Filter: An Algorithm for Reducing Cascaded Rollbacks in Optimistic Distributed Simulations

Atul Prakash
Rajalakshmi Subramanian

Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122.
email: aprakash@zip.eecs.umich.edu

Abstract

We describe a new algorithm, called Filter, that limits the propagation of erroneous computations in optimistic discrete-event distributed simulations. In the proposed algorithm, each message carries a bounded amount of dependency information that describes the assumptions made in the generation of the message, and, in addition, processes keep track of straggler events that have occurred in the system. This knowledge is used by processes to "filter" out messages that depend on a preempted state by discarding them upon receipt. We describe the algorithm and its use in conjunction with time-warp, suggest several ways of reducing its potential overhead by adjusting the extent of filtering, and point out several interesting performance tradeoffs that we are currently exploring.

1 Introduction

Distributed discrete-event simulation is potentially a powerful technique for getting speedups in discrete-event simulations. Discrete-event simulations of large systems in many domains, including computer science, engineering, and military applications, generally take enormous amounts of time and the goal of distributed discrete-event simulations is to get speedups in carrying out simulations by partitioning the system in such a way that parallelism can be obtained. In this paper, we consider those class of techniques where a single simulation program is run on multiple processors by decomposing the program into concurrently running logical processes [1, 2, 6]. Each process maintains its own logical clock, running asynchronously with other processes.

In order to ensure that each process executes events in the correct sequence, there are two broad classes of simulation algorithms, *conservative* and *optimistic*. Some of the conservative algorithms include the null message scheme [1, 2] and deadlock detection and recovery scheme [3], an hierarchical scheme [9], and conservative time windows [7]. Most well known optimistic algorithm is time-warp [6]. A survey of many of the distributed simulation algorithms can be found in [5]. In this paper, we are concerned with reducing overheads of optimistic algorithms for distributed simulations.

In the well known time-warp method [6], a causality error occurs whenever a message is received that contains a time-stamp smaller than that of the last processed message. The event message causing the rollback is called a *straggler*. Since the process being rolled back may have sent messages that are inconsistent with the rolled-back state, cancellation events in the form of *anti-messages* have to be sent to *annihilate* the sent messages. These anti-messages can cause further rollbacks if the next process has already processed the message that was supposed to be annihilated by the anti-message. Unfortunately, it is possible for these rollbacks to propagate for a long time, especially if the process graph has loops, with anti-messages chasing regular messages around the loop. Rollbacks are generally expensive, and a major cause of degradation in potential performance of time-warp algorithms.

Mechanisms have been proposed to prevent the spread of erroneous computation using schemes such as Wolf [8] and optimistic time-windows [10]. These approaches can however also impede the progress of correct computations by unnecessary freezing of correct computations, since they cannot distinguish bad computations from good ones [5]. In this paper, we propose an alternative approach, using *conditional messages*, that is able to distinguish between bad computations from good ones and only prevent bad computations from spreading.

The aim of the algorithm is to avoid processing of messages that would eventually be annihilated by anti-messages currently in transit. This is done by attaching sufficient information with each message sent out so that a process can determine if the message is *definitely* going to be annihilated by a future anti-message, and therefore should not be processed. This is done by propagating with each message a list of *assumptions*, which if they were to be violated, the message would eventually have to be annihilated. These assumptions are simply the *conditions* that must hold true in order for the message to eventually survive the simulation.

Whenever a process receives a message, it checks if any events that could violate the assumptions associated with the message are *known* to have occurred (each process maintains a list, called *rollback list*, of straggler events for the purpose). If a violation of assumptions is known to have occurred, the message is discarded. Otherwise, the message is accepted for processing. So, in the proposed algorithm, acceptance of a message for processing is conditional upon its assumptions being consistent with the current knowledge about the straggler events in the system.

A conservative algorithm based on conditional knowledge has been proposed by Chandy and Sherman [4]. In our algorithm, we are using conditional knowledge to improve performance of optimistic algorithms. The proposed algorithm is quite general, being applicable to all simulations for which time-warp is applicable.

For simplicity of treatment, we make the following assumptions in this paper:

1. Each communication channel is only between two processes, and is FIFO and reliable.

2. Each message on a channel carries a monotonically increasing sequence number, despite rollbacks.

3. The simulation system has a fixed communication topology.

The first two assumptions are easy to satisfy using appropriate communication protocols. The third is really not needed, except to keep our description simple. We also assume that each communication channel has a known globally unique identifier (which is easy to assign before execution, because of fixed communication topology).

