Optimistic Parallel Simulation of Reliable Multicast Protocols [1]

Dan Rubenstein, Jim Kurose, Don Towsley
Department of Computer Science
University of Massachusetts
Amherst MA 01003 USA
{drubenst,kurose,towsley}@cs.umass.edu

Technical Report 97-67
Department of Computer Science
December 1997

Abstract

Parallel discrete-event simulation offers the promise of harnessing the computational power of multiple processors in order to reduce the time needed for simulation-based performance studies. In this paper, we investigate the use of *optimistic parallel simulation techniques* in simulating reliable multicast communication network protocols. Through empirical studies (using the TeD simulation programming language, the Georgia Tech time warp simulator, and a 12-processor SGI Challenge), we find that these parallelized simulations can run noticeably faster than a uniprocessor simulation and, in a number of cases, can make effective use of parallel resources. These results are somewhat surprising because reliable multicast protocols require considerable communication (and hence synchronization) among different network entities.

1 Introduction

There has been a considerable increase in the use of simulation by members of the networking research community as a means for evaluating proposed network architectures and protocols. This is because simulation provides an excellent alternative for examining the large, detailed, and complex network models that are often not amenable to analytic modeling. One potential drawback of simulating such models, however, is the amount of time needed to produce the simulation results. Uniprocessor simulation is always limited by the speed of the underlying processor. Parallel discrete-event simulation, however, offers the promise of harnessing the computational power of multiple processors in order to reduce the time needed for simulation-based performance studies.

In this paper, we empirically investigate the use of so-called "optimistic" parallel simulation techniques in simulating reliable multicast protocols. We use the TeD simulation programming language [1] for specifying the network model and the Georgia Tech time warp simulation engine [2] to run the simulation code produced by TeD. We find that on a 12-processor SGI Challenge shared-memory multiprocessor, these parallelized simulations not only run noticeably faster than an unparallelized uniprocessor simulation, but also make fairly effective use of the available parallel processing resources. This differs from our initial hypothesis that simulation of multicast protocols would not parallelize effectively due to the significant amount of inter-processor communication (and hence synchronization) that such protocols would require. In addition to presenting and analyzing these measurement results, we also discuss the ways in which the multicast models stretched the limits of TeD, in some cases resulting in new language and/or simulator features.

[1] This material was supported in part by the National Science Foundation under Grant No. NCR-9527163 and CDA-9502639. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the University of Massachusetts.

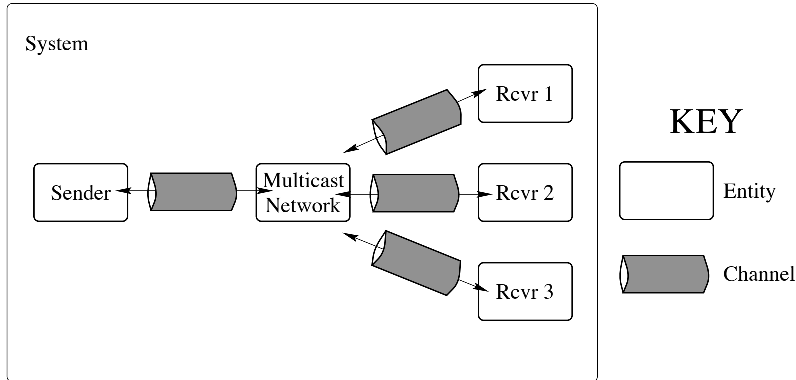


Figure 1: Sample model of the network used in simulator. Here, the multicast session involves 3 receivers.

The remainder of this paper is structured as follows. In the following section, we briefly overview the TeD language and the GTW mechanisms for providing optimistic parallelism. We also describe the protocols that we simulate, and the network topology used. Language and simulation-related issues that we encountered while developing the models are described in Section 3. Our empirical results in Section 4 focus on speedup and rollback in the parallel simulations. Section 5 concludes the paper.

