Network Aware Time Management and Event Distribution

George F. Riley
Richard Fujimoto
Mostafa H. Ammar

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{riley,fujimoto,ammar}@cc.gatech.edu
(404)894-6855, Fax: (404)894-0272

Abstract

In this paper we discuss new synchronization algorithms for Parallel and Distributed Discrete Event Simulations (PDES) which exploit the capabilities and behavior of the underlying communications network. Previous work in this area has assumed the network to be a Black Box *which provides a one-to-one, reliable and in-order message passing paradigm. In our work, we utilize the* Broadcast *capability of the ubiquitous* Ethernet *for synchronization computations, and both unreliable and reliable protocols for message passing, to achieve more efficient communications between the participating systems.*

We describe two new algorithms for computation of a distributed snapshot of global reduction operations on monotonically increasing values. The algorithms require $O(N)$ messages (where N is the number of systems participating in the snapshot) in the normal case. We specifically target the use of this algorithm for distributed discrete event simulations to determine a global lower bound on time-stamp (LBTS), but expect the algorithm has applicability outside the simulation community.

1 Introduction

Distributed applications often require a frequent rendezvous between all participating processes to come to agreement on certain aspects of the distributed computation. For a conservative parallel discrete event simulation, a global concensus on the timestamp of the smallest unprocessed event and the number messages exchanged is a frequent occurrence. Many algorithms exist for distributed consensus agreement. These algorithms typically make some reasonable assumption about the capabilities provided by the underlying communications network, and design the algorithm to work properly given those assumed capabilities A common assumption is the *Black Box* model of network behavior, where messages are injected at one end of a network connection, and reliably come out the other end of the connection at some later time, and possibly out of order.

In actuality, the network can present differing capabilities and reliability guarantees, depending upon how the network is configured and accessed by the application. For example, the ubiquitous *Ethernet* provides, at its lowest level, a simple unreliable in-order datagram service. However, the application can normally request a reliable transport protocol (such as *TCP*), and can also request *Broadcast* service (one-to-all) or *Multicast* service (one-to-many) for individual messages. The performance achieved by the network can vary depending on the services requested by the application, and thus can affect the overall performance of the application. In this paper we show that the capabilities and reliability of the various network models can be exploited in the design of an algorithm, which results in improved performance.

Methods for time management and event message exchange within a parallel discrete event simulation have been examined for some time. Chandy, Misra[1] and Bryant[2] describe the *Null Message Protocol*, in which only the smallest available event at each simulation entity is processed. This protocol assumes there is always an event message available from all peers, and uses a null message as a placeholder when no event message is present. Mattern[3] describes a two pass algorithm for lower bound timestamp computation which is similar in spirit to our method. Mattern's algorithm establishes a consistent cut point in which all messages sent between processors have either been accounted for, or are known to not affect the computation. Chandy's[4] conditional event protocol determines a range of safe events by finding a minimum of all possible events from any peer. Lubachevsky[5] describes a *Bounded Lag* protocol for determining safe events, which takes into account the minimum simulation time delay between any two simulation objects. In the *SPEEDES* simulation engine, Steinman[6] utilizes the *Time Buckets Protocol* in which processes periodically resynchronize to determine a lower bound on events which are safe to process. Time Buckets is also similar to our approach. Nicol[7] describes the *YAWNS* protocol which is also similar to our approach.

The main contributions of our approach are threefold. First we utilize the *Broadcast* capability of an Ethernet network to give an efficient, albeit unreliable, one-to-many message dissemination capability. Secondly, we use knowledge of the inherent message ordering characteristics of the Ethernet to make assumptions about the presence or absence of *Transient Messages*. In traditional time synchronization protocols, a transient message is a message that has not been included in the global concensus because it has not yet been received and processed by the intended receipient. Lastly, we migrate the responsibility of initiating and calculating the concensus to the *slowest running* processor. This results in processors participating in the concensus only when they have no other useful work to perform.

We give two algorithms for computing a *lower bound on time-stamp* (*LBTS*) within a parallel discrete event simulation. For this discussion, we define the *LBTS* to be the minimum timestamp on any message that can possibly be received in the future. The *LBTS* algorithm is a fundamental computation used by conservative synchronization protocols as well as optimistic protocols when computing a *Global Virtual Time* (*GVT*). Our two algorithms assume differing network service models, and we show that the service model assumptions and choices can have a large impact on overall performance. The main contribution of this work is the exploitation of network capabilities (such as broadcast and unreliable event messages) to provide increased performance for large numbers of participating systems.

