Applying Distributed Simulation to a Communication Protocol Development Environment

W. J. Chun *, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal *
Department of Electrical and Computer Engineering
University of California, Santa Barbara
{wesc, moser, pmms, Deb}@alpha.ece.ucsb.edu

Abstract

Creating robust communication protocols for distributed systems is an inherently difficult task due to the many possible executions and message orderings. We describe a communication protocol development environment based on a discrete-event simulator. We have used this development environment in testing and debugging a fault-tolerant totally-ordered multicast protocol operating over a network of multiple interconnected local-area networks (LANs). Each LAN is simulated by a processor participating in the testbed, and a “virtual gateway” mechanism is used to interconnect the LANs. The resulting distributed simulation environment allows the developer to create reproducible test scenarios, to inject faults such as network partitions, to single step protocol execution and, in general, to observe and analyze protocol behavior across each host in the network.

Introduction

As distributed systems become the prevalent choice in computing, increasingly many applications are relying on communication protocols that allow hosts of such a system to collaborate. Creating a robust communication protocol for distributed systems is an arduous task because of the many possible executions and message orderings. Testing, verification, and debugging of such a protocol present even more of a challenge due to the inability to create and control an operational environment in which failures occur exactly as desired. The state of the network is dynamically changing, is difficult to monitor, and is easily adulterated by any direct manipulation of a node.

These problems are addressed in our protocol development environment, the core component of which is a discrete-event simulator. This simulation testbed consists of models of the processors as well as of the network. The processor models execute the Totem protocol and use the model of the communication medium to pass messages.

Motivation

The need for a protocol development environment arose during the development of the Totem multi-protocol system [1][2][4][18][20][23] at the University of California, Santa Barbara. The initial protocol development environment for Totem [10][11] provides a centralized simulator that allows a developer to monitor and control protocol execution for each processor in a single LAN. The user can single-step protocol execution as well as inject faults into the system. A graphical user interface (GUI) monitor [15][16] was added to the development environment to allow developers to visualize the system configuration dynamically, observe processor membership changes, and analyze protocol behavior.

However, as Totem was extended to support totally-ordered multicasting and processor membership throughout a network of multiple interconnected LANs, it became clear that the initial development environment was no longer adequate. The centralized simulator could only simulate a single LAN on a single processor. The creation of the virtual gateway [9] enhanced the development environment so that we can now simulate multiple interconnected LANs simultaneously, providing a distributed simulation of an entire network. In addition to simply connecting and coordinating the individually-simulated single LANs, the development environment also ensures that messages passed between rings are delivered by Totem in the same total order as in a live system.

(*) W. J. Chun is currently with Sun Microsystems, Inc., and D. A. Agarwal is currently with Lawrence Berkeley National Laboratory.
Background

Totem Services

The Totem fault-tolerant communication system provides ordered multicast delivery and processor membership in a distributed environment. Ordered delivery is useful for achieving consistency of replicated data in a system where physically distributed processors communicate via message passing and contribute to a collective computation. A partial order on messages may be derived from the causally-ordered events of sending and receiving messages [17]. A total order on messages ensures that all processors receive and process these messages in the same linear sequence, a sequence that preserves the causal order.

Guaranteeing delivery of messages to each processor in the same total order greatly simplifies the problem of maintaining the consistency of replicated data. In contrast, different message orders may result in inconsistent execution and inconsistent replicated data. We distinguish between the receipt of a message from a lower layer of the protocol hierarchy (such as the communication medium) and the delivery of a message to a higher layer (such as the user application).

Distributed systems may be affected by a variety of events, including processor failures and recoveries, message loss, and network partitions and remerges. Processor membership in such a system is inherently dynamic. To ensure consistency, each processor in the system must maintain the same view of the membership as the other processors at all logical times. As processors or groups of processors join, leave, or rejoin the system, membership configurations at all of the operational processors must adapt to ensure consistency.

The Totem system addresses this need for consistency by providing reliable totally-ordered delivery of messages over both single and multiple interconnected broadcast domains, even across the failures described above. Totem also provides processor membership and topology maintenance services to enable the protocol to function continuously without inconsistency when faults occur.

Totem was developed from prior research into ordered message delivery and processor membership systems. These systems include Isis [7], Amoeba [14], Trans and Total [21][24], Transis [3], and Delta-4/AMp [27].