2 Simulation Environment and Network Protocol Models

In this section we briefly overview the TeD simulation language, the GTW parallel simulator, and the network protocols and network models considered in this study.

Our multicast protocol models are specified in TeD [1], an object-oriented discrete-event modeling language based loosely on the VHSIC Hardware Description Language (VHDL) [3]. Here, we only briefly describe TeD; a more complete description can be found in an accompanying article in this special issue of *Performance Evaluation Review*. TeD provides three basic types of modeling objects: *events, entities*, and *channels*. *Events* are objects that move through the network; in our modeling domain, a simple example of an event is a packet. *Entities* are objects that process events. As shown in Figure 1, we use entities to model packet processing at the sending and receiving network hosts (end-systems). As discussed below, we also use a single entity to model the multicast network; its role is to model the multicast delivery (and loss) of packets from one network host to another. TeD entities are connected to one another via *channels*. A channel has two ports, and can be unidirectional or bidirectional. Network propagation delays can implemented by associating a delay parameter with a channel input event. This results in an event (e.g., packet delivery at the other end of the channel) occurring at the other end of the channel after the given amount of time.

Once a model is defined in TeD, it is pre-processed into a lower level language of choice (e.g., C++ [4] in our studies), which is then compiled to create an executable. The executable makes use of the GTW parallel simulation architecture [2], an event-based optimistic parallel simulator. GTW operates on multiple processors by initiating a parent process which then forks off child processes so that there is a single process on each processor being used in the simulation. Processes communicate through shared memory. Note that because GTW is an an *optimistic* simulator, events occurring in different processes on different processors may require *synchronization* among these processes and *rollback* of the simulation state when a process finds that it has incorrectly proceeded beyond a synchronization point. See [5] and the references therein for a discussion of these issues.

Having briefly described our simulation environment, let us now turn to the network protocols being modeled. We will model a one-to-many reliable multicast session, following the model in [6]. In this model, a single sender distributes data to multiple receivers by partitioning it into *packets*, and transmitting these packets across the network. In a multicast network, the single copy of a packet that is sent by the sender will be duplicated, as needed, within the multicast network and delivered to the individual receivers. A packet can also be *lost* (e.g., as the result of a buffer overflow in a network router) and hence a sent packet may not be received at one or more receivers. We assume here that the loss processes are independent (and homogeneous) among receivers and in time.

The manner in which a receiver recovers from lost data depends on the specific reliable multicast protocol being used. In [6] three generic reliable multicast protocols are described; the reader is referred to [6] for details. One of these is an ACK-based protocol that puts the burden of detecting and recovering from errors on the sender; the other two protocols place more of the burden for loss recovery on the receivers. Each of these protocols makes use of the sender-specified sequence number on each packet to detect lost packets. Each of the three protocols also always multicasts (rather than unicasts) data packets (original packets and retransmissions) to the receivers. The main difference among the protocols is in the manner the sender is notified (if at all) about a lost packet.