In Section 2, we describe our algorithm, called *Filter*, that uses the rollback list for filtering incoming messages and propagating rollbacks quickly and give examples to illustrate its use. In the algorithm as described, time-warp algorithm with anti-messages is still being used to propagate rollbacks, but processing of many unnecessary messages is avoided by selective filtering. In section 3, we explore the computation-filtering tradeoff in the algorithm, suggesting various ways of cutting down the overhead of the algorithm and show that assumption lists can be kept bounded in size. In the Section, we also discuss a variant of the algorithm, called *strong_filter*, that does sufficiently extensive filtering so that anti-messages become redundant and unnecessary. In section 4 we present our conclusions and planned future work.

2 Algorithm Filter

As in the standard time-warp algorithm, we assume that a process selects the message with the least time-stamp from its input channels, does some processing to possibly update its local state, and then forwards zero or more messages on its output channels. When a message is processed, its copy is saved in an Input Queue, so that the input queue can be restored in case of a rollback, as in time-warp. Any messages sent are saved in an Output Queue, so that in case of a rollback, anti-messages can be sent to annihilate each of the messages sent that are not compatible with the restored state.

In the algorithm Filter, each process also maintains an assumption list, consisting of tuples of the form $< c, s, t >$. Each tuple $< c, s, t >$ can be interpreted as follows: the current process state assumes that no straggler message will be received on channel c with a sequence number greater than s, but with time-stamp smaller than t. If such a message is received, the state of this process will eventually have to be rolled back to undo the acceptance of this message. Note that the assumption list is part of the state of the process. It is restored to the saved value whenever a process is rolled back.

Whenever a regular message (data messages, other than anti-messages) is sent, the assumption list of the process is attached to the message. Regular messages in the system are therefore of the format <*channel_id, s, d, t, assum_list*> where *channel_id* is the channel on which message is sent, s is the sequence number, d is the data in message, t is the time-stamp[1] of the

[1]time-stamp refers to the virtual receive time [6], i.e. the

message, and *assum_list* is the message's assumption list. Whenever a message is accepted for processing, the process updates its assumption list to include the assumptions of the message. Assumption list of an arriving message is saved, as part of the message, in the Input Queue upon processing. An assumption list can be thought of as the list of assumptions made (about future events) in coming to the current state and in the generation of outgoing messages.

Each process also maintains a rollback list, consisting of tuples of the form $< c, s, t >$. Each tuple $< c, s, t >$ on the rollback list is interpreted as follows: this process knows that a straggler event was received from channel c with sequence number s and time-stamp t. In this paper, we distinguish between straggler events and anti-messages. Straggler events are regular messages that simply arrive late (out of order). On the rollback list, *only* straggler events are recorded, not the resulting anti-message events. Channel c can be any communication channel in the system, not necessarily incident on this process.

Whenever a rollback occurs in any process due to a straggler event, that process broadcasts a new type of message, called a *rollback-info* message, containing information in the form of the above $< c, s, t >$ tuple about the straggler event to other potentially affected processes. Rollback list is *not* a part of the state of the process; it retains its tuples even when a process has a rollback.

A process accepts the message with the earliest time-stamp from its input. The message can either be a regular message, an anti-message, or a rollback-info message. Below, we describe how each of the message is handled in more detail.

2.1 Processing of regular messages

The processing of a regular message M is done as follows:

1. *Filter:* Check the message's assumption list against the rollback list of the process. If there is a *conflict*, the message is discarded and processing goes on to the next message. There is a conflict if an assumption tuple $< c, s_a, t_a >$ on the message's assumption list is in contradiction with a tuple $< c, s_r, t_r >$ for the same channel on the processes' rollback list. There is a contradiction if $s_a < s_r$, but $t_a > t_r$. For example, an assumption tuple $< c, 10, 20 >$ would conflict with a rollback tuple $< c, 12, 16 >$. The assumption tuple $< c, 10, 20 >$ is saying that no event is assumed to occur on channel c with a sequence number larger than 10, but with time smaller than 20. The rollback tuple $< c, 12, 16 >$ is saying that such an event has already occurred (with sequence number 12 and time-stamp 16), and therefore the message is derived from a rolled-back state and need not be processed.

