The Totem Protocol Development Environment

P. Ciarfella
Cascade Communications Corporation
5 Carlisle Road
Westford, MA 01886

L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106

We describe the protocol development environment for the Totem protocol, a fault-tolerant communication protocol that provides fast reliable ordered message delivery and membership services. The protocol development environment is a distributed discrete-event simulator that provides a testbed for the controlled testing and debugging of the Totem protocol.

Unlike many other protocol simulation environments, the Totem protocol development environment directly executes the implementation code of the protocol being simulated instead of a model of the protocol. Executing the implementation code of the protocol in the simulator facilitates both the testing and debugging of the protocol. Modifications and design changes to the protocol are directly reflected in both the simulation and the implementation.

1 Introduction

The development of communication protocols is significantly more difficult than the development of sequential programs. Communication protocols are harder to develop because of the many possible executions and ordering of events, the difficulty of recording the history of an execution, the inability to stop the system so that its state can be examined, the difficulty of injecting a stimulus or fault at exactly the right moment, and the lack of reproducibility. Fault-tolerance presents additional problems, particularly the need to recover correctly from rare failure modes and the appearance of correct operation even though the system contains serious defects.

The protocol development environment presented here was designed to aid in the development of the Totem reliable ordered broadcast protocol, but its concepts are also applicable to other communication protocols. The development environment is a distributed discrete-event simulator designed to provide a testbed for the controlled testing and debugging of the protocol implementation modules. The novelty of the protocol development environment is that

- The implementation code can be executed in both the development environment and the implemented system, and
- The implementation code can be tested and debugged in the development environment almost as if it were a single sequential program.

1.1 Motivation for the Protocol Development Environment

The construction of the protocol development environment was motivated by the lack of a controlled environment in which to exercise the Totem protocol. Originally, testing and debugging of the protocol implementation were performed by executing the implementation on Sun Sparcstations communicating over an uncontrolled and arbitrarily loaded Ethernet network. The high speed at which the system operated allowed little control to the researchers in monitoring the execution and in injecting faults during testing and debugging. In addition, researchers were not able to use source-level symbolic debuggers because stopping any one processor caused side-effects, such as delaying the transmission of messages, that provoked other processors in the network to time-out and initiate undesired recovery actions. Moreover, investigation of the behavior and performance of the protocol was limited by the size of the network available in the laboratory. Thus, it was decided that a protocol development environment based on a distributed discrete-event simulator should be constructed so that the operation of the Totem protocol could be monitored and changed with ease and flexibility.

A major goal was to provide a development environment in which the Totem implementation code could be executed without modification. In many protocol development projects, the code that simulates

This work was supported by the National Science Foundation, Grant No. NCR-9016361 and by the Advanced Research Projects Agency, Contract No. N00174-93-K-0097. Paul Ciarfella's work was supported by Digital Equipment Corporation's Graduate Engineering Education Program while he was a graduate student at the University of California.

0-8186-6685-4/94 $4.00 © 1994 IEEE

the desired system is different from the code that implements the actual operational system. This can result in two versions, possibly constructed and maintained by different developers, that exhibit inconsistent behavior due to different interpretations of the intended operation of the protocol. The existence of multiple versions also requires a duplication of effort when the protocol is revised including separate coding, testing and debugging efforts that may result in additional inconsistencies. Direct use of the implementation code within the development environment provides a consistent base from which to develop the protocol.

The result is a controlled, flexible and extensible development environment in which the object code modules of the Totem protocol can be debugged and tested, and then simply relinked and executed at full-speed in the operational network (Figure 1).

1.2 Use of the Development Environment

The Totem protocol development environment allows the user to vary the operating conditions of the network including such parameters as message loss and processor failure rates. It thus provides greater control over the operating environment than is available on unpredictable physical networks.

Within the simulated network, the user can control the probabilities of message loss and token loss. Both of these probabilities are based on a Poisson distribution. By adjusting these parameters, the user can create various network scenarios. For example, a high message loss rate can be used to stress the retransmission and flow-control algorithms of the Totem ordering protocol. Likewise, a high token loss rate can be used to stress the robustness of the Totem membership protocol. By utilizing low message and token loss rates, the user can investigate the the stability of the system over a long period of time.

