Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms

Özalp Babaoğlu **Keith Marzullo**

Technical Report UBLCS-93-1

January 1993

Laboratory for Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)

The University of Bologna Laboratory for Computer Science Research Technical Reports are available via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` in compressed PostScript format. Abstracts are available from the same host in the directory `/pub/TR/ABSTRACTS` in plain text format. All local authors can be reached via e-mail at the address *last-name*`@cs.unibo.it`.

UBLCS Technical Report Series

92-1 *Mapping Parallel Computations onto Distributed Systems in Paralex*, by Ö. Babaoğlu, L. Alvisi, A. Amoroso and R. Davoli, January 1992.

92-2 *Parallel Scientific Computing in Distributed Systems: The Paralex Approach*, by L. Alvisi, A. Amoroso, Ö. Babaoğlu, A. Baronio, R. Davoli and L. A. Giachini, February 1992.

92-3 *Run-time Support for Dynamic Load Balancing and Debugging in Paralex*, by Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, L. A. Giachini, September 1992.

92-4 *Paralex: An Environment for Parallel Programming in Distributed Systems*, by Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, L. A. Giachini, October 1992.

93-1 *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanism*, by Ö. Babaoğlu and K. Marzullo, January 1993.

93-2 *Understanding Non-Blocking Atomic Commitment*, by Ö. Babaoğlu and S. Toueg, January 1993.

Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms

Özalp Babaoğlu[1] **Keith Marzullo**[2]

Technical Report UBLCS-93-1

January 1993

Abstract

Many important problems in distributed computing admit solutions that contain a phase where some global property needs to be detected. This subproblem can be seen as an instance of the Global Predicate Evaluation *(GPE) problem where the objective is to establish the truth of a Boolean expression whose variables may refer to the global system state. Given the uncertainties in asynchronous distributed systems that arise from communication delays and relative speeds of computations, the formulation and solution of GPE reveal most of the subtleties in global reasoning with imperfect information. In this paper, we use GPE as a canonical problem in order to survey concepts and mechanisms that are useful in understanding global states of distributed computations. We illustrate the utility of the developed techniques by examining distributed deadlock detection and distributed debugging as two instances of GPE.*

1. Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, 40127 Bologna, Italy. This author was supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Project 6360 (BROADCAST), Hewlett-Packard of Italy and the Italian Ministry of University, Research and Technology.
2. Department of Computer Science, 4130 Upson Hall, Cornell University, Ithaca, New York 14853 USA. This author was supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, and by grants from IBM and Siemens. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

1 Introduction

A large class of problems in distributed computing can be cast as executing some notification or reaction when the state of the system satisfies a particular condition. Examples of such problems include monitoring and debugging, detection of particular states such as deadlock and termination, and dynamic adaptation of a program's configuration such as for load balancing. Thus, the ability to construct a global state and evaluate a predicate over such a state constitutes the core of solutions to many problems in distributed computing.

The global state of a distributed system is the union of the states of the individual processes. Given that the processes of a distributed system do not share memory but instead communicate solely through the exchange of messages, a process that wishes to construct a global state must infer the remote components of that state through message exchanges. Thus, a fundamental problem in distributed computing is to ensure that a global state constructed in this manner is meaningful.

In asynchronous distributed systems, a global state obtained through remote observations could be obsolete, incomplete, or inconsistent. Informally, a global state is inconsistent if it could never have been constructed by an idealized observer that is external to the system. It should be clear that uncertainties in message delays and in relative speeds at which local computations proceed prevent a process from drawing conclusions about the instantaneous global state of the system to which it belongs. While simply increasing the frequency of communication may be effective in making local views of a global state more current and more complete, it is not sufficient for guaranteeing that the global state is consistent. Ensuring the consistency of a constructed global state requires us to reason about both the order in which messages are observed by a process as well as the information contained in the messages. For a large class of problems, consistency turns out to be an appropriate formalization of the notion that global reasoning with local information is "meaningful".

Another source of difficulty in distributed systems arises when separate processes independently construct global states. The variability in message delays could lead to these separate processes constructing different global states for the same computation. Even though each such global state may be consistent and the processes may be evaluating the same predicate, the different processes may execute conflicting reactions. This "relativistic effect" is inherent to all distributed computations and limits the class of system properties that can be effectively detected.

In this paper, we formalize and expand the above concepts in the context of an abstract problem called *Global Predicate Evaluation* (GPE). The goal of GPE is to determine whether the global state of the system satisfies some predicate Φ. Global predicates are constructed so as to encode system properties of interest in terms of state variables. Examples of distributed system problems where the relevant properties can be encoded as global predicates include deadlock detection, termination detection, token loss detection, unreachable storage (garbage) collection, checkpointing and restarting, debugging, and in general, monitoring and reconfiguration. In this sense, a solution to GPE can be seen as the core of a generic solution for all these problems; what remains to be done is the formulation of the appropriate predicate Φ and the construction of reactions or notifications to be executed when the predicate is satisfied.

We begin by defining a formal model for asynchronous distributed systems and distributed computations. We then examine two different strategies for solving GPE. The first strategy, introduced in Section 5, and refined in Section 13, is based on a monitor process that actively interrogates the rest of the system in order to construct the global state. In Section 6 we give a formal definition for consistency of global states. The alternative strategy, discussed in Section 7, has the monitor passively observe the system in order to construct its global states. Sections 8 – 13 introduce a series of concepts and mechanisms necessary for making the two strategies work efficiently. In Section 14 we identify properties that global predicates must satisfy in order to solve practical problems using GPE. In Section 15 we address the issue of multiple monitors observing the same computation. We illustrate the utility of the underlying concepts and mechanisms by applying them to deadlock detection and to debugging in distributed systems.

2 Asynchronous Distributed Systems

A distributed system is a collection of sequential *processes* $p_1, p_2, \ldots, p_n$ and a network capable of implementing unidirectional communication *channels* between pairs of processes for message exchange. Channels are reliable but may deliver messages out of order. We assume that every process can communicate with every other process, perhaps through intermediary processes. In other words, the communication network is assumed to be strongly connected (but not necessarily completely connected).

In defining the properties of a distributed system, we would like to make the weakest set of assumptions possible. Doing so will enable us to establish upper bounds on the costs of solving problems in distributed systems. More specifically, if there exists a solution to a problem in this weakest model with some cost γ, then there is a solution to the same problem with a cost no greater than γ in *any* distributed system.

The weakest possible model for a distributed system is called an *asynchronous system* and is characterized by the following properties: there exist no bounds on the relative speeds of processes and there exist no bounds on message delays. Asynchronous systems rule out the possibility of processes maintaining synchronized local clocks [18,8] or reasoning based on global real-time. Communication remains the only possible mechanism for synchronization in such systems.

In addition to their theoretical interest as noted above, asynchronous distributed systems may also be realistic models for actual systems. It is often the case that physical components from which we construct distributed systems are *synchronous*. In other words, the relative speeds of *processors* and message delays over network *links* making up a distributed system can be bounded. When, however, layers of software are introduced to multiplex these physical resources to create abstractions such as *processes* and (reliable) communication *channels*, the resulting system may be better characterized as asynchronous.

3 Distributed Computations

Informally, a distributed computation describes the execution of a distributed program by a collection of processes. The activity of each sequential process is modeled as executing a sequence of *events*. An event may be either internal to a process and cause only a local state change, or it may involve communication with another process. Without loss of generality, we assume that communication is accomplished through the events $send(m)$ and $receive(m)$ that match based on the message identifier m. In other words, even if several processes send the same data value to the same process, the messages themselves will be unique.[3] Informally, the event $send(m)$ enqueues message m on an outgoing channel for transmission to the destination process. The event $receive(m)$, on the other hand, corresponds to the act of dequeuing message m from an incoming channel at the destination process. Clearly, for event $receive(m)$ to occur at process p, message m must have arrived at p and p must have declared its willingness to receive a message. Otherwise, either the message is delayed (because the process is not ready) or the process is delayed (because the message has not arrived).

Note that this "message passing" view of communication at the event level may be quite different from those of higher system layers. Remote communication at the programming language level may be accomplished through any number of paradigms including remote procedure calls [4], broadcasts [12], distributed transactions [19], distributed objects [20] or distributed shared memory [21]. At the level we observe distributed computations, however, all such high-level communication boil down to generating matching send and receive events at pairs of processes.

The *local history* of process p_i during the computation is a (possibly infinite) sequence of events $h_i = e_i^1 e_i^2 \ldots$. This labeling of the events of process p_i where e_i^1 is the first event executed, e_i^2 is the second event executed, etc. is called the *canonical enumeration* and corresponds to the total order imposed by the sequential execution on the local events. Let $h_i^k = e_i^1 e_i^2 \ldots e_i^k$ denote an initial prefix of local history h_i containing the first k events. We define h_i^0 to be the empty sequence. The *global history* of the computation is a set $H = h_1 \cup \cdots \cup h_n$ containing all of its events.[4]

Note that a global history does not specify any relative timing between events. In an asynchronous distributed system where no global time frame exists, events of a computation can be ordered only based on the notion of "cause-and-effect". In other words, two events are constrained to occur in a certain order only if the occurrence of the first may affect the outcome of the second. This in turn implies that information flows from the first event to the second. In an asynchronous system, information may flow from one event to another either because the two events are of the same process, and thus may access the same local state, or because the two events are of different processes and they correspond to the exchange of a message. We can formalize these ideas by defining a binary relation $\rightarrow$ defined over events such that [17]:

1. If $e_i^k, e_i^\ell \in h_i$ and $k < \ell$, then $e_i^k \rightarrow e_i^\ell$,
2. If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$,
3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

As defined, this relation effectively captures our intuitive notion of "cause-and-effect" in that $e \rightarrow e'$ if and only if e causally precedes e'.[5] Note that only in the case of matching send-receive events is the cause-and-effect relationship certain. In general, the only conclusion that can be drawn from $e \rightarrow e'$ is that the mere occurrence of e' and its outcome *may* have been influenced by event e.

3. For finite computations, this can be easily accomplished by adding the process index and a sequence number to the data value to construct the message identifier.
4. Sometimes we are interested in local histories as *sets* rather than *sequences* of events. Since all events of a computation have unique labels in the canonical enumeration, h_i as a set contains exactly the same events as h_i as a sequence. We use the same symbol to denote both when the appropriate interpretation is clear from context.
5. While "e may causally affect e'", or, "e' occurs in the causal context of e" [28] are equivalent interpretations of this relation, we prefer not to interpret it as "e happens before e'" [17] because of the real-time connotation.

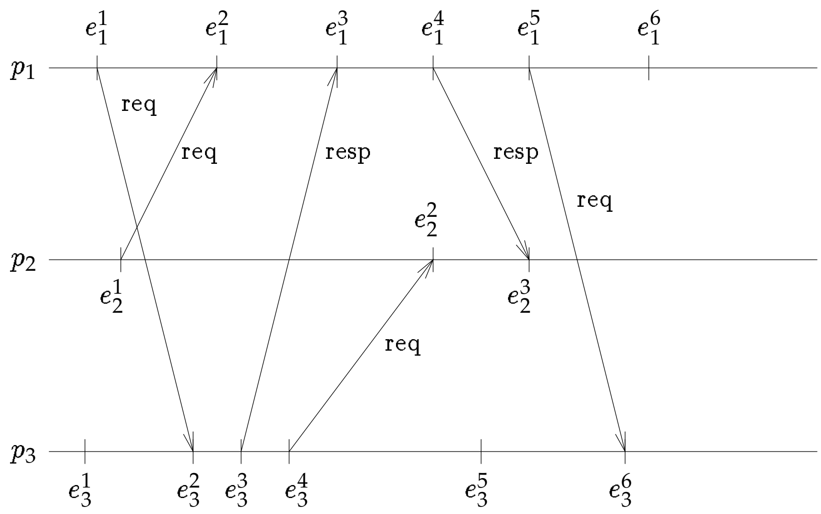

Figure 1. Space-Time Diagram Representation of a Distributed Computation

Certain events of the global history may be causally unrelated. In other words, it is possible that for some e and e', neither $e \rightarrow e'$ nor $e' \rightarrow e$. We call such events *concurrent* and write $e \| e'$.

Formally, a *distributed computation* is a partially ordered set (poset) defined by the pair $(H, \rightarrow)$. Note that all events are labeled with their canonical enumeration, and in the case of communication events, they also contain the unique message identifier. Thus, the total ordering of events for each process as well as the send-receive matchings are implicit in H.

It is common to depict distributed computations using an equivalent graphical representation called a *space-time diagram*. Figure 1 illustrates such a diagram where the horizontal lines represent execution of processes, with time progressing from left to right. An arrow from one process to another represents a message being sent, with the send event at the base of the arrow and the corresponding receive event at the head of the arrow. Internal events have no arrows associated with them. Given this graphical representation, it is easy to verify if two events are causally related: if a path can be traced from one event to the other proceeding left-to-right along the horizontal lines and in the sense of the arrows, then they are related; otherwise they are concurrent. For example, in the figure $e_2^1 \rightarrow e_3^6$ but $e_2^2 \| e_3^6$.