- **ACK**: The ACK protocol requires each receiver to send an acknowledgment (ACK) packet to the sender for each packet it receives. This packet is sent in a point-to-point (non-multicast) manner and thus is carried along the reverse path from the receiver through the network entity to the sender in our model. We assume that ACKs (and NAKs, in the protocols below) are not lost. If, after sending or re-sending a packet and waiting a certain period of time an ACK for the packet with this sequence number has not arrived, the packet is sent again. This process continues for that data packet until the sender receives at least one ACK from each of the receivers. When a sender has no incoming packets (ACKs) to process or retransmissions to perform, it sends a new data packet. We thus assume here that the source effectively has an infinite supply of new messages to send, reflecting our interest here (as in [6]) in determining the maximum supportable throughput of the various reliable multicast protocols. A performance metric of interest not considered here is the delay associated with reliable packet delivery [7].
- **NAK1**: The NAK1 protocol requires each receiver to unicast a negative acknowledgment (NAK) to the sender for each data packet that it has not received. After a NAK transmission, the receiver sets a timer and if the timer expires prior to receipt of the data packet, the receiver resends the NAK. The sender retransmits a packet upon the first receipt of a NAK after a transmission or retransmission of the packet. Since receivers do not explicitly acknowledge received packets, the sender must infer packet receipt by all receivers by the absence of NAKs. As in the ACK protocol, when a sender has no incoming packets (NAKs) to process or retransmissions to perform, it sends a new data packet.
- **NAK2**: The two previous protocols deteriorate in performance as the receiver set grows large [6]. This is because each receiver communicates its status (ACK or NAK) to the sender, which becomes a bottleneck. The NAK2 protocol alleviates this problem by reducing receiver feedback. NAK2 is similar to NAK1 protocol with the following exceptions. Before sending a NAK, a receiver waits first for a random period of time. If it does not receive the data packet or another NAK for the data packet in this period of time, it *multicasts* its NAK. If it does receive a NAK while waiting, it does not send a NAK itself (it *suppresses* its NAK) and resets its timeout period for this packet. This process is repeated until the data packet is received. The NAK2 protocol reduces the burden at the sender but requires that receivers process both data packets and NAKs. It has been shown in practice that reliable multicast protocols adopting this approach perform quite effectively in real networks such as the Internet [8].

3 Simulation Issues

Our goals in undertaking this study were threefold. First, we wanted to verify (though simulation) a number of the modeling assumptions adopted in our earlier analytic work [6, 7]. Second, we wanted to use TeD and GTW in the network modeling domain in order to better understand and test their capabilities. Finally, we wanted to evaluate the benefits of using parallel simulation to model reliable multicast protocols. In this section, we discuss several TeD-related issues encountered; various aspects of parallel simulation of multicast protocols are discussed in the following section.

The most important difficulties we encountered with early versions of TeD concerned checkpointing and rollback. Recall that optimistic simulation requires that a processor roll back its simulation clock whenever an inconsistency is detected. This rollback is accomplished by having a processor periodically *checkpoint* each entity's state. When an inconsistency occurs, the processor restores the most recently checkpointed entity states that are not known to be inconsistent, and rolls back simulation time to the checkpoint time. TeD initially assumed that entities would have only

a small amount of state – an assumption grossly violated in our models. Recall, for example, that each host has a queue (of arbitrary size) of events to be processed. Under the ACK protocol, the sender's queue contains ACK messages and can grow quite large when there are a large number of receivers. The sender must additionally maintain a list of receivers that have not yet acknowledged each individual packet. Initially, TeD simply checkpointed the entire state of an entity at each checkpoint. Given the large amount of entity state in our models, this was inappropriate. Thus, TeD was extended with the capability of checkpointing only the portion of an entity's state that was modified. We redesigned our state-saving mechanism to take advantage of this capability. For example, our queues were converted from linked lists to extendible circular buffers. Popping an element from the queue simply involved shifting a pointer. In this way, we lessened the amount of state that changed between checkpoints.

In early versions of TeD, simulations were terminated simply by stopping the simulation at a given simulated time. For our purposes, results from a particular run were valid only if the protocol reached a certain "stable" state. It was difficult to determine when this would happen, and choosing a time that would conservatively guarantee that we reached such a state was costly in that we would often run the experiment for much longer than needed. We thus created a mechanism that allowed the network entity to determine when the termination condition had been reached, and then to send a message to all other entities to shut down. Similar functionality was later introduced into TeD to make it easier to terminate a simulation in this fashion.

Model initialization procedures also evolved during our study. Initially, model parameters were statically determined at compile time, requiring a recompilation whenever we wished to vary model parameters (e.g., number of receivers). A later version of TeD was released that allowed us to dynamically instantiate networks without recompilation by using dynamic model parameters.