An important and novel feature of the Totem protocol is its support for consistent message ordering in the presence of network partitioning. To test this on a physical network requires the construction, for example, of multiple Ethernet segments that are connected by repeaters or MAC-level bridges. To generate a network partition, the protocol developer must physically change the network infrastructure by disabling a repeater inserted into the network at the appropriate point. This task, though not difficult, can become quite cumbersome when a large number of processors is involved and various configurations are to be tested.

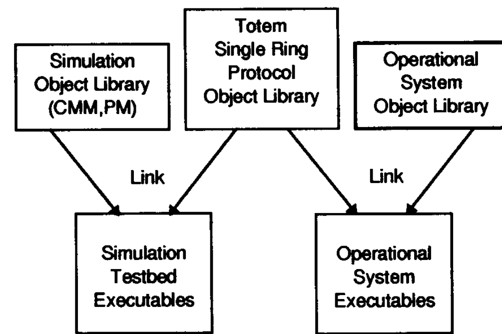

Figure 1: Creation of Executable Modules

In the development environment, the user can define a set of network partition events that occur automatically as the simulation advances in time. Partition events allow the user to monitor and test the formation of processor memberships and the correctness of message ordering during partitioning simply and efficiently without having to reconfigure physical components. Failure of an individual processor can be simulated by defining a partition event that isolates that processor. Similarly, restart of a failed processor or addition of a new processor can be simulated by a partition event that inserts the processor into the network with the other processors. Partition events are defined in a configuration file and specify the time at which an event is to occur together with the components of the partition, where each component comprises one or more simulated processors. Multiple execution histories can be specified in different configuration files.

The user can design different execution scenarios and test cases to be applied to the system. A scenario can be executed many times under the same conditions, providing a stable repeatable execution history in which to debug the implementation or to perform acceptance or regression testing.

2 The Totem Protocol

2.1 Basic Idea of the Totem Protocol

In most distributed applications, it is essential that processors maintain the consistency of replicated data distributed across the system. This consistency is facilitated by a single common order of events seen by all processors in the execution of the distributed computation. A common order of events ensures identical executions and identical data at each processor, while different orders seen by two or more processors may result in inconsistencies in the data.

The Totem protocol [1,2,12] addresses the need for consistency of replicated data by providing reliable totally ordered delivery of messages. The underlying communication medium on which the protocol executes is assumed to have an unreliable broadcast capability. Both processors and the communication medium may suffer from fail-stop, timing, and omission faults, but not malicious faults. In addition, the network may partition and merge at any time and failed processors may restart and rejoin the network at will.

Processors within a network form a logical ring around which a special message, called the token, is circulated. The token mediates access to the ring in that only the processor in possession of the token can broadcast messages. The token contains a sequence number field that is incremented and copied as the sequence number for each successive message broadcast. This results in every message containing a unique sequence number that provides a total order of messages broadcast on the ring. Other fields of the token invoke the retransmission of messages to achieve reliable delivery, and report which messages have been received by all processors.

A message can be delivered to the application when all messages with lower sequence numbers have been delivered (agreed delivery) or when, additionally, each of the other processors on the ring has received and will deliver the message unless they fail first (safe delivery). Effective flow control reduces the rate of message loss due to buffer overflow, and provides low latency from origination to delivery of a message.

A consistent view of the membership of processors on the ring is maintained by all processors on the ring. The presence of new processors in the network is detected, as is the departure of existing processors. When processors enter or leave the network, a new ring is formed with a new token.

The Totem protocol also operates across networks in which two or more rings are connected by gateways. Timestamps from approximately synchronized clocks are used to order messages consistently across rings, while sequence numbers on the individual rings are used to provide reliable delivery of messages. More details of the Totem protocol can be found in [1,2,12].

2.2 Implementation of the Totem Protocol

The implementation of the Totem protocol has been developed for the Unix operating system and consists of a set of portable software modules that provide reliable ordered delivery and processor membership services. Figure 2 presents a block diagram of the Totem protocol.