4 Global States, Cuts and Runs

Let σ_i^k denote the local state of process p_i immediately after having executed event e_i^k and let σ_i^0 be its initial state before any events are executed. In general, the local state of a process may include information such as the values of local variables and the sequences of messages sent and received over the various channels incident to the process. The *global state* of a distributed computation is an n-tuple of local states $\Sigma = (\sigma_1, \ldots, \sigma_n)$, one for each process.[6]

6. We can define global states without referring to channel states since they can always be encoded as part of the process local states. We discuss explicit representation of channel states in Section 13.

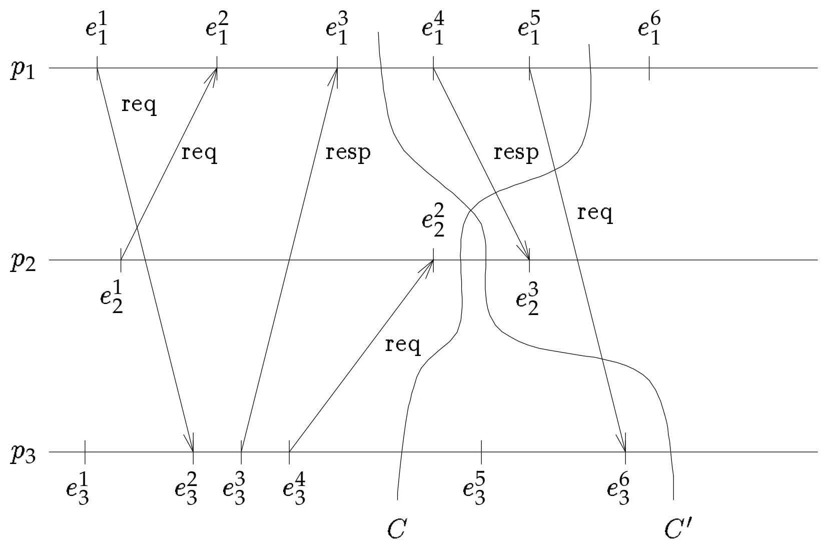

Figure 2. Cuts of a Distributed Computation

A *cut* of a distributed computation is a subset C of its global history H and contains an initial prefix of each of the local histories. We can specify such a cut $C = h_1^{c_1} \cup \cdots \cup h_n^{c_n}$ through the tuple of natural numbers $(c_1, \ldots, c_n)$ corresponding to the index of the last event included for each process. The set of last events $(e_1^{c_1}, \ldots, e_n^{c_n})$ included in cut $(c_1, \ldots, c_n)$ is called the *frontier* of the cut. Clearly, each cut defined by $(c_1, \ldots, c_n)$ has a corresponding global state which is $(\sigma_1^{c_1}, \ldots, \sigma_n^{c_n})$.

As shown in Figure 2, a cut has a natural graphical interpretation as a partitioning of the space-time diagram along the time axis. The figure illustrates two cuts C and C' corresponding to the tuples $(5, 2, 4)$ and $(3, 2, 6)$, respectively.

Even though a distributed computation is a partially ordered set of events, in an actual execution, all events, including those at different processes, occur in some total order.[7] To be able to reason about executions in distributed systems, we introduce the notion of a *run*. A run of a distributed computation is total ordering R that includes all of the events in the global history and that is consistent with each local history. In other words, for each process p_i, the events of p_i appear in R in the same order that they appear in h_i. Note that a run need not correspond to any possible execution and a single distributed computation may have many runs, each corresponding to a different execution.

5 Monitoring Distributed Computations

Given the above notation and terminology, GPE can be stated as evaluating a predicate Φ that is a function of the global state Σ of a distributed system. For the time being, we will assume that a single process called the *monitor* is responsible for evaluating Φ. Let p_0 be

7. If two events actually *do* occur at the same real-time, we can arbitrarily say that the event of the process with the smaller index occurs before the event of the larger-index process.

this process which may be one of $p_1, \ldots, p_n$ or may be external to the computation (but not the system). In this special case, where there is a single monitor, solving GPE reduces to p_0 constructing a global state Σ of the computation (to which Φ is applied). For simplicity of exposition, we assume that events executed on behalf of monitoring are external to the underlying computation and do not alter the canonical enumeration of its events.

In the first strategy we pursue for constructing global states, the monitor p_0 takes on an active role and sends each process a "state enquiry" message. Upon the receipt of such a message, p_i replies with its current local state σ_i. When all n processes have replied, p_0 can construct the global state $(\sigma_1, \ldots, \sigma_n)$. Note that the positions in the process local histories that state enquiry messages are received effectively defines a cut. The global state constructed by p_0 is the one corresponding to this cut.

Given that the monitor process is part of the distributed system and is subject to the same uncertainties as any other process, the simple-minded approach sketched above may lead to predicate values that are not meaningful. To illustrate the problems that can arise, consider a distributed system composed of *servers* providing remote services and *clients* that invoke them. In order to satisfy a request, a server may invoke other services (and thus act as a client). Clients and servers interact through *remote procedure calls*—after issuing a request for service, the client remains blocked until it receives the response from the server. The computation depicted in Figure 1 could correspond to this interaction if we interpret messages labeled *req* as requests for service and those labeled *resp* as responses. Clearly, such a system can deadlock. Thus, it is important to be able to detect when the state of this system includes deadlocked processes.

One possibility for detecting deadlocks in the above system is as follows. Server processes maintain local states containing the names of clients from which they received requests but to which they have not yet responded. The relevant aspects of the global state Σ of this system can be summarized through a *waits-for$^+$ graph* (WFG$^+$) where the nodes correspond to processes and the edges model blocking. In this graph, an edge is drawn from node i to node j if p_j has received a request from p_i to which it has not yet responded. Note that WFG$^+$ can be constructed solely on the basis of local states. It is well known that a cycle in WFG$^+$ is a sufficient condition to characterize deadlock in this system [11]. The nodes of the cycle are exactly those processes involved in the deadlock. Thus, the predicate $\Phi =$ "WFG$^+$ contains a cycle" is one possibility for deadlock detection.[8]

Let us see what might happen if process p_0 monitors the computation of Figure 1 as outlined above. Suppose that the state enquiry messages of p_0 are received by the three application processes at the points corresponding to cut C' of Figure 2. In other words, processes p_1, p_2 and p_3 report local states σ_1^3, σ_2^2 and σ_3^6, respectively. The WFG$^+$ constructed by p_0 for this global state will have edges $(1,3)$, $(2,1)$ and $(3,2)$ forming a cycle. Thus, p_0 will report a deadlock involving all three processes.

An omniscient external observer of the computation in Figure 1, on the other hand, would conclude that at no time is the system in a deadlock state. The condition detected by p_0 above is called a *ghost deadlock* in that it is fictitious. While every cut of a distributed computation corresponds to a global state, only certain cuts correspond to global states that *could* have taken place during a run. Cut C of Figure 2 represents such a global state. On

8. Note that Φ defined as a cycle in WFG$^+$ characterizes a stronger condition than deadlock in the sense that Φ implies deadlock but not vice versa. If, however, processes can receive and record requests while being blocked, then a deadlocked system will eventually satisfy Φ.

the other hand, cut C' constructed by p_0 corresponds to a global state that could never occur since process p_3 is in a state reflecting the receipt of a request from process p_1 that p_1 has no record of having sent. Predicates applied to cuts such as C' can lead to incorrect conclusions about the system state.

We return to solving the GPE problem through active monitoring of distributed computations in Section 13 after understanding why the above approach failed.

6 Consistency

Causal precedence happens to be the appropriate formalism for distinguishing the two classes of cuts exemplified by C and C'. A cut C is *consistent* if for all events e and e'

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C.$$

In other words, a consistent cut is left closed under the causal precedence relation. In its graphical representation, verifying the consistency of a cut becomes easy: if all arrows that intersect the cut have their bases to the left and heads to the right of it, then the cut is consistent; otherwise it is inconsistent. According to this definition, cut C of Figure 2 is consistent while cut C' is inconsistent. A *consistent global state* is one corresponding to a consistent cut. These definitions correspond exactly to the intuition that consistent global states are those that could occur during a run in the sense that they could be constructed by an idealized observer external to the system. We can now explain the ghost deadlock detected by p_0 in the previous section as resulting from the evaluation of Φ in an inconsistent global state.

Consistent cuts (and consistent global states) are fundamental towards understanding asynchronous distributed computing. Just as a scalar time value denotes a particular instant during a sequential computation, the frontier of a consistent cut establishes an "instant" during a distributed computation. Similarly, notions such as "before" and "after" that are defined with respect to a given time in sequential systems have to be interpreted with respect to consistent cuts in distributed system: an event e is *before* (*after*) a cut C if e is to the left (right) of the frontier of C.

Predicate values are meaningful only when evaluated in consistent global states since these characterize exactly the states that could have taken place during an execution. A run R is said to be *consistent* if for all events, $e \rightarrow e'$ implies that e appears before e' in R. In other words, the total order imposed by R on the events is an extension of the partial order defined by causal precedence. It is easy to see that a run $R = e^1 e^2 \ldots$ results in a sequence of global states $\Sigma^0 \Sigma^1 \Sigma^2 \ldots$ where Σ^0 denotes the initial global state $(\sigma_1^0, \ldots, \sigma_n^0)$. If the run is consistent, then the global states in the sequence will all be consistent as well. We will use the term "run" to refer to both the sequence of events and the sequence of resulting global states. Each (consistent) global state Σ^i of the run is obtained from the previous state Σ^{i-1} by some process executing the single event e^i. For two such (consistent) global states of run R, we say that Σ^{i-1} *leads to* Σ^i *in* R. Let $\leadsto_R$ denote the transitive closure of the leads-to relation in a given run R. We say that Σ' is *reachable from* Σ *in run* R if and only if $\Sigma \leadsto_R \Sigma'$. We drop the run subscript if there exists *some* run in which Σ' is reachable from Σ.

The set of all consistent global states of a computation along with the leads-to relation defines a *lattice*. The lattice consists of n orthogonal axes, with one axis for each process. Let $\Sigma^{k_1 \ldots k_n}$ be a shorthand for the global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$ and let $k_1 + \cdots + k_n$ be its *level*. Figure 3

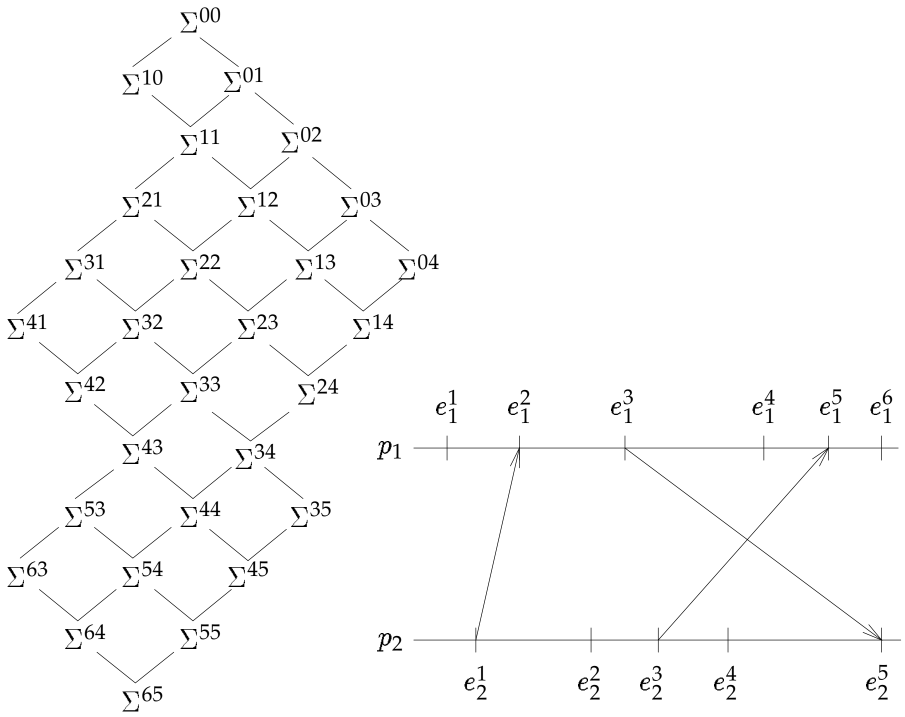

Figure 3. A Distributed Computation and the Lattice of its Global States

illustrates a distributed computation of two processes and the corresponding global state lattice. Note that every global state is reachable from the initial global state Σ^{00}. A path in the lattice is a sequence of global states of increasing level (in the figure, downwards) where the level between any two successive elements differs by one. Each such path corresponds to a consistent run of the computation. The run is said to "pass through" the global states included in the path. For the example illustrated in Figure 3, one possible run may pass through the sequence of global states

$$\Sigma^{00}\ \Sigma^{01}\ \Sigma^{11}\ \Sigma^{21}\ \Sigma^{31}\ \Sigma^{32}\ \Sigma^{42}\ \Sigma^{43}\ \Sigma^{44}\ \Sigma^{54}\ \Sigma^{64}\ \Sigma^{65}.$$

Note that one might be tempted to identify the run corresponding to the *actual* execution of the computation. As we argued earlier, in an asynchronous distributed system, this is impossible to achieve from within the system. Only an omniscient external observer will be able to identify the sequence of global states that the execution passed through.

