To appear in "Journal of Parallel and Distributed Computing", Vol. 18, No. 4, 1993

Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation

Friedemann Mattern

Department of Computer Science, University of Saarland,
Im Stadtwald 36, D 6600 Saarbrücken, Germany

mattern@cs.uni-sb.de

Abstract. This paper presents snapshot algorithms for determining a consistent global state of a distributed system without significantly affecting the underlying computation. These algorithms do not require channels to be FIFO or messages to be acknowledged. Only a small amount of storage is needed. An important application of a snapshot algorithm is Global Virtual Time determination for distributed simulations. The paper proposes new and efficient Global Virtual Time approximation schemes based on snapshot algorithms and distributed termination detection principles.

1 Introduction

A *snapshot* of a distributed system is a *global state* (consisting of the local states of the processes and all the messages in transit) which is meaningful in the sense that it corresponds to a possible global state where the local states of all processes and of all communication channels are recorded simultaneously [5]. In order to get such a *causally consistent* state in a system without a common clock, the local state recording events must be coordinated: If the receipt of a message is recorded, the sending of such message (which usually takes place at another process) must also be recorded. More generally, all causal predecessors of a recorded event must also be recorded. Fortunately, this is possible without freezing the whole system.

Snapshots and snapshot algorithms are fundamental paradigms of distributed computing. Important applications are algorithms for the detection of stable properties such as termination of a distributed computation [18] or deadlock of a distributed system [7]. More generally, snapshot algorithms can be used to compute monotonic functions of the global state such as lower bounds on the simulation time (the so-called *Global Virtual Time* or GVT) to which a distributed simulation system has advanced [13]. Other applications are checkpointing and recovery of distributed data bases [14, 24] and monitoring and debugging of distributed systems.

A snapshot algorithm for systems with FIFO channels was first given by Chandy and Lamport in 1985 [5]. The main idea is that immediately after recording the local state, a process sends control messages along each of its (outgoing) channels. Whenever a process receives a control message for the first time, it takes a local snapshot if it has not already done so "spontaneously." Causal consistency is guaranteed due to the FIFO property of the channels because any message of the underlying application (so-called *basic messages*) sent after the control messages must arrive *after* the local snapshot of the receiver. Messages in transit can easily be recorded because control messages flush the channels.

For non-FIFO systems Taylor showed that a snapshot algorithm is either *inhibitory* (i.e., possibly delaying actions of the underlying application to occur, in particular delaying the sending of basic messages while waiting for a control message from another process), or it relies upon *piggybacking* control information onto the basic messages [9,26]. An inhibitory snapshot algorithm for non-FIFO systems was proposed in [12] by Hélary: After taking a local snapshot, a process does not send basic messages to a neighboring process P until it knows that P has also taken a local snapshot. This is realized by sending control messages to all neighbors after recording the local state and suspending the sending of basic messages along a channel until a control message is received on that channel. Non-inhibitory snapshot algorithms for non-FIFO systems relying on piggybacking a one bit status information onto basic messages were proposed by Lai and Yang in [15]. A new variant based on this principle which uses less space will be described in Section 3 and Section 4.

2 Model and Definitions

Asynchronous distributed systems and distributed computations are modeled as follows. An asynchronous distributed system consists of *processes* which communicate solely via *messages* sent through *channels*. Processes and channels form a strongly connected finite graph. Messages are assumed to be delivered correctly, with arbitrary but finite delay, but not necessarily in the order being sent. There is no common clock or common memory, and the relative speed of processes is undetermined.

A process consists of a set of *states*. Atomic actions which may change the state of a single process are modeled by *events*. They are classified into three types: send events, receive events, and internal events. All events may change the state of the process at which they occur; send events and receive events, however, do change the state of a channel (which is defined to be the set of messages sent but not yet received) by inserting or removing a message. A *local computation* of a process is a sequence of events affecting only the local state and possibly the state of its incident channels. The potential causal dependencies between events of local computations of different processes is modeled by the *happens before* relation (denoted $\rightarrow$) introduced by Lamport in [16]. It is the smallest transitive relation satisfying the following conditions: (1) if e and e' are events of the same local computation and e' is the next event after e, then $e \rightarrow e'$; (2) if e is a send event and e' is the corresponding receive event (i.e., the message sent by e is received by e'), then $e \rightarrow e'$.

A *distributed computation* consists of local computations, one per process, together with a set of corresponding send-receive events such that the set of all its events E is left-closed with respect to $\rightarrow$ (i.e., each message received was sent "earlier"). Furthermore, we require $\rightarrow$ to be a *partial order* (i.e., there are no cyclic dependencies between events). Informally, a distributed computation can be represented by a *time diagram* (Fig. 1). Horizontal lines are time axes of processes, points denote events and arrows represent messages. If $e \rightarrow e'$, then one can follow a "path of causality" from event e to event e' by moving in the direction of the arrows and from left to right on the process lines. Because $\rightarrow$ is a partial order, it is always possible to draw event e to the left of event e' if $e \rightarrow e'$.

If a process sends messages to all other processes in order to initiate local actions, these messages will usually be received at different time instants. Due to unpredictable message delays, it is not possible to guarantee that all local actions triggered by messages are executed simultaneously. However, each

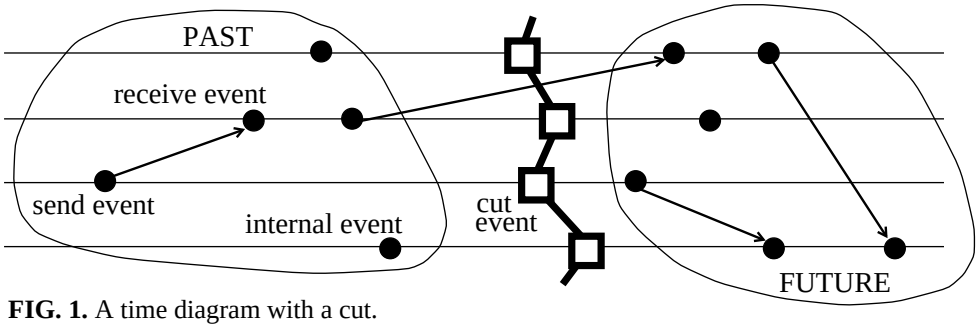

FIG. 1. A time diagram with a cut.

process line is cut into two parts by the receive event of the initiating message. This motivates the notion of "cuts." Graphically, a *cut* can be represented by a zigzag line cutting a time diagram into two parts—a left part called PAST (those events that happen before the cut) and a right part called FUTURE (those events that happen after the cut). The pseudo events where this *cut line* intersects the process lines will be called *cut events*. (Cut events are invisible to the underlying computation, and they do not change the local state of the processes with respect to the underlying computation.) A cut is *consistent* if no message arrow starts in FUTURE and ends in PAST. This notion of consistency fits the observation that a message cannot be received before it is sent [14]. Fig. 2 shows a consistent cut C and an inconsistent cut C'.