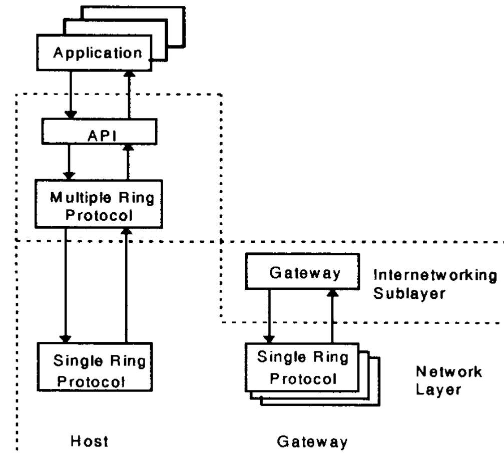

Figure 2: Block Diagram of the Totem Protocol

A host computer consists of a Single Ring Protocol (SRP) module, a Multiple Ring Protocol (MRP) module, and application programs that use the Totem services. A gateway consists of two or more SRP modules and a Gateway module that connects the two rings.

The SRP module is a set of software modules that provide the reliable ordered delivery and processor membership services for processors connected to a single Totem ring. A block diagram of the SRP module is shown in Figure 3. The SRP module is composed of the Data Link, Timer Event Service, Ordering and Membership modules, described below. It provides a

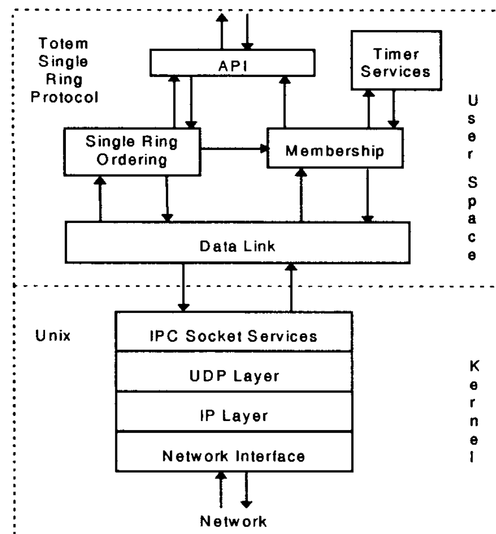

Figure 3: Block Diagram of the Totem Single Ring Protocol

programming interface to the upper layer, i.e., Gateway, Multiple Ring Protocol, or application program.

The Data Link module provides a layer of abstraction for the Totem protocol above the underlying communication medium. It provides functions to send and receive the token and messages with a programming interface that hides the actual interface to the communication services provided by the operating system. With this portable communication interface, any underlying communication subsystem can be supported by the implementation. The Data Link module was developed for the Unix operating system and uses the Unix IPC socket services and the Internet UDP/IP protocols [13,14] to transfer messages across the communication medium. Other Data Link modules, such as one that supports NetBIOS [7], could easily be developed.

The Single Ring Ordering (SRO) module implements the Totem reliable ordered delivery services at the local network level. The Single Ring Ordering module uses the send and receive functions provided by the Data Link module to transmit and receive messages and the token.

The Membership module implements the Totem membership protocol and maintains the continuity of the logical ring. When the Single Ring Ordering module receives membership messages from the Data Link module, it passes them to the Membership module for processing. The Membership module invokes the Data Link module to transmit membership messages and the initial token to the communication medium.

The Timer Event Service module provides a set of functions that schedule a time-out event at a specific time or to cancel an event. A callback option is provided so that a designated function can be executed when a time-out occurs. The Membership module uses these services to implement the token loss time-out, as well as other time-outs that specify the operation of the membership protocol. For example, when the Membership module schedules a token loss time-out, it supplies the address of a function that handles the loss of the token if the time-out occurs.

3 The Simulation Mechanisms

A simulated Totem network (Figure 4) is composed of one or more *Processor Models* (PM) and one *Communication and Medium Model* (CMM). The CMM and each PM are separate processes executing on a single Unix workstation, communicating with each other through the Unix IPC socket and shared memory interfaces.

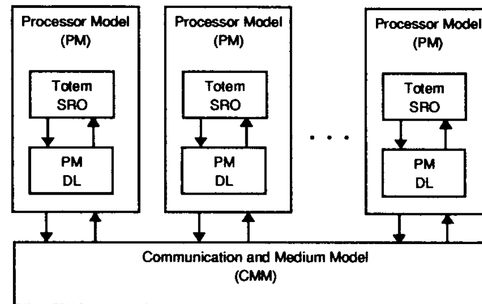

Figure 4: A Simulated Totem Network

3.1 Processor Model

A Processor Model (PM) is a Unix process that simulates a single processor on a Totem network. It executes the Totem protocol implementation code linked to a simulation-specific Data Link module and a set of simulated Unix system services. These are provided by the Simulation Object Library.

