Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

ROLLBACK SOMETIMES WORKS ... IF FILTERED (ABSTRACT)

Boris Lubachevsky
*Adam Shwartz**
Alan Weiss

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract We introduce a new parallel discrete event simulation algorithm called filtered rollback. It is a combination of the Time Warp and bounded lag simulation algorithms introduced previously. The "filter" postpones event processing in some subsystems in favor of safer simulation. The filter may be tuned by the simulationist; at one extreme the algorithm is conservative, i.e., free from rollback, and at the other extreme the algorithm is purely optimistic, i.e., relying exclusively on rollback. We prove that rollback cascading, wherein a "chain reaction" of secondary and higher generation rollbacks appear in the simulation, can be bounded by an appropriate tuning. The tuning achieves a trade-off between the two extremes which yields an efficient and scalable algorithm. Our method of proof uses a representation of the rollback cascading as a tree and models such a tree as a Galton-Watson branching process on which an additional structure is defined, a random walk with a barrier.

1. **Introduction.** "Rollback" simulation refers to algorithms such as Time Warp [4] which allow simulated subsystems to keep different local times, and which recover from errors caused by such asynchrony by rewinding the state of the simulated system to earlier times.

There are at least two modes in which rollback-based simulation may become inefficient, namely *echoing* and *cascading*. *Echoing* is a pattern of self-fueled rollbacks whose *amplitude* increases without bound. In Appendix 1 we construct an echo between just two nodes in a three-node network. *Cascading* is a "chain reaction" of rollbacks where the *number of participants* increases without bound. Although we are currently unable to produce an example of cascading as convincing as the one of echoing in Appendix 1, our Theorem 2 (Part II) gives strong indications that cascading exists.

A *conservative*, i.e., rollback-free algorithm named "bounded lag" was introduced in [6]. It requires the programmer to provide guaranteed lower bounds on the delays between causally connected events. As discussed in [6], good bounds which would assure efficiency may be difficult to provide in certain cases. The "filtered rollback" scheme described in the present paper was introduced in [6] (Conclusion, p.123) as an amalgam of "bounded lag" and "rollback." We call this scheme *filtered rollback*, because the bounded-lag stage "filters" the event stream before it enters the rollback stage. The filter postpones some event processing in favor of safer simulation. The scheme contains tunable parameters which at one extreme make the algorithm identical to a (possibly inefficient) implementation of bounded-lag, and at the other extreme make the algorithm identical to a synchronized "moving time window" rollback [8].

Echoing with unbounded amplitude cannot occur because rollback amplitude cannot exceed the lag bound. However unbounded cascading may still occur. This paper shows that by tuning the rollback filter one may achieve a trade-off between the extremes yielding an efficient algorithm: the rollback cascading is bounded independently of the size of the simulation. Furthermore, the simulation is scalable in the following sense: as both the simulated system size and the number of processing elements in the parallel simulator proportionally increase, keeping other parameters (e.g., local degrees of the simulated network, density of simulated events, length of simulated runs) unchanged, the processing time increases not faster than the logarithm of the size (see Sec. 4).

Our method of proof may also be applicable to the analysis of other rollback

---

Adam Shwartz participated in this work while visiting AT&T Bell Laboratories. His permanent address: Electrical Engineering Technion, Israel Institute of Technology, Haifa 32000, Israel.

schemes. We represent rollback cascading as a tree and model such a tree as a Galton-Watson branching process [1] on which an additional structure is defined, a random walk with a barrier. The damage to performance due to cascading increases when the tree size does and we present tight conditions for the tree boundedness which are also sufficient conditions for the boundedness of the cascading.

2. **The bounded lag algorithm.** To define filtered rollback, we first need a definition of the bounded lag algorithm. Consider the simulation of a static network with nodes 1,2,...N. For simplicity we assume that each node i is permanently hosted by its own processing element (PE). (The results are easily extendible to the case when one PE permanently hosts several nodes.) Node i is represented in the simulation by its (possibly empty) set Π_i of tentative future events, each event being a pair (event content, event occurrence time); by $T(i)$, the minimum of the event times in Π_i; and by an estimate $\alpha(i)$ of the earliest time when the history at node i can be affected by other nodes. Fig. 2.1 represents the conservative bounded-lag algorithm [6]. The convention for interpreting the code in Fig. 2.1 is that all PEs execute it concurrently. Thus all PEs at Step 1 check condition *floor* < *end_time*, then at Step 2 compute $\alpha(i)$ for all nodes $i = 1,2,...N$, then at Step 3 synchronize and so on. No synchronizations are implied except for the ones explicitly stated, e.g., a PE may begin Step 2, while another PE is still at Step 1.

In this algorithm, B is an arbitrarily chosen positive upper bound on the time lag. Line 4 in Fig. 2.1 assures the bounded lag property: the difference in simulated time between any two events processed concurrently does not exceed B.

Let $d(i,j)$ be the *minimum propagation delay* from node i to node j, and $S\downarrow(i,B)$ the incoming reachability sphere of radius B, with the center at i; i.e., the set of nodes j such that $d(j,i) \le B$. Delays $d(i,j)$ are supposed to satisfy the triangle inequality: $d(i,j) + d(j,k) \ge d(i,k)$ for any three nodes i, j, and k.