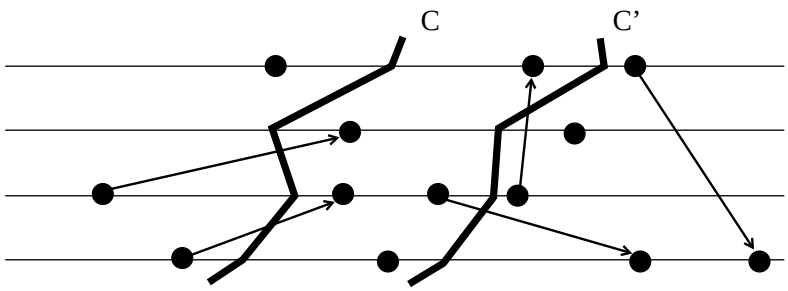

FIG. 2. Consistent (C) and inconsistent (C') cuts.

Formally, a consistent cut of a distributed computation is defined to be a finite subset $C \subseteq E$ (where E is the set of all events of the computation) such that $(e \in C \wedge e' \rightarrow e) \Rightarrow e' \in C$. Notice that a (finite) distributed computation is a "maximal" cut of itself. This motivates the definition of a partial order on the set of all cuts of a distributed computation: A cut C' is said to be *later than* a cut C if $C \subseteq C'$. Informally, this means that in a time diagram the cut line of C' is to the right of the cut line of C. (In Fig. 2 C' is later than C.) It can easily be shown that with operations $\cap$ and $\cup$ the set of all consistent (and all inconsistent) cuts of a distributed computation forms a *lattice* [20].

For a consistent cut the events in a time diagram can be arranged in such a way that the cut line can be drawn as a straight vertical line and all message arrows still go "forward" in time. (This is not possible for inconsistent cuts.) Since for a consistent cut there is a possible execution of the computation in which all cut events occur simultaneously, it makes sense to define the *global state* of a consistent cut (a so-called *causally consistent global state*). It consists of the state

of the channels (those messages that cross the cut line, i.e., which are sent in PAST and received in FUTURE) and the set of local states, one per process, recorded at the moment the corresponding cut event occurs (notice that cut events do not change the state). In the sense of the more formal set theoretic characterization of a consistent cut C given above, the *state of a process* is the state after its last local event in C (or the initial state if such an event does not exist).

Of particular interest for distributed computations are *monotonic functions* of the global state and stable predicates. Let s(C) denote the global state of a consistent cut C and let f be a function from the set of all global states to some partially ordered set. Function f is *monotonic* if C' *later than* C implies $f(s(C)) \leq f(s(C'))$. Typical examples of monotonic functions are the total number of messages sent in a distributed computation or the simulation time to which a distributed simulation system has advanced. *Stable predicates* can be characterized by monotonic boolean indicator functions (where *false* < *true*). That is, a predicate ϕ is *stable* if C' *later than* C implies $\phi(s(C)) \Rightarrow \phi(s(C'))$. Since in distributed systems cut lines (and hence cuts) are adequate substitutes for "points in global time," this means that a stable predicate which becomes true at a point in a distributed computation is true at all later points. For a more detailed discussion of consistent global states and stable predicates the reader is referred to [5].

3 A Consistent Cut Algorithm

A *snapshot algorithm* computes a causally consistent global state of a distributed system. (Alternatively, it may directly compute the value of a monotonic function or a stable predicate for consistent global states.) Such an algorithm is *superimposed* on the underlying computation, that is, it runs concurrently with the underlying computation but should not affect it in a significant way. An important application of a snapshot algorithm is the *detection* of a stable predicate such as distributed termination or distributed deadlock. This can be achieved by repeatedly executing a (somewhat specialized) snapshot algorithm until the predicate holds. This works because for a stable predicate ϕ it is guaranteed that (1) the predicate is still true after the snapshot algorithm has established the truth of ϕ, and (2) if ϕ is true "now," a snapshot algorithm started now or later will also establish the truth of ϕ. Analogous properties (i.e., predicates of the form $f(s) \geq x$) hold for monotonic functions.

In general, the cut defined by the local state recording events of a snapshot algorithm must be consistent in order to be meaningful. (A noteworthy exception is GVT approximation or distributed termination detection discussed in later sections). Therefore, the *determination of a consistent cut* is central to any general snapshot algorithm. In [15] Lai and Yang presented a simple scheme to compute a consistent cut for non-FIFO systems by piggybacking a one bit status information onto basic messages:

(1) Every process is initially white and turns red while taking a local snapshot.
(2) Every message sent by a white (red) process is colored white (red).
(3) Every process takes a local snapshot at its convenience—but before a red message is possibly received.

Obviously, the cut defined by the white events is consistent because no (red) message sent after the cut is received (by a white process) before the cut. In order to guarantee termination, it must be assured

that eventually all processes take a local snapshot, and for most applications the local snapshots must be collected and transmitted to a dedicated process.

In the sequel we assume that a single process initiates the snapshot algorithm. The initiating process becomes red spontaneously and then starts a virtual broadcast scheme by directly or indirectly sending (red) *control messages* to all processes in order to ensure that eventually all processes become red. Virtual broadcast schemes can be implemented in various ways, for example by superimposing a control computation on the underlying basic computation which uses a ring, a spanning tree, or a flooding scheme [27].

Note that a white process can receive a red basic message before receiving a control message (see P_1 in Fig. 3). Because processes do not know whether and when they will receive red basic messages, a white process must be able to take a local snapshot at the moment it receives a red basic message. This local snapshot must reflect the local state *before* the receipt of the message. In practice, this should not be a problem. If it is not possible to "peek" at the message contents before actually receiving it in order to determine its color, it might be possible to take a local snapshot just after receiving the message and before changing the local state. Otherwise a white process must save relevant parts of the local state before receiving a message in order to reproduce the state before the receipt event of a red message.

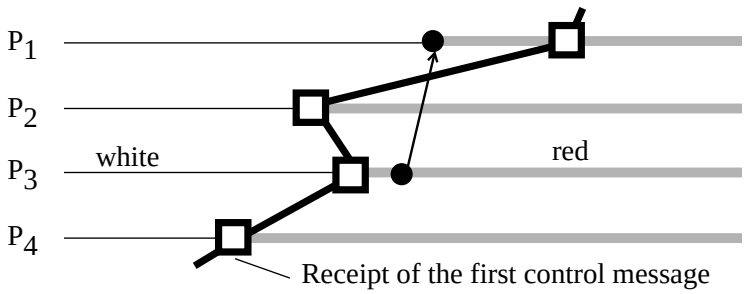

FIG. 3. The receipt of a red message by a white process.