The Data Link model (PM DL) coordinates the transmission and reception of the token and messages with the CMM. When the Totem Single Ring Ordering module invokes the Data Link module to broadcast a message, the PM sends the message to the CMM which then simulates the broadcast by distributing the message to the other processors. Likewise, when the Totem Single Ring Ordering module transmits the token, the PM sends the token to the CMM which then forwards the token to the destination processor.

When the Totem Single Ring Ordering module invokes the Data Link module to receive the token or message, the PM sends a request message to the CMM requesting that the token or the PM's next undelivered message be forwarded. The PM then blocks until the CMM replies with the requested item, which is returned to the routine that invoked the Data Link module.

The Totem Single Ring Ordering module uses the Unix *select* service to determine whether the token or any messages are waiting to be received. In the operational system, the token and undelivered messages are queued within the operating system. In the simulated system, the token and undelivered messages are queued within the CMM and a PM must request these from the CMM. The PM simulates the Unix select function and queries the CMM, instead of the operating system, to determine if data are available for reception. The select function of the PM provides the same programming interfaces as the Unix select function, including support for a time-out option which blocks the simulated processor for a

specified period of time until either the data become available or the time-out expires.

If the token or message is available for delivery to a processor when the select function is called, the CMM returns a positive response to the requesting PM. The PM returns the positive response to the Totem Single Ring Ordering module, which invokes the Data Link module to receive the token or message.

If the token or message is not available for delivery to a processor when the select function is called and the time-out is zero, the CMM returns a negative response to the PM handling the select function, which returns a negative indication to the Totem Single Ring Ordering module.

If the token or message is not available for delivery when the select function is called and the time-out is non-zero, the CMM schedules a time-out event in the Event List but does not yet return a response to the PM.

If the token or message becomes available before the time-out expires, the CMM returns a positive response to the requesting PM. If no token or message becomes available before the time-out expires, a negative response is sent.

3.2 The Communication and Medium Model

The CMM is a Unix process that models an arbitrary communication medium and simulates both broadcast and point-to-point communication. It coordinates the transmission and delivery of the token and messages between PMs and provides the capability to inject faults into the simulated network, such as message loss, token loss and network partition events.

Simulated time is maintained in a variable called *Global Time*, stored in a shared memory segment created and owned by the CMM. *Global Time* is readable by every PM but can only be written by the CMM. A shared memory segment is used so that a PM can quickly read the current time without having to request an update from the CMM.

The flow of control in the CMM is requested in Figure 5. During initialization, the CMM creates and initializes the shared memory segment used to store simulated time. After initialization, the CMM enters a processing loop that never terminates. The loop consists of the sequence of receiving a message from a PM, processing the message, servicing the event queues, if necessary, and then waiting to receive another message, at which point the cycle repeats.

The CMM maintains two event queues, called the Event List and the Special Event List, that simulate the passage of time in the network and synchronize operation

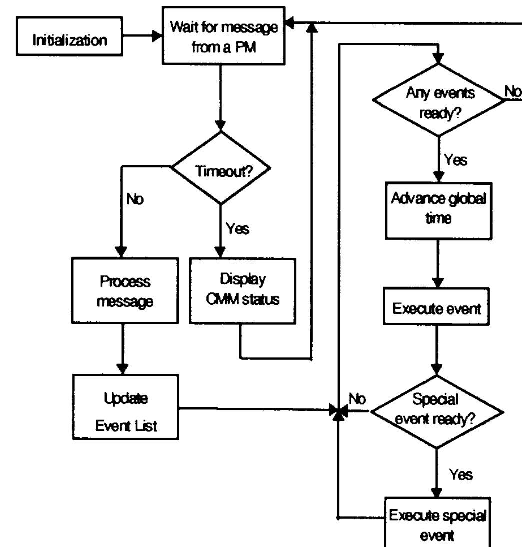

Figure 5: Flow of Control in the CMM

of the PMs. The Event List contains events that are used to drive the execution of PMs and to advance simulated time. The Special Event List contains special events, such as network partition events, that affect the state of the system but do not advance simulated time.