2. *Straggler Check:* If time-stamp of the message is less than that of last accepted message, roll back the process as in the usual time-warp algorithm, sending anti-messages as necessary. Recall that, assumption list is treated as part of the state of the process, and is also rolled back to its earlier value. Append a node containing the following tuple about the straggler event to the rollback list: $< c, M.s, M.t >$, where c is the channel on which M was received, and $M.s$ and $M.t$ are the sequence number and time-stamp of M. A rollback-info message containing the tuple is also broadcast to other nodes so that they can do filtering on any messages currently in transit that originated in a rolled-back state.

3. *Assumption list update:* Otherwise, the message is a non-straggler regular message that is not known to have originated from a rolled-back state to this process. Add the message to the saved message queue and update the assumption list of the process using the algorithm described in Section 2.4.

4. Processing: Do the user-specified processing of the message. With any message sent out, attach the assumption list of the process.

2.2 Processing of rollback-info messages

Rollback-info messages carry information about straggler events that have occurred elsewhere in the system. They are processed as follows:

1. Append the $< c, s, t >$ tuple in the rollback-info message to the rollback list.

2. If the rollback tuple is in *conflict* with any tuple in the assumption list of the process, a rollback is necessary. In case of a conflict, go through the Input Queue looking for the first message that conflicts

virtual time at which a message is supposed to be received at the destination. Virtual send times [6] are not directly used in Filter, but may be added to messages if needed by the underlying time-warp implementation.

with the *rollback-info* message. Rollback to that point, sending anti-messages as in time-warp. No rollback-info messages needs to be sent as a result of the rollback here, since no new straggler message has been received at this process. Rollback-info messages are sent only by the original process at which a straggler message was received.

2.3 Processing of Anti-messages

If the message is an anti-message, go through the input queue to annihilate the corresponding message. If a message is annihilated, carry out a rollback to a state consistent with the anti-message, sending out anti-messages if necessary. If the corresponding message is not found, the anti-message is discarded (the corresponding message must have been received and filtered out earlier since ordered delivery of messages is assumed).

As in the previous case of rollback-info messages, no additional rollback-info message needs to be sent since broadcast of original straggler event will cause the same filtering.

2.4 Update of Assumption List

Upon receiving a regular message M, if the message is not filtered out, a process updates its own assumption list to include the assumptions from the message's assumption list as follows:

1. for all $< c, seq, time >$ tuples on the assumption list of M, append the tuples on to the assumption list of the process. (A more efficient strategy which keeps the list bounded in size is described in Section 3.1.)

2. For all input channels c, append the tuple $< c, M.s, M.t >$ to the assumption list. This assumption says that no straggler event with a time-stamp earlier than this message's time-stamp should arrive in future in order for the present state to avoid a rollback.

A process also updates its assumption list if it simply advances its local virtual clock, without receiving a message. If the local virtual clock is advanced to a time t, then the assumption being made is that no messages will be received with a time-stamp earlier than t. Therefore, for all input channels c, and there corresponding sequence numbers s, tuples $< c, s, t >$ are added to the assumption list of the process.

2.5 Proof of Algorithm Correctness

The proof is based on the fact that the above algorithm *never* filters out a message that should not be, based on the globally known straggler events in the system. The algorithm never rejects a message incorrectly because each message precisely carries the conditions under which processes which it passed through will rollback from their states at that time. If any of the processes rolls back from the state on which the message is dependent, the message would eventually have been annihilated in time-warp too, but using a sequence of anti-messages. In the algorithm, the information about the straggler event is propagated so that the message is either filtered out, and if it had already been accepted by a process, that process is rolled back to a state consistent with the occurrence of the straggler event.

2.6 Examples of the algorithm

Below, we consider two examples that illustrate the use of Filter algorithm.

Example 1:

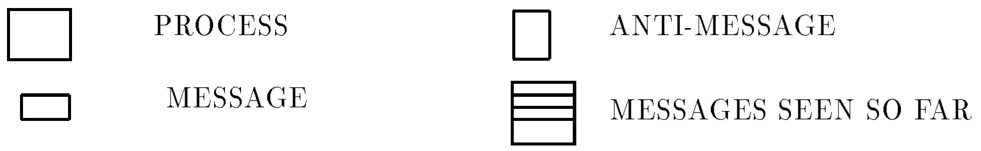
Consider the simulation system shown in Figure 1. Here, P_0 is a source process and P_1 to P_3 are modeling FCFS servers with service time of 5 units. P_0 has already sent messages with time-stamps 5, 35, 40, 65, and 90.

Let us first consider a potential execution sequence in standard time-warp:

1. P_1 receives in succession messages with time-stamp 5, 35, 40 and 65, and sends out messages with time-stamp 10, 40, 45, and 70 to P_2. Figure 1 shows the messages in transit and the Input Queues at that point.

