply have taken the value “5.” All events are timestamped in a way that preserves the *happened-before* relationship. Of course, this one figure does not prove the correctness of the LLC algorithm, but multiple experiments of this type can be performed to provide more insights and confidence in understanding the behavior of the protocol. Note that one advantage of the Xplot tool is its ability to let the user isolate segments of a plot and create expanded views.



a. Original Plot b. Expanded Part (1)

Fig. 8. Time sequence graph displayed by Xplot for LLC.

4.2. Vector Clocks

The vector clock algorithm [1] is the same as LLC, except that individual event timestamps are defined by vectors. In particular, for an *n*-process system, each timestamp is represented as a vector of *n* "logical clock" values. A small modification to the "Packet Process" used for the LLC simulation enabled us to easily simulate vector clocks. The FSM for this process is essentially the same as for LLC. Fig. 9 highlights the modifications made to the FSM. Like the LLC simulation, the Vector Clock simulation is executed for some units of time to collect log files. Then the log files are converted into an appropriate IDL file (using the Log2Xpl utility) and displayed using Xplot.

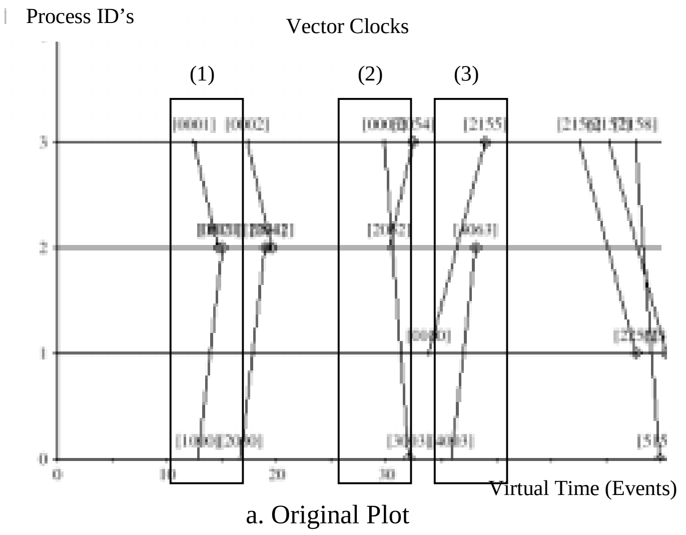
The Vector Clock simulation is shown in Fig. 10. Note that the vectors of the form [abcd] in the figure represent the "clock" values associated with terminals 0, 1, 2, and 3, respectively. As with the LLC simulation, we can observe that for all events E1 and E2, such that E1 ➔ E2, E1's timestamp is less than E2's timestamp (using proper vector comparisons). Furthermore, unlike in the LLC simulation, we can observe that if an event F1 has a timestamp value that is less than the timestamp value of another event F2, then it is true that F1 ➔ F2 – a fundamental difference between standard Lamport timestamps and vector-clock timestamps.

```
ENTER EXECUTIVES FOR INIT:
//num_of_terminals must be known a-priori
for (j = 0; j < num_of_terminals; j++)
state_vector[j] = 0;   //Init. state-vector for Vector Clocks
seq_no = 0;            //Init. seq_no for this Terminal

ENTER EXECUTIVES FOR XMT:
pkptr = op_pk_get(SRC_IN_STRM);//Intrpt from Upper Layer;get packet
seq_no++;              //Incr. seq_no for this terminal;
state_vector[my_address]++;    //Increment current terminal's clock
                       //Get a random Dest.(Code not Shown)
                       //Populate the message with app. values
op_pk_nfd_set(pkptr,"dest_address",destination);
op_pk_nfd_set(pkptr, "src_address",my_address);
op_pk_nfd_set(pkptr,"optional fields",state_vector);
op_pk_nfd_set(pkptr,"Sequence No",seq_no);
print_packet(pkptr,"XMT"); //Record packet in Log as XMT line
op_pk_send_delayed(pkptr, XMT_OUT_STRM);    //Send packet

ENTER EXECUTIVES FOR RCV:
pkptr = op_pk_get(RCV_IN_STRM);  //Msg. rcvd. from other terminal
op_pk_nfd_get(pkptr,"optional fields",&temp_value);
                     //get the vector of the other terminal
for (j = 0; j < num_of_terminals;j++){
    if (j == my_address)
        state_vector[j]++;   //Init. state-vector for Vector Clocks
    else if (state_vector[j] < (temp_value[j]))
        state_vector[j] = temp_value[j];
}
print_packet(pkptr,"RCV");  //Record packet in Log as RCV line
op_pk_destroy(pkptr);       //Destroy packet
```

Fig. 9. Code snippets from Vector Clock Packet Processor.