Totem Model and Architecture

Totem operates over any number of broadcast domains, or LANs, and any number of processors within a broadcast domain. Imposed on each broadcast domain is a logical or virtual “ring” of member processors participating in broadcast message exchange, collectively performing a distributed computation. Broadcast of messages within a domain is controlled by a special message called the token, which is passed sequentially around the ring of processors. Totem is designed to operate on top of an unreliable multicast domain. The current implementation operates on top of the User Datagram Protocol (UDP) [26] with multicasts over the standard 10 Mbps Ethernet, but Internet and Asynchronous Transfer Mode (ATM) multicasts are equally viable.

Like most protocols, Totem segregates functionality in a layered hierarchy. There are four layers in the Totem hierarchy, as shown in Figure 1. They are the Single-Ring Protocol (SRP), the Multiple-Ring Protocol (MRP), the Process Group Interface (PGI), and the User Application (UA).

At the lowest level of the hierarchy, the Totem SRP is responsible for reliable ordered message delivery, fault detection, and membership for the processors in a single broadcast domain. The SRP sends messages by invoking unreliable multicasts in a broadcast domain, and receives and delivers messages up the Totem hierarchy.

At the next higher level in the hierarchy is the Totem MRP. The services provided by the MRP are similar to the SRP in that it ensures reliable ordered delivery of messages and maintains the processor membership. However, the MRP provides these services to processors in a network of multiple interconnected broadcast domains. In addition, the MRP provides topology maintenance services. Processor and communication failures and recoveries are handled consistently throughout the network.

Figure 1. Totem system hierarchy. All four hierarchical layers are linked into a single object module executed by a single processor. Underneath the Totem system is an unreliable broadcast medium.
The notion of process groups in Totem [18] provides a further layer of multicasting abstraction on top of that already imposed by the broadcast domain. An application whose messages cross broadcast domains might desire delivery only to specific processes on selected rings and not to all processes within the network. A process group consists of a list of processes and corresponding processors to which messages may be sent.

The upper-most layer in the Totem hierarchy is the User Application layer. An Application Programmer’s Interface (API) is provided for application developers who wish to use the Totem system as a reliable means of distributed communication.

Totem Development Environment

The protocol development environment created for Totem consists of a discrete-event simulator and a graphical user interface monitor. This environment gives the developer the ability to monitor and visualize protocol behavior step-by-step. By injecting faults into the system, the engineer can verify robustness and correctness of the protocol. Information that would have been impossible to obtain in a live system can be retrieved and analyzed. Although the development environment was created specifically for Totem, modifications can be made to support the design and development of any communication protocol system similar to Totem and its layered architecture.

Use of this simulation environment to aid in the development and debugging of other communication protocols requires the target system to be able to interface to the “network” using the same Unix IPC delivery mechanism (for the CMM to function), and a processor library capable of sending and receiving messages, in addition to the standard application processing of these messages (PM processor participating in a distributed computation). Thus, a target protocol will never require modification to a different set of calls to perform “network communication” in the simulation environment.

A unique feature of our development environment is that it uses the same protocol source and object code as the live system implementation. One of the goals of the simulator was to achieve consistency of execution between the simulated and operational environments. The target protocol should exhibit similar, if not exact, behavior in either environment. In addition to execution consistency, this architecture also allows for source code consistency. Code or design changes and bug fixes in the protocol are reflected immediately in both environments. Figure 2 indicates the relationship between the Totem simulation and implementation object code libraries as components of the final executables.

To build either a simulation testbed or live operational system, the appropriate processor and communication modules are selected and linked to the target protocol — in this case, it is Totem.

Simulation Testbed Architecture

The simulator consists of a module that imitates the physical communication medium and network, called the communication and medium model (CMM), and one or more processor models (PMs), all of which execute as individual processes on a single host, communicating via the CMM using Unix UDP sockets. The combined, concurrent execution of a CMM and a set of PMs comprises a complete protocol simulation as shown in Figure 3.

Communication and Medium Model

If the simulator consisted solely of processor models which executed directly on the network, it would furnish few of the facilities that are necessary to test a complex communication protocol. It would be difficult to monitor processor behavior, and we would lack the ability to control the medium, for example, by injecting faults. Instead, we created a separate model for the network. The CMM serves as an avenue for the PMs to exchange messages within scenarios created and monitored by the user.