2. P_2 receives messages with time-stamp 10, 40, 45, and 70 and sends out messages with time-stamp 15, 45, 50, and 75 in succession. These are then forwarded by P_3 after updating their time-stamp to P_1. Figure 2 shows the Input Queues and messages in transit at this point.

3. P_1 receives the message with time-stamp 20 from P_3, and has to rollback, since there is a causality error. It sends anti-messages with time-stamp 40, 45, and 70 to P_2 and rolls back to time 20. It moves all the messages that it received out of sequence (35, 40, and 65) back to its input and re-processes them in the correct sequence after the



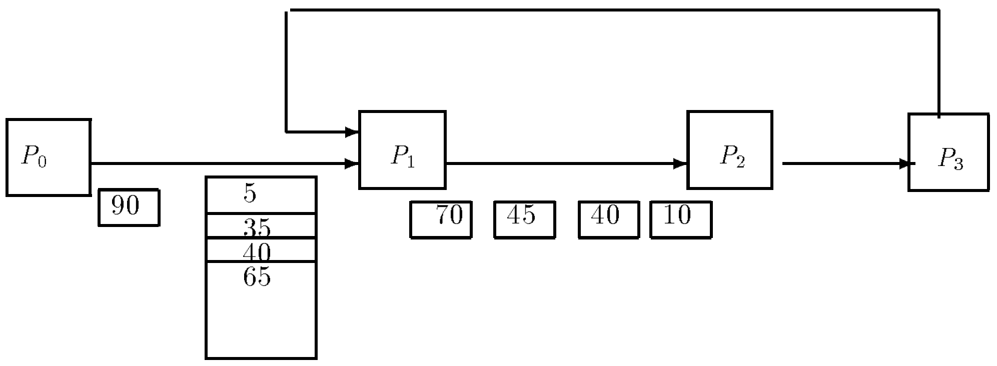


Figure 1: **The state of the Input Queues and the messages in transit in time-warp after P_1 has processed the first 4 messages.**

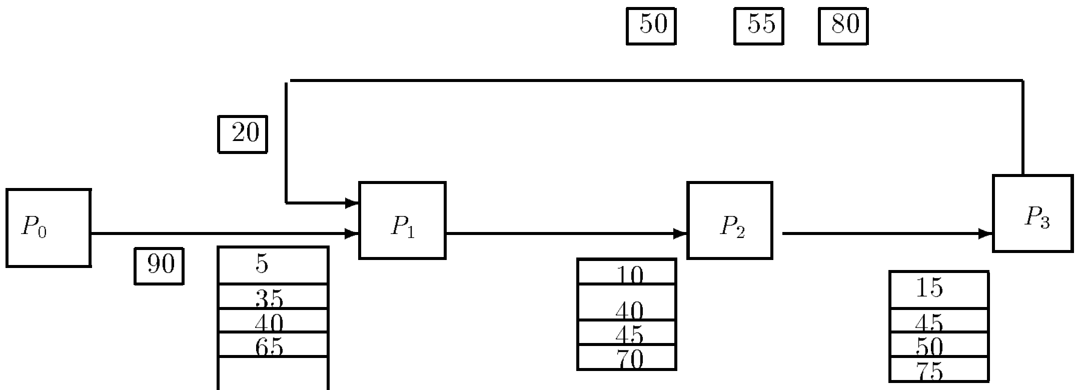


Figure 2: **The state of the Input Queues and the messages in transit in time-warp just before P_1 receives the first straggler message from P_3.**