4.3. Performance Comparison between two Mutual Exclusion Algorithms.

We simulated the Centralized Mutual Exclusion algorithm [1] and the Suzuki-Kasami mutual exclusion algorithm [1] using the basic model with modifications to the Packet Processor (Layer 1, Fig. 5). In order to illustrate distributed ME, a terminal is said to enter the Critical Section (CS) when it can send ten consecutive messages to another terminal denoted as “shared resource.”



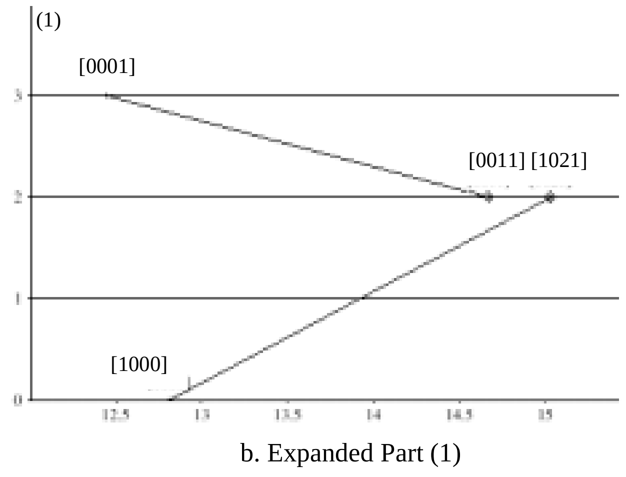


Fig. 10. Time sequence graph displayed by Xplot for Vector Clocks.

For a centralized mutual exclusion algorithm, we had to model a centralized controller, which is again a modification of the basic model. We will not discuss the FSM model for the Controller terminal and the Shared Resource terminal, as it is fairly simple; however those interested can download these models from [16]. The screen-shot of the graph, shown in Fig. 11, illustrates how mutual exclusion is achieved in the centralized scheme. Only a part of the graph is shown, for simplicity.

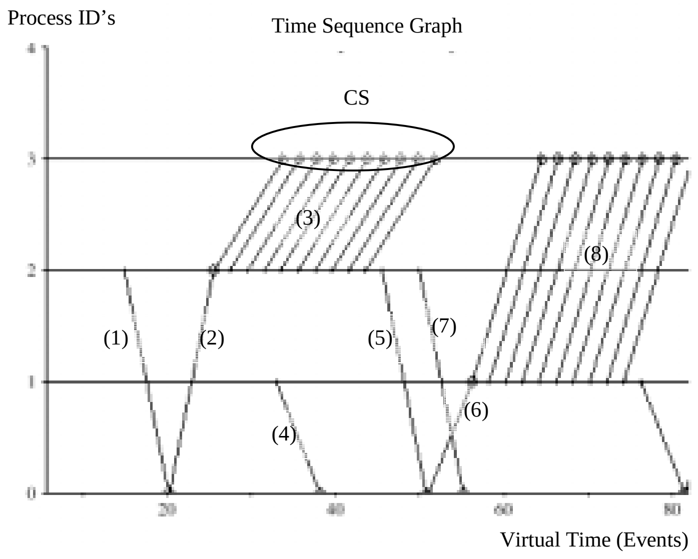


Fig. 11. Time Sequenced Graph for Centralized Mutual Exclusion.

In this simulation Terminal-0 was made to act as the Controller Terminal, giving permissions to other terminals to enter the CS (i.e., to send 10 consecutive messages to a Shared Resource: terminal-3) The other two terminals (Terminal-1, Terminal-2) compete to enter the critical section (CS). In Fig. 12, we can observe the following sequence of events:

(1) Terminal-2 sends a REQuest to access message to the controller (Terminal-0).

(2) The controller sends a REPLY (resource granted) message to Terminal-2.

(3) Terminal-2 enters the critical section (CS).

(4) Terminal-1 also sends a REQuest to access message to the controller. This request gets queued.

(5) Terminal-2 sends a RELEASE message, indicating that it is out of the critical section.

(6) Controller sends REPLY (resource grant) message to Terminal-1.

(7) Terminal-2 sends a second REQuest to access message to the controller. This message gets queued.

(8) Terminal-1 enters the CS.