Several methods for implementing Step 2 are described in [6]. The easiest one to explain is the following:

$$\alpha(i) = \min_{\substack{j \in S\downarrow(i,B) \\ j \neq i}} \{ d(j,i) + \min\{ T(j),\ d(i,j) + T(i) \} \}, \qquad (2.1)$$

In algorithm (2.1), each node i computes its $\alpha(i)$ independently of the other nodes. The PEs hosting nodes $j \in S\downarrow(i,B)$ communicate $T(j)$ to the PE hosting node i.

Figure 2.1. A conservative bounded lag algorithm

initially: *floor* = 0, some $\Pi_i \neq \varnothing$,
time$(e) \ge 0$ for each $e \in \bigcup_{1 \le i \le N} \Pi_i$,
$T(i) = \min_{e_i \in \Pi_i} time(e)$ if $\Pi_i \neq \varnothing$
$T(i) = +\infty$ if $\Pi_i = \varnothing$

1. while *floor* < *end_time* do {
2. compute bound $\alpha(i)$ of the earliest time when the history at node i can be affected by the other nodes ;
3. synchronize ;
4. while $T(i) \le$ *floor* + B and $T(i) < \alpha(i)$ do {
5. process events e with locally minimum time $T(i)$;
if required, schedule new events for node i or other nodes j and/or delete some events from Π_i or other Π_j ;
6. delete the processed events from Π_i and compute new $T(i)$;
} ;
7. synchronize ;
8. compute *floor* $\leftarrow \min_{1 \le i \le N} T(i)$ and broadcast *floor* to all nodes ;
9. synchronize ;
}

Let $t(i,j)$ be a delay in simulated time required for an event at node i to affect the history at node j. Given i and j, for different events at node i, the $t(i,j)$ may be different. A particular value of $t(i,j)$ may not become known until the actual simulation is done. However, the estimate $d(i,j)$ must be known in advance. In the simplest case, $d(i,j)$ is a constant precomputed before the simulation run. (One might choose to update $d(i,j)$ during the run, as discussed in [6].) By definition, in the conservative algorithm the values $d(i,j)$ are chosen so that the conditions

$$d(i,j) \le t(i,j), \quad i,j = 1,2,...,N \qquad (2.2)$$

always hold, which guarantees 100% safety of estimates $\alpha(i)$: no event can affect the history at

node i earlier than at time $\alpha(i)$. The test $T(i) < \alpha(i)$ at Step 4 of Fig. 2.1 prohibits processing of an event scheduled at node i with time greater than or equal to $\alpha(i)$.

3. **The filtered rollback algorithm.** In the bounded lag algorithm the programmer is required to provide positive estimates $d(i,j)$ which guarantee (2.2). This may be difficult, undesirable, or impossible. For example, suppose for fixed i and j, $t(i,j) = 100$ for 99% of the events processed at node i and $t(i,j) = 1$ for 1% of such events. The safe estimate $d(i,j)$ must be valid for all processed events and hence must be no larger than 1. However, taking $d(i,j) = 1$ decreases parallelism and is therefore inefficient. We propose to solve this problem in "filtered rollback" by allowing (2.2) to be violated. For example, if the lower bound for $t(i,j)$ is very small or even zero but the mean, $Et(i,j)$, is positive, we can take $d(i,j) = Et(i,j)/2$, in the hope that (2.2) will be violated rarely. The guidelines for setting $d(i,j)$ will be discussed more thoroughly in Sec. 5.

Once (2.2) is not guaranteed, an event with smaller simulated time can be submitted to a node for processing after the node has already processed an event with larger simulated time. Antievents, similar to antimessages† in [4] should allow recovery from the error. The version of the algorithm that incorporates the additional rollback mechanisms is given in Fig. 3.1. The additional steps in Fig. 3.1 are enumerated by number-letter indices. The reason we adopt this enumeration scheme is so that the steps which are essentially the same as in Fig. 2.1 have the same numbers as the corresponding steps in Fig. 2.1.

A new quantity $t(i)$ is referenced in Steps 5a, 3a, and 3c in Fig. 3.1; $t(i)$ is defined in 5a as the simulated time of the last processed event. If we similarly introduced $t(i)$ in the conservative algorithm in Fig. 2.1 we could have shown that

$$T(i) \geq t(i) \qquad (3.1)$$

and that $t(i)$ does not decrease as the simulation progresses.

Figure 3.1. A filtered rollback algorithm

initially: $floor = 0$, some $\Pi_i \neq \emptyset$,
$\Pi'_i = \emptyset$ and $t(i) = 0$ for $i = 1,...N$,
$time(e) \geq 0$ for each $e \in \bigcup_{1 \leq i \leq N} \Pi_i$,