The remainder of this paper is organized as follows. Section 2 discusses in more detail the network models assumed in our work. Section 3 describes an *LBTS* algorithm which uses broadcast messages for time synchronization, and unreliable datagrams for event messages. Section 4 describes an *LBTS* algorithm which also uses broadcast messages for time synchronization, but uses a reliable transport protocol for event messages. Section 5 describes the experimental methodology used and the hardware platforms on which the experiments were performed, and gives results of the experiments. Finally, Section 6 gives some conclusions and future directions of our work.

2 Network Model

In this section, we discuss the basic behavior of the ubiquitous *Ethernet* network, the programming model used to access the network services, and how these affect the design and performance of our algorithms. We assume that almost all distributed discrete event simulation applications will be executed on a loosely coupled network of workstations connected by an *Ethernet* network. We assume that the programming model allows access to a connection oriented transport protocol such as *TCP*, a connectionless datagram protocol such as *UDP*, and supports both broadcasting and multicasting of datagrams (although we don't exploit the multicast capability in this work). We also assume that an application programmer can choose any combination of these programming models within a single application.

2.1 Network Reliability Models

As mentioned in the previous section, the performance characteristics and delivery guarantees provided by a given network are not as simplistic as the *Black Box* model. We can, therefore, define the several different models of network behavior. For this discussion, we define *reliable* to mean that any message sent to a single destination will eventually be delivered to that destination. We define *in-order* to mean that all messages received are received in the order they were sent.

The *Reliable In-Order Delivery* (*RIOD*) model assumes that all messages sent between any two entities will arrive correctly, and in the order they were sent. This is the model provided by the *TCP* protocol when using a single connection. The *Reliable Non-Ordered Delivery* (*RNOD*) model also assumes that all messages will be received, but they may be received out of order. This is the model provided by the *TCP* protocol when using multiple connections. The *Unreliable In-Order Delivery* (*UIOD*) model assumes that messages may be lost, but messages which are not lost are delivered in the order they were sent. This is the model provided by the *UDP* protocol when using a single port, and when all communicating systems are on a single shared bus local area network. This model applies to both unicast (one-to-one) *UDP* messages, as well as broadcast (one-to-many) *UDP* messages. Finally, the *Unreliable Non-Ordered Delivery* (*UNOD*) model assumes that messages may be lost and may be received out of order. This is the model provided by the *UDP* protocol when using multiple ports, or when the communicating systems are connected on a wide area network.

3 The *BCUDP* Algorithm

In this section, we describe an algorithm for computing a *lower bound on time-stamp* by utilizing the *Broadcast* capability of the underlying network, and assuming the *UIOD* model for message delivery. The algorithm requires exactly $n+1$ messages (n is the number of systems participating) in the best case. Experimental data presented later shows that the best case is in fact the typical case. We call this algorithm the *Broadcast UDP* (*BCUDP*) algorithm.

3.1 Assumptions

The algorithm operates under the following assumptions:

1. The underlying communications medium has a *broadcast* capability. In other words, a single system can communicate with all other systems by sending a single network message.

This implies that all of the systems participating in the algorithm are on a single local area subnetwork.

2. Any broadcast message is *NOT* necessarily received by all other systems. Our algorithm is much more efficient, however, if messages are properly received most of the time.

3. Event messages between systems are sent via some unreliable protocol (eg. UDP), and the application can tolerate lost events. Use of unreliable protocols for event message passing has been used within the *Distributed Interactive Simulation* (*DIS*) community, for example when sending real-time state update messages. Our second algorithm addresses the case where event messages are sent using some reliable protocol.