4 Catching the Messages in Transit

For some stable predicates or monotonic functions of the global state (such as GVT approximation, see Section 6 and Section 7), the messages in transit must be taken into consideration. To catch these messages, Lai and Yang proposed that a process keeps a record of all messages sent and all messages received along its incident channels [15]. After the local snapshots have been "assembled," the messages in transit can be determined for each channel by computing the set theoretic difference. A serious drawback of this method, however, is that complete message histories must be stored and sent to other processes which might require a large amount of space.

We propose a different method to catch the messages in transit which does not suffer from this drawback. Obviously, the messages in transit are precisely the white messages which are received by red processes. Therefore, whenever a red process gets a white message it can send a copy of it to the snapshot initiator. (This message may be sent directly to the initiator or routed on a superimposed control

topology.) After the snapshot initiator has received the last copy of all in-transit messages (and the local snapshots of all processes) it knows the complete snapshot. Fig. 4 illustrates this principle.

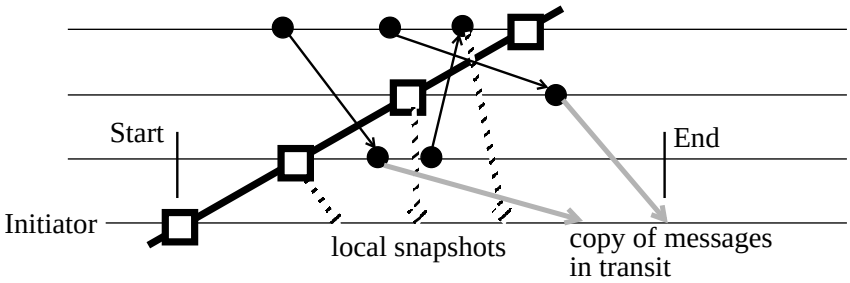

FIG. 4. The snapshot principle.

A problem with the method described so far, however, is termination detection. The initiator gets copies of all messages in transit but it cannot determine when it has received the last one. In principle, the problem can be solved by any *distributed termination detection algorithm* for non-FIFO systems [18] where only white messages are considered. (For that purpose, a process is considered to be *active* iff it is white, and *passive* otherwise. Then, the "white computation" has terminated if no process is white and no white messages—which include the control messages sent towards the initiator—are in transit. Note, however, that since passive processes are never reactivated this is a special case of the general distributed termination detection problem.) A *deficiency counting* termination detection method is particularly attractive in this case. In this method, each process is equipped with a *counter* being part of the process state which counts the number of basic messages the process has sent to any process minus the number of basic messages it has received from any process. By collecting and accumulating these counters together with the local snapshots, the initiating process gets a consistent view of the message counters. It hence knows how many basic messages have been in transit for the cut (i.e., it knows how many copies it will get) and can thus determine the end of the snapshot algorithm.

Because after termination of the snapshot algorithm all processes are red and no white messages are in transit, it is easy to repeatedly execute the snapshot algorithm. For that, one introduces a third color such that the three colors (denoted "0", "1", and "2") are used cyclically. A process whose color is "(i+1) mod 3" can then receive messages with the same color "(i+1) mod 3", or messages with color "i mod 3" (these are messages sent before the previous cut of which copies have to be forwarded to the initiator) or messages with color "(i+2) mod 3" (these are messages sent after the next cut). In the latter case, the process takes a local snapshot before actually receiving it and switches its color to "(i+2) mod 3". Note that the message counters are "color-blind"—they count sent and received messages of any color.

If it is not appropriate that processes inform the initiator every time an in-transit message is received (because, for example, the message contents can be processed locally or because some messages are not relevant to the snapshot) another termination detection method should be considered. By using the *vector counter principle* [18, 19] a scheme that guarantees termination after two control rounds can be devised. A *control round* started by the initiator can be realized by any virtual broadcast scheme (e.g., a so-called wave algorithm) which guarantees that every process is visited by a control message and

which collects distributed information stored by the processes and returns it to the initiator [27]. For ease of illustration we assume here that the processes send control messages along a ring.

In the vector counter method every process P_i counts the number of white messages it has sent to process P_j ($i \neq j$) on the j-th component of a local vector V_i of length n (where n is the number of processes). It decrements its own component ($V_i[i] := V_i[i]-1$) every time it receives a white message from some process. A control message with a control vector C circulating on the ring accumulates the local vectors and resets them to zero ($C := C+V_i$; $V_i := 0$; all operations are performed component-wise). During the first round, it also colors processes red if they are still white and collects the local snapshots. After completion of the first round C[i] indicates the number of white messages that are in transit to P_i for the cut induced by the control round. (This is obvious since with respect to the white messages the cut is consistent.) If at the end of the first round $C[i] > 0$ for some i, a second round is necessary. In the second round the control message waits at each process P_i until all (white) in-transit messages have been received ($V_i[i]+C[i] \leq 0$; notice that after the first round no new white messages have been generated). During the second round the relevant information contained in the in-transit messages is collected.

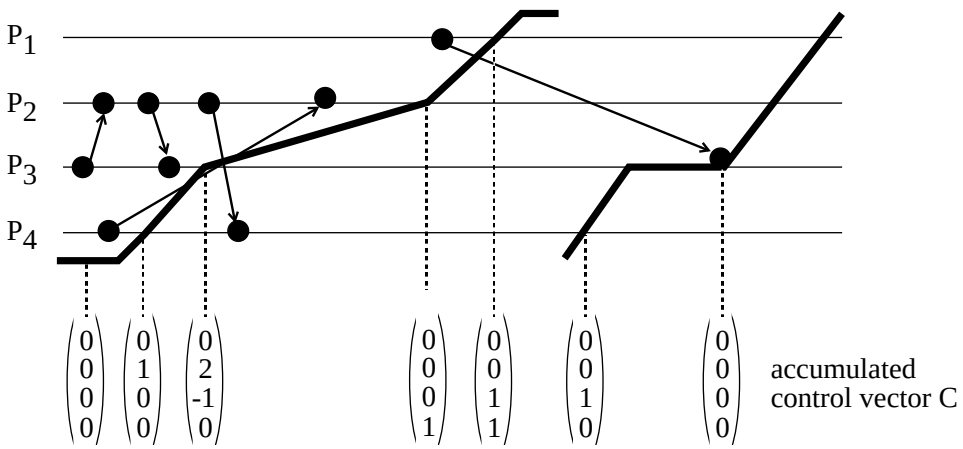

FIG. 5. The vector counter method.