$T(i) = \min_{e_i \in \Pi_i} time(e)$ if $\Pi_i \neq \emptyset$

$T(i) = +\infty$ if $\Pi_i = \emptyset$

1. while $floor < end\_time$ do {
2. compute approximation $\alpha(i)$ of the earliest time when the history at node i can be affected by the other nodes ;
3. synchronize ;
3a. if $t(i) > T(i)$ then {
3b. recover the state of node i including Π_i the node had just before simulated time $T(i)$;
3c. for each event e scheduled previously by node i in a different Π_j on the interval of simulated time $[T(i),t(i)]$, create the matching antievent $\bar{e}$ and insert it into Π'_j ;
$t(i) \leftarrow T(i)$;
} else
4. while $T(i) \leq floor + B$ and $T(i) < \alpha(i)$ do {
5. process events e with time $T(i)$;
if required, schedule new events for node i or delete some events from Π_i and/or schedule some events or antievents e for other node j and insert them into Π'_j ;
5a. $t(i) \leftarrow T(i)$;
6. delete the processed events from Π_i and compute new $T(i)$;
} ;
7. synchronize ;
7a. $\Pi_i \leftarrow \Pi_i \cup \Pi'_i$; $\Pi'_i \leftarrow \emptyset$;
compute new $T(i)$;
8. compute $floor \leftarrow \min_{1 \leq i \leq N} T(i)$
and broadcast $floor$ to all nodes ;
9. synchronize ;
}

† The original formulation of the Time Warp algorithm [4] operates with messages and antimessages. Our algorithm operates with events and antievents. Where in [4], queue i sends a message/antimessage to queue j, in our algorithm, node i inserts an event/antievent into set Π_j. The two approaches are equivalent: it is always possible to recast a message-antimessage processing in terms of an event-antievent processing and vice versa. We believe that our algorithm is easier to formulate in the event-antievent language, which does not prohibit a message-antimessage implementation.

However, in the new version, Fig. 3.1, condition (3.1) is not guaranteed. An instance of violation of (3.1) which is detected by the test in line 3a of Fig. 3.1 means that at Step 7a, $T(i)$ was computed for an event or antievent e with $time(e)$ smaller than $t(i)$; note that $t(i)$ is the previous value of $T(i)$ (Steps 5a and 6). Such an event was necessarily inserted by a node $j, j \neq i$. Such an instance requires a rollback at node i as a result of which $t(i)$ will decrease. Observe that any event or antievent inserted by j is not immediately incorporated into Π_i, but is kept in the auxiliary storage Π'_i.

The rollback branch, lines 3b and 3c, begins with recovering the state to what it was just before the error time $T(i)$. We are not proposing new elements in the way this recovery should be implemented and assume it is done as in the Time Warp [4]. Observe that no rollback extends further in the past than the floor. We exploit this in two ways. First, this simplifies the implementation, e.g., bounded memory suffices to remember old states. Second, in our efficiency analysis in Sec. 4, we can assume the cost associated with this action to be $O(1)$ for each node. Another action in the rollback branch is the scheduling of antievents as stated in line 3c.

The described method corresponds to the *aggressive cancellation* policy [4], wherein the same event e may be canceled by the subsequent event $\bar{e}$, then again scheduled, then again canceled etc. A more elaborate *lazy cancellation* policy [3] can also be used and similarly analyzed within the same filtered rollback framework.

When the test in line 3a fails, execution is directed to the non-rollback branch, lines 4-6 of Fig. 3.1, which are similar to lines 4-6 of Fig. 2.1. There are two differences, however. The first difference is that some events subject for processing are antievents. Processing of an antievent $\bar{e}$ by node i simply means finding its counterpart positive event e in Π_i, such that $time(e) = time(\bar{e})$, and deletion ("annihilation") of both e and $\bar{e}$. It can be shown that such a counterpart must necessarily be present in Π_i when node i executes the non-rollback branch. The other difference is in the way node i schedules events for or deletes events from a different node j. In the conservative version at the same iteration node j can process an event e_j belonging to set Π_j while node i can concurrently insert an event e_i into the same set Π_j or delete an event e_i from Π_j. These two activities do not interfere, since the bound $\alpha(j)$ well separates the times:

$$time(e_j) < \alpha(j) \leq time(e_i).$$

An implementation can be arranged in which both PEi and PEj concurrently access the same data structure which represents Π_j without ambiguity.

In the filtered rollback version, node i might schedule an event e_i for Π_j or might attempt to delete an event e_i from Π_j concurrently with the processing of event e_j by node j with $time(e_i) < time(e_j)$. To avoid ambiguity, we assume that the recipient of the scheduled event, node j, incorporates it into its set of tentative events Π_j only after synchronization at Step 11. Event deletion is reduced to event insertion by creating an antievent $\bar{e}_i$. During Steps 3c and 8 the scheduled events, including antievents, are inserted into the auxiliary set Π'_j. Before computing the new *floor*, at Step 7a Π'_j joins Π_j and this action is followed by updating $T(j)$.

In an alternative method, not shown in Fig. 3.1, updating of Π_j takes place concurrently with processing events in Π_j. In this method, if an (out-of-order) event or antievent e_i is inserted into Π_j while node j is processing an event e_j with $time(e_j) \geq time(e_i)$, processing is aborted and a rollback follows. The efficiency analysis given below is valid for both methods.

4. **Efficiency analysis.** We will try to follow the analysis outlined in [7]. The immediate problem which we encounter on this path is event counting. Note that a rollback algorithm may sweep over the same simulated time more than once. Hence the same event can be processed several times. Moreover, during these sweeps some erroneous events may be introduced and processed. We need to distinguish between *inherent events*, those which actually occur and are counted only once, and *processed events*, which are all events, inherent or not, each counted as many times as processed. Let us denote by I the set of inherent events and by Π the set of processed events. It follows that $I \subset \Pi$ and that all events in $\Pi \setminus I$ are canceled by rollbacks. In Sec. 5 we give a finer classification of the set Π.