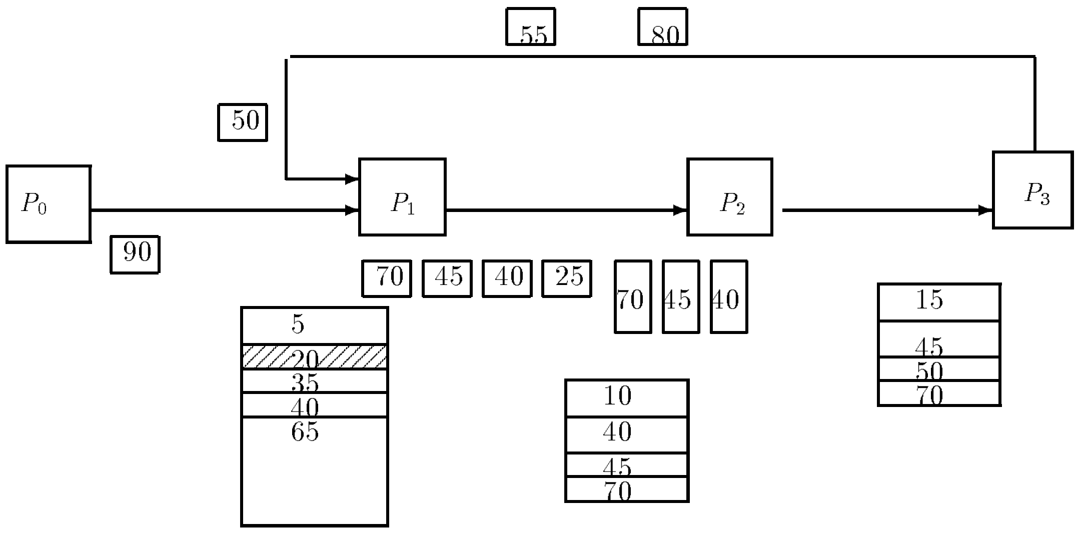


Figure 3: **The state of the Input Queues and the messages and anti-messages in transit just before acceptance of the straggler message with time-stamp 50. Note that this straggler message is being chased by anti-message, currently being P_1 and P_2 with time-stamp 40.**

message with time 20. It then sends out regular messages with time-stamps 25, 40, 45, and 70. Figure 3 shows this situation.

4. Now the interesting situation arises. P_1 next receives the message from P_3 with time-stamp 50. This causes a rollback to time 50. The process sends an anti-message with time-stamp 70 to P_2, rolls back, and then, after rearranging the input queue, sends messages with time-stamp 55, and 70 to P_2 (not shown).
5. P_1 will receive further messages from P_3. Those may cause additional rollbacks to propagate.
6. Eventually, after several steps, P_1 will receive an anti-message with time-stamp 50, undoing step 4 and further steps.

We could go further, but notice that at step 4, the rollback was unnecessary. The message with time-stamp 50 is being chased by an anti-message and need not be accepted. It causes an unnecessary rollback at step 4, and later at step 6 will cause additional rollbacks to undo its acceptance when its anti-message finally arrives at P_1. Not only that, the rollback can potentially propagate along the loop for quite a while.

In the proposed algorithm, situation changes as follows. The second message (time-stamp 35) processed by P_1 will carry an assumption of $< 4, 0, 35 >$, where we let 4 be the id of the channel from P_3 to P_1 (Figure 4). This assumption will propagate with the message and will still be attached to it when the message is later received at P_1 with time-stamp 50. But, before that, at step 2 above, the rollback will occur at P_1 due to the straggler message with time 20. This event is recorded in the rollback list in the form $< 4, 1, 20 >$, where 4 is the channel number, 1 is the sequence number of the message on this channel, and 20 is the time-stamp (Figure 5). Later, at step 4 when the message with time-stamp 50 is received, it is *discarded* since it conflicts with an event in the rollback list. Therefore, an erroneous computation path is avoided by the Use of the Filter algorithm, potentially decreasing the number of messages incorrectly processed and resulting rollbacks.

Example 2:

This example shows that the Filter algorithm can also be effective in controlling the propagation of wrong computations in networks without loops but with multiple feed-forward paths. Consider the situation in Figure 6. Suppose P_1 has to roll back because of a straggler event. In the Filter algorithm, notification of the rollback will be broadcast in the form of a rollback-info message, say A. However, prior to the rollback, P_1 had sent messages B and C which are currently in transit. If the rollback-info message manages to arrive before B and C at P_2 (through the quicker direct channel), we have potential savings. P_2 will correctly discard messages B and C since their assumptions will conflict with the tuple in the rollback-info message. In standard