4. The underlying communications medium delivers all messages with the *UIOD* model. This implies that a single socket per process, and single *UDP* port number is used for all communications between processes

3.2 Overview

The algorithm works by designating a *Master* system that will initiate and compute the final *LBTS* value. It does this by sending a broadcast message to all other processors requesting a reply, and collecting replies until all processors have been heard from. The reply messages contain message counts sent to all peers, received message counts, and local simulation time information. After sending a reply, each system stops processing events and stops generating new messages until the *LBTS* computation is complete. If processors did not receive the initial broadcast, or if their reply was lost, the *Master* asks those processors to report again, plus any processor that might have received a transient message, and the process repeats. After all replies have been gathered, the *Master* computes the *LBTS* and informs the other systems of the computed *LBTS* value via a broadcast. The *Master* then designates a new *Master* for the next instantiation of the algorithm. As the algorithm is executed repeatedly during a distributed computation, the *Master* selection will probabilistically be the system with the *slowest* advancing simulation time. Since the algorithm requires the participants to stop generating new event messages, the slowest running simulation is the ideal choice for the *Master*. Other faster running processes will already be blocked with no more safe events to process, and thus will not be impaired by this requirement.

3.3 *BCUDP* Example

In this section we give a simple example to illustrate a typical execution of the algorithm, under three different scenarios. Assume there are four logical processes (LPs) (each on a separate physical processor) denoted LP_0, LP_1, LP_2, and LP_3. The initial *Master* is LP_0. Assume that all LPs start at simulation time T_0, have all determined it is safe to advance to time T_1, send exactly one event message to all other LPs, and advance time to T_1. At that point, all LPs are no longer able to advance simulation time, and must participate in an *LBTS* computation. Also assume all LPs are trying to advance to simulation time T_2, the time of their next local event. The time of their next local event is called their *Next Event Request* (*NER*) time.

In the first scenario, all LPs receive the initial broadcast from the *Master*, and none of the reply messages are lost. Since LP_0 is the master, it starts the *LBTS* computation by using an *Ethernet* broadcast *Start LBTS* message requesting replies from all peers, and does this at time T_1. Upon receipt of the *Start LBTS* broadcast message, all LPs report to the *Master* (LP_0) their current simulation time (T_1), their *Next Event Request* time (T_2), the count of messages sent to each peer, and the number of messages received. For this example, we assume all event messages have been received properly, so each system reports 1 message sent to each other system, and 3 messages received. The *Master* can simply calculate the *LBTS* as the smallest *Next Event Request* time reported (T_2 in this example). The resultant *LBTS* value is broadcast to all peers, along with an indication of who the next master should be.

In the second scenario, we assume that again all systems receive the initial broadcast, but this time a transient message occurs. LP_0 starts the *LBTS* computation as above and all systems receive the initial broadcast. However, LP_3 has sent a message to LP_1 which has not propogated through the protocol stack and onto the network. All LPs reply, and the *Master* notes that LP_1 has potential transient messages (since more messages were sent to LP_1 than have been received). However, once the *Master* has received the reply from LP_3, the unaccounted for message to LP_1 *must have been either received or lost.* Since the reply by LP_3 was sent after the transient message, the transient message must have also already been sent. Thus, the master can simply broadcast a *Restart LBTS* message, indicating to LP_1 that it should reply again immediately, with no further delay. Once the second reply is received, the *LBTS* can be calculated as above.

In the final scenario, LP_3 does not receive the initial broadcast, and generates transient messages by sending another event to all other LPs after their replies have been generated. Thus the second message from LP_3 to each other peer becomes a transient message. The *Master* notices that LP_3 has not replied (after a suitable timeout period), and requests LP_3 to reply again via a broadcast *Restart LBTS* message. When LP_3 finally replies (after one or more restarts), the *Master* will determine that each system should have received a total of 4 messages, but due to the transients only 3 have been received by LP_1 and LP_2. The *Master* broadcasts a *Round 2 Restart* message, indicating which systems need to reply a second time (in this example it is LP_1 and LP_2). LP_1 and LP_2 will reply again, this time reporting having received 4 messages, and the *LBTS* can be computed by the master as above. If there are still unaccounted for messages after the round 2 reporting, they can be assumed to be lost, due to the message ordering issues described in the previous section. Since all processors in this scenario properly responded to the broadcast, this implies that all event messages sent before the the replies have been either received or lost.