The execution of the CMM is driven by events in the Event List and by the reception of messages from the PMs that comprise the processors in the simulated network. The processing of a received message may result in the CMM sending a reply to the PM or forwarding the message to other PMs. Processing of a message may also result in adding a new event to, or deleting an existing event from, the Event List.

As the CMM processes events from the Event List, it advances *Global Time* and sends messages to the PMs informing them to begin tasks, such as the processing of the token or a message. On completion of those actions, each PM notifies the CMM and then waits for the next event to be delivered to it by the CMM. Ideally, each PM is executing concurrently with all other PMs at all times.

After a message is processed, the CMM invokes a scheduling algorithm to process events on the Event List that are ready to be executed. When an event is processed, *Global Time* is advanced to the time of the event and the event is executed. After the event has been executed, the Special Event List is checked to determine if a special event can be executed. The special event is executed if its scheduled execution time is greater than or equal to *Global Time*.

The Event List and Special Event List are processed until no more events are ready to be executed. At that point, the CMM waits for the arrival of the next message.

4 Simulated Time

One of the most interesting problems confronted by the Totem development environment is that of maintaining the appearance of global synchrony even though each PM is a free-running Unix process. Optimistic concurrent simulation strategies, notably Jefferson's Virtual Time [8], have been developed that allow time to advance aggressively, at the risk of rolling time back to allow a slow processor to catch up. When a rollback occurs, all work must be discarded that was performed after the point at which time is reset. Optimistic algorithms may perform better than conservative algorithms if rollback does not occur frequently.

Implementing an optimistic algorithm with rollback in the development environment would require restructuring the Totem implementation to rollback time. Since one of our goals was to move the Totem code directly and unchanged from the development environment to the operational system, we adopted a more conservative approach that allows concurrency between PMs but restricts asynchrony to ensure that rollback is never required. It also simplifies the presentation of the system state to the user during debugging. Lubachevsky [9] has proposed a similar approach.

4.1 The Scheduler

The CMM executes a scheduling algorithm, the scheduler, that advances the simulated *Global Time* as it processes events from the Event List. When an event is processed at the time specified in the Event List, *Global Time* is advanced to the time of the event. The scheduler ensures that *Global Time* is advanced no faster than the execution rate of the slowest PM and that no PM advances so far ahead of *Global Time* that it could miss an event generated by a slower PM.

When the CMM instructs the PM to begin executing a task, it schedules a completion event in the Event List to record the earliest time at which the PM can generate a further event, typically the sending or receiving of the token or message. The PM cannot influence the behavior of any other PM or the sequence of events in the system prior to that time. Thus, it is safe, relative to that PM, to advance *Global Time* up to the time of the completion event.

When a PM completes the execution of an event, the CMM estimates, based on the type of response generated by the PM, the time at which the PM would have completed its processing, which may be later than the

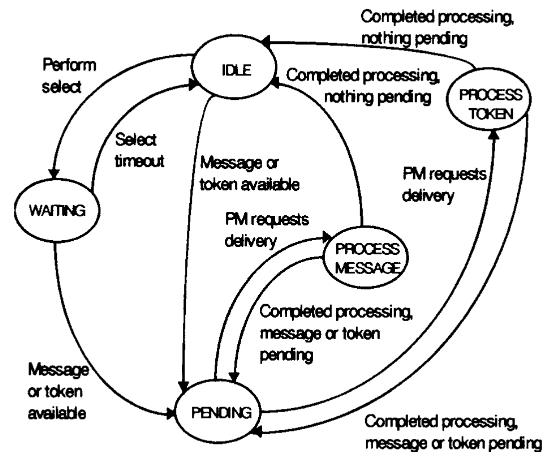

Figure 6: State Machine Maintained by the CMM

completion event. The CMM then removes the PM's completion event from the Event List, possibly replacing it with an event to generate the actions of the PM at the appropriate time.

The CMM uses a highly simplified model of the Totem protocol, based on the state machine shown in Figure 6, to estimate the minimum and actual times for the PM to execute a task. The times required for a PM to execute a task are important parameters of the system, determined from performance measurements made in the operational system. The parameters are defined separately for each PM so that it is easy to model a system in which processors operate at different rates. The values of the parameters for each PM are supplied to the CMM when the PM announces its presence during initialization.