The proof of the log-scalability of the conservative algorithm [7] is based on the following assumptions:

A1. $\Omega(N)$ nodes‡ have events e satisfying $floor \leq time(e) < floor+B$ at one iteration of the algorithm on the average.

A2. The total number $|I|$ of inherent events in the simulation run is $O(N)$ (For a set A we let $|A|$ denote the number of elements in A.)

A3. There exists a constant D, $0 < D < +\infty$, independent of the size N and node i such that $|S^{\uparrow}(i,B)| \leq D$, where $S^{\uparrow}(i,B)$ is the outgoing reachability sphere of radius B with the center at i, i.e. the set of nodes j such that $d(i,j) \leq B$. (Contrast this with $S^{\downarrow}(i,B)$ defined in Sec. 2.)

A4. $|\Pi_i|$, the number of scheduled events for a node i, is bounded from above by a finite fixed constant independent of the node identity i, the problem size N, and the iteration number.

A5. The number of processed events for a node at an iteration is bounded from above by a finite fixed constant independent of the node identity, the problem size N, and the iteration number.

The proof in [7] proceeds as follows:

Step 1. A1 and A3 imply that the number of events processed at an iteration is $\Omega(N)$, averaged over the simulation run.

Step 2. Examining the algorithm implementation and using A3, A4, and A5 imply the estimate $O(\log N)$ for the time required for one iteration.

Step 3. Dividing the latter over the former yields the estimate $O(N^{-1}\log N)$ for the time spent per event on the average over the simulation run.

Step 4. A2 and the estimate in Step 3 imply estimate $O(\log N)$ for the average processing time, i.e., log-scalability.

In applying the same proof scheme for the filtered rollback algorithm, we observe that Steps 1, 2, and 3, and the assumptions involved, are equally valid in the filtered rollback case. However, Step 4 does not work. In the conservative case, $I = \Pi$, but in the filtered rollback case, Π may be substantially larger than I and assumption A2 does not imply log-scalability anymore. In order to prove log-scalability we shall derive an upper bound for the number of processed events $|\Pi|$.

Since rollback cascading is directly responsible for the extra events, we digress into examining this phenomenon in greater detail.

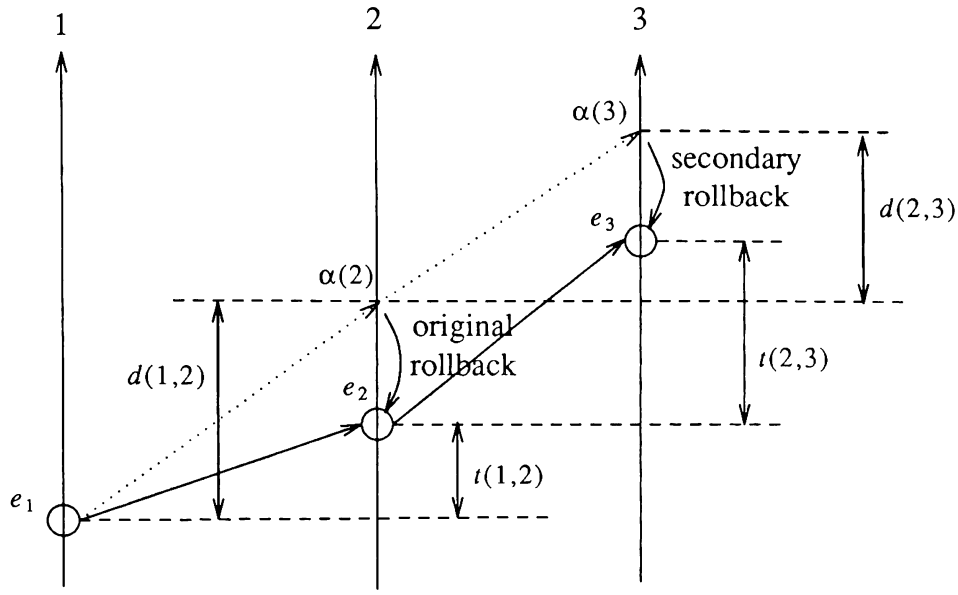
5. **Rollback cascading.** Fig. 5.1 shows a time diagram of a three-node-network simulation. Three time axes numbered 1, 2, and 3 correspond to nodes 1, 2, and 3, respectively, and the simulated time increases upward along the axes. Suppose at iteration 1, nodes 2 and 3 compute their $\alpha(2)$ and $\alpha(3)$ (shown in Fig. 5.1), and suppose Π_2 (resp. Π_3) contains an event e'_2 (resp. e'_3) scheduled at a time slightly less than $\alpha(2)$ (resp. $\alpha(3)$). (Those additional events are not depicted in Fig. 5.1 so as not to clutter the picture.) In the computations of their $\alpha(i)$s both nodes see the minimal-time event e_1 at node 1 and assume that $t(1,2)$ and $t(2,3)$, the delays in propagating this event from node 1 to node 2 and from node 2 to node 3, respectively, will not be smaller than the corresponding $d(1,2)$ and $d(2,3)$. Contrary to this assumption, however, node 1 schedules event e_2 for node 2 and the actual delay $t(1,2)$ happens to be smaller than $d(1,2)$. Node 2 notices at Step 3a (Fig. 3.1) of iteration 2 that rollback is required, at which point it rolls back to $time(e_2)$ invalidating the results of its simulation on the interval from $time(e_2)$ to $\alpha(2)$. This implies that the first occurrence of e'_2 is a non-inherent event. Meanwhile node 3 computes a new $\alpha(3)$ (not shown in Fig. 5.1) which is smaller than the value of $\alpha(3)$ at iteration 1 and is in the past of event e'_3, which has already been processed by node 3. This indicates that a rollback might occur soon, but according to the algorithm in Fig. 3.1 node 3 does not roll back until hard evidence is obtained. Such evidence is generated later, when node 2 processes e_2 and schedules an event e_3 for node 3. Even though the actual delay $t(2,3)$ is larger than its estimate $d(2,3)$, node 3 notices at iteration 3 that event e_3 is in its past and rolls back to $time(e_3)$.