3.4 The *BCUDP* Algorithm

More formally, the *BCUDP* algorithm works as follows:

1. Assume there are k systems participating, designated $S_0, S_1 \ldots S_{k-1}$. The subscript k is known as the *system identifier* (*SID*).
2. Define set R to be the set of systems from which a reply has been received.
3. Define $\text{Rx}[k]$ to be of the total number of messages recieved by system k since the last successful *LBTS* computation.
4. Define $\text{Tx}[k]$ to be the total number of messages sent to system k since the last successful *LBTS* computation.
5. A single system is initially designated the *Master*. A simple way to do this is to assign the first *Master* to be system S_0.
6. When the *Master* system can no longer safely process events, it will initiate an *LBTS* computation by broadcasting a *Start LBTS* (*SLBTS*) message. The *SLBTS* message will contain:
   - The *epoch* for this LBTS computation. The epoch values simply count sequentially by one for each *LBTS* computation. The epoch values allow a system to determine if this request is a duplicate of one already processed.
   - The *SID* for the current *Master*. The replies in step 8 below are unicast directly to the *Master*.
7. The *Master* clears $R = \emptyset$, $\text{Rx}[k] = 0 \ \ \forall k$ and $\text{Tx}[k] = 0 \ \ \forall k$.
8. When system S_j receives the *SLBTS* broadcast message, it will respond to the *Master* by sending a *Reply LBTS* (*RLBTS*) message unicast to the *Master*. Note that the *Master* also replies to itself. The replies will consist of:
   - The replier's (*SID*) j.
   - The replier's current simulation time (ST_j).
   - The replier's next event time (NER_j).
   - A count of the total number of event messages received by S_j since the last *LBTS* computation completed (*ThisRx*).
   - An array $\text{MyTx}[k]$ to be an array of the number of messages sent by S_j to each other system k since the last *LBTS* computation completed.
   - An indication of whether this system was unable to continue safely processing events before this *SLBTS* message was received. This is used to determine which S_j will be the next *Master*.

   System S_j will stop processing events (and sending event messages to other systems) until the *Done LBTS* message is received (see step 13 below). However, each S_j will continue to poll the message socket, receiving, counting, and enqueing any event new event messages. (These messages are by definition *Transient Messages*, since they were received after the *RLBTS* message was sent, and thus were not accounted for and may affect the NER_j value reported).
9. The *Master* waits for replies from all other systems, or a suitable timeout. The timeout period for the replies is determined heuristically by adapting to the smallest timeout period that still allows all other systems time to reply. Upon receipt of a reply from system S_j, the *Master* will:
   (a) Add j to set R,
   (b) Set $\text{Rx}[j]$ = ThisRx, the received message count reported by j,
   (c) Set $\text{Tx}[i] = \text{Tx}[i] + \text{ThisTx}[i] \ \forall \ i = 0 \ldots k-1$. In other words, add to the count of messages sent to each system the number of messages sent to that system by S_j.
   (d) Store the reported simulation time ST_j.
   (e) Store the reported next event time NER_j.
10. If all systems have replied, there are three possible cases:
    (a) $\text{Tx}[i] = \text{Rx}[i] \ \forall \ i = 0 \ldots k-1$. In other words, there are no transient messages. The *LBTS* is the calculated as the smallest NER_j, plus the lookahead value. Proceed to step 13.
    (b) At least one system has *Transient Messages* or *lost* messages. Note that at this point in the algorithm it is not possible to distinguish between the two possibilities. We simply know that there are some messages which have been sent but have not been received. For each S_j that has missing messages ($\text{Tx}[j] \neq \text{Rx}[j]$), remove j from set R, subtract out the transmit counts from step 9c above, set a local flag indicating round 2 is in progress, and restart the computation (step 11).
    (c) At least one system has lost messages. After the second round of replies have been received, any unaccounted for messages are lost and can be ignored. This is because all systems have replied, and no system sends new messages after a reply. With the *UIOD* model, all messages sent before the reply have also been received or lost. The *LBTS* is the calculated as the smallest NER_j, plus the lookahead value. Proceed to step 13.