Fig. 5 illustrates the method. Various optimizations and variants are possible. For example, it is only necessary to revisit those processes P_i for which $C[i] > 0$ holds after the first round. Vectors can be coded in such a way that only non-zero components are stored and transmitted. It is also possible to use a scalar counter s_i (where $s_i = \sum_j V_i[j]$) instead of a vector counter. This, however, is merely a variant of the deficiency counting method described earlier and might require more than two rounds. If the snapshot algorithm is repeatedly executed, two local vectors should be used—one counting the white messages, the other one counting the red messages. By swapping the meaning of the two colors, the algorithm can then be executed again. An adaptation of this algorithm for GVT approximation will be presented in Section 7.

5 DMC — a Distributed Monotonic Computation Scheme

In general, the global state determined by a snapshot algorithm (such as the algorithm presented in the previous section) is out of date and no longer valid. For monotonic functions of the global state, however, even an outdated snapshot is often of interest since it yields a lower bound on the "current" value of the function.

In this section we consider a particular distributed computation scheme we call DMC (Distributed Monotonic Computation) which computes a monotonic function of the global state. In the next section we will see that this scheme has many interesting applications, for example it is used in distributed simulations to represent simulation time. Specialized snapshot algorithms, so-called GVT algorithms, can be used to compute approximations of the monotonic function; such algorithms will be presented in Section 7 and Section 8. The notions in this section are due to Tel [27, Chapter 4].

Let X be a partially ordered set where every finite non-empty subset has an infimum. Each process P_i maintains a variable x_i whose domain is X, and all basic messages are "stamped" with a value of X. A send event stamps the message with a value greater than or equal to the current value of x_i. An internal event may only increase x_i. A receive event may also change x_i; the new value of x_i, however, must be greater than or equal to the value of x_i before the receive event *or* it must be greater than or equal to the X-stamp of the message received. (In [27] Tel generalizes this to the requirement that the new value be at least the *infimum* of the current value and the value of the X-stamp.) Fig. 6 gives an example of a computation which behaves according to those rules.

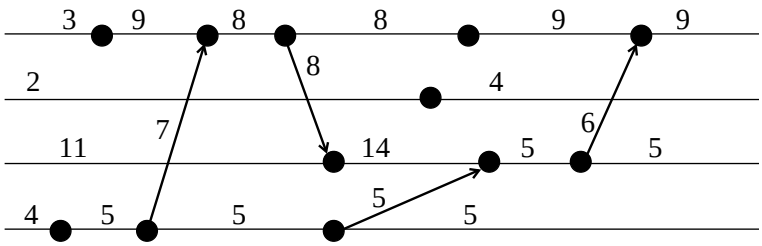

FIG. 6. A distributed monotonic computation.

It is often interesting to know how "far" the computation has proceeded. This is formalized by the definition of a *global infimum function* f on the set of all cuts which computes the "current" infimum. For a cut C we define f(C) as the infimum of all local values x_i (taken at the moment of the local cut event) and all X-stamps of messages which cross the cut line, i.e., which are in transit. (Notice that in Section 2 we did only define the global state of a *consistent* cut; it will shortly become clear, however, that messages from FUTURE to PAST do no harm.)

In order to prove that the global infimum function f is monotonic, we consider a cut C and another cut C' which is later than C (see Fig. 7). To simplify the proof we concentrate on the messages in transit (i.e., we assume that an event sends a *virtual message* to its locally next event stamped with the current value of x_i). Without loss of generality we can assume that a special

internal initialization event is in the past of every cut. Therefore, for each message that crosses cut C' from the PAST of C' to the FUTURE of C', it is possible to construct a finite *backward chain* of directly related events that ends in the PAST of C and therefore also crosses C. Let "value of an event" denote the value of the local variable x_i immediately after the event. Send events and internal events (different from initialization events) always have directly preceding events with a value which is smaller than or equal to their own value; for receive events it is always possible to choose one of the two directly preceding events (i.e., the corresponding send event or the previous local event) in such a way that this holds. By induction on the length of such a monotonic backward chain, it easily follows that the X-stamps of its messages which cross C' (messages a and d, but also message c in Fig. 7) are greater than or equal to the X-stamps of the messages on the chain that cross C (message b in Fig. 7). Since such a construction is possible for *all* messages that cross C', this applies to the infimum, and hence $f(C) \leq f(C')$. The reader may prove Tel's generalization of the scheme in a similar way (see also [27]).

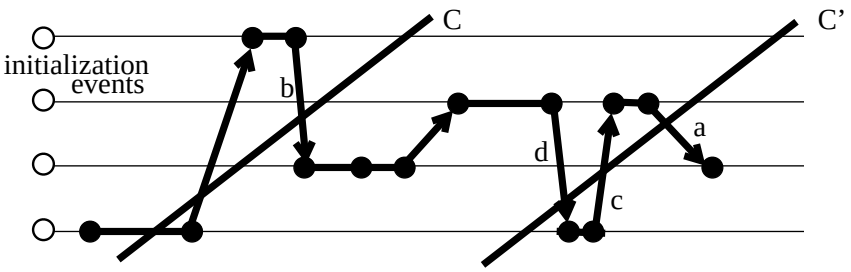

FIG. 7. Construction of a monotonic backward chain.

Fig. 7 also illustrates that it does not matter whether a message that crosses an inconsistent cut from FUTURE to PAST (message c) is taken into consideration: For each of those messages there exists a message with an equal or smaller X-stamp (message d in Fig. 7) which goes from PAST to FUTURE, therefore the X-stamp of message c is not relevant to the infimum.

As is the case for general snapshot computation, it is generally not possible to compute the "current" value of a monotonic function f. However, it is possible to approximate that value (i.e., to determine a lower bound) by calculating f for a "recent" cut C. Interestingly, for global infimum functions of DMC schemes this can be done *without coordinating the local cut events.* Fig. 8 explains why this is the case. For an inconsistent (or consistent) cut C we can define its *consistent prefix closure* $C^* = C \cup \{e' \in E \mid \exists\, e \in C : e' \rightarrow e\}$. Graphically this means that we adjust the cut line by pushing it somewhat to the right in order to include those events which happened before in the FUTURE. By construction, C* is later than C and hence $f(C^*) \geq f(C)$. (Notice that f(C) is well-defined for any global state function f if the messages coming from FUTURE are ignored.) However, C* is still "earlier" than a virtual consistent cut C' depicted by a vertical cut line through the rightmost cut event of cut C in the diagram. Therefore, $f(C) \leq f(C^*) \leq f(C')$ which signifies that f(C) is a lower bound on the value of function f at the moment f(C) is determined. For DMC schemes, this works even if C is inconsistent[1] and even if we do not care about messages coming from the FUTURE. We will make use of this result in Section 7 where we develop algorithms for GVT approximation.