In this example, node 2 has an original rollback and node 3 has a rollback of the second generation. There may be, of course, third and higher generations of rollbacks. Observe that we can always, at least conceptually, trace each non-original rollback to its origin. Specifically, if node i receives a rollback-causing event from node j we put link $i \leftarrow j$ into our trace. If two or more rollback-causing events are received by node i we trace the rollback only by the event with the smallest time, because this rollback overrides the others. Thus, we can visualize rollback trees growing from iteration to iteration with branches sometimes dying out as the simulation progresses.

---

‡ For non-negative functions $\phi(N)$ and $\psi(N)$: $\phi(N) = \Omega(\psi(N))$ and $\psi(N) = O(\phi(N))$ if there exists constant $0 < C < +\infty$ such that $\phi(N) > C\psi(N)$ for all sufficiently large N.

Figure 5.1. Two generations of cascading



Let i be the node at the root of such a tree. In the neighborhood of i we can always find a unique node j and a unique event $e_{starter}$ whose processing at j causes the original rollback at i which starts the tree. In Fig. 5.1, the root is at node $i = 2$ and this original rollback is caused by processing event $e_{starter} = e_1$ at node $j = 1$. A starter event $e_{starter}$ such as e_1 may or may not be inherent, $e_1 \in I$ or $e_1 \in \Pi \setminus I$.

We now wish to decompose Π by classifying the degree to which each event is "non-inherent." We do this by constructing sets $I^{(k)}$ such that

$$\Pi = I^{(0)} \cup I^{(1)} \cup I^{(2)} \cup \ldots,$$

where $I^{(i)} \cap I^{(j)} = \varnothing$ when $i \neq j$. Define $I^{(0)} \stackrel{def}{=} I$. Recursively define $I^{(k+1)}$ as the set of events which are part of trees rooted in $I^{(k)}$, $k = 0,1,\ldots$.

Returning to Fig. 5.1, we observe that the diagram does not specify to which $I^{(k)}$ the events on this rollback tree belong. If there are other nodes in the network besides nodes 1, 2, and 3, it is possible that e_1 may be canceled by a rollback (not shown) at some future iteration. In this case $e_1 \in I^{(k)}$ for some $k \geq 1$, and then $e'_2, e'_3 \in I^{(k+1)}$.

We present a model which upper-bounds the described cascading effect. This model is a tree defined by a standard Galton-Watson branching process [1] on which an additional structure is defined, a random walk with a barrier. This structure represents "survival" of the rollback tree branches. The root of the tree corresponds to an original rollback. In Fig. 5.1, a rollback tree originates at node 2 with positive "survival number" $X_{01} = d(1,2) - t(1,2)$. Node 3 is a first generation child. Suppose it is the first child in this generation; its "survival number" $X_{11} = \alpha(3) - time(e_3)$ can also be represented as $X_{11} = X_{01} + W_{11}$ where $W_{11} = d(2,3) - t(2,3)$. At each level of the tree there may be several children of a particular father because an event processing at a node may schedule events in several neighboring nodes. Each child (k,m) (k-generation number, m-child number) inherits the survival number from the father and adds its own positive or negative increment W_{km} to it. A rollback branch (k,m) terminates (a desirable situation) when its survival number X_{km} becomes negative.

For simplicity of the analysis, let us assume a statistically homogeneous simulation with i.i.d. $t(i,j)$ and with independent event-producers at each node and where the distribution of $t(i,j)$ is independent of the number of nodes N. Note that this homogeneity assumption applies only to the simulated system. Topology and architecture of the host parallel computer is irrelevant, as long as the computer works efficiently for the given topology of the simulated system, e.g., it does not take long to synchronize, or to send a message from a node to a neighboring node. Then we will have

homogeneous branching and i.i.d. survival increments W_{km}. The distribution of the increments is that of $d(i,j) - t(i,j)$. Estimates $d(i,j)$ are in the control of the simulationist. How should these $d(i,j)$ be chosen?

An obvious guideline is to choose $d(i,j)$ so that the $d(i,j) - t(i,j)$ are i.i.d. with negative mean i.e.,

B1. W_{km} are i.i.d and $E(W_{km}) < 0$,

(note that even if the $t(i,j)$ are not identically distributed, the $d(i,j)$ may be chosen so that B1 holds). How negative should $E(W_{km})$ be? Theorems 1 and 2 below provide explicit bounds in terms of the distribution of W_{km} which ensure boundedness of cascading.