7 Observing Distributed Computations

Let us consider an alternative strategy for the monitor process p_0 in constructing global states to be used in predicate evaluation based on a *reactive* architecture [13]. In this approach, p_0 will assume a passive role in that it will not send any messages of its own. The application processes, however, will be modified slightly so that whenever they execute an event, they notify p_0 by sending it a message describing the event.[9] As before, we assume that monitoring does not generate any new events in that the send to p_0 for notification coincides with the event it is notifying. In this manner, the monitor process constructs an *observation* of the underlying distributed computation as the sequence of events corresponding to the order in which the notification messages arrive [14].

We note certain properties of observations as constructed above. First, due to the variability of the notification message delays, a single run of a distributed computation may have different observations at different monitors. This is the so-called "relativistic effect" of distributed computing to which we return in Section 15. Second, an observation can correspond to a consistent run, an inconsistent run or no run at all since events from the same process may be observed in an order different from their local history. A *consistent observation* is one that corresponds to a consistent run. To illustrate these points, consider the following (consistent) run of the computation in Figure 1:

$$R = e_3^1\ e_1^1\ e_3^2\ e_2^1\ e_3^3\ e_3^4\ e_2^2\ e_1^2\ e_3^5\ e_1^3\ e_1^4\ e_1^5\ e_3^6\ e_2^3\ e_1^6$$

All of the following are possible observations of R:

$$\begin{aligned}
O_1 &= e_2^1\ e_1^1\ e_3^1\ e_3^2\ e_3^4\ e_1^2\ e_2^2\ e_3^3\ e_1^3\ e_1^4\ e_3^5 \ldots \\
O_2 &= e_1^1\ e_3^1\ e_2^1\ e_3^2\ e_1^2\ e_3^3\ e_3^4\ e_1^3\ e_2^2\ e_3^5\ e_3^6 \ldots \\
O_3 &= e_3^1\ e_2^1\ e_1^1\ e_1^2\ e_3^2\ e_3^3\ e_1^3\ e_3^4\ e_1^4\ e_2^2\ e_1^5 \ldots
\end{aligned}$$

9. In general, the application processes need to inform p_0 only when they execute an event that is relevant to Φ. A local event e_i^k is said to be *relevant* to predicate Φ if the value of Φ evaluated in a global state $(\ldots, \sigma_i^k, \ldots)$ could be different from that evaluated in $(\ldots, \sigma_i^{k-1}, \ldots)$. For example, in the client-server computation of Figure 1, the only events relevant to deadlock detection are the sending/receiving of request and response messages since only these can change the state of the WFG^+.

Given our asynchronous distributed system model where communication channels need not preserve message order, *any* permutation of run R is a possible observation of it. Not all observations, however, need be meaningful with respect to the run that produced them. For example, among those indicated above, observation O_1 does not even correspond to a run since events of process p_3 do not represent an initial prefix of its local history (e_3^4 appears before event e_3^3). Observation O_2, on the hand, corresponds to an inconsistent run. In fact, the global state constructed by p_0 at the end of observation O_2 would be $(\sigma_1^3, \sigma_2^2, \sigma_3^6)$, which is exactly the global state defined by cut C' of Figure 2 resulting in the detection of a ghost deadlock. Finally, O_3 is a consistent observation and leads to the same global state as that of cut C in Figure 2.

It is the possibility of messages being reordered by channels that leads to undesirable observations such as O_1. We can restore order to messages between pairs of processes by defining a *delivery rule* for deciding when received messages are to be presented to the application process. We call the primitive invoked by the application *deliver* to distinguish it from *receive*, which remains hidden within the delivery rule and does not appear in the local history of the process.

Communication from process p_i to p_j is said to satisfy *First-In-First-Out (FIFO) delivery* if for all messages m and m'

FIFO Delivery: $send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$.[10]

In other words, FIFO delivery prevents one message overtaking an earlier message sent by the *same* process. For each source-destination pair, FIFO delivery can be implemented over non-FIFO channels simply by having the source process add a sequence number to its messages and by using a delivery rule at the destination that presents messages in an order corresponding to the sequence numbers. While FIFO delivery is sufficient to guarantee that observations correspond to runs, it is not sufficient to guarantee consistent observations. To pursue this approach for solving the GPE problem where Φ is evaluated in global states constructed from observations, we need to devise a mechanism that ensures their consistency.

We proceed by devising a simple mechanism and refining it as we relax assumptions. Initially, assume that all processes have access to a global real-time clock and that all message delays are bounded by δ. This is clearly not an asynchronous system but will serve as a starting point. Let $RC(e)$ denote the value of the global clock when event e is executed. When a process notifies p_0 of some local event e, it includes $RC(e)$ in the notification message as a *timestamp*. The delivery rule employed by p_0 is the following:

DR1: At time t, deliver all received messages with timestamps up to $t - \delta$ in increasing timestamp order.

To see why an observation O constructed by p_0 using DR1 is guaranteed to be consistent, first note that an event e is observed before event e' if and only if $RC(e) < RC(e')$.[11] This is true because messages are delivered in increasing timestamp order and delivering only messages with timestamps up to time $t - \delta$ ensures that no future message can arrive with a timestamp smaller than any of the messages already delivered. Since the observation coincides with the delivery order, O is consistent if and only if

Clock Condition: $e \rightarrow e' \Rightarrow RC(e) < RC(e')$.

This condition is certainly satisfied when timestamps are generated using the global real-time clock. As it turns out, the clock condition can be satisfied without any assumptions—

10. Subscripts identify the process executing the event.
11. Again, we can break ties due to simultaneous events based on process indexes.

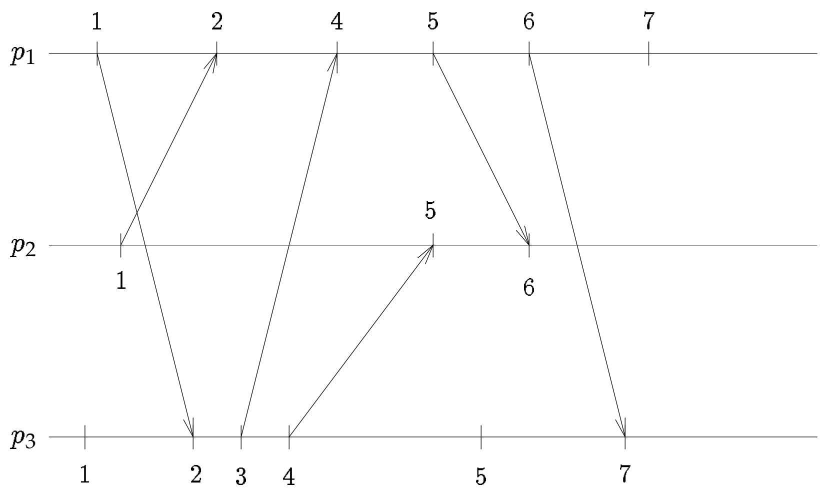

Figure 4. Logical Clocks

in an asynchronous system.

8 Logical Clocks

In an asynchronous system where no global real-time clock can exist, we can devise a simple clock mechanism for "timing" such that event orderings based on increasing clock values are guaranteed to be consistent with causal precedence. In other words, the clock condition can be satisfied in an asynchronous system. For many applications, including the one above, any mechanism satisfying the clock condition can be shown to be sufficient for using the values produced by it as if they were produced by a global real-time clock [27].

The mechanism works as follows. Each process maintains a local variable LC called its *logical clock* that maps events to the positive natural numbers [17]. The value of the logical clock when event e_i is executed by process p_i is denoted $LC(e_i)$. We use LC to refer to the current logical clock value of a process that is implicit from context. Each message m that is sent contains a timestamp $TS(m)$ which is the logical clock value associated with the sending event. Before any events are executed, all processes initialize their logical clocks to zero. The following update rules define how the logical clock is modified by p_i with the occurrence of each new event e_i:

$$LC(e_i) := \begin{cases} LC + 1 & \text{if } e_i \text{ is an internal or send event} \\ \max\{LC, TS(m)\} + 1 & \text{if } e_i = receive(m) \end{cases}$$

In other words, when a receive event is executed, the logical clock is updated to be greater than both the previous local value and the timestamp of the incoming message. Otherwise (i.e., an internal or send event is executed), the logical clock is simply incremented. Figure 4 illustrates the logical clock values that result when these rules are applied to the computation of Figure 1.

Note that the above construction produces logical clock values that are increasing with respect to causal precedence. It is easy to verify that for any two events where $e \rightarrow e'$, the logical clocks associated with them are such that $LC(e) < LC(e')$. Thus, logical clocks satisfy the clock condition of the previous section.[12]

Now let us return to the goal at hand, which is constructing consistent observations in asynchronous systems. In the previous section, we argued that delivery rule DR1 lead to consistent observations as long as timestamps satisfied the clock condition. We have just shown that logical clocks indeed satisfy the clock condition and are realizable in asynchronous systems. Thus, we should be able to use logical clocks to construct consistent observations in asynchronous systems. Uses of logical clocks in many other contexts are discussed in [29].

Consider a delivery rule where those messages that are delivered, are delivered in increasing (logical clock) timestamp order, with ties being broken as usual based on process index. Applying this rule to the example of Figure 4, p_0 would construct the observation

$$e_1^1\, e_2^1\, e_3^1\, e_1^2\, e_3^2\, e_3^3\, e_1^3\, e_3^4\, e_1^4\, e_2^2\, e_3^5\, e_1^5\, e_2^3\, e_1^6\, e_3^6$$

which is indeed consistent. Unfortunately, the delivery rule as stated lacks liveness since, without a bound on message delays (and a real-time clock to measure it), no message will ever be delivered for fear of receiving a later message with a smaller timestamp. This is because logical clocks, when used as a timing mechanism, lack what we call the *gap-detection* property:

Gap-Detection: Given two events e and e' along with their clock values $LC(e)$ and $LC(e')$ where $LC(e) < LC(e')$, determine whether some other event e'' exists such that $LC(e) < LC(e'') < LC(e')$.

It is this property that is needed to guarantee liveness for the delivery rule and can be achieved with logical clocks in an asynchronous system only if we exploit information in addition to the clock values. One possibility is based on using FIFO communication between all processes and p_0. As usual, all messages (including those sent to p_0) carry the logical clock value of the send event as a timestamp. Since each logical clock is monotone increasing and FIFO delivery preserves order among messages sent by a single process, when p_0 receives a message m from process p_i with timestamp $TS(m)$, it is certain that no other message m' can arrive from p_i such that $TS(m') \leq TS(m)$. A message m received by process p is called *stable* if no future messages with timestamps smaller than $TS(m)$ can be received by p. Given FIFO communication between all processes and p_0, stability of message m at p_0 can be guaranteed when p_0 has received at least one message from *all* other processes with a timestamp greater than $TS(m)$. This idea leads to the following delivery rule for constructing consistent observations when logical clocks are used for timestamps:

DR2: Deliver all received messages that are stable at p_0 in increasing timestamp order.[13]

Note that real-time clocks lack the gap-detection property as well. The assumption, however, that message delays are bounded by δ was sufficient to devise a simple stability check in delivery rule DR1: at time t, all received messages with timestamps smaller than $t - \delta$ are guaranteed to be stable.

12. Note that logical clocks would continue to satisfy the clock condition with any arbitrary positive integer (rather than one) as the increment value of the update rules.

13. Even this delivery rule may lack liveness if some processes do not communicate with p_0 after a certain point. Liveness can be obtained by the monitor p_0 requesting an acknowledgement from all processes to a periodic empty message [17]. These acknowledgements serve to "flush out" messages that may have been in the channels.

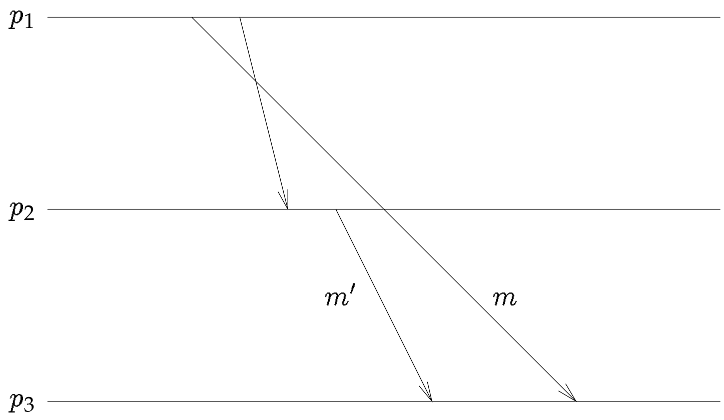

Figure 5. Message Delivery that is FIFO but not Causal

9 Causal Delivery

Recall that FIFO delivery guarantees order to be preserved among messages sent by the *same* process. A more general abstraction extends this ordering to all messages that are causally related, even if they are sent by different processes. The resulting property is called *causal delivery* and can be stated as:

Causal Delivery (CD): $send_i(m) \rightarrow send_j(m') \Rightarrow deliver_k(m) \rightarrow deliver_k(m')$
for all messages m, m', sending processes p_i, p_j and destination process p_k. In other words, in a system respecting causal delivery, a process cannot known about the existence of a message (through intermediate messages) any earlier than the event corresponding to the delivery of that message [31]. Note that having FIFO delivery between all pairs of processes is not sufficient to guarantee causal delivery. Figure 5 illustrates a computation where all deliveries (trivially) satisfy FIFO but those of p_3 violate CD.