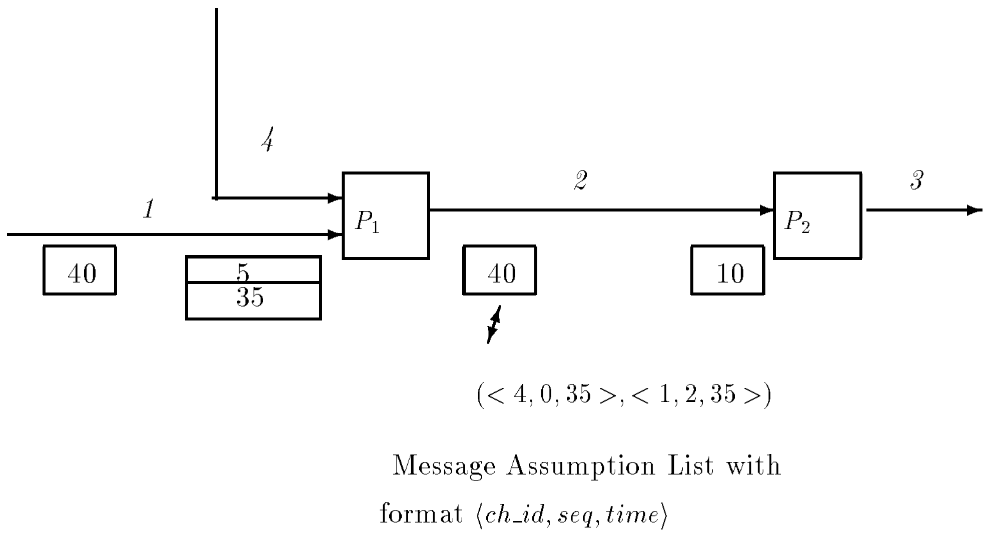

4
1
40
5
35
P_1
2
40
10
P_2
3
$(<4,0,35>,<1,2,35>)$
Message Assumption List with
format $\langle ch\_id, seq, time\rangle$

Figure 4: **The assumption list that propagates with the 2nd message, currently with time-stamp 40.**

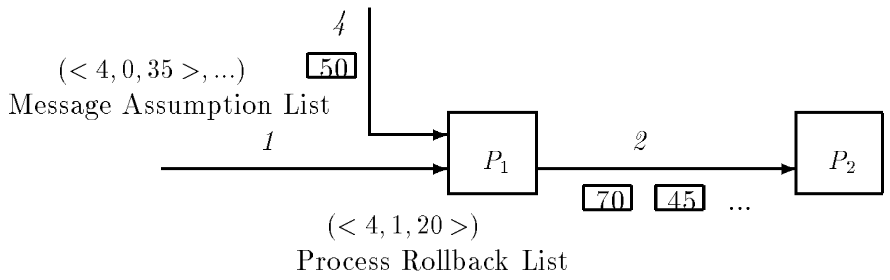


Figure 5: **The assumption list that propagates with the message with time-stamp 50 has a tuple $< 4, 0, 35 >$. The rollback list for process P_1 has the tuple $< 4, 1, 20 >$. Since there is a conflict, the message with time-stamp 50 will be discarded even though its anti-message is still in transit.**

time-warp, both B and C would have been processed by P_2 and the incorrect computation could potentially have propagated beyond P_2 by the time anti-messages for B and C were received. In general, in the Filter algorithm, broadcasts of rollback-info messages should be given priority over other messages so that computing along incorrect execution paths in the simulation can be avoided as much as possible.

3 Performance Issues

There are several performance issues that are raised by the Filter algorithm. It is cutting down on number of messages wrongly processed but at the expense of maintaining assumption lists and rollback lists. While only experimentation will tell whether the tradeoff is effective, we discuss below some of the ways in which the overhead can be cut down.

3.1 Limiting the size of the assumption list

Whenever a process decides to accept a message, each assumption made in generation of the message has to be added to the list of assumptions made by the process in arriving at the current state. We could simply append all the tuples in the arriving message to the assumption list of the process, but, even after eliminating duplicates, the list would tend to grow over time.

Since the Filter algorithm is being used in conjunction with time-warp, one way to keep assumption lists small (and therefore message overhead small) is to simply prune the assumption list at any time by discarding some of the assumption tuples. This reduces the amount of filtering, but the algorithm would still work since anti-messages will eventually annihilate the rest of inconsistent messages not caught by Filter. Ways to keeping assumption list small include keeping only the assumption with highest sequence number for each channel, or to keep track of assumptions only for selected channels. Both strategies can significantly reduce the computational and communication overhead of the Filter algorithm.

It turns out that the assumption lists can be kept bounded in the number of channels by maintaining only one tuple per channel, and without loss of filtering action, provided rollback-info messages are broadcast to all the potentially affected nodes (i.e. those that are reachable from the broadcasting process) using the following broadcast algorithm. The broadcast algorithm is to simply forward the rollback-info message on all the output channels. Processes, upon receiving a rollback-info message, immediately forward it on all their output channels, if they had not done so previously. Assuming ordered delivery of messages, this algorithm guarantee that rollback-info messages are received before any messages that are derived from next state of the process initiating the broadcast.