11. If some systems have not replied, after a suitable timeout period the *Master* will will broadcast a *Restart LBTS* (*RstLBTS*) message containing:
    (a) The *epoch* for this LBTS computation.
    (b) A copy of set R indicating which systems need not reply (those systems *not* in set R should reply).
12. Upon receipt of the *RstLBTS* message, any system S_j will check if j is in set R. If j is in set R, S_j just ignores the *RstLBTS*. If not, S_j sends a new *RLBTS* message as in step 8. Return to step 9.
13. When a valid *LBTS* has been calculated, the *Master* will broadcast a *Done LBTS* (*DLBTS*) message, with the following information:
    - The value of the computed *LBTS*.
    - The epoch for this *LBTS*.
    - An indication of the *SID* of the *Master* for the next iteration. The next *Master* will be chosen at random among those systems that reported they were not yet ready for the current *LBTS* computation. The rationale for this is that the *slowest* system will be the best choice to decide when the next *LBTS* should start.
14. Upon receipt of the *DLBTS* message, all systems will note the calculated *LBTS* and resume processing events. Should any system not receive the *DLBTS* message, it will be stuck until the next *SLBTS* is processed. (Note: As an efficiency enhancement, we suggest that the current *LBTS* value be included with all event messages, so any system which missed the *DLBTS* can still determine the *LBTS* value when it receives an event message.)
15. The newly appointed *Master* will respond to the old *Master* with a *Master Accept LBTS* (*MaLBTS*) message, indicating it has received the *DLBTS* message and recognizes it is responsible for the next *SLBTS*. If it is not received in a reasonable time period, the old *Master* will resend the previous *DLBTS* message (unicast directly to the new *Master*) and repeat this step.

3.5 *BCUDP* Correctness Proof

In this section we outline a proof that the above algorithm gives a *consistent cut*. For this discussion, we define a *consistent cut* to be a point in time where we can be certain that no message sent by any system prior to the *consistent cut* will be received by any other system after the *consistent cut*.

Define set R as above, to be the set of all systems from which a reply has been received. The focus of the proof is to show that at any point in time, we are assured to have a *consistent cut* between all systems in R. As more and more systems reply, the cut remains consistent between those replying systems, until the set R contains all systems. Define an *incremental consistent cut* to be a consistent cut among systems presently in set R. Also define S_m to be the *Master* system.

1. The initial set R is empty, which by definition is an *incremental consistent cut*.
2. The *Master* S_m initiates an *LBTS* computation and immediately adds itself to set R. By definition, system S_m has processed any messages sent to itself, so at this point set R contains an *incremental consistent cut* set of exactly one system.
3. The *Master* receives a reply from some system S_j. S_j is added to set R. By assumption 4, any message sent by S_j to any other system will have been received (or lost) prior to S_j sending the reply. In other words, any messages sent by S_j prior to S_j sending the reply is either already received or lost by all other systems. Thus we have an *incremental consistent cut* with the newly added S_j. Note that systems in set R may still be receiving messages from systems *NOT* in set R, perhaps due to those systems having lost the original *SLBTS* broadcast, or not having processed it yet. However, this does not violate our definition of the *incremental consistent cut* as above.
4. If not all replies have been received, return to step 3
5. At this point, set R contains all systems, and the *incremental consistent cut* is now a *complete consistent cut* for all systems. This implies only that all systems in set R (all participants at this point) are now guaranteed to have received all messages that are going to be received. This does *not* guarantee that the reported simulation time values were correct when reported, only that they are correct *now*.
6. Since some systems may have have reported incorrect values for the local minima (due to *Transient Messages*), the round 2 processing of the algorithm collects new minima from those systems which experienced *Transient Messages* during round 1 (as determined by the reported message counts). The round 2 processing simply allows all systems a chance to report revised minima as determined by the *consistent cut* of set R at the end of round 1.

4 The BCTCP Algorithm

In this section, we describe an algorithm for computing a *lower bound on time-stamp* by utilizing the *Broadcast* capability of the underlying network, and assuming *RIOD* model for event messages, and applying the observations made previously about message ordering using the *sockets* network programming abstraction. We call this algorithm the *Broadcast TCP* (*BCTCP*) algorithm.

4.1 Assumptions

The algorithm operates under the same assumptions as the *BCUDP* algorithm (given in section 3.1), excepting assumptions 3 and 4, which are revised below:

3. Event messages between systems are sent via some reliable protocol (eg. TCP), and all event messages will be delivered to the intended receiver.

4. The underlying communications medium delivers messages with the *RIOD* model when using a single socket, and the *RNOD* model when using multiple sockets.

4.2 Overview

The *BCTCP* algorithm works identically to the *BCUDP* algorithm, except in the handling of *Transient Messages*. With *BCUDP* we were able to assume that unaccounted for messages at the end of round 1 were lost and could be ignored, but with the *RIOD* model used for *BCTCP* we must assume that the messages will eventually be delivered. When processing a round 2 *RstLBTS* message, all peers must wait until all expected messages have been received before replying. An outline of a correctness proof is given in [8], but omitted here for brevity.