The relevance of causal delivery to the construction of consistent observations is obvious: if p_0 uses a delivery rule satisfying CD, then all of its observations will be consistent. The correctness of this result is an immediate consequence of the definition of CD, which coincides with that of a consistent observation. In retrospect, the two delivery rules DR1 and DR2 we developed in the previous sections are instances of CD that work under certain assumptions. What we seek is an implementation for CD that makes no assumptions beyond those of asynchronous systems.

10 Constructing the Causal Precedence Relation

Note that we have stated the gap-detection property in terms of clock values. For implementing causal delivery efficiently, what is really needed is an effective procedure for deciding the following: given events e, e' that are causally related and their clock values, does there exist some other event e'' such that $e \rightarrow e'' \rightarrow e'$ (i.e., e'' falls in the causal "gap" between e and e')?

By delivering event notification messages in strict increasing timestamp order, rules DR1 and DR2 assume that $RC(e) < RC(e')$ (equivalently, $LC(e) < LC(e')$) implies $e \rightarrow e'$. This is a conservative assumption since timestamps generated using real-time or logical clocks only guarantee the clock condition, which is this implication in the opposite sense. Given $RC(e) < RC(e')$ (or $LC(e) < LC(e')$), it may be that e causally precedes e' or that they are concurrent. What is known for certain is that $\neg(e' \rightarrow e)$. Having just received the notification of event e', DR1 and DR2 could unnecessarily delay its delivery even if they could predict the timestamps of all notifications yet to be received. The delay would be unnecessary if there existed future notifications with smaller timestamps, but they all happened to be for events concurrent with e'.

The observations of the preceding two paragraphs suggest a timing mechanism *TC* whereby causal precedence relations between events can be deduced from their timestamps. We strengthen the clock condition by adding an implication in the other sense to obtain:
Strong Clock Condition: $e \rightarrow e' \equiv TC(e) < TC(e')$.

While real-time and logical clocks are consistent with causal precedence, timing mechanism *TC* is said to *characterize* causal precedence since the entire computation can be reconstructed from a single observation containing *TC* as timestamps [9,34]. This is essential not only for efficient implementation of CD, but also for many other applications (e.g., distributed debugging discussed in Section 14.2) that require the entire global state lattice rather than a single path through it.

10.1 Causal Histories

A brute-force approach to satisfying the strong clock condition is to devise a timing mechanism that produces the set of all events that causally precede an event as its "clock" value [34]. We define the *causal history* of event e in distributed computation $(H, \rightarrow)$ as the set

$$\theta(e) = \{e' \in H \mid e' \rightarrow e\} \cup \{e\}.$$

In other words, the causal history of event e is the smallest consistent cut that includes e. The projection of $\theta(e)$ on process p_i is the set $\theta_i(e) = \theta(e) \cap h_i$. Figure 6 graphically illustrates the causal history of event e_1^4 as the darkened segments of process local histories leading towards the event. From the figure, it is easy to see that $\theta(e_1^4) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_2^1, e_3^1, e_3^2, e_3^3\}$.

In principle, maintaining causal histories is simple. Each process p_i initializes local variable θ to be the empty set. If e_i is the receive of message m by process p_i from p_j, then $\theta(e_i)$ is constructed as the union of e_i, the causal history of the previous local event of p_i and the causal history of the corresponding send event at p_j (included in message m as its timestamp). Otherwise (e_i is an internal or send event), $\theta(e_i)$ is the union of e_i and the causal history of the previous local event.

When causal histories are used as clock values, the strong clock condition can be satisfied if we interpret clock comparison as set inclusion. From the definition of causal histories, it follows that

$$e \rightarrow e' \equiv \theta(e) \subset \theta(e').$$

In case $e \neq e'$, the set inclusion above can be replaced by the simple set membership test $e \in \theta(e')$. The unfortunate property of causal histories that renders them impractical is that they grow rapidly.

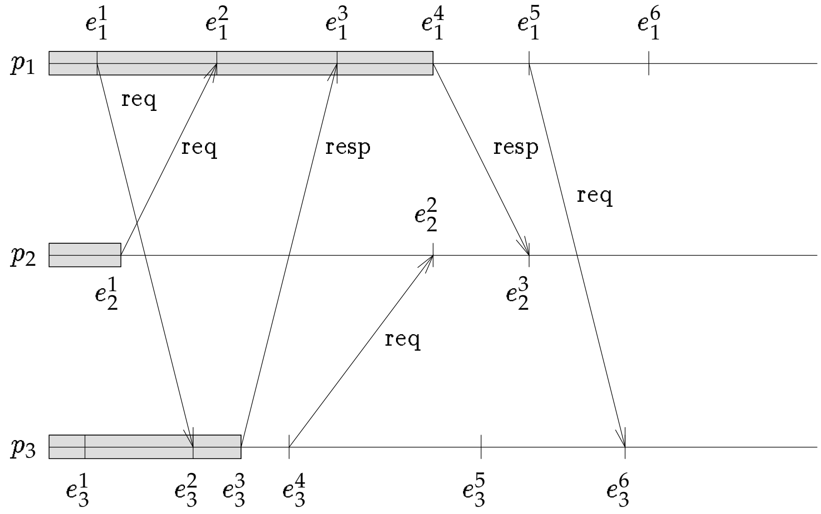

Figure 6. Causal History of Event e_1^4

10.2 Vector Clocks

The causal history mechanism proposed in the previous section can be made practical by periodically pruning segments of history that are known to be common to all events [28]. Alternatively, the causal history can be represented as a fixed-dimensional vector rather than a set. The resulting growth rate will be logarithmic in the number of events rather than linear. In what follows, we pursue this approach.

First, note that the projection of causal history $\theta(e)$ on process p_i corresponds to an initial prefix of the local history of p_i. In other words, $\theta_i(e) = h_i^k$ for some unique k and, by the canonical enumeration of events, $e_i^\ell \in \theta_i(e)$ for all $\ell < k$. Thus, a single natural number is sufficient to represent the set $\theta_i(e)$. Since $\theta(e) = \theta_1(e) \cup \cdots \cup \theta_n(e)$, the entire causal history can be represented by an n-dimensional vector $VC(e)$ where for all $1 \leq i \leq n$, the ith component is defined as

$$VC(e)[i] = k, \quad \text{if and only if } \theta_i(e) = h_i^k.$$

The resulting mechanism is known as *vector clocks* and has been discovered independently by many researchers in many different contexts (see [34] for a survey). In this scheme, each process p_i maintains a local vector VC of natural numbers where $VC(e_i)$ denotes the vector clock value of p_i when it executes event e_i. As with logical clocks, we use VC to refer to the current vector clock of a process that is implicit from context. Each process p_i initializes VC to contain all zeros. All messages contain a timestamp $TS(m)$ which is the vector clock value of m's send event. The following update rules define how the vector clock is modified by p_i with the occurrence of each new event e_i:

$$\begin{array}{ll} VC(e_i)[i] := VC[i] + 1 & \text{if } e_i \text{ is an internal or send event} \\ & \\ VC(e_i) := \max\{VC, TS(m)\} & \text{if } e_i = receive(m) \\ VC(e_i)[i] := VC[i] + 1 & \end{array}$$

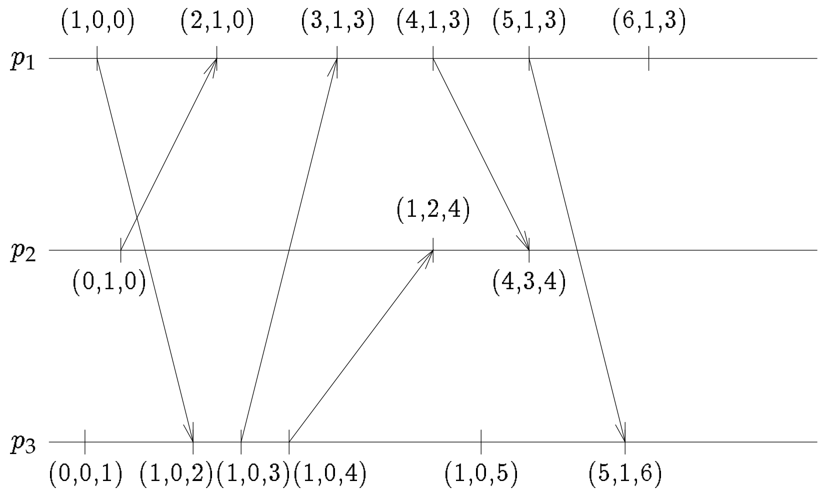

Figure 7. Vector Clocks

In other words, an internal or send event simply increments the local component of the vector clock. A receive event, on the other hand, first updates the vector clock to be greater than (on a component-by-component basis) both the previous value and the timestamp of the incoming message, and then increments the local component. Figure 7 illustrates the vector clocks associated with the events of the distributed computation displayed in Figure 1.

Given the above implementation, the jth component of the vector clock of process p_i has the following operational interpretation for all $j \neq i$:

$$VC(e_i)[j] \quad \equiv \quad \text{number of events of } p_j \text{ that causally precede event } e_i \text{ of } p_i.$$

On the other hand, $VC(e_i)[i]$ counts the number of events p_i has executed up to and including e_i. Equivalently, $VC(e_i)[i]$ is the ordinal position of event e_i in the canonical enumeration of p_i's events.

From the definition of vector clocks, we can easily derive a collection of useful properties. Given two n-dimensional vectors V and V' of natural numbers, we define the "less than" relation (written as $<$) between them as follows

$$V < V' \equiv (V \neq V') \wedge (\forall k : 1 \leq k \leq n : V[k] \leq V'[k]).$$

This allows us to express the strong clock condition in terms of vector clocks as

Property 1 *(Strong Clock Condition)*

$$e \rightarrow e' \equiv VC(e) < VC(e').$$

Note that for the above test, it is not necessary to know on which processes the two events were executed. If this information is available, causal precedence between two events can be verified through a single scalar comparison.

Property 2 *(Simple Strong Clock Condition) Given event e_i of process p_i and event e_j of process p_j, where $i \neq j$*

$$e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i].$$

Note that the condition $VC(e_i)[i] = VC(e_j)[i]$ is possible and represents the situation where e_i is the latest event of p_i that causally precedes e_j of p_j (thus e_i must be a send event).

Given this version of the strong clock condition, we obtain a simple test for concurrency between events that follows directly from its definition

Property 3 *(Concurrent) Given event e_i of process p_i and event e_j of process p_j*

$$e_i \| e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j]).$$

Consistency of cuts of a distributed computation can also be easily verified in terms of vector clocks. Events e_i and e_j are said to be *pairwise inconsistent* if they cannot belong to the frontier of the same consistent cut. In terms of vector clocks, this can be expressed as

Property 4 *(Pairwise Inconsistent) Event e_i of process p_i is pairwise inconsistent with event e_j of process p_j, where $i \neq j$, if and only if*

$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j]).$$

The two disjuncts characterize exactly the two possibilities for the cut to include at least one receive event without including its corresponding send event (thus making it inconsistent). While this property might appear to be equivalent to $\neg(e_i \| e_j)$ at first sight, this is not the case; it is obviously possible for two events to be causally related and yet be pairwise consistent.

We can then characterize a cut as being consistent if its frontier contains no pairwise inconsistent events. Given the definition of a cut, it suffices to check pairwise inconsistency only for those events that are in the frontier of the cut. In terms of vector clocks, the property becomes

Property 5 *(Consistent Cut) A cut defined by $(c_1, \ldots, c_n)$ is consistent if and only if*

$$\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i].$$

Recall that, for all $j \neq i$, the vector clock component $VC(e_i)[j]$ can be interpreted as the number of events of p_j that causally precede event e_i of p_i. The component corresponding to the process itself, on the other hand, counts the total number of events executed by p_i up to and including e_i. Let $\#(e_i) = (\sum_{j=1}^{n} VC(e_i)[j]) - 1$. Thus, $\#(e_i)$ denotes exactly the number of events that causally precede e_i in the entire computation.

Property 6 *(Counting) Given event e_i of process p_i and its vector clock value $VC(e_i)$, the number of events e such that $e \rightarrow e_i$ (equivalently, $VC(e) < VC(e_i)$) is given by $\#(e_i)$.*

Finally, vector clocks supply a weak form of the gap-detection property that logical and real-time clocks do not. The following property follows directly from the vector clock update rules and the second form of the Strong Clock Condition. It can be used to determine if the causal "gap" between two events admits a third event.

Property 7 *(Weak Gap-Detection) Given event e_i of process p_i and event e_j of process p_j, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists an event e_k such that*

$$\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j).$$

The property is "weak" in the sense that, for arbitrary processes p_i, p_j and p_k, we cannot conclude if the three events form a causal chain $e_i \rightarrow e_k \rightarrow e_j$. For the special case $i = k$, however, the property indeed identifies the sufficient condition to make such a conclusion.

11 Implementing Causal Delivery with Vector Clocks

The weak gap-detection property of the previous section can be exploited to efficiently implement causal delivery using vector clocks. Assume that processes increment the local component of their vector clocks only for events that are notified to the monitor.[14] As usual, each message m carries a timestamp $TS(m)$ which is the vector clock value of the event being notified by m. All messages that have been received but not yet delivered by the monitor process p_0 are maintained in a set $\mathcal{M}$, initially empty.

