Deadlock Management Protocol for Distributed Discrete Event Simulation

Maxim Jovanovich
CSCI 232, Computer Science Department, George Washington University
mdjovano@gwmail.gwu.edu

Abstract

A protocol for managing deadlock in Distributed Discrete Event Simulation (DDES, which is also known by the name Parallel Discrete Event Simulation, or PDES) is described. DDES may be one of the keys to simulating complex living organisms, among many other interesting simulation goals. The new protocol is named the Deadlock Management Protocol (DMP) and is developed under the approach of minimizing system lifetime cost primarily through software engineering principles. The main idea is to take the simplest possible approach, but also to ensure that the solution is robust. As such, only DDES is considered; solving deadlock in a manner that would also benefit other systems such as general-purpose Internet computing or database management systems would only add complexity to the solution. Previous methods of maintaining causal relationships in DDES are researched to find the strengths and weaknesses of each in an attempt to take the best strengths without the weaknesses. A framework approach and layered architecture is suggested to prevent re-work once an implementation is developed. Within the framework, any approach to maintaining causality could be used. Deadlock management is suggested as the method of choice given current trends in hardware capabilities.

1. Introduction

Consider advances in biology research made in the last decade that have been made possible by computing. One widely publicized example is the Human Genome project, which has enabled researchers to determine the locations of genes responsible for a variety of diseases, down to a precise location on a specific chromosome, and even to know the exact DNA sequence of these genes [1]. Now consider lesser known examples, such as the simulation of the biochemical reaction pathways in a single yeast cell [2,3,5]. One might ask, "why should I care about the simulation of a yeast cell?" This question would arise naturally, because the typical use of yeast is for baking. However, in terms of cell simulation, it is a proving ground for much loftier goals. The really interesting goal is the simulation of human cells [4,6]. Being able to simulate human cells would mean that many potential drugs [5] (maybe someday even a cure for cancer) could be tested in simulation before ever going to trials in living specimens, saving time, money, and the lives of animals. At this point, it's just a matter of time and computing resources, for the correct models to be developed. The ultimate goal would of course be to simulate the biochemical system that composes an entire living organism.

Discrete Event Simulation (DES) is an important tool in modeling complex systems such as living cells. DES is usually done by maintaining in a priority queue a set of events that are to occur. Events are pulled off the queue in chronological order. The simulation as a whole works by using causality of events. Each event is caused by a preceding event. Some calculation is

performed to see when the caused event(s) will take place. New events are placed into the queue for eventual processing. When these events are pulled off the queue, they might cause additional events, thus perpetuating the simulation [7].

For the introductory example, there are ways to simulate biological systems (for example, metabolic cycles) completely as systems of differential equations of chemical reaction rates [5]. However, when protein synthesis, mRNA, tRNA, and DNA interaction are considered in the system, chemical reaction rates are a poor way to do the simulation. The latter interactions are more event-based than anything else, so DES is the method of choice [23]. When the problems get large enough in size and fine in granularity, Distributed Discrete Event Simulation (DDES) is the next logical step, because it enables the simulation to run in a reasonable amount of time on multiple computers, through the speedup associated with parallel processing [21,24]. When the simulation is distributed, there are multiple event queues that need to synchronize with each other, and message passing is used to place events in remote queues. Because of the synchronization requirement, there is potential (and likelihood) of deadlock [7,9]. The simulation on each computer cannot be allowed to move forward until it is certain it will receive no more messages for events that could be earlier than the event being processed. The reason for this is that there could be some causal link that is missed if events are processed out of order. This need for synchronization is the underlying cause of deadlock in these systems. There are two major problems with transitioning from DES to DDES. First is actually detecting deadlock, when (or if) it occurs. The second problem is finding the way to resume the simulation if it is stuck in deadlock.

There are a few viable solutions to the deadlock problems [8,9,10,11,12,21]. One is to do advanced analysis and design to ensure that deadlock is impossible. (This is extremely difficult in practice.) A second solution is to prevent deadlock by being optimistic in the synchronization, assuming that there will never be out-of-order events. Since it is still possible for out-of-order events to occur, this solution must include a method of rolling the simulation back to where the violation occurred, so that the events can be replayed in the correct order [11]. (Can be costly in time and memory depending on the rollback probability.) Another method is to allow deadlock, but to find occurrences of deadlock with a distributed algorithm [8,9,10], and then resume the simulation by kick-starting it with the single earliest possible message in the queues. (This is conceptually very simple.) A more complicated method is to do the same, but kick-start the simulation with a maximal set of the earliest possible messages [9]. (This is conceptually very difficult.) The last solution is to insert messages for events very far in the future into the system that force the simulation to continue. (This is primarily applicable to terminal deadlock, which is when the simulation reaches the end of events to be processed, although it can apply to other situations where a limit can be place on the earliest time of the next message that will be sent.) This last solution is typically referred to as the passing of null messages.

2. Problem statement

Simply put: What is the best overall DDES solution method?

To answer this, some other questions must be asked. First, how does one objectively determine what the ideal method solution should be? Next, what metrics should be used to measure against this ideal? My argument is that the ideal would be the optimistic solution described above, if the possibility of rollback was zero. However, this is not achievable in practice, so the various solutions must be weighed against each other to see which is best in a given situation, possibly compared to the ideal. I propose using system lifetime cost as the performance metric.

System lifetime cost. This should include the cost of time, hardware costs, and software costs. The time it takes to run a given solution is obviously important. The whole idea of going to a distributed simulation, besides being able to conquer larger-sized problems, is speedup [21], and both really go hand-in-hand. A

small problem that takes too long would be cost prohibitive, and so would a problem that is fast when small, but takes too long when size of the problem is increased. Hardware maintenance can be a big issue over the lifetime of a product. Volatile memory is cheap and fairly reliable these days, so it is not much of an issue, but non-volatile storage requirements can impact lifetime costs greatly. For example, the reliability of a system worsens, in terms of Mean Time Between Failure (MTBF), dramatically as more non-redundant storage systems are introduced. Depending upon the system complexity, failures in non-volatile storage can lead to lengthy down time, which leads to costly project delays in addition to adding maintenance cost.