With the above broadcast protocol in place, suppose that a tuple $< c, s1, t1 >$ from the message M's assumption list is to be added to the assumption list L of the process. If L does not contain any tuple for the same channel c, then $< c, s1, t1 >$ can simply be added as a new tuple to L. However, if it does contain another tuple for the same channel, say $< c, s2, t2 >$, the action taken is to simply discard the tuple with the lower sequence number and keep the one with the higher sequence number on the list. If the sequence numbers are the same, the one with higher time is kept. Let us consider the various possibilities to see why this action makes sense.

Case $s1 > s2$ and $t1 \geq t2$: The action is to replace $< c, s2, t2 >$ by $< c, s1, t1 >$. The justification is that no straggler event violating $< c, s2, t2 >$ with sequence numbers between $s2$ and $s1$ could have occurred on c, and therefore the entry is not needed. Had such a straggler event occurred, a rollback-info message would already have arrived through the above broadcast protocol, causing a rollback to a time before $< c, s2, t2 >$ was added to the assumption list, thereby removing $< c, s2, t2 >$ from the list.

Case $s1 > s2$ and $t1 < t2$: In this case, a straggler event *must* have occurred on the channel c with sequence number between $s1$ and $s2$, violating $< c, s2, t2 >$. However, using the above broadcast protocol for rollback-info messages, straggler event notifications are received before messages derived from a subsequent state of the process, and therefore the tuple $< c, s2, t2 >$ should already been discarded. Therefore, this case cannot occur with the above broadcast protocol.

If $s1 < s2$, it can be similarly argued that only $< c, s2, t2 >$ needs to be kept in an assumption list. Therefore, we have the conclusion that, with the above broadcast protocol for rollback-info messages, only one tuple per channel needs to be kept in the assumption list of a process without loss of filtering action.

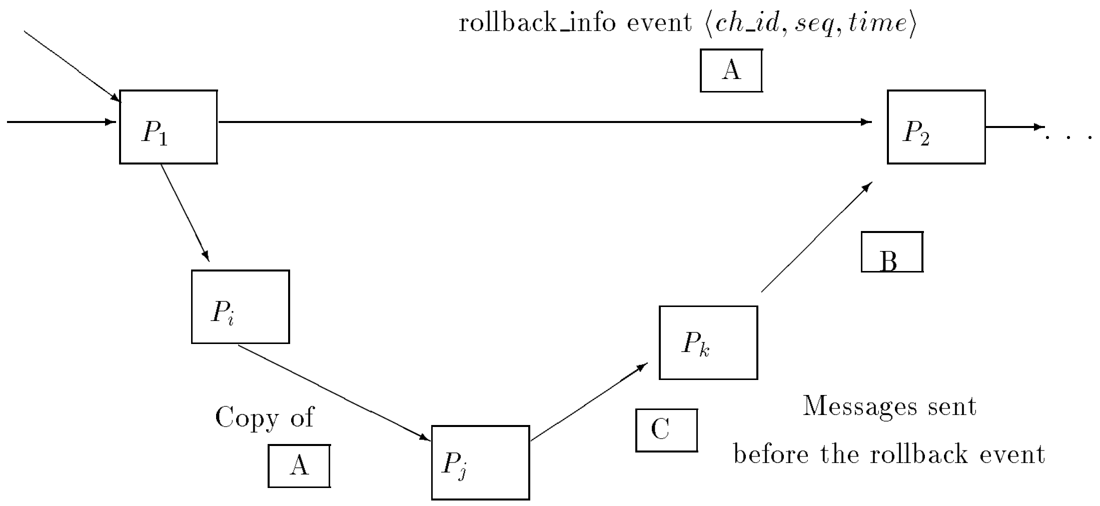

rollback_info event $\langle ch\_id, seq, time \rangle$
A
P_1
P_2
. . .
P_i
B
P_k
Copy of
A
P_j
C
Messages sent
before the rollback event

Figure 6: **Messages B and C were derived from a state at P_1 prior to the rollback. If the rollback-info event A reaches P_2 prior to B and C, then B and C will be filtered out by P_2. In standard time-warp, anti-messages for B and C would have traveled the same path as B and C, potentially taking a long time before catching up, leading to spread of an erroneous computation from P_2.**

3.2 Limiting the size of the rollback list

If the Filter algorithm is being used in conjunction with time-warp, There are several ways for reducing overhead associated with rollback lists without affecting the correctness of the simulation. Note that any inconsistent messages not filter out by use of rollback lists are eventually annihilated by anti-messages. Therefore, rollback list can be pruned at any point, say by discarding very old tuples.