1. This generalizes the notions of *strongly* and *weakly stable properties* introduced by Lai and Yang in [15].

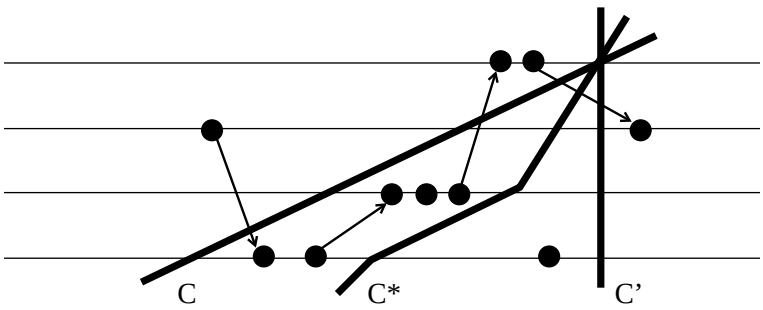

FIG. 8. Construction of the consistent prefix closure of a cut.

There exist several examples of DMC schemes for which good lower bounds on the current value of the global infimum function are important. Among the most important applications are *distributed simulation schemes* (with the problem of GVT approximation) and *message-driven distributed computations* (with the problem of distributed termination detection). Other examples comprise approximations of logical time defined by Lamport clocks [16] or vector clocks [10, 20].

6 Distributed Simulation

A particular interesting application of DMC schemes can be found in the theory of distributed simulation. Here, the global infimum function is called *Global Virtual Time* (GVT) and algorithms to compute lower bounds are of great practical importance.

A distributed discrete simulation system consists of a set of sequential event-driven simulators implemented by autonomous processes which interact by so-called event messages. (See Fujimoto's excellent survey [11] on parallel and distributed simulation which also contains further references.) Each simulator has its own simulation clock and simulates its part of the model independently from the other simulators. However, a simulator can schedule an event for execution by another simulator at a simulation time later than or equal to its own local simulation time. This remote event scheduling feature is implemented by sending an *event message* to the other simulator which contains among other things a timestamp that determines the simulation time at which the event is to occur.

A fundamental requirement to guarantee the correctness of a simulation is that events be executed *chronologically*, i.e., in the order of the simulation times. In general, however, event messages need not arrive at a simulator ordered by simulation time. Notice that this can also be the case if message transmission is FIFO because event messages need not be *generated* with monotonically increasing timestamps and because usually event messages are received from different sources along different channels. A central problem of distributed simulation theory is to guarantee by some decentralized control mechanism that each simulator executes its events in chronological order.

In *conservative* distributed simulation schemes [22] a simulator is only allowed to advance its simulation clock to the simulation time of the earliest event in its local event scheduling list (in order to execute that event) if it can be assured that no event message with a timestamp in the past will ever arrive. In *optimistic* distributed simulation schemes (such as Time Warp [13]) a simulator may directly execute the earliest event (if any) in its event scheduling list. If an event message with an earlier timestamp

subsequently arrives, the simulator *rolls back* to an earlier simulation time, cancelling all intermediate side effects, and re-executes from that point now including the event that arrived late. The rollback operation therefore requires that the state of each simulator process be saved regularly.

It should be clear that the handling of simulation time conforms to the distributed monotonic computation scheme defined in Section 5. For optimistic schemes a simulation clock may be set back when an event message is received, but it is never set to a value earlier than the timestamp value of the message. (For technical reasons it is necessary to go back to the most recent local recovery point before that instant in time and replay the computation; this is a transparent mechanism, however.) In [13] Jefferson defines Global Virtual Time (GVT) at real time t as the minimum of all local clocks at real time t and of the timestamps of all event messages that are in transit at real time t. Clearly, GVT is a global infimum function in the sense defined in Section 5. The determination of a tight lower bound on the current GVT value is important for conservative as well as for optimistic distributed simulation systems.

For conservative schemes (which often suffer from deadlock situations [6]), GVT approximation allows "accelerating" the simulation time by directly advancing all local clocks to the calculated lower bound. (In this case, however, GVT should be defined to consist of the minimum of the message timestamps only.) After that, some simulators may be able to give new model specific *guarantees* (i.e., lower bounds on the timestamps of event messages they will generate) to other simulators, and consequently several simulators may be able to proceed by executing events in parallel. In any case, when a conservative distributed simulation is globally deadlocked, GVT approximation yields the simulation time of the next event which can be safely executed [6].

For optimistic distributed simulation schemes, GVT approximation is important for memory management and output commitment: GVT serves as a floor of the simulation times to which any simulator can ever roll back. Thus, for each simulator there must exist at least one checkpoint that is older than GVT. Only the most recent of all checkpoints older than GVT must be kept, however. All other checkpoints are obsolete and should be removed to save memory. In case of interactive and animated simulation it is necessary to present to the outside world a consistent and chronological view of what is going on inside the simulation system. Since physical output cannot be undone, animation events can only be released when GVT exceeds their timestamp values.

7 GVT Approximation

Several algorithms for GVT approximation (or *distributed infimum approximation* as it is called by Tel [27]) are known. In principle, existing general distributed snapshot algorithms can be used—it is a straightforward exercise to apply those algorithms [5, 15] or the variants described in Section 4 for GVT approximation. However, specialized and therefore potentially more efficient solutions are preferable. In [27] Tel gave several wave based solutions for different communication models. For the asynchronous model two solutions are suggested. The first solution relies on sending an acknowledgment for every received message and maintaining a data structure to represent the timestamp of unacknowledged messages. (Note that a message may still be in transit if the message is received after the cut but the acknowledgment is received *before* the cut, Fig. 9 illustrates this situation which is handled correctly in [27].) Basically the same algorithm was presented by Samadi et al. [23]. The second solu-

tion keeps a list of received messages and a list of sent messages for each process. After having collected and accumulated all local lists, the messages in transit are determined by taking the appropriate difference. (Notice that for an inconsistent cut it is possible that the receipt of a message is reported, but not its sending.) This is the same principle used by Lai and Yang in their general snapshot algorithm [15] and it suffers from the same drawback that control messages are rather long and that a large amount of space is used for keeping the message lists.