Software maintenance is probably one of the most overlooked items in these considerations [13]. Often the most complex solution to a problem is chosen because it has been proven to be the fastest in actual execution time. However, if it take months to program and debug, it might be a waste compared to a solution that is inefficient in execution but only takes a day to program. In addition, code maintenance is often overlooked. Most software code is used much longer than it was ever intended or expected. It is usually inadequate over time, requiring periodic upgrades. The difficulty of distributed programming in general and salary cost due to DDES being a niche programming area mean that developing and maintaining DDES projects is very costly.

3. Related research work

The typical approach to creating a DDES starts with the normal way of creating any DES. First, the system to be simulated is analyzed and decomposed into a set of physical processes, the action of which can produce the predicted (or perhaps observed) system behavior. Each physical process is represented by a logical process (LP) in a message-passing system that can be executed as a combination of a sequence of events and sequence of messages passed between LPs on a computer [7]. In the DES, all of the LPs are run with a single event queue, and messages are treated as a sub-type of events. The single event queue handles synchronization and causality automatically, because events are always processed from the queue in order.

To make a DDES from a given DES, the set of LPs for the system simulation is partitioned into smaller disjoint sets of LPs, the union of which is the original set of LPs for the entire system. In the DDES, some degree of message-passing takes place over the interconnection network. Typically, the choice is made to have a one-to-one relation between LPs and processors. That is, each LP has its own processor, and every processor is used for just one LP. In this case, every message that is not self-directed must travel over the interconnection network. The possible incoming lines of communication from which each LP can receive messages are termed "dependencies" and vary with the decomposition of the system being simulated. To maintain causality in the system, as explained before, each LP must ensure that it will not receive any messages which have a time stamp earlier than the time of the next event to process. This is done by tracking the times of messages for each dependency. If all of the dependency times are greater than or equal to the lowest event time in an LP's queue (the event at the top of the queue), then the next event is safe to process. This dependency of event processing on incoming messages is one area for potential deadlock. The other is that an LP may be trying to transmit a message to another LP but, for some reason or another, the destination LP is not able to receive any messages due to finite buffer size limitations [9].

The earliest work on deadlock in distributed systems is by K. M. Chandy and J. Misra [8]. Chandy and Misra worked out the basic conservative solutions to any deadlock problem that can be encountered, including terminal deadlock, building on the work of Dijkstra and Scholten [18]. Their solutions, as presented by them, have a couple of flaws, which will be highlighted later.

It is worth noting that they studied these problems at a time where processors and memory were fairly limited but interconnection networks were fast by comparison. In this situation, the minimum amounts of processing and storage were strict requirements of

whatever solution was chosen. As such, the optimistic approach was rejected because it requires check-pointing (thus requiring large amounts of storage of some kind) and will inevitably have to implement a rollback mechanism, which takes processing time. Also, Chandy and Misra were approaching the problems from the typical distributed applications of the time, which were databases. In actuality, this is still the predominant distributed application today, especially given that every major software application today is effectively a database application. Optimistic methods don't even make sense in database applications.

More recently, the processing, memory, and interconnection rate situation was reversed from when Chandy and Misra started their work on deadlock. The interconnection network speeds of today (in commodity networks) are well behind what processors are capable of in terms of bandwidth. Memory also lags behind processors in these terms, but still outdoes the typical computer-to-computer interconnection. So, the accepted solution today is different from what Chandy and Misra chose, and is somewhat based on technology from a few years back. The generally accepted solution today is to use the optimistic method, implement check-pointing, and implement rollback mechanisms.

This solution has many attractive points [11]. First, through check-pointing, the simulation state can be saved and restored. This can be useful not only for rollback mechanisms, but for recovery from system failure. Also, if rollback can be implemented in each processor individually, without requiring interconnection utilization, it can be very fast. Unfortunately, rollback usually requires use of message -passing between processors because some messages may have been sent out speculatively, requiring some sort of communicated "undo" feature (sending a message to another process indicating "some messages you received were sent out by accident") when it is determined that a synchronization violation has occurred [12].

The details of the optimistic methods of DDES are left to the reader. There are many references to date, and the optimistic methods seem to be the standard way of doing any type of DDES in recent years [11,12,19]. The references included in this paper all indicate the same drawbacks of a large memory footprint and time-consuming, complicated, garbage collection schemes. As noted earlier, the optimistic methods have their merits as well, so they are not to be ignored. The answer is perhaps not in the pessimistic method or in the optimistic method, but somewhere in between.

On the horizon, faster and faster interconnection networks are on the way, possessing bandwidth (10 and 100 G-bits per second) greater than a single processor (typically 2 to 6 GB/s) can utilize. It appears that the times are returning to the situation for which Chandy and Misra had originally devised their solutions. Given this scenario and because of rollback mechanism complexity, the focus of this paper is where the least attention seems to be devoted: the conservative solution. What follows is an explanation of the best-known conservative approaches to deadlock management.

4. Deadlock detection and recovery

Chandy and Misra [8] realized that the first step is to devise a distributed algorithm for detecting deadlock. As an indicator of the importance of this first step, many papers have been written on the subject of deadlock detection alone. To state the obvious, nothing can be done to recover from deadlock unless it is known that deadlock exists, or many recovery attempts would be wasteful in the least. A distributed algorithm for this is key, because having a centralized algorithm gives a single point of failure, and as stated earlier, as the amount of hardware increases, the system MTBF decreases, making component failure likely in a distributed system. The next step is an algorithm to recover from deadlock, which again is a distributed algorithm.

4.1 Deadlock detection

The basic system of Chandy and Misra for detecting deadlock requires some description to understand the

development of my solution. Rather than require the reader to find their paper and review it side-by-side with this one, a full description of everything necessary to understand their analyses is included here. Their method is devised where the distributed processing system is viewed as the set of vertices and colored edges of a directed graph [8]. The vertices represent processes, identified by a process index number. The directed edges represent wait-for relationships.