The Suzuki-Kasami algorithm also provides mutual exclusion, but by a decentralized control scheme [1]. The basic idea of the algorithm is that a site can enter the critical section only if the site possesses a token (some special purpose, shared, message), and if a site that is wanting to enter the critical section does not have possession of the token, then the site broadcasts a “request” message to all other sites and waits to receive the token. We do not present the time sequence graph that plots the messages from the simulation, as it is trivial. Instead, we focus now on a method to compare the performance of the two algorithms.

OPNET provides certain functions to collect statistical data for a particular variable of interest. It is the programmer’s responsibility to use these functions to collect appropriate sample points for these statistics. For example, in the mutual exclusion models, we added the capability to collect statistics for the delay observed by any terminal between the following events: sending a request to enter the CS (event E1) and getting the permission to enter the CS (event E2). Thus if E1 occurs at some time, say X secs, and E2 occurs at a future time, say X+Y, then the delay will be recorded as Y units at time X+Y. This is treated as a sample point. A number of such sample points are collected as the simulation proceeds. We can also program OPNET so that it will not register every sample point, but the average of the last, say 10, sample points, or the average of the number of sample points collected in the last, say 10, seconds. These sample points can then be, either plotted using a built in graphing tool, or can be exported in the form of a spreadsheet.

For our simulation we collected every sample point with 14 different terminals. Every Terminal was programmed to generate a request to enter the critical section every 100 secs, and once any terminal enters the critical section, it remains in the critical section for around 10 secs. The result shown in Fig. 12 represents the running average of all the collected sample points and is plotted by the built-in graphing tool provided by OPNET. This graph clearly shows that the average time taken by any terminal to enter the critical section using centralized mutual exclusion is more than the time required by using the Suzuki-Kasami algorithm. Similar other statistics can be collected to compare the performance.

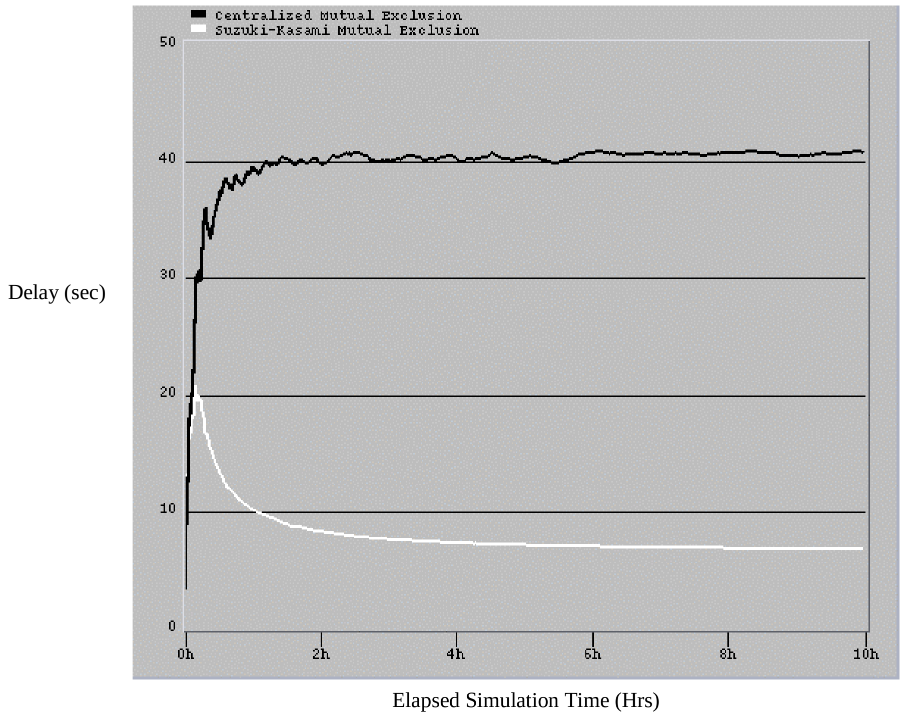


Fig. 12. Average delay for entering the CS.

5. CONCLUSION AND FUTURE RESEARCH

We conclude that simulation of distributed algorithms can become fairly simple by using some existing tools, and the approach provides a highly flexible simulation environment. Using a tool like OPNET on the front-end empowers a user with a model-based approach for distributed algorithm specification. Likewise, a tool like Xplot on the back-end facilitates the (portable) generation of very easy to manipulate graphical views of message sequences, which are at the heart of all distributed algorithms. Our foremost objective in this paper is to introduce the existence of tools that can be integrated to create a powerful environment to simulate distributed algorithms.