The CMM models a communication medium by accepting messages for broadcast and delivering them to the intended recipients, just like a real network. Two key components of the CMM are a simulated Unix select...
system call and simulated network input/output (I/O) buffers.

The CMM maintains and processes two event queues as part of the event-driven simulation environment. Events generated by PM messages and subsequent results of those messages are placed in the main event queue for processing. When messages are received by the CMM, they are timestamped according to an auxiliary timing model maintained by the CMM and inserted into the event queue. The CMM processes all events in the queue, and when no earlier event is possible, the CMM returns to waiting for messages from the PMs. Simplified models of the CMM’s state machine and flow of control are shown in Figure 4. A secondary event queue is used to process special events, such as a network partition.

Failures are currently injected into the system via a configuration file which is scanned when the CMM starts up. In this file are predetermined events and (global) times for network failures. In the future, the CMM will be enhanced so that users can inject faults into the network dynamically.

Simulated time is maintained in a global time variable that is shared among the CMM and all of the PMs. Since this is a discrete-event simulator, each event processed by the CMM drives global time forward. The management of global time is similar to Jefferson’s optimistic Virtual Time algorithm [13] in which local time is advanced aggressively as possible, but the Global Virtual Time (GVT) is only advanced when it can do so “safely,” without violating any logical process’ local time or events in the system. Just as in a live system, neither the Time Warp algorithm nor our simulator allow (global) time to roll back. Our approach differs, however, from Jefferson’s in that our logical processes don’t maintain local times. Rather, all the PMs share the same global time as the CMM.

Although the CMM and all of the PMs have access to the shared-memory global time variable, only the CMM can update it since the CMM is the only entity in the system to have a global view of all the events that occur in the network. As the CMM processes events from the event queue and advances global time, each PM is notified of events such as the arrival of a message or the token. As the CMM executes its event loop, all of the PMs process their queues in parallel independently. The CMM scheduling algorithm ensures that global time is not advanced faster than the execution rate of the slowest PM and prevents any PM from advancing too far into the future, causing it to miss an event generated by a slower PM. This property lets us avoid the rollback (and its cascading effect) of Local Virtual Time (LVT) as in Time Warp. Details of the CMM state time, flow of execution, and scheduling algorithm can be found in [10].

The Processor Model

The PMs, like the CMM, execute as individual processes on a single Unix workstation. Processor failures during protocol execution are simulated by simply killing PM processes. Likewise, new or recovering processors are simulated by (re)starting new PMs.

A PM is composed of the Totem object modules and a simulation processor interface library, which provides the
simulated functionality of a real processor. For every CMM, there are one or more PMs executing concurrently. The processor model, with its embedded Totem multi-protocol stack, is presented in Figure 5.

When a simulated processor sends a message, the transparency provided by the PM and the Totem interface creates the appearance that a real processor is sending a message on a real communication link between sites! In actuality, we are still using interprocess communication, but between three processes instead of two. The source PM (a single process) sends a message to the CMM (another process), which in turns forwards it to the destination PM (yet a third process).

**Motivation for the Virtual Gateway**

By using one CMM and one or more PMs, we can simulate a single broadcast domain using the Totem SRP. In general, a network is comprised of a collection of any number of LANs interconnected with one another, and it is expected that messages will be forwarded from one LAN to another in order to reach their destinations.

The Totem MRP addresses this issue. Likewise, so should the simulator. A PM on one simulated ring should be able to transmit a message onto its CMM, be sent through a gateway (PM), and be forwarded by another CMM to the destination PM. The key question here is how to create the image of a single gateway processor which interfaces to a pair of CMMs, each of which executes on a distinct host. The simulated gateway itself would also have to execute as two processes, one on each host.

In a live implementation, this issue does not arise since a single gateway processor is a physical member of both rings. Because of the testbed design and constraint of one ring per CMM and per processor, simulating multiple rings requires multiple processors. The MRP layer of Totem handles a live gateway as a single process (which it is); however, two processes are required to create a simulated gateway. Since we refrain from modifying the protocol code directly, we have devised a simulation mechanism called a virtual gateway so that a pair of processes, each running on separate hosts, can cooperate to provide gateway functionality. Independently executing centralized simulations can now work together in the resulting distributed simulation.