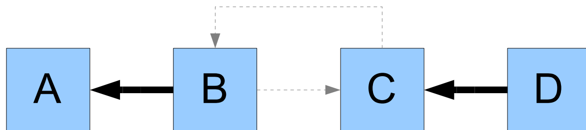

Illustration 1: Wait-for graph

The dashed lines show possible locations for edges where edges currently do not exist. The edges (identified by two values, the first being the waiting process, and the second being the waited-for process) go from the waiting process to the waited-for process. For Chandy and Misra, these wait-for relationships were explicit request/reply channels, where one process requests data from another process, and then waits for a reply with the requested data. In our DDES, the wait-for relationships are implicit: they are the inter-process dependencies for which the time-stamp is less than the time of the top event in the queue. The graph colors of Chandy and Misra are as follows:

- Grey: if process i has sent a request to process j which process j has not received (yet).
- Black: if process j has received a request from process i and has not sent the corresponding reply to process i.
- White: if process j has sent a reply to process i which process i has not received (yet).

Their color scheme is valid for distributed databases, but for generalized systems, it doesn't account for cases like our DDES, which can have implicit (unrequested) responses for which a process must wait. In particular, the black coloring must be extended to cover this case. Also, for our DDES, there is no need for using the grey coloring because there will not be any explicit requests for messages.

There are also sets of axioms for operations on the graph, and operations on processes. These are the graph axioms used by Chandy and Misra:

- G1 (Creation): A grey edge (i,j) may be created if edge (i,j) does not exist.
- G2 (Blackening): A grey edge will turn black after an arbitrary, finite time.
- G3 (Whitening): A black edge (i,j) may turn white only if process j has no outgoing edges. (Only active processes may reply).
- G4 (Deletion): A white edge will disappear after an arbitrary, finite time.

The set of graph axioms (G1 through G4) all apply to our DDES. For a complete generalization, this set should include in G2 that a black edge can be created black in addition to being recolored from an originally grey edge.

As stated before, all messages in the DDES are implicit. As such, there needs to be a determination of when a black edge should be created. Here, a process is truly waiting for another process when any one of the input dependencies has a last-message time that's less than the time of the next event to process. In fact, each input dependency with a last-message time less than the time of the event at the top of the queue corresponds to the existence of a black outgoing edge.

After they describe the graph axioms, Chandy and Misra introduce the concept of "probes" and describe the basic solution to the deadlock detection problem. The detection problem amounts to detecting dark cycles in the wait-for graph, where a dark cycle is defined as "a cycle in which all edges are grey or black (some may be grey and others black)." They also state that a dark cycle "will persist forever because, it follows from the graph axioms that edges in a dark cycle cannot be whitened or deleted."

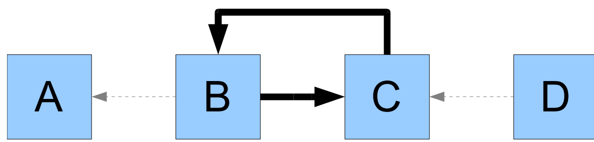

Illustration 2: Dark cycle

This is their probe description: "The algorithm by which [process] i determines if it is part of a dark cycle is called a probe computation. In probe computations vertices send messages, called probes, to one another; probes are concerned with deadlock detection exclusively and are distinct from requests and replies." This project uses the same form of probe computations. One important point is that the following assumptions are made: it is assumed that messages sent between processes arrive in finite time and in the order sent, indicating the requirement of some sort of reliable data transport protocol like TCP.

Next are the process axioms used by Chandy and Misra:

- P1: If a probe is sent by process i to process j when edge (i,j) is grey, edge (i,j) will turn black sometime after this probe is sent and before it is received. If a probe from process i is received by process j when edge (i,j) is black then edge (i,j) existed and was dark (grey or black) at all times from the instant at which the probe was sent, to the instant the probe was received.
- P2: If a probe is sent by process j to process i when (i,j) is white, then (i,j) will disappear sometime after this probe is sent and before it is received.
- P3: A process i can determine (locally) if there is an outgoing edge (i,j) to any process j, though it cannot determine its color (locally). A process j can determine (locally) if there is an incoming black edge (i,j) from any process i.
- P4: Every probe will be received in some arbitrary finite time after it is sent.

The process axioms (P1 through P4) all apply to a fully generalized system and to our DDES. Axiom P3 may seem strange in that a process can't tell the color of an outgoing edge, even though it was responsible for the creation of the edge. Given that there is an outstanding request from process i to process j, process i can only know if the edge is not grey. The unknown is, once the edge has turned black, whether the edge has turned white. Because the color is dependent upon whether process j has sent a reply (which may not have reached the process i yet), process i doesn't know the color until after the reply actually reaches process i.

Finally is the deadlock-detection algorithm. According to Chandy and Misra, "To determine whether it is on a dark cycle, a vertex i initiates a computation called a probe computation. Several vertices may initiate probe computations and the same vertex may initiate several probe computations. To distinguish each probe computation, the messages and variables used in the n-th computation initiated by vertex i are tagged (i,n)."

To help explain the algorithm, they introduce the concept of "meaningfulness" with: "A vertex j will send at most one probe to any vertex k in one probe computation. The probe is said to be meaningful if and only if edge (j,k) exists and is black at the time that vertex k receives the probe. From P3, vertex k can determine if a probe is meaningful." The summary of their algorithm is that the initiator of a probe computation sends probes along all outgoing edges, and upon receiving the first meaningful probe declares that it is on a dark cycle, and therefore that deadlock exists. For non-initiator processes, their job is to send probes along all outgoing edges upon receiving the first meaningful probe.

The Chandy/Misra deadlock-detection algorithm is insufficient for determining deadlock. Granted, if a dark cycle is detected, it is guaranteed that deadlock exists. However, there are cases where deadlock can exist but not every probe computation will not detect it,

depending upon the location of the initiator or the nature of the deadlock.

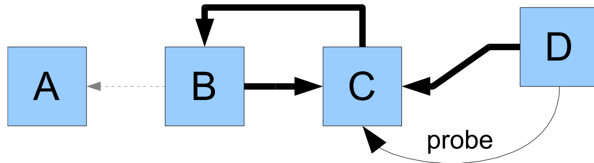

Illustration 3: No return path to initiator

One configuration (Illustration 3) where their algorithm doesn't detect deadlock is where the initiator is dependent on a node that is in a dark cycle. Here, the initiator will send a probe into the cycle, but the probe will never get back to the initiator. Deadlock will only be declared in this case if one of the nodes in the cycle becomes an initiator.