Let $p(r)$ be the branching distribution of the process, i.e. $p(r)$ is the probability that a father has exactly r children. Observe that $p(r) = 0$ for values r larger than the local nodal degree D_i at node i. We assume that

B2. $\max_{i \text{ is a node of the network}} D_i \stackrel{def}{=} D < +\infty$.

Assume also

B3. $h \stackrel{def}{=} \mathrm{Sup}_\theta(-\log E\exp(\theta W_{km})) < \infty$,

and define

$b \stackrel{def}{=} E(\text{number of children of a father}) = \sum_k kp(k)$,

which is the branching factor. Straightforward calculations show that by choosing $d(i,j)$ small enough (but positive), h can be made arbitrarily large.

Theorem 1. Under assumptions B1-B3:

if $D < e^h$ then $E(\text{size of the tree}) < +\infty$.

Proof is given in Appendix 2.

6. **Efficiency analysis (cont'd).** Returning to the main-stream of the efficiency analysis, we introduce the following assumptions:

A6. The mean size of all rollback cascading trees is $O(1)$ uniformly over the simulation run.

A7. $E|\Pi| = O(N)$.

A6 follows from Theorem 1. A7 follows from A6 and the analysis below. We will show that for an appropriate choice of $d(i,j)$, there exists a $\beta < 1$ so that

$$E|I^{(k+1)}| \le \beta E|I^{(k)}|, \; k=0,1,2,... \quad (6.1)$$

From (6.1) and A2 it follows that

$$E|\Pi| = E\sum_{k=0}^{\infty} |I^{(k)}| \le |I^{(0)}| \sum_{k=0}^{\infty} \beta^k = O(N) \quad (6.2)$$

To show (6.1), choose $d(i,j)$ small enough so that $D < e^h$. Let q be an upper bound on the probability that an event starts a rollback tree. Since by Theorem 1 the expected size of a tree started by event $e_{starter}$, $|\text{tree}(e_{starter})|$, is bounded from the above, say by a constant $C < +\infty$ which is independent of k, N and $e_{starter}$, we have

$$E|I^{(k+1)}| = E\sum_{e \in I^{(k)}} |\text{tree}(e)| \, \mathbf{1}(e \text{ roots a tree}) \quad (6.3)$$

Since the size of any particular tree is independent of the size of $I^{(k)}$, Wald's Lemma applied to (6.3) yields

$$E|I^{(k+1)}| = E|I^{(k)}| \, E|\text{tree}(e)| \, P(e \text{ roots a tree}) \quad (6.4)$$

From the proof of Theorem 1 it follows that C decreases when the $d(i,j)$ decrease. Since q depends on all $d(i,j)$ and is zero if all $d(i,j)$ are smaller than the respective $t(i,j)$, choosing $d(i,j)$ small will ensure that $\beta \stackrel{def}{=} qC < 1$ and (6.1) follows.

Now since A7 is assured, the expected number of processed events $|\Pi|$, counting each time the same event is processed as a separate event, and including all events wrongly introduced by the rollback optimism, is $O(N)$. Therefore we can complete Step 4 of the proof.

The analysis above also implies that the total number of events in Π is finite with probability one, i.e. in almost all realizations of the stochastic system, Π is finite. Moreover, the number of events processed is $O(N)$ with high probability. The actual system being simulated may not be stochastic in any sense, and furthermore the simulation may not be stochastic. There is a connection, however, between random processes and deterministic processes that is often exploited in information theory (e.g. to show existence of efficient codes) that enables us to apply these results to deterministic simulations. If the event times are regarded as a realization of a stochastic simulation, or as samples of such a simulation, we can conclude that, in this deterministic system, Π is still finite and is even $O(N)$.

7. Final remarks.

Theorem 2. Under assumptions B1, B3 and $\sum_k k^2 p(k) < \infty$:
(I) if $b < e^h$ then P(tree is finite) = 1 and E(size of the tree) $< \infty$;
(II) if $b > e^h$ then P(tree is finite) < 1 and E(size of the tree) = ∞.

Theorem 2 substantially improves Theorem 1 by establishing a condition for the finiteness of the tree which is tight. Its lengthier proof is omitted.

We have only proved sufficiency of the tree boundedness but not its necessity for the rollback simulation efficiency. This is so because the tree model establishes only the upper bound for the possible damage due to rollback. Specifically, if several rollback trees coexist in the simulation, the cumulative damage can be smaller than the sum of the damages due to each tree. However, in our analysis we use the latter as representing this damage. Also, a single rollback tree can fork children to itself which may also decrease the damage as compared with its upper bound from our model.

These damage alleviating phenomena depend on the topology of the simulated network and their effects as a function of the topology require further investigations.

Also note that the analysis outlined above applies only to our specific algorithm, filtered rollback. However, we believe it sheds some light to the efficiency or inefficiency of the general rollback simulation paradigm. In particular, the lower bound in Theorem 2, part (II), strongly suggests that if $b > e^h$ then unbounded cascading may develop.

Appendix 1: An echo in a rollback algorithm. It is easy to generate an echo by "maliciously" manipulating processing rates for different types of simulated activities *during simulation*. An attractive and realistic feature of the example below is that processing rates for the simulated activities are assigned *in advance* so that the echo is generated without outside intervention, just by virtue of the pattern of node interactions. The end result is that after achieving simulated time t_s the lengths of simulated time intervals which must be rolled back and the number of wrongly processed events are $\Omega(t_s)$, so that as the real time t_r advances, the processing rate (which determines how many inherent events are processed per unit of physical time) decreases as $O(1/\sqrt{t_r})$. Long cascading is not possible, since the number of nodes is fixed at 3.