4.3 The *BCTCP* Algorithm

More formally, the *BCTCP* algorithm works the same as *BCUDP* excepting the handling of step 12, which is replaced below:

12. Upon receipt of the *RstLBTS* message, each system S_j will check if j is in set R. If j is in set R, S_j just ignores the *RstLBTS*. If not, and if the round 2 flag in the *RstLBTS* is *NOT* set, S_j will respond immediately with a *RLBTS* message, as in step 8. If the round 2 flag is set, S_j must wait for all transient messages to be received. The *RstLBTS* message contains an array (TotTx[k]) which notifies each system of the total number of messages that should be received before proceeding. Once all messages have been received, send the *RLBTS* message as above. Return to step 9.

5 Experiments and Results

In this section, we describe the environment we used to implement and test the algorithms described in the previous sections. The hardware used is a collection of 48 Intel Pentium-II 300Mhz systems each with 512Mb main memory and two processors. The systems are connected with a 100BT *Ethernet* network (100 Mbps). Each system runs the Intel Solaris operating system, version 5.5.1.

The software used for testing was a simple airport simulation. The program models 480 airports, and 48,000 airplanes. Airplanes travel between airports with a travel time uniformly distributed between 30 and 360 minutes. When arriving at an airport, they remain on the ground for a time interval uniformly distributed between 10 and 60 minutes. The simulation was run on a varying number of systems between 1 and 48. As more systems were used, the overall size of the simulation remained the same (when running on 2 systems, each system managed 240 airports and 24,000 planes initially). This test simulation is very fine grained, with little CPU processing per event, and thus the *LBTS* computation time and the message transmission times will dominate the overall performance.

For a baseline, the airport simulation was first implemented and tested using the *RTIKIT* software developed at Georgia Tech. The *RTIKIT* uses *TCP* sockets (the*RIOD* model) for all communication, and a classical butterfly barrier[9] for *LBTS* calculations. The RTIKIT implementation is described in [10]. We refer to the *RTIKIT* version as *Base1*.

Then both of our new algorithms were implemented and tested for overall performance using the same airport simulation. Three performance metrics were used:

1. Elapsed time. The overall running time of the system, in wall-clock seconds.

2. Time spent per *LBTS* calculation. The total time spent calculating new *LBTS* values was tracked, and divided by the number of *LBTS* calculations made.

3. Reliability metrics, specifically the number of lost event messages and the number of *LBTS* computations that were successful on the first try.

All tests were performed on dedicated systems, with no other user jobs running (normal operating system daemons were present and running). No network traffic was present other than that generated by our tests.

5.1 Results

The results of these experiments are shown in the graphs above. All graphs in this section have the number of systems used for this simulation on the X-Axis, with the various performance metrics on the Y-Axis, and separate curves for each of the implementations.

Figure 1 shows the overall elapsed time of the simulation runs, for systems counts of 1, 2, 4, 6, 8, 12, 16, 24, 32, and 48. (Note. The metrics for the one system case is identical in all cases, since no network activity is needed and no *LBTS* computations are used, and thus the *LBTS* algorithm and the network event passing mechanisms are not used). The results show substantial reduction in overall running time, with algorithm *BCUDP* being the best.

Another way to measure the efficiency of the *LBTS* computation algorithm is to measure the average time spent calculating the *LBTS*. Figure 2