One final issue that is still under investigation is to determine how TeD/GTW should best handle the high bandwidth that we sometimes require in the modeled network. In a multicast network, numerous packets (events) can be in transit among the many different links in the network (channels) at any given point in simulation time. TeD/GTW queues these events for delivery, and was designed with the assumption that few events would be in transit within channels at any point in time. The fact that this is not the case in our simulation has led to several interesting research issues in parallel simulation that are still under investigation.

4 Results

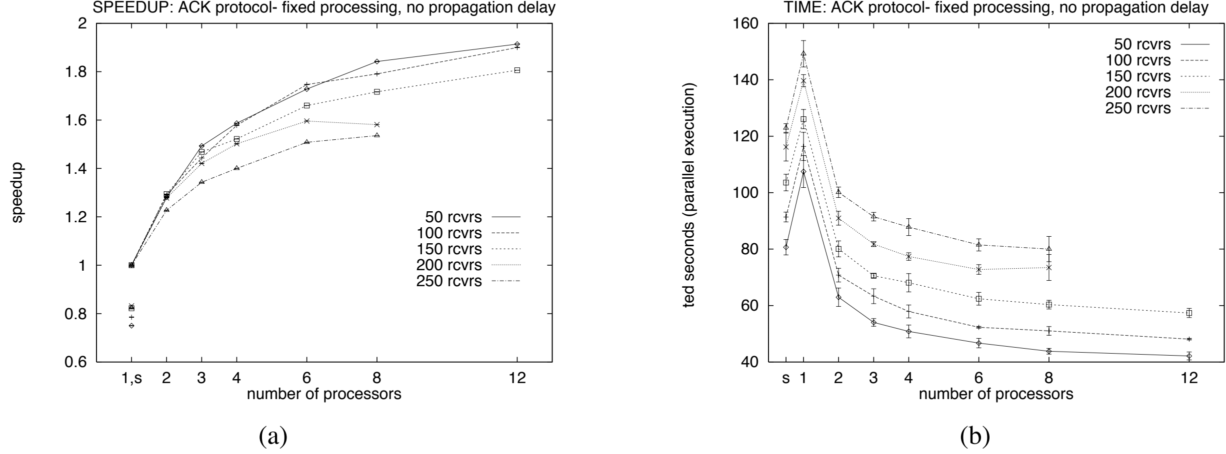
Simulations were performed on a SGI Challenge XL with 12 250 MHz R4400 processors, each with a 2-level cache, consisting of a 32 KB primary cache split equally between instructions and data, and a 4 MB secondary cache. During our simulations, no other users were logged into the machine, but standard IRIX processes were present. We use three metrics to evaluate the the performance of parallel simulation:

- The simulation *running time* itself was measured three ways: the clock time (wall time) that elapsed from the start of the simulation to its end, the clock time used by the parent process (that spawns the simulation and continues to exist until the simulation terminates) within the simulation (i.e., not counting the time used for system calls), and time used by the parent process while the simulation clock was active (i.e., ignoring both system time and simulation setup/teardown time). The variation between these times for any particular experiment was small and produced negligible differences in our observations, so all further discussion uses our third description of running time.

- The *speedup* of the simulator using n processors is measured as RT_{seq}/RT_n. RT_n is the running time of the simulator with n processors. RT_{seq} is the running time of the simulator using a special version of TeD that compiled to a sequential simulation. This sequential version is used as our normalization point for speedup (rather than the parallel version of TeD running on a single processor) because the latter performs operations (e.g., checkpointing) not needed in a true sequential simulation. The ratio RT_{seq}/RT_1 quantifies the difference between the sequential and parallel versions on a single processor. We also note that every simulation terminated after 50,000 milliseconds of virtual time had elapsed so the two simulations used to compute each speedup point always observe identical systems for identical periods of (virtual) time.