First, we describe the *simulated system*. It consists of three nodes A, B, and C which exchange messages:
node A may send messages to B and C,
node B may send messages to A and C.
after receiving a message from B, node A processes the message, then sends a reply message to B.
similarly, after receiving a message from A, node B processes the message, then sends a reply message to A.

Another activity A and B can be engaged in is sending messages to C. Node A starts preparing a message to C when A has no other work to do. The message is sent as soon as it is ready, and then if A again has nothing to do, preparation of another message for C is started. Similarly B prepares and sends messages to C if B has otherwise nothing to do.

For both A and B, their mutual message exchange has a higher priority than sending messages to C. Thus, if A receives a message from B while preparing a message for C, the preparation is aborted and processing of the message from B is started. Similarly B aborts a preparation of a message for C and starts processing a message from A if such a message is received.

Sending/receiving messages takes no time. But processing the message between A and B takes time u and preparation of messages for C takes time $2u$, $u > 0$.

Suppose that initially, at time $t = 0$, A receives the first message from B. Then for $t > 0$ the only message traffic will be between A and B: A sends a message to B at $t = u$, then B sends a message to A at time $t = 2u$, then A sends a message to A at time $t = 3u$, and so on. The idle periods of length u for both A and B would not be long enough to complete the preparation of a message to C; each time such a preparation is started it is later aborted and no message to C is sent.

Our final assumption about simulated system is that C has very little work to do, and sends no messages.

Now we describe the work of the *simulator*. The simulation is hosted by three processors, $h(A)$, $h(B)$, and $h(C)$, each of which performs a simulation of the corresponding node of the simulated system. Suppose that the simulation of processing one message and sending one reply by $h(A)$ to $h(B)$ or by $h(B)$ to $h(A)$ takes one time unit of physical time, and similarly preparing and sending a message to $h(C)$ by either $h(A)$ or $h(B)$ takes one time unit.

Also one time unit is required for preparation and sending an antimessage during a rollback. Since C has little work to do, $h(C)$ increases its simulated time very fast, so the simulated time of C is always very large ($+\infty$).

According to the rollback strategy the events will be processed as follows:

1. At physical time 1, $h(A)$ completes processing the message which was received by A from B at simulated time 0 and sends a message to $h(B)$ with time stamp u. Meanwhile, since $h(B)$ received no time stamp during interval of physical time (0,1), it has completed preparation and sending a message by B to C. Thus, at physical time 1, the simulated time of A is u, the simulated time of B is $2u$, $h(A)$ has correctly sent a message to $h(B)$, and $h(B)$ has incorrectly sent a message to $h(C)$.
2. Receiving time stamp u with the message from $h(A)$, $h(B)$ starts rolling back its simulated time from $2u$ to u after physical time 1. One antimessage is to be sent from $h(B)$ to $h(C)$, so the rollback would be completed by physical time 2. Meanwhile, since $h(A)$ received no time stamp during interval of physical time (1,2), by time 2, $h(A)$ has simulated preparation and sending a message from A to C.
3. At physical time 2 the rollback by $h(B)$ is completed and $h(B)$ starts simulation of processing the message which B received from A at simulated time u. This processing is completed with sending a message to $h(A)$ at physical time 3. The message has time stamp $2u$. Meanwhile, since $h(A)$ still received no time stamp neither at physical time 2 nor during the following interval (2,3), by time 3, $h(A)$ has simulated preparation and sending a second message from A to C. Thus, at physical time 3, the simulated time of B is $2u$, the simulated time of A is $5u$, $h(B)$ has correctly sent a second message to $h(A)$, and $h(A)$ has incorrectly sent two messages to $h(C)$.
4. Following the same pattern, at physical time 6, the simulated time of B is $8u$, the simulated time of A is $3u$, $h(A)$ has correctly sent a message to $h(B)$, and $h(B)$ has incorrectly sent three messages to $h(C)$. The next similar physical time would be 10=6+4, then 15=10+5, and so on.

By induction, at physical time $n(n+1)/2 = 1+2+....+n$, the floor of the simulation is nu which is also the simulated time of A if n is odd, or of B if n is even. The other of the two nodes is incorrectly advanced to simulated time $(3n-1)u$ and has incorrectly sent n messages to C.

Thus, the simulated time advancement during one unit of the physical time advancement degrades in inverse proportion to the square root of the physical time, as the rollback amplitude rises in direct proportion to the square root of the physical time.

Appendix 2: Proof of Theorem 1. We have a standard (Galton-Watson) branching process [1] defined by a distribution on the nonnegative integers ($\{p(i)\}$, $i = 0, 1, ...$) and a sequence of i.i.d. random variables $w(i,j)$ with $P(w(i,j) = n) = p(n)$.

The process realization can be thought of as a random tree with root labeled (0,1) which spawns $w(0,1) = k(1)$ children labeled $(1,1),(1,2),...,(1,k(1))$; each child $(1,j)$ spawns $w(1,j)$ grandchildren. The $k(2) = w(1,1) + \cdots + w(1,k(1))$ grandchildren are labeled $(2,1),(2,2),...(2,k(2))$ and so on. The number of descendents of the nth generation, $k(n)$, is defined recursively: $k(0) = 1$; $k(n+1) = \sum_{j=1}^{k(n)} w(n, j)$.