Some efficiency is possible in the broadcasts of rollback-info messages. Rollback-info messages need to be sent to only those processes that are reachable from the process. Other processes cannot be affected by a straggler event notification since they cannot see any messages that depend on the state of the rolled-back process.

In fact, since the algorithm is being used in conjunction with time-warp, rollback-info messages need not even be sent. A process receiving a straggler message can simply update its own rollback list and make no broadcast. Since anti-messages are being sent anyway, they would annihilate any messages not filtered out as a result.

3.3 Limiting the information to selected channels

In many simulations, the only place where straggler events can occur are at merge points, where a process has more than one input channel. If that is the case, assumption lists and rollback lists can be restricted to those channels that are part of a merge point. For instance, in Example 1, the only channels on which straggler events can be received are the channel from P_0 to P_1 and the channel from P_3 to P_1. On all other channels, messages will be received in increasing timestamp order, or with an intermediate anti-message or rollback-info message. Therefore, in that example, assumption lists and rollback lists need to keep track of only two of the channels, those input to P_1, significantly reducing the overhead.

3.4 Avoiding sending unnecessary anti-messages

If all processes keep all tuples on their assumption and rollback lists for a particular channel, and if rollback-info messages for that channel are always broadcast reliably, then it is unnecessary for the process making the broadcast, as a result of a straggler message on that channel, to also send anti-messages. Since messages carry all the assumptions regarding that channel, a rollback-info message will eventually annihilate all the messages that originated in the state prior to the receipt of the corresponding straggler event, either by causing filtering or by causing rollbacks.

An extreme implementation of the algorithm is that no anti-messages are ever sent out, and only rollback-info messages are relied upon for all rollbacks and annihilation. We call this the *Strong_Filter* algorithm. In this case, all processes *have* to make broadcasts of rollback-info messages, assumption lists cannot be pruned arbitrarily[2], and rollback-info events cannot be pruned from the rollback lists. The advantage here is that Output Queue is no longer necessary, since anti-messages are not sent. The disadvantage is that rollback lists can grow over time and there is a higher computation overhead per message, in the form of checking conflicts over the whole assumption list, etc. It remains to be examined whether the higher overhead in computation compensates for the reduction in number of messages and rollbacks.

Clearly, there are many ways to reduce overhead of the proposed algorithm by varying the degree of filtering and taking advantage of knowledge about channels on which straggler events can occur. We are currently implementing time-warp algorithm on a set of workstations, with and without Filter, and will report on our experimental results in the future. Further performance studies should show under what situations the algorithm is effective.

4 Conclusions

We have proposed an algorithm for reducing the number of rollbacks in optimistic distributed simulation. The algorithm is interesting in the way it keeps track of dependencies between messages so that messages can be discarded if they are known to have originated in a state incompatible with current knowledge about the global state. The Filter algorithm is likely to increase the overhead per message in the system, but potentially reduce the total number of messages sent around and processed, and the number of rollbacks. The algorithm allows some control over how much filtering is carrying out in a particular simulation all the way from no filtering (completely relying on anti-messages of time-warp), to complete filtering (completely relying on rollback-info messages). Future performance studies will indicate how effective filtering is in practice and how much filtering should be carried out for different types of simulations.

[2]Message assumption lists can still be kept bounded used the strategy described in Section 3.1.

Acknowledgments

This work was supported by the National Science Foundation under the Grant NSF-CCR-8909674.

References

[1] R.E. Bryant. Simulation on a distributed system. In *Proc. of COMPSAC*, 1979.

[2] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEETSE*, 1979.

[3] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *CACM*, 24(11):198–206, April 1981.

[4] K.M. Chandy and R. Sherman. The conditional event approach to distributed simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):95–99, March 1989.

[5] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[6] David R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, July 1985.

[7] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32:111–123, January 1989.

[8] V. Madisetti, J. Walrand, and D. Messerschmitt. WOLF: a rollback algorithm for optimistic distributed simulation systems. In *1988 Simulation Conference Proceedings*, pages 296–305, December 1988.

[9] Atul Prakash and C.V. Ramamoorthy. Hierarchical distributed simulations. In *Proc. of the Eighth International Conference on Distributed Computing, San Jose*, pages 341–348, 1988.

[10] L.M. Sokol, D.P. Briscoe, and A.P. Wieland. MTW: A strategy for scheduling discrete simulation events for concurrent execution. In *Proc. of the SCS Multiconference on Distributed Simulation*, pages 34–42, July 1988.