Parameter	Parameter Values
Number of receivers	50, 100, 150, 200, and 250 for ACK and NAK1, 25 and 50 for NAK2
Probability of loss	Data packets were lost with probability 0.1, acknowledgments were never lost
Propagation Delay	Fixed at 0, or varied uniformly between 10 and 500 ms
Processing Delay	Fixed at 10 ms, or varied uniformly between 10 and 50 ms

Figure 2: Values of parameters varied during experimentation



(a) (b)

Figure 3: (a) Speedup and (b) running time for the ACK protocol with a varying number of receivers.

- The *rollback percentage*. The TeD simulator returns the percentage of events that are rolled back, or the percentage of time that the state of an entity changed and that state was later considered invalid.

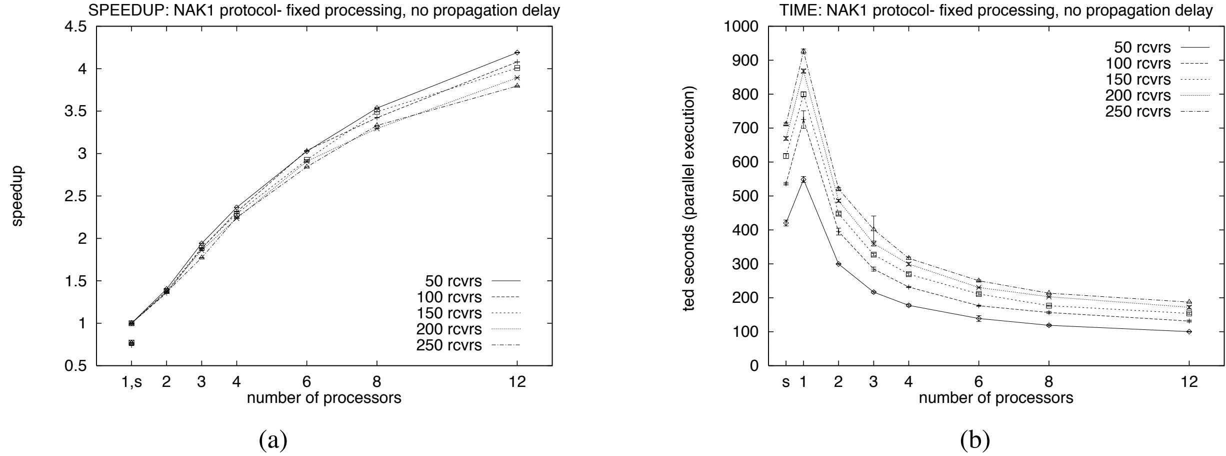
We ran the simulator for 50,000 milliseconds of virtual time under a variety of network configurations (summarized in Figure 2) in order to evaluate how the speedup offered by optimistic simulation varied with the protocol (ACK, NAK1, or NAK2), the size of the simulation (number of receivers), and the number of processors used. We note that we were unable to obtain results for the NAK1 protocol for more than 150 receivers on 12 processors, and could not get results for the NAK2 protocol when the number of receivers was larger than 50, due to TeD memory constraints. (TeD is still under revision and we anticipate that future versions will alleviate this problem.)

We ran each experiment 10 times to obtain 95% confidence intervals, and varied the network propagation delays and host processing delays (e.g., time to process a data, ACK, or NAK packet) in order to determine speedup sensitivity due to possible synchronizations. Though minor differences occurred through these variations, the overall conclusions that we draw are similar. Thus, we present results obtained from simulations with a propagation delay of zero in the network and a fixed processing time of one.