4.2 Scheduling Events

An event has two parameters: the time at which the event was scheduled, called *start_time*, and the estimated completion time, called *est_compl_time*. When an event is added to the Event List, *start_time* is set to the value of *Global Time* at that instant, and *est_compl_time* is set to the sum of *Global Time* and the minimum length of time that it will take a PM to execute the task associated with the event, or the time-out value if the event is a time-out.

For each PM, *last_ctime* defines the time at which the PM most recently completed execution of an event. If *last_ctime* is less than *Global Time* when a new event is scheduled for the PM, then the system has advanced in time faster than the PM has executed, the PM is thus idle and a new task can be assigned to the PM. When

last_ctime becomes greater than *Global Time*, the PM has processed a task that will be completed in the near future. In this case, the CMM does not assign a new task to the PM until *Global Time* has reached *last_ctime*, thus preventing the PM from advancing faster than other slower PMs in the system.

When the scheduler is invoked, the Event List is processed until the event at its head is the anticipated completion of a task on the PM that has not yet reported completion of the task. Such events are not yet ready to be processed and the CMM blocks, waiting for a message from a PM.

Special events are processed only when simulated time has advanced to or beyond the event's *est_compl_time*. The execution of a special event does not advance simulated time, although it may result in the modification of the Event List.

4.3 Time-out Events

When the Totem protocol schedules a time-out through the Timer Event Services, the PM sends a time-out request to the CMM. Upon receiving the request, the CMM creates a time-out event for the indicated time and inserts it into the Event List. If the value of the time-out is less than the current value of *Global Time* when the CMM receives the request message from the PM, the time-out event is not scheduled.

When the CMM receives a select request from a PM with a non-zero time-out and the token or message is waiting to be delivered to the requesting PM, the CMM schedules a select time-out event. The select time-out is computed as the time-out plus the maximum of *Global Time* and *last_ctime* for the PM.

From the CMM's point of view, a PM that is in the IDLE state (see Figure 6) is not processing the token or a message. Due to the control flow in the Totem Single Ring Ordering module, an idle PM is guaranteed to perform a select call to request the delivery of the token or message, taking it into the WAITING state. A pure time-out event is processed only when none of the PMs is in the IDLE state. If time-out events were processed immediately when they appeared at the head of the Event List, simulated time could advance prematurely, preventing the execution of events generated by other PMs that should occur before the time-out.

4.4 Task Completions

An anticipated minimum-time completion event is scheduled by the CMM when the token or a message is forwarded to a PM. When the PM finishes processing the token or message, it sends a completion message to the CMM. When the completion message is received by the CMM, the associated completion event is deleted from the Event List.

When a processor transmits the token or a message, an event is scheduled for the end of the time required for that transmission. At that time, the message is delivered to all other PMs in the simulated network or, in the case of the token, to the next PM on the ring. The PM that transmitted the token or message blocks until it receives an acknowledgment from the CMM that the transmission has been completed.

5 Experience with the Development Environment

5.1 Performance of the Development Environment

The ability to model accurately the performance of the Totem protocol was not a goal of this work. Rather, we aimed to model accurately the functional behavior of the protocol. Thus, we do not present a comparison of the performance of the Totem protocol as predicted by the development environment and its performance in the operational system. Instead, we present measurements of the rate at which the development environment can model broadcasts by the Totem protocol as seen by the protocol developer.

Figure 7 shows the measured average number of messages broadcast in a simulated network over a 10 second execution of the development environment. This count was kept by the CMM and represents the number of broadcast messages sent to the CMM. The measurements were made on a single processor Sun Sparcstation IPC.