**The Virtual Gateway**

Figure 6a illustrates a gateway processor in the live system. The dashed line indicates where we divide gateway functionality for the simulator. The resulting simulation virtual gateway is shown in Figure 6b. Because we must run a gateway as two separate PMs on separate rings, some interesting issues arise. For example, which protocols should each side of a virtual gateway execute? Or, which variables must maintain a “global” status between
both sides of the virtual gateway? And which type of message management should exist between two broadcast domains connected by a gateway?

Non-gateway PMs execute both the single and multiple LAN protocols (SRP and MRP, respectively). In a virtual gateway, however, only one of the two components executes the MRP. Recall that the system is simulating one gateway on two processors, rather than two gateways on two processors. If both PMs were to execute the MRP, then the system would consist of the invalid configuration of two adjacent gateways situated between both rings.

Although there could be many PMs per ring, only one can participate as part of an instance of a virtual gateway. A ring could be interconnected to two others, with two PMs serving as components of distinct virtual gateways. Two gateways between the same pair of rings (in serial or in parallel) are prohibited.

To illustrate the concept of a virtual gateway, Figure 7 presents the gateways in both the live implementation and simulated Totem environments. A single processor serves as a gateway in the live system whereas, in the simulated system, two are required to provide gateway functionality.

**Virtual Gateway Components**

For a virtual gateway to operate, we have two physical processors, each executing a CMM and one or more PMs. Since message management and processor membership in each ring are managed by the Totem SRP, all virtual gateway functionality occurs at the MRP level. Each virtual gateway component is a single PM with some gateway functionality.

Although we are using two processes to simulate a single gateway, the segregation of duties does not divide evenly. Both virtual gateway processes execute their own SRP, but only one executes the MRP, as described above. We designate a master-slave or major-minor relationship between the two components of a virtual gateway. The “lifeline” between components of a virtual gateway is a lossless communication link, implemented using the Transmission Control Protocol (TCP) [25].

The processor acting as the major component serves as the portion of the virtual gateway that executes both the single and multiple LAN protocols (i.e., Totem SRP and MRP) as well as any higher-level layers. Initialization includes setting up the locally-simulated ring followed by initialization of the remote ring.

The MRP within the major component of the virtual gateway must be able to receive messages from, and deliver messages to, each of the two SRPs and their rings. In a simulation, one of these two SRPs must operate on a remote processor interfaced to a different CMM. Instead of directly contacting the (remote) SRP, the MRP interfaces to a communication stub, the major stub, that provides the abstraction of a real SRP, a “virtual SRP,” if you will. Similarly, the layer above the remote SRP is the minor stub which serves as a “virtual MRP.”

When the MRP forwards a message to the remote ring, its call to the SRP is actually linked to the major stub, which encapsulates the message in a virtual gateway packet and passes it to the minor stub. The minor stub extracts the message, invokes a call to its local SRP, which transmits the message to be broadcast by the CMM for the (remote) ring. Messages traveling in the opposite direction are subject to a symmetric procedure.

The code for the Totem MRP is unchanged and is unaware that it is interfacing to the major stub acting as a
virtual SRP rather than as a real SRP. Similarly, the code for the remote SRP is identical and also not cognizant of the fact that it is interfacing to the minor stub acting as a virtual MRP rather than to a real MRP.

Virtual Gateway Failure

If the lossless communication link is severed in any way, this constitutes a (virtual) gateway failure. Such a fault occurs when one of the components of a virtual gateway determines that the communication port to its partner is no longer valid, due to a system failure or loss of a message between components. When one component detects the failure of its partner, it simulates a complete gateway failure by terminating itself. Termination involves closing all network ports and exiting with an error status from the PM. The Totem protocol executing in the other PMs will detect the failure, reconfigure the domain, and recover from the fault.

Congestion Control

There are two methods of congestion control built into a virtual gateway, complementing those provided by the Totem MRP. One is a filtering mechanism for preventing messages from returning to their source, and the other is a network congestion control mechanism.

Filtering mechanism. Since the minor component does not run the multiple-LAN protocol, it has no knowledge of the messages it has received from the major component to forward. If a PM on the major component’s CMM originates a message, that same message should not be resent when the minor component’s ring has messages to forward. The minor component filters messages that return to the originating ring, preventing unnecessary duplication.