The other configuration (Illustration 4) is where there is no cycle at all, but instead there is a dependency (node A) that will never produce any messages because it has terminated. Note that this is not always an indicator that the entire simulation should be terminated. On the contrary, the rest of the simulation may still be alive and well. It may just need to sever or bypass the dependency link(s) to the terminated node. One real-world example of this would be when a grocery checker in the supermarket goes home for the day. Just because the particular check stand the checker was using is no longer ringing up groceries doesn't mean that the store is (or should be) closed.

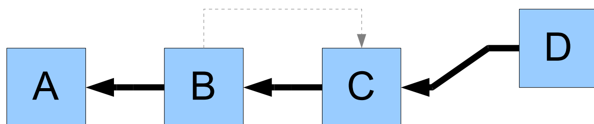

Illustration 4: Linear wait-for deadlock

4.2 Deadlock recovery

In "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Chandy and Misra describe a complicated computation mechanism to determine the optimal way to resume computations from a deadlocked distributed discrete event simulation. Sufficiently, an alternative simple method of resumption is to find the queue with the lowest time tag for its top-most event, and to find the process that has this queue [9]. Because of the assumption that messages arrive in order and because of the nature of the event queues, all events must be processed in order, it follows that there can never be a message sent that will have a time lower that the global minimum top-most event time. Therefore, it is safe for the process that has this queue to resume by processing the queue's top-most event. It can do this by setting all of its dependencies' last-message times, if lower than the top-of-queue time, to the top-of-queue time, thus allowing the next event to execute.

Now we have seen the basic methods of discrete event simulation. In particular, the mechanisms for detecting and recovering from deadlock have been explained. The next step is to find an overall optimal solution that uses the best parts of all of the available methods, but is as simple as possible to minimize system lifetime cost.

5. Solutions and analysis

To help describe the potential solutions and analyze the results, the reader will benefit from a common example. The car wash simulation (Illustration 5) is one of the simplest models that has all of the complications that a DDES will encounter.

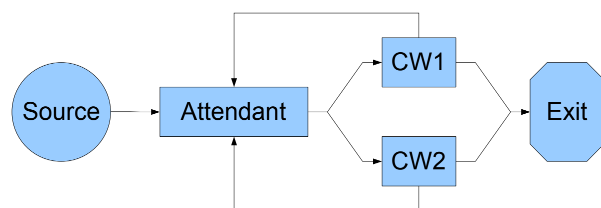

Illustration 5: Car Wash Simulation

A variation of this was first described by Birtwistle and later modified to its current form by Misra [7] to illustrate DDES in a paper where Misra develops an early form of deadlock management. Arrows in the illustration show the message passing scheme. The description of the car was system is: "A car wash system consists of an attendant and two car washes, abbreviated CW1 and CW2. Cars [generated by the source] arrive at random times at the attendant. The attendant directs cars to CW1 or CW2 according to the following rule: If both car washes are busy, that is, washing cars, any arriving car is queued at the attendant; if exactly one car wash is idle, the car at the head of the queue, if any, is sent to that idle car wash; if both car washes are idle, the car at the head of the queue, if any, is sent to CW1." Once cars are finished at the washing station, they proceed to the exit node (which could log a "Car Complete!" message) and disappear from the simulation.

Two of the cases not handled by Chandy and Misra's deadlock detection algorithm can be illustrated in the car wash example. The first is where there is a dark cycle and the initiator is outside of the cycle. As an example, the exit could be waiting for a message from CW1 to process its next event. CW1 could be waiting on the attendant to send another car. The attendant could be waiting on the the source, or another message from CW2 in order to process the "station CW1 is free" message on the queue, and thus the simulation is in deadlock. If the exit is the initiator of a probe computation, it will never receive a message back with the same probe computation, because it is not a dependency of any of the other nodes.

The second case could be that CW1 is waiting on the attendant for a car, and the attendant is waiting on the source for a car. However, the source could be stopped from producing any more cars as a way to end the simulation. But cars that are still in the simulation should continue. Because the attendant and CW1 have no knowledge of how the simulation is being shut down, the only thing they see is a deadlock condition. However, there is no dark cycle in the wait-for graph in this case, only a dark path.

The optimistic methods are so well-developed that an entire book could be written on the subject, so the reader is directed to other sources for more information. Instead, this paper will focus on making a fully-developed solution using the conservative method, compare this solution to the existing (optimistic) methods, and then pick the best pieces of each to recommend the final solution. The overall approach is to create a simulation framework, upon which anyone (with sufficient knowledge of DES in general) can create a simulation of anything (within reason). This way, the implementation details are hidden from the DES "programmer", freeing this person to think about the really important things in the simulation. It is a layered approach in which the simulation framework provides a set of services (those of running event queues, passing messages, and managing deadlock) to a higher layer that is only concerned with event content, message content, and creation of new events and/or messages. It also allows analysis of the solution independent of any specific simulation types [13].

5.1 Deadlock management.

The complete conservative method involves detecting deadlock and then determining the method of recovery. Because it never avoids deadlock, but instead works around it, I call this method deadlock management. Development of this solution will proceed in two phases. First, a protocol is developed for a theoretical system where there are separate simulation and management messages, that are always delivered in-order and in finite time. Afterwards, the protocol is examined to determine how it can be made a reality in a system where processes do not fail, which would be the limit of the capabilities brought by deadlock management alone. (This is explained in Section 5.3.)