A message $m \in \mathcal{M}$ from process p_j is deliverable as soon as p_0 can verify that there are no other messages (neither in $\mathcal{M}$ nor in the network) whose sending causally precede that of m. Let m' be the last message delivered from process p_k, where $k \neq j$. Before message m of process p_j can be delivered, p_0 must verify two conditions:

1. there is no earlier message from p_j that is undelivered, and
2. there is no undelivered message m'' from p_k such that

$$send(m') \rightarrow send(m'') \rightarrow send(m), \forall k \neq j.$$

The first condition holds if exactly $TS(m)[j] - 1$ messages have already been delivered from p_j. To verify the second condition, we can use the special case of weak gap-detection where $i = k$ and $e_i = send_k(m')$, $e_k = send_k(m'')$ and $e_j = send_j(m)$. Since the two events $send_k(m')$ and $send_k(m'')$ both occur at process p_k, Property 7 can be written as

(Weak Gap-Detection) If $TS(m')[k] < TS(m)[k]$ for some $k \neq j$, then there exists event $send_k(m'')$ such that

$$send_k(m') \rightarrow send_k(m'') \rightarrow send_j(m).$$

Thus, no undelivered message m'' exists if $TS(m')[k] \geq TS(m)[k]$, for all k. These tests can be efficiently implemented if p_0 maintains an array $D[1 \ldots n]$ of counters, initially all zeros, such that counter $D[i]$ contains $TS(m_i)[i]$ where m_i is the last message that has been delivered from process p_i. The delivery rule then becomes:

14. Equivalently, processes send a notification message to the monitor for *all* of their events.

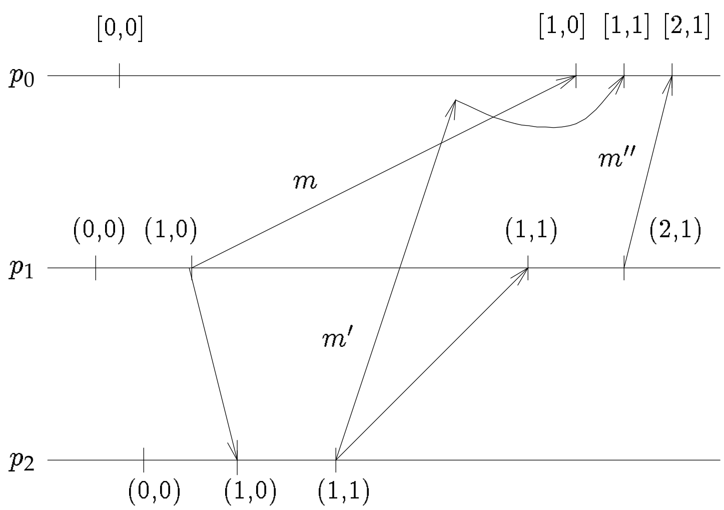

Figure 8. Causal Delivery Using Vector Clocks

DR3: (Causal Delivery) Deliver message m from process p_j as soon as both of the following conditions are satisfied

$$\begin{aligned} D[j] &= TS(m)[j] - 1 \\ D[k] &\geq TS(m)[k], \forall k \neq j. \end{aligned}$$

When p_0 delivers m, array D is updated by setting $D[j]$ to $TS(m)[j]$.

Figure 8 illustrates the application of this delivery rule by p_0 in a sample computation. The events of p_1 and p_2 are annotated with the vector clock values while those of p_0 indicate the values of array D. Note that the delivery of message m' is delayed until message m has been received and delivered. Message m'', on the other hand, can be delivered as soon as it is received since p_0 can verify that all causally preceding messages have been delivered.

At this point, we have a complete reactive-architecture solution to the GPE problem in asynchronous distributed systems based on passive observations. The steps are as follows. Processes notify the monitor p_0 of relevant events by sending it messages. The monitor uses a causal delivery rule for the notification messages to construct an observation that corresponds to a consistent run. The global predicate Φ can be applied to any one of the global states in the run since each is guaranteed to be consistent. An application of this solution to deadlock detection is given in Section 14.1.

Causal delivery can be implemented at any process rather than just at the monitor. If processes communicate exclusively through broadcasts (rather than point-to-point sends), then delivery rule DR3 remains the appropriate mechanism for achieving causal delivery at all destinations [3]. The resulting primitive, known as *causal broadcast* (c.f. Section 15), has been recognized as an important abstraction for building distributed applications [3,15,28]. If, on the other hand, communication can take place through point-to-point sends, a delivery rule can be derived based on an extension of vector clocks where each message carries a timestamp composed of n vector clocks (i.e., an $n \times n$ matrix) [32,30].

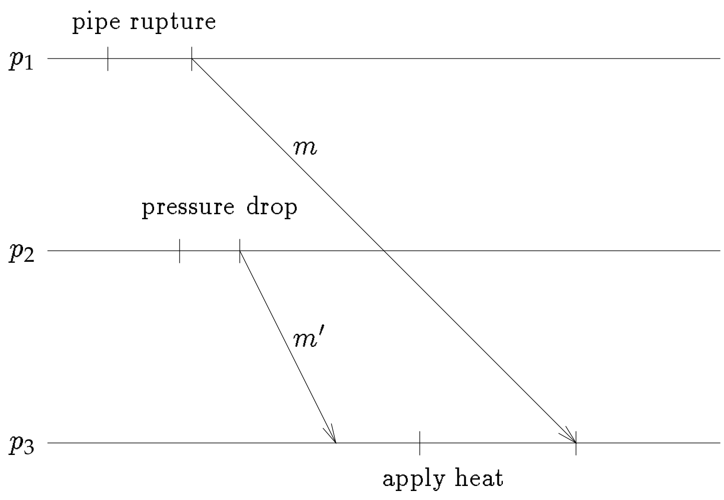

Figure 9. External Environment as a Hidden Channel

12 Causal Delivery and Hidden Channels

In general, causal delivery allows processes to reason globally about the system using only local information. For such conclusions to be meaningful, however, we need to restrict our attention to *closed* systems—those that constrain all communication to take place within the boundaries of the computing system. If global reasoning based on causal analysis is applied to systems that contain so-called *hidden channels*, incorrect conclusions may be drawn [17].

To illustrate the problem, consider the example taken from [16] and shown in Figure 9. A physical process is being monitored and controlled by a distributed system consisting of p_1, p_2 and p_3. Process p_1 is monitoring the state of a steam pipe and detects its rupture. The event is notified to the controller process p_3 in message m. Process p_2 is monitoring the pressure of the same pipe, several meters downstream from p_1. A few seconds after the rupture of the pipe, p_2 detects a drop in the pressure and notifies p_3 of the event in message m'. Note that from the point of view of explicit communication, messages m and m' are concurrent. Message m' arrives at p_3 and is delivered without delay since there are no undelivered messages that causally precede it. As part of its control action, p_3 reacts to the pressure drop by applying more heat to increase the temperature. Some time later, message m arrives reporting the rupture of the pipe. The causal sequence observed by p_3 is $\langle$*pressure drop*, *apply heat*, *pipe rupture*$\rangle$ leading it to conclude that the pipe ruptured due to the increased temperature. In fact, the opposite is true.

The apparent anomaly is due to the steam pipe which acts as a communication channel external to the system. The rupture and pressure drop events are indeed causally related even though it is not captured by the $\rightarrow$ relation. When the pipe is included as a communication channel, the order in which messages are seen by p_3 violates causal delivery. In systems that are not closed, global reasoning has to be based on totally-ordered observations derived from global real-time. Since this order is consistent with causal precedence, anomalous conclusions such as the one above will be avoided.

13 Distributed Snapshots

In Section 5 we presented a strategy for solving the GPE problem through active monitoring. In this strategy, p_0 requested the states of the other processes and then combined them into a global state. Such a strategy is often called a "snapshot" protocol, since p_0 "takes pictures" of the individual process states. As we noted, this global state may not be consistent, and so the monitor may make an incorrect deduction about the system property encoded in the global predicate.

We will now develop a snapshot protocol that constructs only consistent global states. The protocol is due to Chandy and Lamport [5], and the development described here is due to Morgan [26]. For simplicity, we will assume that the channels implement FIFO delivery, and we omit details of how individual processes return their local states to p_0.

For this protocol, we will introduce the notion of a *channel state*. For each channel from p_i to p_j, its state $\chi_{i,j}$ are those messages that p_i has sent to p_j but p_j has not yet received. Channel states are only a convenience in that each $\chi_{i,j}$ can be inferred from just the local states σ_i and σ_j as the set difference between messages sent by p_i to p_j (encoded in σ_i) and messages received by p_j from p_i (encoded in σ_j). In many cases, however, explicit maintenance of channel state information can lead to more compact process local states and simpler encoding for the global predicate of interest. For example, when constructing the waits-for graph in deadlock detection, an edge is drawn from p_i to p_j if p_i is blocked due to p_j. This relation can be easily obtained from the local process states and channel states: process p_i is blocked on p_j if σ_i records the fact that there is an outstanding request to p_j, and $\chi_{j,i}$ contains no response messages.

Let IN_i be the set of processes that have channels connecting them directly to p_i and OUT_i be the set of processes to which p_i has a channel. Channels from $p_j \in IN_i$ to p_i are called *incoming* while channels from p_i to $p_j \in OUT_i$ are called *outgoing* with respect to p_i. For each execution of the snapshot protocol, a process p_i will record its local state σ_i and the states of its incoming channels ($\chi_{j,i}$, for all $p_j \in IN_i$).

13.1 Snapshot Protocols

We proceed as before by devising a simple protocol based on a strong set of assumptions and refining the protocol as we relax them. Initially, assume that all processes have access to a global real-time clock *RC*, that all message delays are bound by some known value, and that relative process speeds are bounded.

The first snapshot protocol is based on all processes recording their states at the same real-time. Process p_0 chooses a time t_{ss} far enough in the future in order to guarantee that a message sent now will be received by all other processes before t_{ss}.[15] To facilitate the recording of channel states, processes include a timestamp in each message indicating when the message's send event was executed.

Snapshot Protocol 1

1. Process p_0 sends the message "take snapshot at t_{ss}" to all processes.[16]

---

15. Recall that there need not be a channel between all pairs of processes, and so t_{ss} must account for the possibility of messages being forwarded.
16. For simplicity, we describe the protocols for a single initiation by process p_0. In fact, they can be initiated by any process, and as long as concurrent initiations can be distinguished, multiple initiations are possible.

2. When clock RC reads t_{ss}, each process p_i records its local state σ_i, sends an empty message over all of its outgoing channels, and starts recording messages received over each of its incoming channels. Recording the local state and sending empty messages are performed before any intervening events are executed on behalf of the underlying computation.
3. First time p_i receives a message from p_j with timestamp greater than or equal to t_{ss}, p_i stops recording messages for that channel and declares $\chi_{j,i}$ as those messages that have been recorded.

For each $p_j \in IN_i$, the channel state $\chi_{j,i}$ constructed by process p_i contains the set of messages sent by p_j before t_{ss} and received by p_i after t_{ss}. The empty messages in Step 2 are sent in order to guarantee liveness:[17] process p_i is guaranteed to eventually receive a message m from every incoming channel such that $TS(m) \geq t_{ss}$.

Being based on real-time, it is easy to see that this protocol constructs a consistent global state—it constructs a global state that did in fact occur and thus could have been observed by our idealized external observer. However, it is worth arguing this point a little more formally. Note that an event e is in the consistent cut C_{ss} associated with the constructed global state if and only if $RC(e) < t_{ss}$. Hence,

$$(e \in C_{ss}) \wedge (RC(e') < RC(e)) \Rightarrow (e' \in C_{ss}).$$

Since real-time clock RC satisfies the clock condition, the above equation implies that C_{ss} is a consistent cut. In fact, the clock condition is the only property of RC that is necessary for C_{ss} to be a consistent cut. Since logical clocks also satisfy the clock condition, we should be able to substitute logical clocks for real-time clocks in the above protocol.

There are, however, two other properties of synchronous systems used by Snapshot Protocol 1 that need to be supplied:

- The programming construct "**when** $LC = t$ **do** S" doesn't make sense in the context of logical clocks since the given value t need not be attained by a logical clock.[18] For example, in Figure 4 the logical clock of p_3 never attains the value 6, because the receipt of the message from p_1 forces it to jump from 5 to 7. Even if LC does attain a value of t, the programming construct is still problematic. Our rules for updating logical clocks are based on the occurrence of new events. Thus, at the point $LC = t$, the event that caused the clock update has been executed rather than the first event of S.

  We overcome this problem with the following rules. Suppose p_i contains the statement "**when** $LC = t$ **do** S", where S generates only internal events or send events. Before executing an event e, process p_i makes the following test:
  - If e is an internal or send event and $LC = t - 2$, then p_i executes e and then starts executing S.
  - If $e = receive(m)$ where $TS(m) \geq t$ and $LC < t - 1$, then p_i puts the message back onto the channel, re-enables e for execution, sets LC to $t - 1$ and starts executing S.
- In Protocol 1, the monitor p_0 chooses t_{ss} such that the message "take snapshot at t_{ss}" is received by all other processes before time t_{ss}. In an asynchronous system, p_0 cannot

17. Our use of empty messages here is not unlike their use in distributed simulation for the purposes of advancing global virtual time [25].
18. Note that this problem happens to be one aspect of the more general problem of simulating a synchronous system in an asynchronous one [1].

compute such a logical clock value. Instead, we assume that there is an integer value ω large enough that no logical clock can reach ω by using the update rules in Section 8.