Although this filtration method is quite suitable for preventing the round-trip of messages between the major and minor components of a virtual gateway, it does not help control the overall system message flow among all the processors. To avoid unlimited buffering in the system for messages destined for the communication medium, or at the application layer for messages intended for the user, a network-wide congestion control mechanism is necessary to control the total number of messages in the network.

Token Block Mechanism. The Totem SRP provides fairness in the form of flow control, limiting the number of messages which a processor can transmit in one circulation of the token. When messages are produced faster than a receiving host can process them, this information is passed to the rest of the system to reduce network traffic temporarily until more buffers are available, thus preventing the loss of messages due to lack of buffers.

Saturation can occur at the user-level as well as at the network level. For situations in which the application cannot process messages as fast as they arrive or the application transmits more messages than the network buffers can hold, a mechanism for network-wide congestion control is again required.

In the Totem MRP, one such mechanism is implemented using the token. A simple algorithm is employed to determine the maximum buffer size before overflow and the minimum buffer size before messages are allowed again. The form of backpressure applied is a block notification that is disseminated by the token to all processors on the ring.

In a live implementation, the token blocks all processors of a ring in a single rotation. Gateways, which receive more than one token, pass on the blocking signal to all of the interconnected rings. To mimic this functionality in the simulator, this blocking signal is communicated between the virtual gateway components, which then pass it on to the rest of the network.

TCP Deadlock and Resolution

Although TCP provides a seemingly reliable means of communication between the major and minor components of a virtual gateway, the possibility of deadlock does still exist. Because messages are sent asynchronously, messages can be forwarded across the gateway faster than either partner can receive and process them. When the components are forwarding data to each other in such a manner, and input buffers fill to capacity, blocking occurs until there is room available to receive incoming messages. If both sides are transmitting and the buffers fill simultaneously, neither has the opportunity to clear its receive buffers, resulting in deadlock.

Both Lubachevsky [19] and Misra [22] note how deadlock may occur in a distributed system, and ours is no exception. The deadlock problem in the development environment was first identified in [10]. Ciarfella also provides two suggestions for resolving the deadlock. An independent query to a panel of experts [8] also yielded a potpourri of viable alternatives.

Our solution to the TCP deadlock problem consisted of two steps: revising the data structures and preventing buffer overflow. Instead of the operating system kernel’s TCP buffers, we implemented a user-level queue. This data structure is managed by a simple algorithm, based on a sliding window protocol, to ensure that the kernel’s TCP buffers never reach capacity.

Queue management functions were developed to determine whether to call the transmit function immediately, or
to delay the transmission by buffering the message. The
decision of whether to send or enqueue a message is based
on the size of the sliding window, the current number of
remaining slots in the window, and the current size of the
queue.

Although the size of the queue is unbounded in theory,
it is preferable if this size is less than the remaining num-
ber of TCP buffers. Therefore, in the name of simplicity
and code reuse, we again employ Totem’s blocking mecha-

nism. We control the overall flow of messages for these
queues in the same way that Totem does for its network
queues. When the queue reaches the predetermined “max-
imum” size, we set the blocking flag and an internal flag as
a reminder that it was this queue that set the flag. This flag
is then distributed to all of the other rings in the manner
described earlier. Similarly, once the queue has decreased
to an appropriate level, the blocking flag is unset and oper-
ation proceeds as normal. By limiting the total number of
messages in the network at any given time, no buffers
should ever fill to capacity, thus avoiding deadlock. Spe-
cific details on the data structures used can be found in [9].

Rather than creating a completely new scheme to man-
age the queues, adding to the complexity of the virtual
gateway operation (and quite possibly affecting perfor-
ance), we opted for the above method as being a simple
and effective solution. The virtual gateway is “just another
client” of the congestion control mechanism already pro-
vided by Totem.

System Implementation

The development environment was created on Sun
Microsystems SPARCstation 1s and IPCs running the
SunOS 4.1.3 Unix operating system and now also executes
on the more powerful Sun SPARCrStation 20s running
Solaris 2.4. Using the distributed simulation capability of
the simulator, we’ve been able to set up a variety of net-
work topologies for Totem. We have also been able to
identify and resolve problems in Totem that would have
been difficult to ascertain using a live environment.