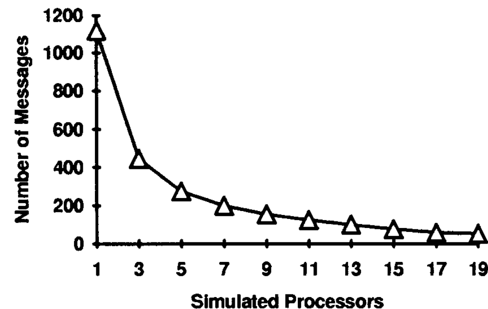

Figure 7: Messages Broadcast in 10 Seconds

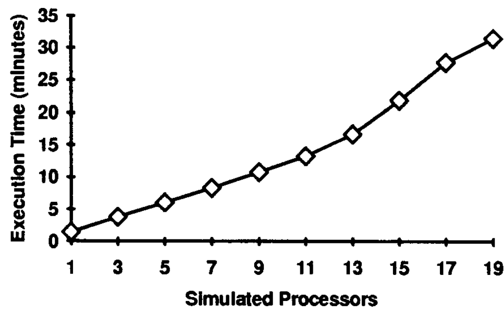

Figure 8: Time to Broadcast 10,000 Messages

A simulated network containing a single processor generated over 1100 messages every 10 seconds, or 110 per second; with a second processor the rate of broadcasting dropped to 625 messages per 10 seconds. With eight processors, the rate of broadcasting fell below 200 messages every 10 seconds and, with 15 processors, it leveled off at approximately 60 messages every 10 seconds. The measurements in Figure 7 were generated from processors that were all members of a single simulated network.

When a set of six processors was partitioned into two networks, each containing three processors, the aggregate rate of broadcasting was 421 messages every 10 seconds. This is almost double the rate generated by the processors within the single network; however, each message was delivered to only half as many processors.

With three networks containing three processors each, the aggregate rate of broadcasting was 436 messages every 10 seconds, compared to a single network of nine processors that generated 154 messages over the same period. The increased rate of broadcasting is due to the reduced number of processors that receive each message and, thus, the reduced processing in the development environment to represent such receptions.

Figure 8 presents the execution time of the development environment to broadcast and deliver 10,000 messages. The execution time is roughly linear up to about 14 processors, above which the Sparcstation became heavily loaded by the simulation of many processors and the operating system's response to user input became very slow. The level of performance by the development environment was entirely adequate and allowed us to perform testing and debugging without concern for processing speed.

The performance of the system can be improved by code optimization. The data obtained in these figures were obtained using a version of the development environment built from non-optimized code. When the system was regenerated with compiler and linker optimizations enabled, the performance of the system increased by approximately 10 to 15 percent.

5.2 Use of the Development Environment

The protocol development environment has been extensively used in the development of the Totem protocol. We have found no circumstance in which the development environment has failed to provide a representation of the operational system sufficiently faithful to allow testing and debugging. When coupled with a commercially available source-level symbolic debugger that provides single-instruction execution and breakpoint capabilities, the development environment becomes a powerful tool for testing and debugging the implementation code.

We have found many instances where the precise control and monitoring provided by the development environment revealed flaws in the protocol implementation that had not been observed in the operational system. For example, early versions of the membership protocol contained a flaw that occasionally caused the protocol to need two or three attempts to form a new membership rather than the single attempt that should have sufficed. Because of the high speeds involved, even multiple attempts to form a new membership were completed within one-tenth of a second and were not apparent in the operational system, but were obvious in the development environment.

Since our workstations are small single processor systems, we have not been able to measure the improved performance that should be obtained from the development environment running on a multiprocessor workstation. Instead, to exploit our network of small workstations, we are currently building a version of the development environment to model multiple rings interconnected by gateways. Each ring and the processors attached to it are simulated on a single workstation. The code representing the gateway between rings is split into two pieces, one for each ring to which it is attached. Each piece runs on the workstation that simulates its ring, and the pieces communicate over the Ethernet using the Totem protocol itself to relate the activities on the various rings. We believe that this version of the development environment will allow us to model quite large networks without loss of the benefits experienced from the current version.

6 Related Work

The approach we have adopted in the development environment derives in part from research into the use of parallel or distributed systems for simulation, in particular the research of Jefferson [8] and Misra [11]. Jefferson's Virtual Time is based on a highly optimistic concurrency control algorithm within which processes always proceed immediately without waiting for complete information and, thus, may be required to roll back. Misra's approach, in contrast, is based on a more conservative concurrency control algorithm in which roll back is never needed. Our approach is, however, much closer to that of Lubachevsky [9], which exploits the known minimum computation time of processes to allow a bounded lag between one process and another without risk of needing rollback. A recent theory developed by Bagrodia, Chandy and Liao [3] unifies these approaches.

Less satisfying is the current state of research into the development, testing and debugging of parallel and distributed systems and communication protocols. Several systems, such as those surveyed by Marinescu *et al* [10], focus on recording the sequence of events during an execution, disturbing the operation of the system as little as possible, and repeating the sequence of events in a controlled fashion to allow traditional invasive monitoring and debugging techniques. We have not adopted this approach because, without extensive hardware support, accurate monitoring of event sequences introduces significant perturbation, provides no control over the execution, and offers little ability to inject faults at precisely the points required for our test scenarios.