The ideal deadlock management protocol is as follows. The DDES uses simulation message-passing as described before, but includes management-only messages that are out-of-band and are always processed by the simulation framework, even if the simulation

itself is deadlocked. In previous DDES descriptions a computing node consisted of a processor and an LP running on that processor. For the deadlock management solution, a computing node consists of a processor that runs both an LP and an LP manager. All simulation and management messages go through the manager. This protocol implements fully distributed algorithms, except in that the simulation requires a single external input to start the simulation once all nodes are up and running. (Note that using the same techniques in this protocol, that startup can be implemented in a fully distributed manner as well. This is left as an exercise for the reader.) There are seven different types of management messages:

- Start Simulation Message: this is the type of message that is sent to each node to trigger the execution of the simulation. It is sent by a controller process once each node is up and running. For example, a controller could be a computer from which an operator runs all of the remote-procedure calls which cause background execution the LP/manager code in each computer in a cluster. The start messages would be sent to each node once all of the remote procedure calls return.
- Probe Computation Message: this is the same as the probe computation message used by Chandy and Misra. It includes the initiating node identifier and the computation number for the initiating node. In addition to those parameters identified explicitly by Candy and Misra, it must also include the last simulation message time for the dependency that determines the destination of the particular Probe Computation message.
- Poll Query Message: this is a message used to implement a distributed algorithm that finds the global minimum top-of-queue event time. This message communicates to each node that the minimum-finding algorithm is in progress.
- Poll Flood Message: this is another message used in the minimum-finding algorithm. The purpose of this message is to ensure that a globally consistent minimum value is found, regardless of in-transit messages [14].
- Poll Reply Message: this message is used to communicate the results of the minimum-finding algorithm to the originator of the query.
- Break-Lock Command Message: this message causes a node to process its next event. Once a determination has been made as to which node LP holds the minimum top-of-queue time, this message is sent to that node to trigger resumption of the simulation.
- Termination Message: this message is sent to terminate the simulation. A specific type of deadlock, terminal deadlock is distinguishable from the deadlock previously described. It is where there is no LP in the system with an event to process. When terminal deadlock is detected, a termination message is sent to each node to indicate that the simulation is complete. There may be simulations which have no end, so this message is useful for the controller to dictate an arbitrary stopping point.

All of the management messages necessary to implement deadlock detection, deadlock recovery, and overall simulation management are given above. Once the simulation is running, the event queue is processed independent of any management messages. The message time of each outgoing message is tracked by destination node. This will be used to determine probe computation meaningfulness. The special value of positive infinity is used as the top-of-queue time if there are no messages in the queue.

As done with Chandy and Misra, the manager implements a watchdog timer that will time-out when no events have been processed from the queue recently, for example due to unsatisfied dependency times. The timeout action is to initiate a probe computation. Nodes also implement a state machine that changes in

response to the different management messages. The three possible states are:

- Not Started: the node has no events to process and is ready to receive messages but will not have a running watchdog timer. This is the initial node state.
- Running: the node may or may not have events to process and will initiate a probe computation when the watchdog timer expires.
- Deadlocked: the node has determined through the distributed deadlock detection algorithm that deadlock has occurred and therefore it no longer needs to operate the watchdog timer because deadlock is a (locally) stable property [15]. The node is ready to receive messages, but will not resume running until a message is received along the right input channel or until commanded to break lock. It is necessary to distinguish between deadlock existing in the simulation and the node being in the deadlocked state. Note that the node not being in this state doesn't mean that deadlock doesn't exist. Being in this state doesn't even mean that there is currently deadlock in the simulation. But if the node is in the deadlocked state, then deadlock definitely existed at one time and may persist into the future if further actions are not taken.

Many of the node behaviors are dependent more upon the message type than the node state, so the overall protocol is described in the manner of Chandy and Misra, by describing what a node does in response to receiving each different management message type or to watchdog timer expiration.

Upon receiving a simulation message, the node adds this message to the event queue and updates the last message time for the dependency from which the message came. It also restarts the watchdog timer and transitions to the running state. The node then processes any events on the queue that are safe to process.

Upon receiving a Start Simulation message, for at least one of the nodes in the cluster, the manager posts an initial (kick-off) message into its simulation event queue and transitions to the running state. The exact implementation is irrelevant, but it would likely be through a callback function.

Upon receiving a Probe Computation message, the manager stops the watchdog timer. The manager rejects the message if already in the deadlocked state or if the Probe Computation message time is less than the time of last message received along the given input channel. Next, if the message computation number matches the computation number last recorded for the initiator, then the manager declares deadlock (the node transitions to the deadlocked state). This handles the case of the first of the car wash examples. (See Illustrations 3 and 5.) Otherwise, if the message computation number is greater than the computation number last recorded for the initiator, then the probe computation is meaningful, so the manager saves the new computation number and continues the probe computation process.

The preceding account is the old algorithm described by Chandy and Misra [8] with the exception that the node doesn't need to be the initiator to declare deadlock. Another difference in the new algorithm is that at this point, once the computation has been determined to be meaningful, if there are no events on the queue and there are no dependencies for this node, the manager immediately declares deadlock. This is because this node is a dependency for some other node (the node that sent it the Probe Computation message) and because it will never generate another outgoing simulation message. This handles the case of the second car wash example which the previous algorithm would not have handled. (See Illustrations 4 and 5.)

If the probe computation is meaningful and deadlock hasn't been declared at this point, the manager forwards the Probe Computation message to all of the dependencies for which the last simulation message time is less than the top event time. As mentioned before for the Probe Computation message type, the last simulation message time for a dependency is included in the Probe Computation message to that

dependency. Note that the only way for a node to reach the deadlocked state is through the receipt of a Probe Computation message.

If not previously in the deadlocked state but now deadlock is declared, the manager sends Poll Query messages to all other nodes in the system. The current node is recorded as the initiator, and a new query number is generated. The query number must be greater than the last query number used by this node as the initiator. At this time, this node must create a data structure to record the status of replies to this query. Also, this node's top-of-queue time is used to seed the minimum time value computation, which is stored with the data structure that tracks the replies to the query.

Upon receiving a Poll Query message, the query number is checked against the last number recorded for the given initiator. If the message query number is the greater of the two, this is a new query. The new number is saved. At this time the manager initializes a data structure to record flood messages. This data structure must be specific to the initiator, because there could potentially be several queries initiated by different nodes all at the same time. Also at this time, the manager sends Poll Flood messages (which include the query initiator identifier and the query number) to all nodes besides itself and the initiator. The initiator doesn't need to receive flood messages because it knows its own top-of-queue time, and only needs the minimum-time query responses from the other nodes.

If the message number is greater than or equal to the last number recorded for the initiator, then the flood status structure is checked to see if there are any outstanding flood (or query) messages left to process for this query (as identified by the initiator and query number). If all outstanding messages have been received, the current top-of-queue time is sent in a Poll Reply message back to the initiator.