4.1 Discussion of Results

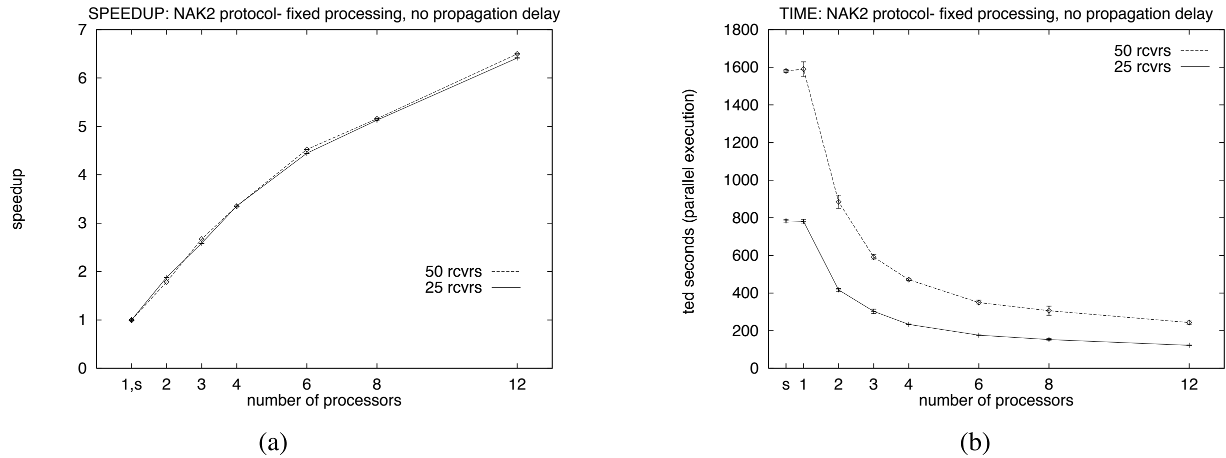
Figures 3, 4, and 5 plot the speedup and running times we measured using optimistic parallel simulation. Speedup plots are normalized to results from the sequential simulator. The disconnected points that lie below the line $speedup = 1$ gives the slowdown observed for various numbers of receivers when the parallel version of the simulator on one processor is compared to the sequential version. In the running time plots, the sequential version is plotted at points whose x-coordinate is marked with an 's'. The running time plots also display the 95% confidence intervals. Rollback percentages for the three protocols appear in Figure 6.

For the ACK protocol, increasing the number of processors to 12 almost doubles the speed of the simulation for 50 receivers. As the number of receivers increases to 250, this speedup lessens to about 1.5. For the NAK1 protocol, we see that the speedups of 12 processor simulations vary between 3.7 and 4.4. As in the ACK protocol, we see larger



(a) (b)

Figure 4: (a) Speedup and (b) running time for the NAK1 protocol with a varying number of receivers.



(a) (b)

Figure 5: (a) Speedup and (b) running time for the NAK2 protocol with 25 and 50 receivers.

gains in speedup for a smaller numbers of receivers, but the difference is not as significant. For the NAK2 protocol, we observe speedups close to 7 for simulations involving either 25 or 50 receivers.

A closer look at the manner in which simulation entities are mapped to processors, and an understanding of the computational load of each entity under each reliable multicast protocol, suggest some possible explanations for the trends observed in Figures 3 through 6. Simulation entities are placed into *processing groups* in a round robin fashion, where each processing group represents the state of a single process in the simulation. This round robin distribution ensures that the sender and network entities are mapped into different processing groups, and that a processor can contain at most one entity or one receiver more than any other process. For a simulation of a network with r receivers, simulated over p processes, every processor contains approximately r/p receiver entities. We note that the processes themselves may migrate among processors under the control of the IRIX operating system. However, the operating system uses cache-affinity based processing, so such migration rarely occurs.

We conjecture that for all three protocols, the process that contains the sender advances at a slower rate through simulated time than any other process, and thus a simulation's running time correlates directly with this process' running time. For the ACK protocol, a receiver responds with an ACK to every packet it receives. Thus, receivers require very little processing time, and so reducing the number of receivers that map to the processor containing the sender (e.g., by increasing the number of processors in the simulation) will not result in large speedup gains. This could explain the relatively small speedup we see in the ACK protocol simulation as processors are added. We conjecture that the reduction in speedup that occurs as the number of receivers is increased is due to the increased