The basic models that we have developed in Section 2 can be readily modified to simulate many existing, as well as new, algorithms. Our Log2Xpl conversion utility, which converts the log files generated by OPNET models to an appropriate IDL file for Xplot, is general enough to be used with other existing toolkits as well as with data collected from actual system execution in a real (not simulated) environment. It can be easily modified and used depending on the requirements. Also OPNET provides useful mechanisms to compare the performance of different algorithms based on statistical data. We feel that this is a very important feature that can help students or researchers choose the most efficient algorithm from the available lot. We have highlighted this fact in this paper by a simple comparison of the Centralized and Suzuki-Kasami mutual exclusion algorithms.

We wish to point out that, although OPNET provides very good flexibility for simulating distributed algorithms, it is not at all trivial to learn programming in OPNET. We anticipate that for any student/researcher adopting this approach, the most difficult step will be to design the FSM models. But, in our opinion, development of such models is an extremely valuable skill since FSM modeling is an important abstraction in many areas of computer science (e.g., model checking [17] for system verification). Till now we have tested our approach only with people who are familiar with OPNET. The approach appears to be very convenient, and has aided the development of a new token-based algorithm for simulating distributed mutual exclusion. In the future, we plan to introduce the approach to students not familiar with OPNET, to obtain a broader perspective on the approach’s efficacy.

In this paper, we have ignored fault tolerance issues, such as what happens if a message gets lost, or delayed, or how the algorithm will react, should some terminal crash. The terminal models are not based on IP, so they cannot be used with any general network topology. Currently only the model shown in Fig. 2 is supported. Also, the Log2Xpl utility is not optimized, and can have a long runtime if the simulation is run for a very long time or if there are a large number of terminals in the simulation. All these issues are pending and will be dealt with in future research.

ACKNOWLEDGEMENTS

We wish to thank Prof. Oliver Yu, for lending us, both the software and the hardware for carrying out our experiments. We also thank the anonymous referees for providing helpful suggestions that improved the quality of this paper.

REFERENCES

1. M. Singhal, *Advanced Concepts in Distributed Operating Systems*, McGraw Hill, 1994.
2. A. Tanenbaum, *Modern Operating System*, Prentice Hall, 1995.
3. E. Knapp, ‘Deadlock detection in distributed databases,’ *ACM Computing Surveys*, 1987; 19:303–328.
4. M. Raynal, ‘A simple taxonomy for distributed mutual exclusion algorithms,’ *Operating System Review*, 1991; 25:47–50.
5. J. Matocha and T. Camp, ‘A taxonomy of distributed termination detection algorithms,’ *Journal of Systems and Software*, 1998; 43(3):207–221.
6. R. Fujimoto, ‘Parallel and distributed simulation,’ *Proceedings of the 1999 Winter Simulation Conference*, 122-131.
7. M. Ben-Ari, ‘Distributed algorithms in Java,’ *Proceedings of the Conference on Integrating Technology into Computer Science Education*, 1997, 62 – 64.
8. S. Hartley, ‘Alfonse, You have a message!,’ *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, 2000, 60-64.
9. B. Koldehofe, M. Papatriantafilou and P. Tsigas, ‘Distributed algorithm visualization for educational purpose,’ *Proceedings of the 4th Annual SIGCSE/SIGCUE on Innovation and Technology in Computer Science Education*, 1999, 103 – 106, http://www.cs.chalmers.se/~lydian/
10. W. Schreiner, ‘Toolkit for Simulating Distributed Algorithms in Java,’

http://www.risc.uni-linz.ac.at/software/daj/

11. M. Bauderon, M. Mosbah, S. Gruner and A. Sellami, ‘A New tool for the simulation and visualization of distributed algorithms,’ *LaBRI Report No. RR-1245-00*, *Université Bordeaux 1*, October 2000. http://dept-info.labri.u-bordeaux.fr/~stefan/abstract.html
12. S. Burdette, T. Camp and B. Bynum, ‘Distributed BACI: A toolkit for distributed applications,’ *Concurrency: Practice and Experience*, Vol. 12, 2000, 35-52. http://www.mines.edu/fs_home/tcamp/dbaci/
13. OPNET modeler, Version 7.0.B, OPNET Technologies, Inc. Bethesda, Md., 1999, http://www.opnet.com
14. Network Simulator, http://www.isi.edu/nsnam/ns/
15. T. Shepard, ‘TCP packet trace analysis,’ Technical Report, MIT Laboratory for Computer Science, MIT-LCS-TR-494, February 1991. ftp://ftp.lcs.mit.edu/pub/lcs-pubs/tr.outbox/MIT-LCS-TR-494.ps.gz
16. http://ftp.evl.uic.edu/pub/OUTgoing/shashank/
17. E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 2000.