Upon receiving a Poll Flood message, the manager handles it the same way as the Poll Query message. The only difference between these messages is that the query is sent by the initiator, while the flooding is done by the non-initiator participants of the query. An implementation of this algorithm could merge these two types into a single message type, but they are kept separate here to clarify the steps of the algorithm.

Upon receiving a Poll Reply message, the manager checks the query number for the reply. If it is equal to the number of the most recent query sent by this node, then the manager records the reply and updates the running minimum (if the new top-of-queue time in the reply is lower than the lowest so far), also tracking the node from which this reply came. If all replies are accounted for, the manager checks the running minimum. A minimum value of positive infinity means that no nodes anywhere had events to process. This signals terminal deadlock. The response to terminal deadlock is to immediately send Termination messages to all other nodes. There is no additional state to handle terminal deadlock, and no transition takes place, because the state doesn't matter at this point. If there is a minimum value less than infinity, then the manager sends a Break-Lock Command message to the node which replied with this minimum value and transitions to the running state, but does not yet re-enable the watchdog timer. The destination of the Break-Lock Command could be the same node as the initiator.

Upon receiving a Break-Lock Command message, the manager checks all of the last simulation message times for the input dependencies. Any times lower than the top-of-queue event time are set to the top-of-queue event time. The manager then triggers the simulation to resume, transitioning the node to the Running state. Also, it restarts the watchdog timer, regardless of whether there are any events in the queue to process.

Upon receiving a Termination message, the node frees any allocated data structures used for tracking flood or reply messages, flushes the event queue (if there are any events left), and stops execution of any processes or threads used to implement the simulation or management functions.

Finally, in response to timeout of the watchdog timer, if the node has any dependencies, the manager initiates a probe computation. For the probe computation, it generates a new computation number and sends out Probe Computation messages to any dependencies for which the time of last simulation message is less than the top-of-queue event time.

That is the basic description of the new algorithm as it applies to an ideal communication system where the simulation and management messages always arrive in the order sent and within finite time of being sent. The next step is to describe how this could be implemented over a currently existing (realistic) transport protocol, but under the assumption that the processing elements themselves are not susceptible to corruption.

5.2 Deadlock Management Protocol (DMP)

The obvious approach is to start with existing protocols, see which protocol offers most of the services we need, and then tailor the protocol from there. The transport services required (for both simulation and management message types) by a DDES framework with deadlock management are in-order message delivery, reliable data transport, finite delivery time, and the ability to send simulation messages of variable length. The framework also requires the ability to transport management messages at any time without any buffers getting full or causing blocking of message delivery, while keeping receipt of management and simulation messages in-order [14]. It also requires the prevention of more simulation messages from arriving than it can handle. A final suggestion (not necessarily a hard requirement) is that the service-providing protocol should be available on multiple operating systems for cross-platform simulation capability.

At first glance, it seems that most, if not all, of these requirements are met by the Transmission Control Protocol (TCP) [25]. TCP definitely offers in-order deliver, reliable data transport, finite delivery time, flow control, and even allows for some out-of-band data for management messages by using the (non-standard) urgent data capability. One possible implementation would be to send all message types directly over a single TCP byte stream and to apply the urgent data flag for management messages. This would require the simulation to flush the input buffer in order to access the management messages, thus negating the ability to ensure in-order reliable delivery. Perhaps a separate connection could be used to send the management messages? Unfortunately, this would not work directly because the probe and poll messages need to be synchronized with the simulation messages for the distributed algorithms to work properly [14].

Another possible implementation with TCP would be to send simulation messages directly over TCP and then to use the TCP header options fields as a method of passing management messages. There are a couple of problems with using the protocol this way. First, there are no guarantees that the options (management messages) and data (simulation messages) will be synchronized, because packet transport isn't tied to any particular bytes in the byte stream. In addition, it would not allow for reliable data transport of the management messages. Any dropped packets would mean dropped management messages. A reliable data transport mechanism (with sequence numbers, a separate watchdog timer callback, etc.) would have to be built on top of the options fields.

Obviously, TCP would require extensive modification to meet all of the service needs. There might be other protocols that are more ideally suited to the needs of the DDES framework, but their obscurity makes them choices that would be very difficult to implement in practice. The only choice left is UDP. Obviously, it lacks reliability, in-order delivery, and flow control, so these would have to be programmed in at the framework level. A wrapper around UDP supplying services to the framework layer could essentially mirror all of the TCP functionality by using connection setup, sequence numbers, a send window, a simulation message receive window, a message type field, other fields that would be required by the management messages, and a simulation message data field. However, unlike TCP, it could work on more of a message level instead of as a byte stream. This UDP wrapper would also require access to the same watchdog timer used by the manager to do the deadlock detection processing. The management watchdog timer could serve a dual purpose for these.

5.3 Deadlock management in the real world

Field	Mnemonic	Size	Offset	Bit Offset
MD5 Sum	md5Sum	6 bytes	0	0
Flags	flags	1 byte	6	0
Synchronization Flag	SYNC	1 bit	6	4
Acknowledgment Flag	ACK	1 bit	6	5
Finis Flag	FIN	1 bit	6	6
Destination Flag	DEST	1 bit	6	7
Message Type	msgType	4 bytes	7	0
Sequence Number	seqNum	4 bytes	8	0
Acknowledgment Number	ackNum	4 bytes	12	0
Receive Window	rcwWindow	4 bytes	16	0
Source Node	srcNode	4 bytes	20	0
Initiator Node	initNode	4 bytes	24	0
Computation Number	compNum	4 bytes	28	0
Minimum Time	minTime	8 bytes	32	0
Message Time	msgTime	8 bytes	40	0
Destination Node	destNode	4 bytes	48	0
Message Data	data	4 bytes	52	0

Table 1: DMP Header

From the above discussion, it should be apparent that a deadlock management protocol could be made to work in a situation where the underlying data transport is unreliable, but where processes (such as the DDES framework) never fail. If a simulation needs to run for a long period of time in a cluster environment, it's basically asking for trouble. Given the considerations of MTBF, the simulation is likely to run into some sort of hardware glitch that causes one or more of the processes to malfunction. As alluded to earlier, the limitation of applying only deadlock management as the method of dealing with the difficulties of DDES is that it cannot handle this situation.