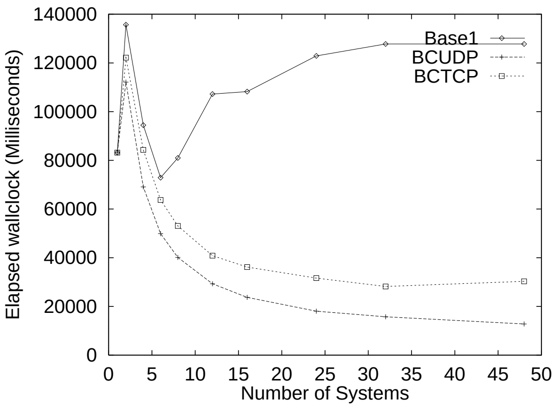

Figure 1. Elapsed Running Time

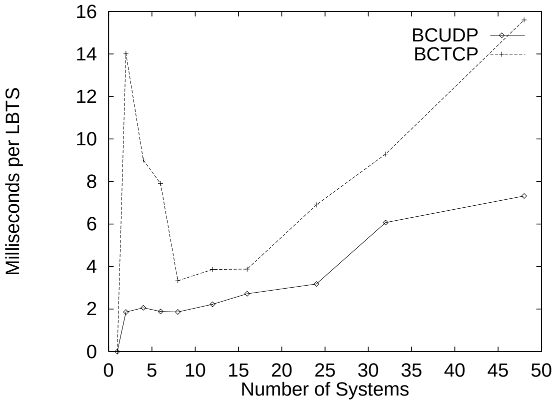

Figure 2. Milliseconds per LBTS

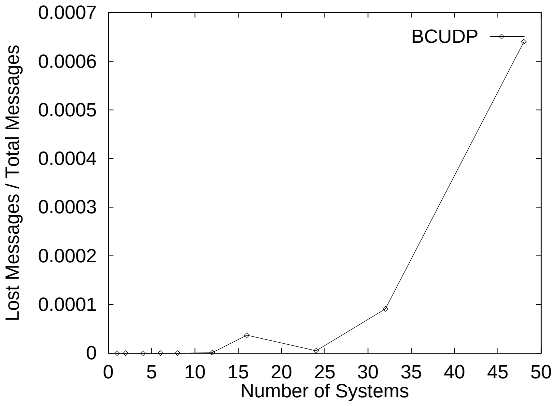

Figure 3. Fraction of Lost Event Messages

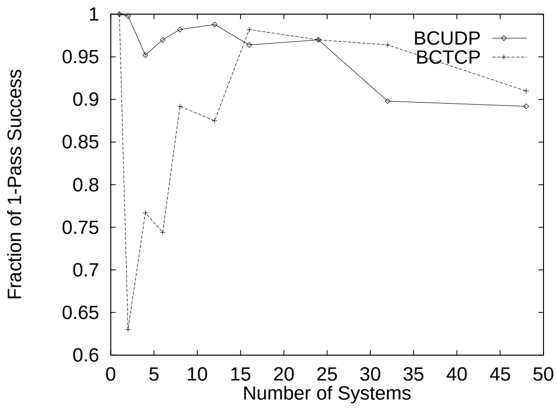

Figure 4. Fraction of Successful One-Pass

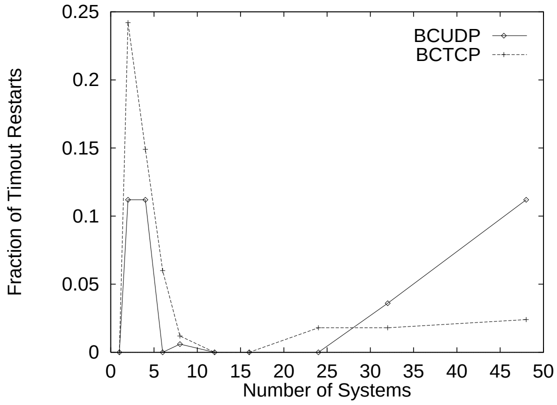

Figure 5. Fraction of Timeout Restarts

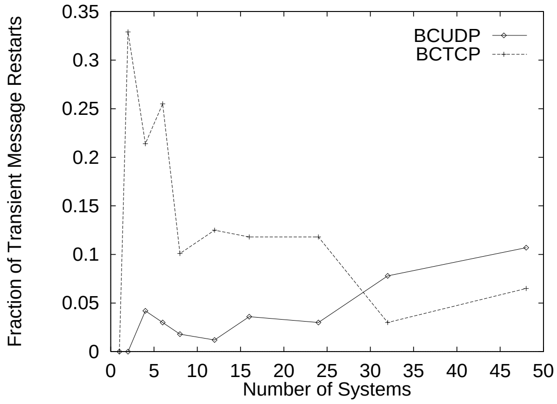

Figure 6. Fraction of Transient Restarts

shows the amount of wall-clock time spent on average performing each *LBTS* computation for algorithms *BCUDP* and *BCTCP*. (Note. The *LBTS* computation times were not available for the baseline runs). Intuitively we expect *BCUDP* to be slightly better (less time spent) since the *BCUDP* algorithm never waits for transient messages in round 2, where the *BCTCP* algorithm does. Figure 2 shows that the times spent in *LBTS* computations is fairly high for the *BCTCP* algorithm with small numbers of processors, we suspect due to the timeout value being too low in this case (note the large number of restarts due to timeouts in figure 5). For processor counts between 8 and 32 the time spent processing *LBTS* computations is roughly comparable between our two algorithms, with *BCUDP* slightly better as expected.