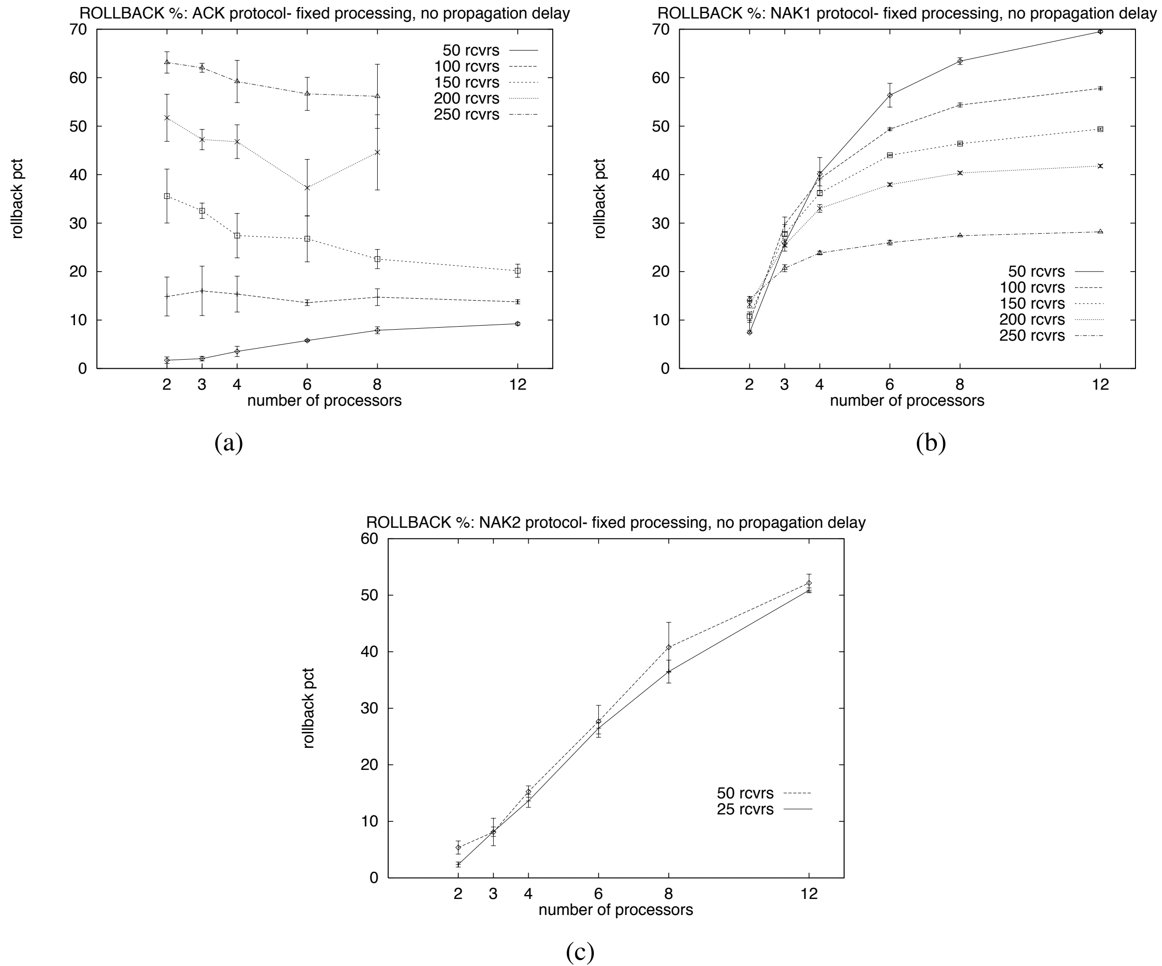


Figure 6: Percentage of events rolled back

rollback percentage that occurs as the number of receivers increases. It is possible that a portion of these rollbacks result from the process containing the sender advancing in simulated time beyond the process containing the network entity.