Assuming the existence of such an ω requires us to bound both relative process speeds and message delays, and so we will have to relax it as well. Given the above considerations, we obtain Snapshot Protocol 2, which differs from Protocol 1 only in its use of logical clocks in place of the real-time clock.

Snapshot Protocol 2

1. Process p_0 sends "take snapshot at ω" to all processes and then sets its logical clock to ω.
2. When its logical clock reads ω, process p_i records its local state σ_i, sends an empty message along each outgoing channel, and starts recording messages received over each of its incoming channels. Recording the local state and sending empty messages are performed before any intervening events are executed on behalf of the underlying computation.
3. First time p_i receives a message from p_j with timestamp greater than or equal to ω, p_i stops recording messages for that channel and declares $\chi_{j,i}$ as those messages that have been recorded.

Channel states are constructed just as in Protocol 1 with ω playing the role of t_{ss}. As soon as p_0 sets its logical clock to ω, it will immediately execute Step 2, and the empty messages sent by it will force the clocks of processes in OUT_0 to attain ω. Since the network is strongly connected, all of the clocks will eventually attain ω, and so the protocol is live.

We now remove the need for ω. Note that, with respect to the above protocol, a process does nothing between receiving the "take snapshot at ω" message and receiving the first empty message that causes its clock to pass through ω. Thus, we can eliminate the message "take snapshot at ω" and instead have a process record its state when it receives the first empty message. Since processes may send empty messages for other purposes, we will change the message from being empty to one containing a unique value, for example, "take snapshot". Furthermore, by making this message contain a unique value, we no longer need to include timestamps in messages—the message "take snapshot" is the first message that any process sends after the snapshot time. Doing so removes the last reference to logical clocks, and so we can eliminate them from our protocol completely.

Snapshot Protocol 3 (Chandy-Lamport [5])

1. Process p_0 starts the protocol by sending itself a "take snapshot" message.
2. Let p_f be the process from which p_i receives the "take snapshot" message for the first time. Upon receiving this message, p_i records its local state σ_i and relays the "take snapshot" message along all of its outgoing channels. No intervening events on behalf of the underlying computation are executed between these steps. Channel state $\chi_{f,i}$ is set to empty and p_i starts recording messages received over each of its other incoming channels.
3. Let p_s be the process from which p_i receives the "take snapshot" message beyond the first time. Process p_i stops recording messages along the channel from p_s and declares channel state $\chi_{s,i}$ as those messages that have been recorded.

Since a "take snapshot" message is relayed only upon the first receipt and since the network is strongly connected, a "take snapshot" message traverses each channel exactly

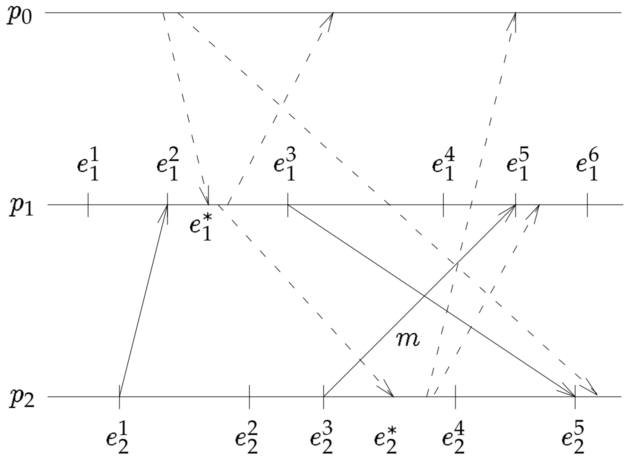

Figure 10. Application of the Chandy-Lamport Snapshot Protocol

once. When process p_i has received a "take snapshot" message from all of its incoming channels, its contribution to the global state is complete and its participation in the snapshot protocol ends.

Note that the above protocols can be extended and improved in many ways including relaxation of the FIFO assumption [23,35] and reduction of the message complexity [24,31].

13.2 Properties of Snapshots

Let Σ^s be a global state constructed by the Chandy-Lamport distributed snapshot protocol. In the previous section, we argued that Σ^s is guaranteed to be consistent. Beyond that, however, the actual run that the system followed while executing the protocol may not even pass through Σ^s. In this section, we show that Σ^s is not an arbitrary consistent global state, but one that has useful properties with respect to the run that generated it.

Consider the application of Chandy-Lamport snapshot protocol to the distributed computation of Figure 3. The composite computation is shown in Figure 10 where solid arrows indicate messages sent by the underlying computation while dashed arrows indicate "take snapshot" messages sent by the protocol. From the protocol description, the constructed global state is Σ^{23} with $\chi_{1,2}$ empty and $\chi_{2,1}$ containing message m. Let the run followed by processes p_1 and p_2 while executing the protocol be

$$r = e_2^1\, e_1^1\, e_1^2\, e_1^3\, e_2^2\, e_1^4\, e_2^3\, e_2^4\, e_1^5\, e_2^5\, e_1^6$$

or in terms of global states,

$$r = \Sigma^{00}\, \Sigma^{01}\, \Sigma^{11}\, \Sigma^{21}\, \Sigma^{31}\, \Sigma^{32}\, \Sigma^{42}\, \Sigma^{43}\, \Sigma^{44}\, \Sigma^{54}\, \Sigma^{55}\, \Sigma^{65}.$$

Let the global state of this run in which the protocol is initiated be Σ^{21} and the global state in which it terminates be Σ^{55}. Note that run r does not pass through the constructed global state Σ^{23}. As can be verified by the lattice of Figure 3, however, $\Sigma^{21} \leadsto \Sigma^{23} \leadsto \Sigma^{55}$ in this example. We now show that this relationship holds in general.

Let Σ^a be the global state in which the snapshot protocol is initiated, Σ^f be the global state in which the protocol terminates and Σ^s be the global state constructed. We will show that there exists a run R such that $\Sigma^a \leadsto_R \Sigma^s \leadsto_R \Sigma^f$. Let r be the actual run the system followed while executing the snapshot protocol, and let e_i^* denote the event when p_i receives "take snapshot" for the first time, causing p_i to record its state. An event e_i of p_i is a *prerecording* event if $e_i \rightarrow e_i^*$; otherwise, it is a *postrecording* event.

Consider any two adjacent events $\langle e, e' \rangle$ of r such that e is a postrecording event and e' is a prerecording event.[19] We will show that $\neg(e \rightarrow e')$, and so the order of these two events can be swapped, thereby resulting in another consistent run. If we continue this process of swapping ⟨postrecording, prerecording⟩ event pairs, then we will eventually construct a consistent run in which no prerecording event follows a postrecording event. The global state associated with the last prerecording event is therefore reachable from Σ^a and the state Σ^f is reachable from it. Finally, we will show that this state is Σ^s, the state that the snapshot protocol constructs.

Consider the subsequence $\langle e, e' \rangle$ of run r where e is a postrecording event and e' a prerecording event. If $e \rightarrow e'$ then the two events cannot be swapped without resulting in an inconsistent run. For contradiction, assume that $e \rightarrow e'$. There are two cases to consider:

1. Both events e and e' are from the same process. If this were the case, however, then by definition e' would be a postrecording event.
2. Event e is a send event of p_i and e' is the corresponding receive event of p_j. If this were the case, however, then from the protocol p_i will have sent a "take snapshot" message to p_j by the time e is executed, and since the channel is FIFO, e' will also be a postrecording event.

Hence, a postrecording event cannot causally precede a prerecording event and thus any ⟨postrecording, prerecording⟩ event pair can be swapped. Let R be the run derived from r by swapping such pairs until all postrecording events follow the prerecording events. We now argue that the global state after the execution of the last prerecording event e in R is Σ^s. By the protocol description and the definition of prerecording, postrecording events that record local states will record them at point e. Furthermore, by the protocol, the channel states that are recorded are those messages that were sent by prerecording events and received by postrecording events. By construction, these are exactly those messages in the channels after the execution of event e, and so Σ^s is the state recorded by the snapshot protocol. □

14 Properties of Global Predicates

We have derived two methods for global predicate evaluation: one based on a monitor actively constructing snapshots and another one based on a monitor passively observing runs. The utility of either approach for solving practical distributed systems problems, however, depends in part on the properties of the predicate that is being evaluated. In this section, some of these properties are examined.

14.1 Stable Predicates

Let Σ^s be a consistent global state of a computation constructed through any feasible mechanism. Given that communication in a distributed system incurs delays, Σ^s can only reflect

19. Adjacent event pairs $\langle e_1^3, e_2^2 \rangle$ and $\langle e_1^4, e_2^3 \rangle$ of run r are two such examples.

```
process p(i): 1 ≤ i ≤ n
    var pending: queue of [message, integer] init empty;      % pending requests to p(i)
        working: boolean init false;                          % processing a request
        m: message; j: integer;
    while true do
        while working or (size(pending) = 0) do
            receive m from p(j);                    % m set to message, j to its source
            case m.type of
                request:
                    pending := pending + [m, j];
                response:
                    [m, j] := NextState(m, j);
                    working := (m.type = request);
                    send m to p(j);
            esac
        od ;
        while not working and (size(pending) > 0) do
            [m, j] := first(pending);
            pending := tail(pending);
            [m, j] := NextState(m, j);
            working := (m.type = request);
            send m to p(j)
        od
    od
end p(i);
```

Figure 11. Server Process

some past state of the system— by the time they are obtained, conclusions drawn about the system by evaluating predicate Φ in Σ^s may have no bearing to the present.

Many system properties one wishes to detect, however, have the characteristic that once they become true, they remain true. Such properties (and their predicates) are called *stable*. Examples of stable properties include deadlock, termination, the loss of all tokens, and unreachability of storage (garbage collection). If Φ is stable, then the monitor process can strengthen its knowledge about when Φ is satisfied.

As before, let Σ^a be the global state in which the global state construction protocol is initiated, Σ^f be the global state in which the protocol terminates and Σ^s be the global state it constructs. Since $\Sigma^a \leadsto \Sigma^s \leadsto \Sigma^f$, if Φ is stable, then the following conclusions are possible

$$(\Phi \text{ is true in } \Sigma^s) \Rightarrow (\Phi \text{ is true in } \Sigma^f)$$

and

$$(\Phi \text{ is false in } \Sigma^s) \Rightarrow (\Phi \text{ is false in } \Sigma^a).$$

As an example of detecting a stable property, we return to deadlock in the client-server system described in Section 5. We assume that there is a bidirectional communication channel between each pair of processes and each process when acting as a server runs the same program, shown in Figure 11. The behavior of a process as a client is much simpler: after sending a request to a server it blocks until the response is received. The server is modeled as a *state machine* [33]: it repeatedly receives a message, changes its state, and optionally sends one or more messages. The function NextState() computes the action a server next takes: given a message m from process p_j, the invocation NextState(m, j) changes the state of the server and returns the next message to send along with its destination. The resulting message may be a response to the client's request or it may be a further request whose response is needed to service the client's request. All requests received by a server, including those received while it is servicing an earlier request, are queued on the FIFO queue pending, and the server removes and begins servicing the first entry from pending after it finishes an earlier request by sending the response.

Figures 12 shows the server of Figure 11 with a snapshot protocol embedded in it. Each server maintains a boolean array blocking that indicates which processes have sent it requests to which it has not yet responded (this information is also stored in pending, but we duplicate it in blocking for clarity). When a server p_i first receives a snapshot message, it sends the contents of blocking to p_0 and relays the snapshot message to all other processes. Subsequent snapshot messages are ignored until p_i has received n such messages, one from each other process.[20]

The conventional definition of "p_i waits-for p_j" is that p_i has sent a request to p_j and p_j has not yet responded. As in Section 5, we will instead use the weaker definition p_i *waits-for*$^+$ p_j which holds when p_j has received a request from p_i to which it has not yet responded [11]. By structuring the server as a state machine, even requests sent to a deadlocked server will eventually be received and denoted in blocking. Hence, a system that contains a cycle in the conventional WFG will eventually contain a cycle in the WFG$^+$, and so a deadlock will be detected eventually. Furthermore, using the WFG$^+$ instead of the WFG has the advantage of referencing only the local process states, and so the embedded snapshot protocol need not record channel states.