The interesting and important work of Bates [5] applies sophisticated abstraction and filtering techniques to reduce the large amount of information generated by the execution of a concurrent system, allowing the user to focus on one specific aspect of the behavior of the system. We have developed monitoring, filtering and graphical visualization tools for our communication protocols. By running such monitoring tools in conjunction with the development environment, we avoid the substantial perturbations that are inevitable when computationally expensive abstraction functions are used to produce displays meaningful to the user.

Much closer to our approach is the work of Bagrodia and Shen [4], who envisage a design process that begins with a conventional performance simulation and then refines that performance model in many steps into the operational system. Their simulation environment is more general than ours and is capable of accurately modeling the performance of a system. It is, however, considerably more invasive and the transition between a partially implemented process and a physical process is nontrivial. We achieve a seamless transition from development environment to operational system in part because our development environment is custom coded with knowledge of the Totem protocol, which Bagrodia and Shen strive to avoid. Gburzynski and Rudnicki [6] have adopted an approach similar to that of Bagrodia and Shen with more focus on performance modeling.

7 Conclusion

We have described a development environment for the Totem reliable ordered broadcast protocol. The development environment is a discrete-event simulation testbed that models the execution of the protocol implementation and provides support for testing and debugging of the protocol. It has proven to be a valuable and effective tool in developing the Totem protocol.

The development environment is novel in that it provides a seamless transition between executing the protocol within the operational system and within the simulation environment. By sharing the implementation code of the Totem protocol between the simulation environment and the operational system, we are able to model the protocol as it would execute in the operational system without introducing inconsistencies in the protocol's behavior. By measuring the performance of the implementation in the operational system and using those measurements in the development environment, we are able to maintain realistic timing information even though the protocol code being executed contains invasive debugging mechanisms.

The development environment is coded in a manner that is relatively specific to the Totem protocol. It does, however, embody the interesting idea that a development environment should operate by deriving functional behavior from an execution of the implementation code and timing behavior from a performance model of the implementation. We believe that the idea can be made completely general although that was not our aim. The current version of the development environment can probably be made to run much faster. We have not tried to do that since our objective was simply to provide an environment to aid in the testing and debugging of the Totem protocol.

References

[1] D. A. Agarwal, P. M. Melliar-Smith and L. E. Moser. Totem: A protocol for message ordering in a wide-area network. In *Proceedings of the First International Conference on Computer Communications and Networks*, pages 1-5, San Diego, CA , June 1992.

[2] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems*, pages 551-560, Pittsburgh, PA, May 1993.

[3] R. L. Bagrodia, K. M. Chandy and W. T. Liao. A unifying framework for distributed simulation. *ACM Transactions on Modeling and Computer Simulation*. 1(4):348-385, October 1991.

[4] R. L. Bagrodia and C. C. Shen. Midas: Integrated design and simulation of distributed systems. *IEEE Transactions on Software Engineering*, 17(10):1042-1058, October 1991.

[5] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*. Sigplan Notices, 24(1):11-22, January 1989.

[6] P. Gburzynski and P. Rudnicki. LANSF: A protocol modeling environment and its implementation. *Software Practice and Experience*, 21(1):51-56, January 1991.

[7] IBM Corp. NetBIOS Application Development Guide, S68X-2270-00, April 1987.

[8] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404-425, July 1985.

[9] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple loop networks. *Communications of the ACM*, 32(1):111-123,131, January 1989.

[10] D. C. Marinescu, J. E. Lumpp, T. L. Casavant and H. J. Siegel. Models for monitoring and debugging tools for parallel and distributed software. *Journal of Parallel and Distributed Computing*, 9,(2):171-184, June 1990.

[11] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39-65, March 1986.

[12] P. M. Melliar-Smith, L. E. Moser and D. A. Agarwal. Ring-based ordering protocols. In *Proceedings of the International Conference on Information Engineering*, pages 882-891, Singapore, December 1991.

[13] J. Postel. User Datagram Protocol. RFC 768, DDN Network Information Center, SRI International, August 1980.

[14] J. Postel. User Datagram Protocol. RFC 791, DDN Network Information Center, SRI International, September 1981.