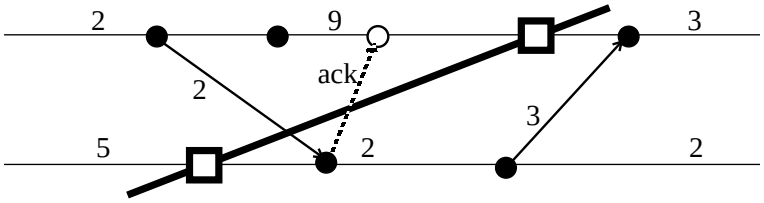

FIG. 9. Acknowledgment for in-transit message.

In [17] Lin and Lazowska proposed an algorithm which is similar to Samadi's algorithm but which does not use an acknowledgment message for every single message. The idea is that every message carries a sequence number and when a process P gets a control message it sends to every neighboring process Q the smallest sequence number which is missing from that process (thereby implicitly acknowledging all messages with a smaller sequence number). Q assumes that this message and all messages with larger sequence numbers are still in transit. For the messages in transit, Q is able to compute a lower bound on the timestamps because, for unacknowledged messages, it keeps the local minima (and the corresponding sequence numbers) of the timestamps as a function of sequence numbers. The algorithm works in asynchronous systems without explicit acknowledgments, but it has two drawbacks: First, the approximated GVT value is not optimal, and second, space is needed for the data structures which keep the missing sequence numbers and the local minima of sent timestamp values. Other solutions to the GVT approximation problem were given by Baldwin et al. [2], by Bauer et al. [3], and by Bellenot [4].

We propose a different and rather simple solution whose main idea is easily explained using Fig. 10. We determine two cuts C and C' such that C' is later than C. (This can be realized, e.g., by two control waves or by two rounds of a control message on a ring.) Our goal is to compute a GVT approximation along cut C'. For that, we have to determine the minimum of all local clock values (which is easy) and the minimum of the timestamps of all messages that cross cut C'. As has been shown in Section 5, messages sent after C' that are received before C' can be ignored. The remaining set of messages that cross C' can be divided into a set of messages that are sent between C and C' and a set of messages that are sent before C. However, by "pushing" C' to the right, the algorithm guarantees that the second set is empty, i.e., the algorithm makes sure that a message sent before C is received before C'. For the first set a lower bound on the smallest timestamp can easily be determined.

Let M denote the set of all messages that cross C' from PAST to FUTURE (conceptually, we can again include the virtual messages introduced in Section 5, therefore $M \neq \emptyset$), and let M' denote the set of all messages sent between C and C'. Because $M \subseteq M'$ (the algorithm descibed below guarantees that no message crosses both cuts), $\min\{\text{timestamps of } M'\} \leq \min\{\text{timestamps of } M\}$. Hence, if we determine

at C’ the minimum of all timestamps of messages sent after C, we get a *lower bound* on the timestamps of messages that are in transit at C’. To get a good approximation, C and C’ should be "close together."

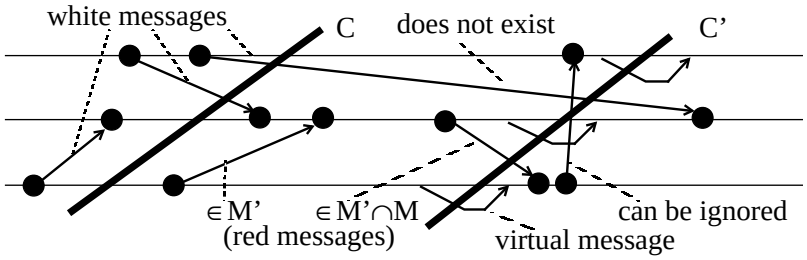

FIG. 10. The GVT approximation principle.

The algorithm assumes that processes and messages are colored in a similar way as described in Section 3 and Section 4. Initially, every process is white. A process is colored red to the right of cut C. Every process counts the number of white messages it sends and it receives. After cut C (i.e., after becoming red) every process remembers the minimum of the timestamp values of all (red) messages it sends. Since C is consistent with respect to white messages, it is possible to determine the number of white messages that are in transit for this cut. If *no* messages are in transit, the minimum of the local variables computed along C is already a valid GVT approximation (i.e., a lower bound). Otherwise, a second control round yielding cut C’ is initiated. Here, the global minimum of all local variables and of all timestamps of sent red messages can be determined.

By collecting and accumulating the counters of received white messages, it can be checked whether all white messages have been received before cut C’. If this is not the case, the (possibly invalid) result is ignored and another control round is started—this can be repeated until eventually the counters signal that the last white message was received. If vector counters are used (see Section 4) at most one round C’ after the first control round C is needed, since it is possible to wait for slow white messages. For a ring topology, this solution will now be presented in a more formal way.

Processes are denoted by $P_1,...P_n$; the index of the unique initiator is denoted by `init`. Each process P_i has the following local variables:

`T` : the local simulationclock
t_{min}: the minimal timestamp of red messages
`V`: vector counter for white messages
`color`: the color of the process (white or red), initialized to white
`GVT_approx`: the result of the algorithm (only for the initiator)

Each basic message has a header with two fields to represent the color of the message and the timestamp. (The timestamp is equal to or greater than the value of the local simulation clock at the moment of sending the message.) When *sending* a message, the following actions take place:

```
send <color, timestamp,...> to Pj;
if color = white
   then V[j] := V[j]+1;
   else tmin := min(tmin, timestamp);
fi;
```

That is, white messages are counted and for red messages the minimal timestamp value is determined. When a message <msg_col,timestamp,...> is *received* by a process P_i, the following actions take place:

```
if msg_col = white
     then V[i] := V[i]-1;
fi;
Process the message and update T such
that T >= min(T', timestamp) where T'
is the value of T before the receipt of
the message. After that, check whether
a control message is waiting (see be-
low) and can now be propagated.
```

A control message has three fields, it accumulates the local minimum of the local clocks on m_clock, the minimum of the timestamps on m_send, and it accumulates the vector counters on count. When a control message <m_clock, m_send, count> is received by a process P_i which is not the initiator, the following self-explanatory actions take place:

```
if color = white then
   tmin := ∞;
   color := red;
fi;
wait until V[i]+count[i] ≤ 0;
send <min(m_clock,T),min(m_send,tmin),
      V+count> to P(i mod n)+1;
V:=0;
```

When a control message <m_clock, m_send, count> is received by the initiator P_{init}, the following actions take place:

```
wait until V[init]+count[init] ≤ 0;
if count = 0
   then GVT_approx := min(m_clock,m_send);
   else send <T,min(m_send,tmin),V+count>
             to P(init mod n)+1; V:=0;
fi;
```

When P_{init} gets back the control message after a complete first round, it is already red. It then starts a second round if necessary (i.e., if count≠0). Notice that m_send and count must be accumulated

over both rounds, whereas `m_clock` is calculated individually for each round. When the initiator gets back the control message after the second round, `count` is guaranteed to be the zero vector and the GVT approximation is found.