In the NAK1 protocol, receivers require more processing – they maintain state that tracks which packets have been received, generate timeouts and send NAKs. Reducing the number of receivers in a processor by adding processors to the simulation therefore has a larger impact on speedup than in the ACK case. However, the sender must process a packet from each sender that loses a packet. Consequently, the sender still has a relatively large processing load that is proportional to the number of receivers. Hence, speedup is still somewhat limited.

In the NAK2 protocol, the NAK suppression mechanism significantly reduces the sender's processing burden, bringing it much closer to that of a receiver's processing load. In this case, increasing the number of processors results in a more significant decrease in the processing load of the process containing the sender entity. Hence, we see higher speedups under NAK2 and a more effective use of the parallel resources.

Because all events are passed among entities through the network entity, all rollbacks involve the process that contains the network. This process either has to be rolled back to a state that is determined by a process that proceeds at a slower rate through simulated time (such as the process containing the sender), or triggers a rollback in another process that proceeds at a faster rate through simulated time (such as a process containing only receivers). A more detailed discussion of how this affects rollback is beyond the scope of this paper.

5 Conclusion

We have experimented with an optimistic parallel simulator to simulate a variety of different reliable multicast protocols and have also examined how changing the number of processors, simulation size (number of receivers), and synchronization among processors impacts the speedup of the simulation and percentage of rolled-back events. Our findings indicate that the speedup achieved depends strongly on the specific reliable multicast protocol being simulated.

Although the speedup results are positive, one can argue that there are still cases where parallel simulation might not provide sufficient benefits. For instance, if confidence intervals need to be computed, multiple simulation runs are needed. In such cases, an alternative approach to exploiting parallelism is to distribute these multiple runs among the various processors, rather than sequentially running parallel simulations. (We note, however, that computing confidence intervals by running sequential simulations on n processors does not necessarily give a speedup of n, as one might expect [9].) Yet another alternative is possible when multiple simulation experiments are to be run (e.g., under different parameter values). In this case, one could divide the experiments among the various processors and perform the different experiments in parallel. In such a case, a speedup of close to n is likely.

There are two directions for future work. First, we can expand on the network model in many ways, varying network topology, network delay, host processing times, and the number of senders in a multicast session. Second, we can examine the speedups achieved if conservative, rather than optimistic, simulation is used.

References

[1] K. Perumalla, A. Ogielski and R. Fujimoto, *MetaTeD: A Meta Language for Modeling Telecommunication Networks*, GIT-CC-96-32, Technical Report, College of Computing, Georgia Institute of Technology, 1997. URL: *http://www.cc.gatech.edu/computing/pads/ted.html*

[2] R. M. Fujimoto, *Time Warp on a Shared Memory Multiprocessor*, Transactions of the Society for Computer Simulation, Vol. 6, No. 3, pp. 211-239, July 1989.

[3] J. Bhasker, *A VHDL Primer*, Prentice Hall, 1995.

[4] K. Perumalla and R. Fujimoto, *A C++ Instance of TeD*, GIT-CC-96-33, Technical Report, College of Computing, Georgia Institute of Technology, 1997. URL: *http://www.cc.gatech.edu/computing/pads/ted.html*

[5] R.M. Fujimoto, *Parallel Discrete Event Simulation*, Communications of the ACM, pp. 30-53, Vol. 33, No. 10, Oct., 1990.

[6] D. Towsley, J. Kurose, S. Pingali, *A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols*, IEEE Journal on Selected Areas in Communications, 1997.

[7] M. Yamamoto, J. Kurose, D. Towsley, H. Ikeda, *A Delay Analysis of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols*, Proc. IEEE Infocom97, Kobe Japan, April 97.

[8] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, *A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. Proc. ACM SigComm 1995.*

[9] Phil Heidelberger, *Discrete Event Simulations and Parallel Processing: Statistical Properties*, SIAM Journal on Scientific and Statistical Computing. 9, 1114-1132.