Related Work

Misra [22] describes traditional discrete-event simula-
tion systems as lacking in terms of simulating a large sys-
tem. With the advent of modern computer communica-
tions, it has become feasible to execute such simulations
across a network of computers. In his paper, Misra
describes how a simulation can be distributed among a set
of interacting machines. The initial simulator of our proto-
col development environment follows a similar transi-
tion. As a valuable tool in helping to develop Totem’s
capabilities, it was limited in that simulations of the proto-
col were centralized. The “body” of a large simulation can
be carried out effectively over multiple hosts, and the vir-
tual gateway allows for such a distributed simulation.

Another protocol development environment is the
NEST system [12]. The motivation behind NEST is simi-
lar to that of our protocol development environment. They
both provide execution and code-based consistency
between the simulated and live environments. In NEST,
there are separate processor or “node” modules, but
instead of utilizing a communication medium model, each
processor model makes calls to “link functions” that simu-
late the behavior of the communication link. The node and
link functions are then integrated into a single simulation
process to be run by a “simulation server.” The specific
objective of NEST differs, however, from ours. NEST is a
more general development tool particularly oriented
towards modeling the performance of network routing pro-
tocols, whereas our development environment is focused
on the design, testing, and debugging of reliable ordered
cast protocols.

Another simulation system whose objective is to mini-
mize the difference and maximize the consistency between
the model and actual systems is MIDAS [5]. Instead of
creating separate simulated and live system modules for
Totem, we linked in the actual protocol code to obtain an
accurate model of how Totem will behave in a live envi-
nronment. The MIDAS approach uses “hybrid models,”
which contain iterative and incremental partial implemen-
tations of simulated and actual models. The MIDAS sys-
tem is more general than our protocol development
environment in that it can aid in designing and building
both the hardware and software for a distributed system.
Another attribute of MIDAS is that it supports accurate
performance measurements. The goal of our development
environment is to facilitate the development of reliable
ordered message delivery protocols by providing a suitable
and controllable environment in which to test and debug
the target protocol.

Although we do not provide a full treatment of the
topic, the simulation time of the development environment
is a crucial component to the accurate modeling of events
in the system. Using multiple computers to perform dis-
tributed simulation introduces new problems in reflecting
event occurrences. Approaches can be highly aggressive,
where rollbacks may occur [13], or ultra-conservative,
where rollback is never required [22]. Lubachevsky’s
approach to event scheduling [19] is based on using what
he calls the bounded lag time and minimum propagation
delay to determine whether or not an event should be
admitted for execution, thus reducing the chance of roll-
back. The simulator uses a method similar to Lubachev-
sky, but adapted to an environment where rollback cannot, and does not, occur.

An alternative mechanism for debugging distributed systems comes from Bates [6]. In his paper, Bates describes how the use of behavioral models can be used to analyze the difference between expected and actual behavior. These models can be employed to “classify” events according to particular model characteristics. Once this model is identified, appropriate action can be taken to debug the system based on the expected behavior. In the protocol development environment for Totem, such models are not necessary since we have incorporated the protocol code directly into the simulation models. Actions taken by Totem are exactly those which would occur in a live system environment.

**Conclusion**

Communication protocols that provide reliable ordered message delivery are important for distributed systems. The lack of a controlled environment impedes the testing, debugging, and validation of such protocols. This motivation led to our creation of the Totem protocol development environment. The discrete-event simulator in this development environment allows a developer to inject faults in a simulated environment and to obtain an accurate representation of how the protocol handles these faults. As our target protocols matured with new capabilities that provide reliable ordered delivery to more than one interconnected LAN, the simulator was unable to continue providing the service for which it was created.

The simulator was thus enhanced with the virtual gateway mechanism. It allows us to link concurrent simulations of single LANs on individual processors into one large distributed simulation of interconnected LANs on multiple hosts. Although the possibility of deadlock exists, we were able to devise a congestion control scheme to limit the number of messages in the system overall.

The refined simulator and its potential to be adapted to similar protocols make it a useful tool in designing, developing, analyzing, debugging, and testing reliable and robust communication protocols. As computing services become more distributed over a computer network, applications will depend more on communication protocols that provide reliable ordered message delivery. The protocol development environment will enable protocol developers to meet the needs of these applications, now and for the future.

**References**