20. By the assumption that the network is completely connected, each invocation of the snapshot protocol will result in exactly n snapshot messages to be received by each process.

```
process p(i): 1 ≤ i ≤ n
  var pending: queue of [message, integer] init empty;          % pending requests to p(i)
      working: boolean init false;                                % processing a request
      blocking: array [1..n] of boolean init false;    % blocking[j] = “p(j) is blocked on p(i)”
      m: message; j: integer; s: integer init 0;
  while true do
      while working or (size(pending) = 0) do
          receive m from p(j);                        % m set to message, j to its source
          case m.type of
              request:
                  blocking[j] := true;
                  pending := pending + [m, j];
              response:
                  [m, j] := NextState(m, j);
                  working := (m.type = request);
                  send m to p(j);
                  if (m.type = response) then blocking[j] := false;
              snapshot:
                  if s = 0 then
                       % this is the first snapshot message
                      send [type: snapshot, data: blocking] to p(0);
                      send [type: snapshot] to p(1),...,p(i−1),p(i+1),...,p(n)
                  s := (s + 1) mod n;
          esac
      od ;
      while not working and (size(pending) > 0) do
          [m, j] := head(pending);
          pending := tail(pending);
          [m, j] := NextState(m, j);
          working := (m.type = request);
          send m to p(j);
          if (m.type = response) then blocking[j] := false;
      od
  od
end p(i);
```

Figure 12. Deadlock Detection through Snapshots: Server Side

```
process p(0):
    var wfg: array [1..n] of array [1..n] of boolean;          % wfg[i, j] = p(j) waits-for p(i)
        j, k: integer; m: message;
    while true do
        wait until deadlock is suspected;
        send [type: snapshot] to p(1), ..., p(n);
        for k := 1 to n do
            receive m from p(j);
            wfg[j] := m.data;
        if (cycle in wfg) then system is deadlocked
    od
end p(0);
```

Figure 13. Deadlock Detection through Snapshots: Monitor Side

Figure 13 shows the code run by the monitor p_0 acting as the deadlock detector. This process periodically starts a snapshot by sending a snapshot message to all other processes. Then, p_0 receives the arrays blocking from each of the processes and uses this data to test for a cycle in the WFG$^+$. This approach has the advantage of generating an additional message load only when deadlock is suspected. However, the approach also introduces a latency between the occurrence of the deadlock and detection that depends on how often the monitor starts a snapshot.

Figures 14 and 15 show the server and monitor code, respectively, of a reactive-architecture deadlock detector. This solution is much simpler than the snapshot-based version. In this case, p_i sends a message to p_0 whenever p_i receives a request or sends a response to a client. Monitor p_0 uses these notifications to update a WFG$^+$ in which it tests for a cycle. The simplicity of this solution is somewhat superficial, however, because the protocol requires all messages to p_0 to be sent using causal delivery order instead of FIFO order. The only latency between a deadlock's occurrence and its detection is due to the delay associated with a notification message, and thus, is typically shorter than that of the snapshot-based detector.

14.2 Nonstable Predicates

Unfortunately, not all predicates one wishes to detect are stable. For example, when debugging a system one may wish to monitor the lengths of two queues, and notify the user if the sum of the lengths is larger than some threshold. If both queues are dynamically changing, then the predicate corresponding to the desired condition is not stable. Detecting such predicates poses two serious problems.

The first problem is that the condition encoded by the predicate may not persist long enough for it to be true when the predicate is evaluated. For example, consider the computation of Figure 16 in which variables x and y are being maintained by two processes p_1 and p_2, respectively. Suppose we are interested in monitoring the condition $(x = y)$. There are seven consistent global states in which the condition $(x = y)$ holds, yet if the monitor evaluates $(x = y)$ after state Σ^{54} then the condition will be found not to hold.

The second problem is more disturbing: if a predicate Φ is found to be true by the monitor, we do not know whether Φ *ever* held during the actual run. For example, suppose

```
process p(i): 1 ≤ i ≤ n
    var pending: queue of [message, integer] init empty;          % pending requests to p(i)
        working: boolean init false;                                % processing a request
        m: message; j: integer;
    while true do
        while working or (size(pending) = 0) do
            receive m from p(j);                       % m set to message, j to its source
            case m.type of
                request:
                    send [type: requested, of: i, by: j] to p(0);
                    pending := pending + [m, j];
                response:
                    [m, j] := NextState(m, j);
                    working := (m.type = request);
                    send m to p(j);
                    if (m.type = response) then
                        send [type: responded, to: j, by: i] to p(0);
            esac
        od ;
        while not working and (size(pending) > 0) do
            [m, j] := first(pending);
            pending := tail(pending);
            [m, j] := NextState(m, j);
            working := (m.type = request);
            send m to p(j);
            if (m.type = response) then
                send [type: responded, to: j, by: i] to p(0)
        od
    od
end p(i);
```

Figure 14. Deadlock Detection using Reactive Protocol: Server Side

```
process p(0):
    var wfg: array [1..n, 1..n] of boolean init false;      % wfg[i, j] = "p(j) waits-for p(i)"
        m: message; j: integer;
    while true do
        receive m from p(j);                                % m set set to message, j to its source
        if (m.type = responded) then
            wfg[m.by, m.to] := false
        else
            wfg[m.of, m.by] := true;
        if (cycle in wfg) then system is deadlocked
    od
end p(0);
```

Figure 15. Deadlock Detection using Reactive Protocol: Monitor Side

in the same computation the condition being monitored is $(y - x) = 2$. The only two global states of Figure 16 satisfying this condition are Σ^{31} and Σ^{41}. Let a snapshot protocol be initiated in state Σ^{11} of the run

$$\Sigma^{00}\ \Sigma^{01}\ \Sigma^{11}\ \Sigma^{12}\ \Sigma^{22}\ \Sigma^{32}\ \Sigma^{42}\ \Sigma^{43}\ \Sigma^{44}\ \Sigma^{45}\ \Sigma^{55}\ \Sigma^{65}.$$

From the result of Section 13.2, we know that the snapshot protocol could construct either global state Σ^{31} or Σ^{41} since both are reachable from Σ^{11}. Thus, the monitor could "detect" $(y - x) = 2$ even though the actual run never goes through a state satisfying the condition.

It appears that there is very little value in using a snapshot protocol to detect a nonstable predicate—the predicate may have held even if it is not detected, and even if it is detected it may have never held. The same problems exist when nonstable predicates are evaluated over global states constructed from observations of runs: if a nonstable predicate holds at some state during a consistent observation, then the condition may or may not have held during the actual run that produced it.

With observations, however, we can extend nonstable global predicates such that they are meaningful in the face of uncertainty of the run actually followed by the computation. To do this, the extended predicates must apply to the entire distributed computation rather than to individual runs or global states of it. There are two choices for defining predicates over computations [6,22]:

1. **Possibly**(Φ): There exists a consistent observation O of the computation such that Φ holds in a global state of O.
2. **Definitely**(Φ): For every consistent observations O of the computation, there exists a global state of O in which Φ holds.

The distributed computation of Figure 16 satisfies both predicates **Possibly**($(y-x) = 2$) and **Definitely**($x = y$). As with stable predicates, by the time they are detected, both of these predicates refer to some past state or states of a run. The predicate "Φ currently holds" can also be detected, but to do so will introduce blocking of the underlying computation.

An application of these extended predicates is when a run is observed for purposes of debugging [6]. For instance, if Φ identifies some erroneous state of the computation, then **Possibly**(Φ) holding indicates a bug, even if it is not observed during an actual run. For

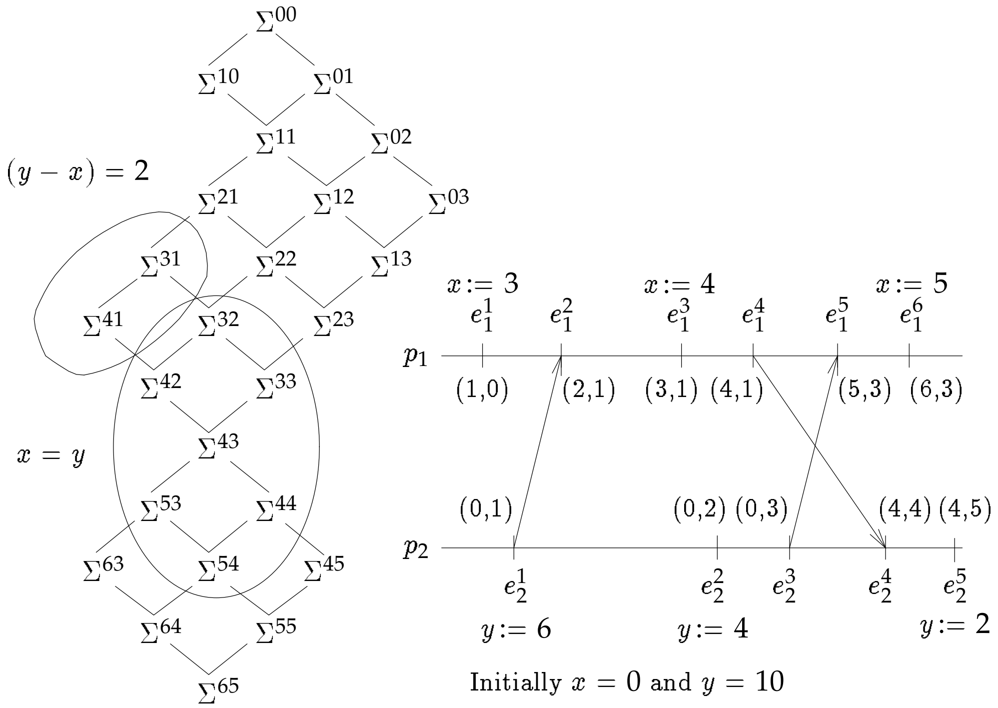

Figure 16. Global States Satisfying Predicates $(x = y)$ **and** $(y - x) = 2$

example, if $(y - x) = 2$ denotes an erroneous state, then the computation of Figure 16 is incorrect, since there is no guarantee that the erroneous state will not occur in some run.

The intuitive meanings of **Possibly** and **Definitely** could lead one to believe that they are duals of each other: $\neg$**Definitely** (Φ) being equivalent to **Possibly**$(\neg\Phi)$ and $\neg$**Possibly** (Φ) being equivalent to **Definitely**$(\neg\Phi)$. This is not the case. For example, while it is true that $\neg$**Definitely**(Φ) holding through a computation does imply **Possibly**$(\neg\Phi)$ (there must be an observation O in which $\neg\Phi$ holds in all of its states), it is possible to have both **Possibly**$(\neg\Phi)$ and **Definitely**(Φ) hold. Figure 16 illustrates the latter: the computation satisfies both **Possibly**$(x \neq y)$ and **Definitely**$(x = y)$. Furthermore, if predicate Φ is stable, then **Possibly**$(\Phi) \equiv$ **Definitely**(Φ). The inverse, however, is not true in general.

The choice between detecting whether Φ currently holds versus whether Φ possibly or definitely held in the past depends on which restrictions confound the debugging process more. Detecting a condition that occurred in the past may require program replay or reverse execution in order to recover the state of interest, which can be very expensive to provide. Hence, detection in the past is better suited to a *post-mortem* analysis of a computation. Detecting the fact that Φ currently holds, on the other hand, requires delaying the execution of processes, which can be a serious impediment to debugging. By blocking some processes when the predicate becomes potentially true, we may make the predicate either more or less likely to occur. For example, a predicate may be less likely to occur if processes "communicate" using timeouts or some other uncontrolled form of communication. The latter is a particular problem when processes are multithreaded; that is, consisting of multiple, independently schedulable threads of control which may communicate through shared memory. In fact, it is rarely practical to monitor such communication when debugging without hardware or language support.

14.3 Detecting Possibly and Definitely Φ

The algorithms for detecting **Possibly**(Φ) and **Definitely**(Φ) are based on the lattice of consistent global states associated with the distributed computation. For every global state Σ in the lattice there exists at least one run that passes through Σ. Hence, if any global state in the lattice satisfies Φ, then **Possibly**(Φ) holds. For example, in the global state lattice of Figure 16 both Σ^{31} and Σ^{41} satisfy $(y-x) = 2$ meaning that **Possibly**$((y-x) = 2)$ holds for the computation. The property **Definitely**(Φ) requires all possible runs to pass through a global state that satisfies Φ. In Figure 16 the global state Σ^{43} satisfies $(x = y)$. Since this is the only state with level 7, and all runs must contain a global state of each level, **Definitely**$(x = y)$ also holds for the computation.

Figure 17 is a high-level procedure for detecting **Possibly**(Φ). The procedure constructs the set of global states current with progressively increasing levels (denoted by ℓ). When a member of current satisfies Φ, then the procedure terminates indicating that **Possibly**(Φ) holds. If, however, the procedure constructs the final global state (the global state in which the computation terminates) and finds that this global state does not satisfy Φ, then the procedure returns $\neg$ **Possibly**(Φ).

In order to implement this procedure, the monitored processes send the portion of their local states that is referenced in Φ to the monitor p_0. Process p_0 maintains sequences of these local states, one sequence per process, and uses them to construct the global states of a given level. The basic operation used by the procedure is "current := states of level ℓ", and so we must be able to determine when all of the global states of a given level can be assembled and must be able to effectively assemble them.

```
procedure Possibly(Φ);
    var current: set of global states;
        ℓ: integer;
    begin
        % Synchronize processes and distribute Φ
        send Φ to all processes;
        current := global state Σ^{0...0};
        release processes;
        ℓ := 0;
        % Invariant: current contains all states of level ℓ that are reachable from Σ^{0...0}
        while (no state in current satisfies Φ) do
            if current = final global state then return false
            ℓ := ℓ + 1;
            current := states of level ℓ
        od
        return true
    end
```

Figure 17. Algorithm for Detecting Possibly(Φ).