Finally we note that the initiator P_{init} starts the algorithm by setting its local variable `color` to `red` and t_{min} to ∞, and then executing

send <T,∞,V> **to** $P_{(init \bmod n)+1}$; V:=0;

Instead of vector counters, simple *scalar counters* can be used—this only requires a few obvious adjustments. (In particular, since by reducing the vector to a scalar, pertinent information is lost, it is not possible to wait at specific processes for messages which are known to arrive. Therefore, the wait statement must be removed.) Because in that case it is not guaranteed that `count` is 0 after the second round, further rounds might be necessary. They are started in the same way as the second round. The algorithm is also easily changed to allow repeated executions and other control topologies than rings; the details, however, are left to the reader.

8 A Parallel Snapshot and GVT Approximation Algorithm

Although the snapshot algorithm of Section 4 and the GVT algorithm of Section 7 were presented in a way which is particularly well suited for implementation on a ring, they can also be realized on other control topologies, e.g. spanning trees. In this section we develop an "inherently parallel" snapshot and GVT approximation algorithm which does not need a specific control topology. Another advantage compared to the previous solutions is that it does not use vector counters although only a single control "round" is necessary. The basic techniques (coloring of processes and messages; waiting for messages that are in transit for the first cut) are the same as those used in the previous solutions.

Interestingly, the algorithm can be derived by adapting Chandy's and Lamport's elegant snapshot algorithm [5] to non-FIFO systems. We therefore first present the outline of a slight variant of their FIFO-based algorithm:

When process P receives a marker control message along an input channel c:

if P has not recorded its state
  then P records its state;
       P sends one marker along each of its incident output channels;
       P records the state c as the empty sequence;
  else P records the state c as the sequence of messages received along c
       after P's state was recorded and before P received a marker along c;
fi;

If the underlying network is strongly connected then after finite time, once the algorithm has been initiated, each process will have recorded its state and the states of all incoming channels. A process spontaneously starts the algorithm as if it received a marker from some virtual, non-existing channel. The reader may easily check that the global state is consistent; for the details we refer to [5].

If communication is non-FIFO, two cases must be considered. (1) A message sent by process P to process Q *after* the marker can be received by Q *before* the marker; (2) a message sent by P to Q *before* the marker can be received by Q *after* the marker. Fig. 11 illustrates the two cases.

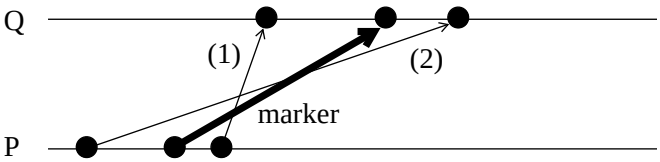

FIG. 11. Non-FIFO communication.

We already know how to deal with case (1): A process becomes red at the moment it records its local state, and the color of a process is piggybacked onto every basic message. Therefore, when process Q receives message (1), it knows that a marker would have arrived if communication were FIFO. Upon receiving a red message, a white process becomes red and does all the actions (i.e., state recording and marker propagation) it would do upon receiving a marker. The subsequent arrival of the marker is simply ignored. Notice that for GVT approximation those messages do no harm and therefore no premature actions have to be taken (i.e., it is correct to record the state and to propagate the marker when the first marker is received as in the FIFO case).

Case (2) essentially is the problem of knowing when the last white message along a channel arrives. For FIFO systems this is easy because markers *flush* the channels; when the marker has arrived there are no more white messages in transit for that channel. For non-FIFO systems the same can be achieved by an extension of the flushing principle that makes markers "slower" than basic messages: A marker is augmented with the number of (white) messages that are sent before it along the channel. The acceptance of the marker can then be deferred until all messages sent before it have been received. Fig. 12 illustrates the idea. To realize it, a process must count for each channel the number of white messages received and the number of white messages sent. The two solutions for cases (1) and (2) can be combined by piggybacking the number of white messages sent along a channel onto every red message sent along that channel. Notice that for non-FIFO systems the state of a channel is the *set* of all *white* messages received by a red process.

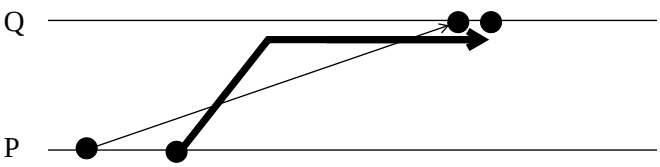

FIG. 12. Deferment of acceptance of a marker.

In [1], Ahuja presented a similar solution. In order to extend the applicability of the Chandy-Lamport algorithm (and other algorithms) to non-FIFO systems, the author proposes *flush primitives* for non-FIFO channels. All messages sent along a channel before sending a *forward-flush marker* are received before the marker (this solves case 2 of Fig. 11), and all messages sent along a channel after sending a

backward-flush marker are received after the marker (this solves case 1). The author observes that sending a combined forward-backward marker (instead of an ordinary marker) is a sufficient condition for the correct working of the Chandy-Lamport algorithm for non-FIFO systems. However, as our discussion shows, explicit use of flush primitives is not necessary for an efficient solution of the non-FIFO snapshot problem.

In order to form a global state, the local snapshots (including the channel states) must be collected. For a single initiator and a system with bidirectional communication channels this can be done in an elegant way by the *echo algorithm* [8, 25]: Upon receipt of the first marker or the first red message a process sends markers to all neighbors but the one from which the marker or the message was received. A process having no other channel waits until all white messages have been received and then returns an *echo* message that contains the local snapshot. (Echoes behave like markers, they also contain the number of white messages sent before them.) When a process has received markers *or* echoes along every incident channel and when it has received all white messages, it returns an echo with all accumulated snapshots to the process from which it first received a marker or a red message. Eventually the "echo wave" reaches the initiator. It is easy to see that the channels travelled by the echoes form a *spanning tree* of the network.