The method that can handle malfunctioned processes is the optimistic method, so long as a malfunction-detection algorithm can be applied. As luck would have it, for the optimistic method, using check-pointing prepares for such situations. When a malfunction is detected, the node can be restarted. If the node check-points are saved into some sort of redundant non-volatile memory, the node state can be recovered during restart and the simulation can resume where it left off.

So, now it seems that we're back to the optimistic method and check-points. But, this isn't the entire story. A compromise can be made. With a full deadlock management scheme in place, the check-pointing method doesn't need to be as robust as it would be for a fully-optimistic method. The interval between saving check-points could be extended, since they would no longer be used for rollback recovery. The only requirement for the check-points is that they should represent a globally consistent state. This can be done using the method of distributed snapshots outlined by Chandy and Lamport [14]. When a failure in any one of the processes is detected, the entire simulation can be rolled-back in a distributed manner to the most recent global snapshot. The algorithm used to record a global snapshot uses similar mechanisms to the minimum-time polling algorithm used in the deadlock recovery method, so there is opportunity for combining these algorithms.

5.4 The final solution

My recommended final solution for DMP is to use a TCP-like connection-oriented protocol over UDP with a fixed-size protocol header that includes some built-in security features. It uses sequence numbers, acknowledgment numbers, and receive window size that correspond to units of discrete messages. The receive window is used specifically for preventing event queue overflows due to receipt of simulation-type (event) messages. (See Table 1.)

The DMP header has 13 fields as follows:

1. MD5 Sum: a message digest [26]. It is the least-significant six bytes of the MD5 message digest (the digest[] array indices 10-15) of the concatenation of

the DMP header, any data following the header (for simulation messages), and any shared key that is desired to create a message authentication code from the digest. The key can be any length and can be left out if the simulation is in a local (trusted) cluster that doesn't allow external (Internet) communication. How a key would be determined is left to the user and/or the implementation. The purpose of using a key would be to help prevent tampering with the simulation.

2. Flags: a one-byte bit field with a reduced set of TCP flags and a destination flag. The SYN flag is for connection creation. The ACK flag is to indicate that the acknowledgment field is valid. The FIN flag is used to indicate that it is the final message, for closing a connection, and would likely be used with the termination type of management message. The DEST flag is to indicate that the destination node field is valid.
3. Message Type: uses the following types:
   - 0 - Simulation Message
   - 1 - Probe Computation Message
   - 2 - Termination Message
   - 3 - Poll Query Message
   - 4 - Poll Reply Message
   - 5 - Break-Lock Command Message
   - 6 - Poll Flood Message
   - 7 - Start Simulation Message
   - 255 - Rollback Message
4. Sequence Number: a 4-byte number (in units of discrete messages) used for reliable data transport and in-order delivery of messages.
5. Acknowledgment Number: a 4-byte number (in units of discrete messages) used to indicate the next message expected in the connection.
6. Receive Window: a 4-byte number indicating the amount of space (in units of discrete messages) available in the event queue.
7. Source Node: the 4-byte identifier of the node sending this message.
8. Initiator Node: the 4-byte identifier of the node that is the initiator of a probe computation (for Probe Computation Message types) or of a minimum-time query (for Poll Query, Flood, and Reply Message types) for implementing distributed computations.
9. Computation Number: a 4-byte number used to specify the distributed computation number for the initiator for tracking the processing of distributed computations.
10. Minimum Time: an 8-byte field used to store the minimum-time replies of minimum-time query computations. This is a user-definable format that must allow for values of +INF and presumably could be an IEEE-754 double-precision floating point value.
11. Message Time: an 8-byte field used to indicate the logical clock value of the sending simulation node, for simulation-type messages. This is a user-definable format that must allow for values of +INF and presumably could be an IEEE-754 double-precision floating point value.
12. Destination Node: an optional 4-byte identifier of the node which is the destination of this message.
13. Message Data: the event data for a simulation-type message. It can be any length (up to the limitations of UDP packets as further limited by the link-layer technology in use), allowing for user-definable data types.

Connection setup should follow the same basic procedures as TCP. There must be a manager process for each network interface that is to be involved in the simulation on the host computer. Each manager process acts as both a client and a server. Each process binds to UDP port DMP_PORT, a number which would be determined in an actual implementation. Server sequence numbers are generated using SYN cookies based on the node identifier numbers used in the actual connection creation, or randomly by the client. The receive window size must be valid for each message.

Once the connection is operating and each node manager has received Start Simulation messages, the simulation runs as described previously for DDES. Messages sent using DMP are processed by the receiver, as long as received in order. Acknowledgments are cumulative. Out of order packets (in addition to packets with invalid checksums or md5sum fields) are rejected. In-order management

messages are processed immediately and as previously described. When an in-order simulation message (type 0) is received, the manager attempts to add it to the node's event queue. If this fails, the message must be saved in an input line receive buffer, but the receive window size must be set to zero. When the receive window size is zero, management messages may still be sent to the node, but simulation messages must not be sent. When the node state is saved for rollback recovery, messages in the input line receive buffers must also be accounted for. Once events in the queue have been processed, the events from the input line buffer can be accessed and added to the queue. Once the input line queue size is down to zero and there is more room on the queue for messages, then the receive window size can be increased again.

The information above is enough to create a protocol over UDP that does only deadlock management. To include the benefits of rollback-recovery techniques, the managers should piggyback the query computations with saving the global state. Saving the global state for every query computation would probably be prohibitive and unnecessary, so a user-definable parameter for the manager should be how many query computations should be done between save. How often they should be done would depend upon reliability considerations of the computing cluster (or grid) being used to run the simulation. The more likely failures are expected, the more often state-saving should be done. The user-definable parameter should be tuned so that state-saves are done at least more often than the MTBF. The Poll Flood messages are used as marker messages [14] to ensure that the global state is consistent. When a component fails, the simulation on the host needs to be restarted, once repairs are made. Once it is up and running, it needs to load the last saved node state(s) for the simulation on that host, and then broadcast Rollback messages to all other nodes. The Rollback messages include the initiator and query number of the the query computation that was done when the state was saved. All other nodes will respond by rolling-back to the last saved state as identified by the initiator and query number.