We also measured the amount of *unreliability* exhibited by the *UIOD* network models assumed by our algorithms. Specifically, we measured the number of lost event messages when using *BCUDP* and the number of successful *one pass LBTS* computations. A successful *one pass LBTS* computation indicates that the *SLBTS* broadcast message was received properly by all peers, the *RLBTS* was received properly by the *Master* from all peers, there were no *Transient Messages*, and the *DLBTS* message was received successfully by all peers. Figure 3 shows the fraction of event messages which were lost when using the *BCUDP* algorithm (for our simulation there were over 1 million event messages transmitted). We note that less than 0.0007 of all event messages were lost, even when 48 systems were used. Figure 4 shows the fraction of all *LBTS* computations that were successful in one pass. A further breakdown of the reasons for one pass failures is shown in figures 5 (restarted due to lost *SLBTS* or lost *RLBTS* message) and 6 (restarted due to *Transient Messages*). For the most part, we see that most of the multiple pass *LBTS* computations are due to transient messages, excepting the *BCTCP* algorithm when using a small number of processors (less than 8). Again we believe the large number of timeout restarts are due to the timeout value being too small in this case.

6 Conclusions and Future Directions

The performance results show conclusively that careful attention to and exploitation of the underlying network model can give a substantial improvement in overall performance. Simply choosing the *RIOD* model of network behavior and counting on reliable, in-order delivery of all communications is not always the best choice. Distributed algorithms should be designed with the simplest and most efficient network model in mind, and optimized to work well in the *expected* case. Both of our algorithms, *BCUDP* and *BCTCP* are optimized to perform well in the case where all messages arrive correctly, even when using the *UIOD* network model. When unusual events occur, such as lost or delayed network messages, the algorithms expend extra time handling these events and may not perform as well. However, if the *expected* case occurs often enough, and the unusual events occur infrequently enough, the algorithms perform well.

For future research, we intend to examine how to combine the *LBTS* algorithm and the message retransmission request mechanism, to give way to an apparent *RIOD* model when in fact using the *UIOD* model provided by *UDP* sockets. Since the *Master* system is able to collect a complete record of message counts to and from every pair of systems, the *Master* can determine which systems have lost messages and need retransmissions. Again, the expected case will be where no messages are lost, resulting in little additional overhead most of the time.

References

[1] K. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," in *Communications of the ACM*, vol. 24, November 1981.

[2] R. E. Bryant, "Simulation of packet communications architecture computer systems," in *MIT-LCS-TR-188*, 1977.

[3] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," in *Journal of Parallel and Distributed Computing*, 1993.

[4] K. M. Chandy and R. Sherman, "The conditional event approach to distributed simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989.

[5] B. D. Lubachevsky, "Efficient distributed event-driven simulations of multiple-loop networks," *Communications of the Association for Computing Machinery*, vol. 32, no. 1, pp. 111–123, 1989.

[6] J. Steinmann, "Speedes: Synchronous parallel enviornment for emulation and discrete event simulation," *Advances in Parallel and Distributed Simulation, SCS Simulation Series*, vol. 23, pp. 95–103, 1991.

[7] D. M. Nicol, "The cost of conservative synchronization in parallel discrete event simulations," *Journal of the Association for Computing Machinery*, vol. 40, no. 2, pp. 304–333, 1993.

[8] G. F. Riley, R. M. Fujimoto, and M. A. Ammar, "Network aware time management and event distribution," Feb 2000. Technical Report GIT-CC-00-11.

[9] D. E. Brooks, "The butterfly barrier," *The International Journal of Parallel Programming*, vol. 14, pp. 295–307, 1986.

[10] R. M. Fujimoto and P. Hoare, "HLA RTI performance in high speed lan environments," in *Proceedings of the Fall Simulation Interoperability Workshop*, 1998.