For GVT approximation one can take the cut defined by those instants when the echo message is sent. (For the initiator this is the instant when it *would* send an echo.) By construction, there is no white message in transit for this cut, therefore it qualifies for cut C' of Fig. 10. A process becomes red when it receives the first marker. The instants when the processes become red form cut C of Fig. 10. The global minimum of the local clocks and of the red messages sent before C' is accumulated by the echo wave. We shall not discuss the technical details, but it should become clear that the resulting algorithm is a *parallel variant* of the algorithm presented in Section 7. Instead of control rounds on rings it uses a *wave* of control messages which consists of two phases. The first phase (which is realized by markers) corresponds to the first round, and the second phase (which is realized by echoes) corresponds to the second round. Exactly 2e control messages are used (where e is the number of channels).

9 Conclusions

We presented new distributed algorithms for computing snapshots of distributed computations and for approximating the Global Virtual Time of a distributed simulation system. The algorithms are simple and efficient and do not require channels to be FIFO or messages to be acknowledged. The basic idea is to use two colors which indicate whether a process has already taken its local snapshot and whether a message was sent before or after the local snapshot of a process. Thus, messages which would make a snapshot inconsistent can easily be recognized and avoided, and messages which are in transit can be caught by the receiving process. The problem of knowing when the snapshot is complete (i.e., all in-transit messages have been caught) is solved by a distributed termination detection scheme. While in the paper this was illustrated using the vector counter termination detection method, using any other termination

detection scheme (e.g., based on simple scaler counters instead of vector counters) will also work.

GVT approximation is done by applying a somewhat specialized variant of this principle. In particular, messages which cross a cut in the "wrong" direction can simply be ignored. The main idea of GVT approximation is to use two cuts and to make sure that no messages cross both cuts. Hence, the minimum of the timestamps of all messages which cross the second cut can easily be determined by considering all messages which are sent between the two cuts.

Cuts can be realized in various ways. We first presented a simple implementation using a token circulating on a ring, and then sketched how the echo algorithm with its two phases can be used to realize the two cuts by traversing an arbitrarily connected network in parallel. An additional advantage of this variant is the fact that only a single "round" of the echo algorithm is necessary although the control messages are rather short. Since distributed termination detection is a particular case of GVT approximation, this scheme does also yield an elegant parallel termination detection algorithm for distributed computations. Interestingly, it is also possible to derive GVT approximation algorithms from termination detection algorithms [21] which shows that the two problems are closely related instances of the general snapshot problem.

References

1. Ahuja, M. Flush Primitives for Asynchronous Distributed Systems. *Information Processing Letters* 34 (1990), 5-12.

2. Baldwin, R., Chung, M.J., and Chung, Y. Overlapping Window Algorithm for Computing GVT in Time Warp. In *Proc. 11th International Conference on Distributed Computing Systems*, 1991, pp. 534-541.

3. Bauer, H., Sporrer, C., and Krodel, T.H. On Distributed Logic Simulation Using Time Warp. In *Proc. VLSI International Conference*, IFIP, Edinburgh, 1991.

4. Bellenot, S. Global Virtual Time Algorithms. In *Proc. of the SCS Multiconference on Distributed Simulation*, 1990, pp. 122-127.

5. Chandy, K., and Lamport, L. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems* 3, 1 (1985), 63-75.

6. Chandy, K., and Misra, J. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. *Comm. of the ACM* 24, 2 (1981), 198-205.

7. Chandy, K., Misra, J, and Haas, L. Distributed Deadlock Detection. *ACM Transactions on Computer Systems* 1, 2 (1983), 144-156.

8. Chang, E. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Transactions on Software Engineering* SE-8, 4 (1982), 391-401.

9. Critchlow, C., and Taylor, K. The Inhibition Spectrum and the Achievement of Causal Consistency. *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990, pp. 31-42.

10. Fidge J. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. *Proc. 11th Australian Computer Science Conference*, 1988, pp. 56-66.

11. Fujimoto, R.M. Parallel Discrete Event Simulation. *Comm. of the ACM* 33, 10 (1990), 30-53.

12. Hélary, J. Observing Global States of Asynchronous Distributed Applications. In Bermond, J-C, and Raynal, M. (Eds.). *Proc. of the 3rd International Workshop on Distributed Algorithms.* Springer-Verlag, LNCS 392, 1989, pp. 45-56.

13. Jefferson, D. Virtual Time. *ACM Trans. Program. Lang. Syst.* 7, 3 (1985), 404-425.

14. Koo, R., and Toueg, S. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering* SE-13, 1 (1987), 23-31.

15. Lai, T., and Yang, T. On Distributed Snapshots. *Information Processing Letters* 25 (1987), 153-158.

16. Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM* 21, 7 (1978), 558-565.

17. Lin, Y. B., and Lazowska, D. Determining the Global Virtual Time in a Distributed Simulation. *Proc. of the International Conference on Parallel Processing,* 1990, pp. 201-209.

18. Mattern, F. Algorithms for Distributed Termination Detection. *Distributed Computing* 2 (1987), 161-175.

19. Mattern, F. Experience with a New Distributed Termination Detection Algorithm. In: Van Leeuwen, J. (Ed.). *Proc. of the 2nd International Workshop on Distributed Algorithms.* Springer-Verlag, LNCS 312, 1988, pp. 127-143.

20. Mattern, F. Virtual Time and Global States of Distributed Systems. In Cosnard M. et al. (Eds). *Proc. Workshop on Parallel and Distributed Algorithms* (Chateau de Bonas, France, Oct. 1988), Elsevier, 1989, pp. 215-226.

21. Mattern, F., Mehl, H., Schoone, A., Tel, G. Global Virtual Time Approximation with Distributed Termination Detection Algorithms. Tech. Rep. RUU-CS-91-32, Department of Computer Science, University of Utrecht, The Netherlands, 1991.

22. Misra, J. Distributed Discrete-Event Simulation. *Computing Surveys* 18, 1 (1986), 39-65.

23. Samadi, B., Muntz, R.R., and Parker, D.S. A Distributed Algorithm to Detect a Global State of a Distributed Simulation System. In *Proc. IFIP Conference on Distributed Processing,* Amsterdam, North-Holland, 1987.

24. Sarin, S., and Lynch, N. Discarding Obsolete Information in a Replicated Database System. *IEEE Transactions on Software Engineering* SE-13, 1 (1987), 39-47.

25. Segal, A. Distributed Network Protocols. *IEEE Transactions on Information Theory* IT-29, 1 (1983), 23-35.

26. Taylor, K. The Role of Inhibition in Asynchronous Consistent-Cut Protocols. In Bermond, J-C, and Raynal, M. (Eds.). *Proc. of the 3rd International Workshop on Distributed Algorithms.* Springer-Verlag, LNCS 392, 1989, pp. 280-291.

27. Tel, G. *Topics in Distributed Algorithms.* Cambridge University Press, Cambridge, 1991.