6. Summary

The subject of deadlock management and other techniques for maintaining causality while still allowing the simulation to move forward is in its infancy. Much work has been done in the area of optimistic methods, but these methods are mostly plagued with overly-complicated rollback mechanisms. The conservative methods have largely been ignored. This paper merely hits the surface of what might be possible with conservative methods. The solution presented here is only an incremental improvement over previous versions through improved deadlock detection (handling all possible cases regardless of which node is the probe computation initiator) and through recovery that noes not require null messages, instead using explicit resumption commands based on real-time analysis of the deadlock situation.

There is much more work that needs to be done in determining the optimal methods. A quantitative comparison needs to be done in real implementations of the latest techniques and more methods could be developed that use combinations of techniques not previously used. The conservative method proposed in this paper has been made as simple as possible. In doing so, there are other deadlock recovery techniques that have the potential to cause more than one node to resume operation simultaneously, that have been ignored because of the degree of complexity they would add [9]. This solution presented here, however, should be a good starting point for further improvement.

7. References

[1] Carina Dennis (Senior Editor), Richard Gallagher (Chief Biology Editor), and Philip Campbell (Editor-in-chief), "The Human Genome: Everyone's Genome", *Nature*, Volume 409, 15 Feb 2001, p. 813

[2] T. Antoine-Santoni, F. Bernardi, and F. Giamarchi, "General Methodology for Metabolic Pathways Modeling and Simulation using Discrete Event Approach. Example on glycolysis of Yeast.", *BIOCOMP'07*, Las Vegas, Nevada, USA, Jun 25-28, 2007

[3] Sachie Fujita, Mika Matsui, Hiroshi Matsuno, and Satoru Miyano, "Modeling and Simulation of Fission Yeast Cell Cycle on Hybrid Functional Petri Net", *IEICE Transaction Fundamentals*, Volume E86–A, No.5, May 2003

[4] Natsumi Shimizu, Shunsuke Yamamichi, and Hiroyuki Kurata, "CADLIVE-Based Analysis for the Budding Yeast Cell Cycle", *The 14th International Conference on Genome Informatics (GIW)*, Dec 14-17, 2003, pp. 609-610

[5] Chen Yu Zong, "Biological Pathway Simulation", LSM3241: Bioinformatics and Biocomputing, Lecture 9, Computational Science Department, National University of Singapore

[6] M. L. Hines and N. T. Carnevale, "Discrete event simulation in the NEURON environment", *Neurocomputing*, Volumes 58-60, Jun 2004, pp. 1117-1122

[7] Jayadev Misra, "Distributed Discrete-Event Simulation", *ACM Computing Surveys (CSUR)*, Volume 18, Issue 1, Mar 1986, pp. 39-65

[8] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *Annual ACM Symposium on Principles of Distributed Computing, Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, Ottawa, Canada, 1982, pp. 157-164

[9] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", *Communications of the ACM*, April 1981, Volume 24, Number 11, Issue 4, Apr 1981, pp. 198-206

[10] K. Mani Chandy, Jayadev Misra, and Laura M. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computer Systems*, Volume 1, Issue 2, May 1983, pp. 144-156

[11] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", *ACM Computing Surveys (CSUR)*, Volume 34, Issue 3, Sep 2002, pp. 375-408

[12] Keith Marzullo, "Rollback Recovery", CSE 223A: Principles of Distributed Computing, Winter 2009

[13] Lorin Hochstein, Forrest Shull, and Lynn B. Reid, "The role of MPI in development time: a case study", *Conference on High Performance Networking and Computing, Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, Article Number 34, 2008

[14] K. Mani Chandy and Leslie Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, Volume 3, Number 1, Feb 1985, pp. 63-75

[15] Keith Marzullo and Laura S. Sabel, "Efficient detection of a class of stable properties", Distributed Computing, Volume 8, Issue 2, Oct 1994, Springer-Verlag, pp. 81-91

[16] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Volume 21, Issue 7, Jul 1978, pp. 558-565

[17] Jayadev Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes", ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 4, Issue 1, Jan 1982, pp. 37-43

[18] Edsger W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations", Information Processing Letters 11, 1: 1-4, Aug 1980, The Netherlands

[19] Keith R. Bisset, "An Adaptive Synchronization Protocol for Parallel Discrete Event Simulation", *Proceedings 31st Annual Simulation Symposium 1998*, 5-9 Apr 1998, pp. 26-33

[20] K. V. Anjan and Timothy Mark Pinkston, "An Efficient, Fully Adaptive Deadlock Recovery Scheme: DISHA", *ACM SIGARCH Computer Architecture News*, Volume 23, Issue 2, May 1995, pp. 201-210

[21] Bruno R. Preiss, "Performance of Discrete Event Simulation on a Multiprocessor using Optimistic and Conservative Synchronization", *Proceedings of the 3rd International Conference on Parallel Processing*, 1990, pp. 218-222

[22] Jayadev Misra and K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection", *ACM Transactions on*

Programming Languages and Systems, Volume 4, Number 4, Oct 1982, pp. 678-686

[23] D. K. Reed, S. P. Levitan, J. Boles, J. A. Martinez, and D. M. Chiarulli, "An Application of Parallel Discrete Event Simulation Algorithms to Mixed Domain System Simulation", *Design, Automation, and Test in Europe, Proceedings of the conference on Design, automation and test in Europe*, Volume 2, 16-20 Feb 2004, pp. 1356-1357

[24] Richard Fujimoto, Kalyan Perumalla, George Riley, and Jean Walrand (Series Editor), *Network Simulation*, Synthesis Lectures on Communication Networks, Morgan & Claypool Publishers, 2007

[25] "Transmission Control Protocol", RFC 793, prepared for Defense Advanced Research Projects Agency, Information Processing Techniques Office, 1400 Wilson Boulevard, Arlington, Virginia 22209, by Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291

[26] R. Rivest, "The MD5 Message-Digest Algorithm", RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992