Let Q_i be the sequence of local states, stored in FIFO order, that p_0 maintains for process p_i, where each state σ_i^k in Q_i is labeled with the vector timestamp $VC(e_i^k)$. Define $\Sigma_{min}(\sigma_i^k)$ to be the global state with the smallest level containing σ_i^k and $\Sigma_{max}(\sigma_i^k)$ to be the global state with the largest level containing σ_i^k. For example, in Figure 16, $\Sigma_{min}(\sigma_1^3) = \Sigma^{31}$ and $\Sigma_{max}(\sigma_1^3) = \Sigma^{33}$. These states can be computed using Property 5 of vector clocks as follows:

$$\Sigma_{min}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots, \sigma_n^{c_n}) : \forall j : VC(\sigma_j^{c_j})[j] = VC(\sigma_i^k)[j]$$

and

$$\begin{aligned}\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots, \sigma_n^{c_n}) &: \\ \forall j : (VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]) \wedge ((\sigma_j^{c_j} = \sigma_j^f) &\vee (VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))\end{aligned}$$

where σ_j^f is the state in which process p_j terminates.

Global states $\Sigma_{min}(\sigma_i^k)$ and $\Sigma_{max}(\sigma_i^k)$ bound the levels of the lattice in which σ_i^k occurs. The minimum level containing σ_i^k is particularly easy to compute: it is the sum of components of the vector timestamp $VC(\sigma_i^k)$. Thus, p_0 can construct the set of states with level ℓ when, for each sequence Q_i, the sum of the components of the vector timestamp of the last element of Q_i is at least ℓ. For example, if p_0 monitors the computation shown in Figure 16, then p_0 can start enumerating level 6 when it has received states σ_1^5 and σ_2^4 because any global state containing σ_1^5 must have a level of at least 8 (5 + 3) and any global state containing σ_2^4 must also have a level of at least 8 (4+4). Similarly, process p_0 can remove state σ_i^k from Q_i when ℓ is greater than the level of the global state $\Sigma_{max}(\sigma_i^k)$. For the example in Figure 16, $\Sigma_{max}(\sigma_1^2) = \Sigma^{23}$, and so p_0 can remove σ_1^2 from Q_1 once it has set ℓ to 6.

Given the set of states of some level ℓ, it is also straightforward (if somewhat costly) to construct the set of states of level $\ell + 1$: for each state $\Sigma^{i_1, i_2, \ldots, i_n}$ of level ℓ, one constructs the n global states $\Sigma^{i_1+1, i_2, \ldots, i_n}$, ..., $\Sigma^{i_1, i_2, \ldots, i_n+1}$. Then, Property 5 of vector clocks can be used to determine which of these global states are consistent. One can be careful and avoid

```
procedure Definitely(Φ);
    var current, last: set of global states;
        ℓ: integer;
    begin
        % Synchronize processes and distribute Φ
        send Φ to all processes;
        last := global state Σ^{0...0};
        release processes;
        remove all states in last that satisfy Φ;
        ℓ := 1;
        % Invariant: last contains all states of level ℓ − 1 that are reachable
        % from Σ^{0...0} without passing through a state satisfying Φ
        while (last ≠ { }) do
            current := states of level ℓ reachable from a state in last;
            remove all states in current that satisfy Φ;
            if current = final global state then return false
            ℓ := ℓ + 1;
            last := current
        od
        return true
    end ;
```

Figure 18. Algorithm for Detecting Definitely(Φ).

redundantly constructing the same global state of level $\ell + 1$ from different global states of level ℓ, but the computation can still be costly because the number of states of level ℓ grows as $O(\ell^{n-1})$ (for n constant).

Figure 18 gives the high-level algorithm used by the monitoring process p_0 to detect **Definitely**(Φ). This algorithm iteratively constructs the set of global states that have a level ℓ and are reachable from the initial global state without passing through a global state that satisfies Φ. If this set of states is empty, then **Definitely**(Φ) holds and if this set contains only the final global state then $\neg$ **Definitely**(Φ) holds. Note that, unlike detecting **Possibly**(Φ), not all global states need be examined. For example, in Figure 16, suppose that when the global states of level 2 were constructed, it was determined that Σ^{02} satisfied Φ. When constructing the states of level 3, global state Σ^{03} need not be included since it is reachable only through Σ^{02}.

The two detection algorithms are linear in the number of global states, but unfortunately the number of global states is $O(k^n)$ where k is the maximum number events a monitored process has executed. There are techniques that can be used to limit the number of constructed global states. For example, a process p_i need only send a message to p_0 when p_i potentially changes Φ or when p_i learns that p_j has potentially changed Φ. Another technique is for p_i to send an empty message to all other processes when p_i potentially changes Φ. These, and other techniques for limiting the number of global states are discussed in [22]. An alternative approach is to restrict the global predicate to one that can be efficiently detected, such as the conjunction and disjunction of local predicates [10].

15 Multiple Monitors

There are several good reasons for having multiple monitors observe the same computation for the purposes of evaluating the same predicate [14]. One such reason is increased performance—in a large system, interested parties may have the result of the predicate sooner by asking the monitor that is closest to them.[21] Another reason is increased reliability—if the predicate encodes the condition guarding a critical system action (e.g., shutdown of a chemical plant), then having multiple monitors will ensure the action despite a bounded number of failures.

The reactive-architecture solution to GPE based on passive observations can be easily extended to multiple monitors without modifying its general structure. The only change that is required is for the processes to use a causal broadcast communication primitive to notify the group of monitors [12]. In this manner, each monitor will construct a consistent, but not necessarily the same, observation of the computation. Each observation will correspond to a (possibly different) path through the global state lattice of the computation. Whether the results of evaluating predicate Φ by each monitor using local observations are meaningful depends on the properties of Φ. In particular, if Φ is stable and some monitor observes that it holds, then all monitors will eventually observe that it holds. For example, in the case of deadlock detection with multiple monitors, if any one of them detects a deadlock, eventually all of them will detect the same deadlock since deadlock is a stable property. They may, however, disagree on the identity of the process that is responsible for creating the deadlock (the one who issued the last request forming the cycle in the WFG^+).

Multiple observations for evaluating nonstable predicates create problems similar to those discussed in Section 14.2. There are essentially two possibilities for meaningfully evaluating nonstable predicates over multiple observations. First, the predicate can be extended using **Definitely** or **Possibly** such that it is made independent of the particular observation but becomes a function of the computation, which is the same for all monitors. Alternatively, the notification messages can be disseminated to the group of monitors in a manner such that they all construct the same observation. This can be achieved by using a *causal atomic broadcast* primitive that results in a unique total order consistent with causal precedence for all messages (even those that are concurrent) at all destinations [7,12].

Now consider the case where the monitor is replicated for increased reliability. If communication or processes in a distributed system are subject to failures, then sending the same notification message to all monitor replicas using causal delivery is not sufficient for implementing a causal broadcast primitive. For example if channels are not reliable, some of the notification messages may be lost such that different monitors effectively observe different computations. Again, we can accommodate communication and processes failures in our reactive architecture by using a *reliable* version of causal broadcast as the communication primitive [12]. Informally, a reliable causal broadcast, in addition to preserving causal precedence among message send events, guarantees delivery of a message either to all or none of the destination processes.[22] A formal specification of reliable causal broadcast in the presence of failures has to be done with care and can be found in [12].

Note that in an asynchronous system subject to failures, reliable causal broadcast is the best one can hope for in that it is impossible to implement communication primitives

21. In the limit, each process could act as a monitor such that the predicate could be evaluated locally.
22. Note that in [12], this primitive is called *casual broadcast* without the *reliable* qualifier since all broadcast primitives are specified in the presence of failures.

that achieve totally-ordered delivery using deterministic algorithms. Furthermore, in an environment where processes may fail, and thus certain events never get observed, our notion of a consistent global state may need to be re-examined. For some global predicates, the outcome may be sensitive not only to the order in which events are observed, but also to the order in which failures are observed by the monitors. In such cases, it will be necessary to extend the causal delivery abstraction to include not only actual messages but also failure notifications (as is done in systems such as ISIS [2]).

16 Conclusions

We have used the GPE problem as a motivation for studying consistent global states of distributed systems. Since many distributed systems problems require recognizing certain global conditions, the construction of consistent global states and the evaluation of predicates over these states constitute fundamental primitives with which one can build such systems.

We derived two classes of solutions to the GPE problem: a snapshot based one, that constructs a consistent global state and a reactive-architecture based one, that constucts a consistent run of the system. To derive these two classes, we have developed a set of basic concepts and mechanisms for representing and reasoning about computations in asynchronous distributed systems. These concepts represent generalizations of the notion of *time* in order to capture the uncertainty that is inherent in the execution of such systems. Two kinds of clocks were presented: *logical clocks* that generalize real-time clocks, and *vector clocks* that are incomparable to real-time clocks but exactly capture the causal precedence relation between events.

We illustrated the practicality of our concepts and mechanisms by applying them to distributed deadlock detection and distributed debugging. Reactive-architecture solutions based on passive observations were shown to be more flexible because they capture a run instead of just a global state. In particular, these solutions can be easily adapted to deal with nonstable predicates, multiple observations and failures. Each extension can be easily accommodated by using an appropriate communication primitive for notifications, leaving the overall reactive architecture unchanged.

Acknowledgments The material on distributed debugging is derived from joint work with Robert Cooper and Gil Neiger. We are grateful to them for consenting to its inclusion here. The presentation has benefited greatly from extensive comments by Friedemann Mattern, Michel Raynal and Fred Schneider on earlier drafts.

References

[1] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[2] K. Birman. The process group approach to reliable distributed computing. Technical Report TR91-1216, Department of Computer Science, Cornell University, January 1993. To appear in Communications of the ACM.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] Andrew D. Birrel and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[6] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, May 1991.

[7] Flaviu Cristian, H. Aghili, H. Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, Michigan, June 1985. A revised version appears as IBM Technical Report RJ5244.

[8] D. Dolev, J.Y. Halpern, and R. Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 504–511, April 1984.

[9] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Eleventh Australian Computer Science Conference*, pages 55–66, University of Queensland, February 1988.

[10] V. K. Garg and B. Waldecker. Unstable predicate detection in distributed programs. Technical Report TR-92-07-82, University of Texas at Austin, March 1992.

[11] V. Gligor and S. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, SE-6:435–440, September 1980.

[12] Vassos Hadzilacos and Sam Toueg. Reliable broadcast and agreement algorithms. In S.J. Mullender, editor, *Distributed Systems*, chapter 5. ACM Press, 1993.

[13] David Harel and Amir Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI, pages 477–498. Springer-Verlag, 1985.

[14] J. Helary, C. Jard, N. Plouzeau, and M. Raynal. Detection of stable properties in distributed applications. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 125–136, Vancouver, British Columbia, August 1987.

[15] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the amoeba distributed operating system. In *Proceedings of the Eleventh International Conference on Distributed Computer Systems*, pages 222–230, Arlington, Texas, May 1991. IEEE Computer Society.

[16] Hermann Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, pages 460–467, Yokohama, Japan, June 1992. IEEE Computer Society.

[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[18] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[19] Butler Lampson. Atomic transactions. In B. Lampson et. al., editor, *Distributed Systems – Architecture and Implementation: An Advanced Course*, volume 105 of *Lecture Notes on Computer Science*, pages 246–265. Springer-Verlag, 1981.

[20] H. M. Levy and E. D. Tempero. Modules, objects, and distributed programming: Issues in rpc and remote object invocation. *Software - Practice and Experience*, 21(1):77–90, January 1991.

[21] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[22] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *Proceedings of the Fifth International Workshop on Distributed Algorithms (WDAG-91)*, Lecture Notes on Computer Science. Springer-Verlag, Delphi, Greece, October 1991.

[23] Friedemann Mattern. Virtual time and global states of distributed systems. In Michel Cosnard et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, October 1989.

[24] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 1993. To appear.

[25] J. Misra. Distributed-discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[26] Carroll Morgan. Global and logical time in distributed algorithms. *Information Processing Letters*, 20:189–194, May 1985.

[27] Gil Neiger and Sam Toueg. Substituting for real time and common knowledge in asynchronous distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 281–293, Vancouver, British Columbia, August 1987.

[28] Larry L. Peterson, Nick C. Bucholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[29] M. Raynal. About logical clocks for distributed systems. *Operating Systems Review*, 26(1):41–48, January 1992.

[30] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, September 1991.

[31] A. Sandoz and A. Schiper. A characterization of consistent distributed snapshots using causal order. Technical Report 92-14, Departement d'Informatique, Ecole Polytechnique Federale de Lausanne, Switzerland, October 1992.

[32] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In J.-C. Bermond and M. Raynal, editors, *Proceedings of the Third International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes on Computer Science*, pages 219–232, Nice, France, September 1989. Springer-Verlag.

[33] Fred Schneider. Replication management using the state machine approach. In S.J. Mullender, editor, *Distributed Systems*, chapter 7. ACM Press, 1993.

[34] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the Holy Grail. Technical Report SFB124-15/92, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 1992.

[35] Kim Taylor. The role of inhibition in asynchronous consistent-cut protocols. In J.-C. Bermond and M. Raynal, editors, *Proceedings of the Third International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes on Computer Science*, pages 280–291. Springer-Verlag, Nice, France, September 1989.