B2 yields $p(i) = 0$ for $i > D$, therefore $E(w(i,j)) = \sum_{i=0}^{D} ip(i) \stackrel{def}{=} b < +\infty$.

We need the genealogy of each descendent. We label the descendents at the nth generation

$$(n, 1), (n, 2), \cdots, (n, k(n)).$$

The ancestry of descendent (n, j) is given by the following sequence

$(0, 1)$; $(1, L(1, n, j))$; $(2, L(2, n, j))$; ...; (n, j),

where $L(i,n,j)$ is the label of the direct ancestor of node (n,j) in the ith generation.

Once our branching process which generates random trees with nodes (n,j) has been constructed, we define two new random processes on it, $x(n,j)$ and $\tilde{x}(n,j)$. We are given a probability distribution $dF(t)$ with $\int_{-\infty}^{\infty} tdF(t) < 0$, $\int_{0}^{\infty} dF(t) > 0$, and $\int_{-\infty}^{\infty} e^{\theta t} dF(t) < \infty$ for some $\theta > 0$. We assume that the i.i.d. sequence of random variables W_{ij} introduced in Sec. 6 is distributed according to $F(t)$. It follows from Sec. 6 that W_{ij} is assigned to descendent (i, j). Now, given a positive number x_0, define

1) $\tilde{x}(0, 1) = x_0$
2) $\tilde{x}(n, m) = \tilde{x}(n-1, L(n-1, n, m)) + W_{nm}$.

That is,

$$\tilde{x}(n, m) = x_0 + \sum_{i=1}^{n} W_{i, L(i, n, m)}$$

3)

$$x(n, m) = \begin{cases} \tilde{x}(n, m) & \text{if } \tilde{x}(i, L(i, n, m)) \text{ are} \\ & \text{positive for all } 0 \le i \le n \\ 0 & \text{otherwise .} \end{cases}$$

We also define $z(n)$ = the number of $x(n, j)$ $1 \le j \le k(n)$ which are positive.

We further define a process $x^*(n,i)$ in the same way as $\tilde{x}(n,i)$, except that the carrier branching process is deterministic with branching factor D. It is always possible to construct $x^*(n,i)$ so that

$$x^*(n,i) = \tilde{x}(n,i), \qquad 1 \le i \le k.$$

This yields

$$\sum_{i=1}^{k(n)} \mathbf{1}(x(n,i) > 0) \le \sum_{i=1}^{D^n} \mathbf{1}(x^*(n,i) > 0).$$

Lemma. Given any $\varepsilon > 0$ there exists an $n_0 > 0$ such that if $n > n_0$ then $P(x^*(n,i) > 0) \le e^{-n(h-\varepsilon)}$.

Proof is a direct application of Chernoff inequality [2].

Proof of Theorem 1.

$$\begin{aligned} & E(\sum_{n=1}^{\infty} z(n)) \\ & = E(\sum_{n=1}^{\infty} \sum_{i=1}^{k(n)} \mathbf{1}(x(n,i) > 0)) \\ & \le \sum_{n=1}^{\infty} E(\sum_{i=1}^{D^n} \mathbf{1}(x^*(n,i) > 0)) \\ & \le \sum_{n=1}^{\infty} D^n P(x^*(n,i) > 0) \\ & \le \sum_{n=1}^{n_0} D^n + \sum_{n=n_0+1}^{\infty} D^n e^{-n(h-\varepsilon)} < \infty \end{aligned}$$

where ε is small enough so that $De^{-h+\varepsilon} < 1$ and n_0 is given by the lemma.

References

[1] Asmussen, S. and Hering, H., *Branching Processes*, Birkhauser, 1983.

[2] Chernoff, H., A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Statist.*, **23** (1952), 494-507.

[3] Gafni, A., Berry, O., Jefferson, D. Optimized virtual synchronization, *Proc. 2nd Int. Workshop on Applied Math. and Performance/Reliability Models*, Univ. of Rome II (1987), 229-244.

[4] Jefferson, D.R., Virtual time. *ACM Transactions on Programming Languages and Systems*, **7**, 3 (1985), 404-425.

[5] Lubachevsky, B.D. Bounded lag distributed discrete event simulation (Extended Abstract), in *Distributed Simulation*, B.Unger, and D.Jefferson (eds.), SCS, *Simulation Series*, **19**, 3 (1988), 183-192.

[6] Lubachevsky, B.D. Efficient distributed event driven simulations of multiple-loop networks, *Communications of the ACM*, **32**, 1 (1989), pp.111-131.

[7] Lubachevsky, B.D. Scalability of the bounded lag distributed discrete event simulation, in *Distributed Simulation*, B.Unger, and R.Fujimoto (eds.), SCS, *Simulation Series*, **21**, 2 (1989), 100-107.

[8] Sokol, L.M., Briscoe, D.P, and Wieland, A.P. MTW: a strategy for scheduling discrete simulation events for concurrent execution, in *Distributed Simulation*, B.Unger, and D.Jefferson (eds.), Simulation Series, SCS, **19**, 3 (1988) 34-42.