Distributed Snapshots in Spite of Failures *

Amitabh Shah
Sam Toueg

TR 84-624
July 1984
(Revised February 1985)

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Partial support for this work was provided by the National Science Foundation under grant No. 83-03135.

Distributed Snapshots In Spite of Failures†

Amitabh Shah
Sam Toueg

Cornell University
Ithaca, New York 14853

ABSTRACT

An extension of the Chandy-Lamport algorithm ([Chan84]) to find global states of distributed systems is presented where benign failures of processes and channels are permitted. The scope of the algorithm in detecting stable properties in distributed systems is discussed. As an application, an algorithm to detect deadlocks in failure-prone distributed systems is presented.

1. Introduction

In absence of perfectly synchronized real-time clocks, a general notion of "consistency" in distributed systems has emerged. In terms of the basic primitives of distributed systems, viz. "send" and "receive", this notion of consistency translates to: the receive of a message can not precede the sending of that message under any temporal frame of reference.

Various mechanisms to achieve consistency have been proposed, the notable being: two-phase locking ([Eswa76]), logical time and partial ordering of events ([Lamp78]), checkpointing and rollback ([Russ80,Pres83]), distributed synchronization ([Schn82]), virtual time ([Jeff83]) etc. Chandy and Lamport ([Chan84]) have proposed a theory of stable property detection in a distributed system based on the partial ordering of events model of achieving consistency. They have given an algorithm to determine the global state of the system at some consistent point of computation. Stable properties like termination of computation, deadlocks etc., which by definition, persist through future computations of the system, can then be extracted out of the global state.

Their model of a distributed system assumes a totally failure-free environment. Real systems, however, are quite prone to failures. In this note, we extend the Chandy-Lamport algorithm to operate in a realistic environment and explore under what conditions stable properties can be detected. We assume the reader is familiar with the notions presented in [Chan84].

† Partial support for this work was provided by the National Science Foundation under grant MCS 83-03135.

2. Model of the system

A distributed system consists of processes $P_1, \cdots, P_n$ and directed channels connecting the processes. A channel c directed from P_i to P_j is denoted as (i,j). Processes do not share any memory but communicate via the channels which are assumed to have infinite buffers. The channels referred to here are virtual channels. Initially, the system is completely connected: there is a directed channel from every process to every other process.

In [Chan84] both, the processes and the channels, were assumed to be *failure free.* In addition, channels were assumed to be *loss-free, error-free* and *first-in-first-out (fifo)* . We assume only the last two in our model of the distributed system. We allow both the processes and the channels, to fail benignly. Processes can fail by halting and revive arbitrarily and channels can go down and come up, losing messages in transit.

A process is defined by its state and the events occurring on it. Events are of two types : **Autonomous** events - transitions in the process state that do not involve communication with other processes and **Communication** events which, further, are of two kinds : **send** a message to a process and **receive** a message from a process. (We also allow a process to send a message to itself and receive such a message; since such a message is internal to the process, we assume that it is not lost in transit unless the process itself fails.) Each event has with it, well-defined *before* and *after* states: the event can occur at a process only is the process is in the *before* state; the process is in the *after* state if the event occurs.

3. Definitions

We recall some definitions from [Chan84].

A **computation of a process** is a tuple $(s_0, <e_i>, i = 0,1,....)$ where s_0 is the initial state of the process and $<e_i>$ is the sequence of events occurring at the process. A **computation of the distributed system** is a set of computations of its component processes such that: the events are *partially ordered* by a " *happened before* " relation "$<$" on the events in the system ([Lamp78]). The relation "$<$" is defined as the closure of the following base relation:

$e_i < e_j$ if

1. both e_i and e_j occurred at P_k and e_i occurred before e_j, or
2. e_i is the sending of a message and e_j is the receipt of that message.

A **point** T in a distributed system computation is defined to be a set of events such that $\forall\ e, e' : e < e'$ and e' $\in$ T then e $\in$ T. A **cut** is defined as a list of events $(e_1^*, \ldots, e_n^*)$ where $\forall$ i, i=1, . . . , n, e_i^* is an event in process P_i's computation. A cut $(e_1^*, \ldots, e_n^*)$ is said to be **proper** if the function F defined by

$$F\ (\ (e_1^*, \ldots, e_n^*)\ ) = \bigcup_{i=1}^{n} \{\ e \mid e \text{ in } P_i\text{'s computation and not later than } e_i^*\}$$

is a point in the system computation. Proper cuts are also referred to as **consistent** cuts in literature and they represent the notion of consistency in our discussion. Since there is a one-one correspondence between points of computation and proper cuts, we shall refer to them interchangeably.

We represent the progress of the system by a directed event graph as in Figure 1.

The horizontal lines represent temporal progress of the constituent processes, the diagonal lines represent the inter-process communication, the points represent events and the direction of the arcs represents the partial order "<". We also allow processes to send messages to themselves and these are represented by curved arcs. An event $e < e'$ iff there is a path from e to e' in the event graph.

A cut c is then a partition of the event graph into two sets, $PAST_c$ and $FUTURE_c$. The cut c is proper if $FUTURE_c$ is closed under "<". In Figures 2a and 2b we show a proper cut c and an improper cut c' respectively.

In [Chan84] the global state of the system is determined at a point of computation or by a proper cut. Special messages called ***signals*** (as opposed to normal messages) are sent in the system. Failure-free environment guarantees that every process will eventually receive the signal. Moreover, the channels, being fifo, ensure that the cut $(e_1{}^*, \ldots, e_n{}^*)$ determined by the latest events $e_i{}^*$ ($\forall$ i= 1, . . . , n) before the receipt of the signals is a proper cut.

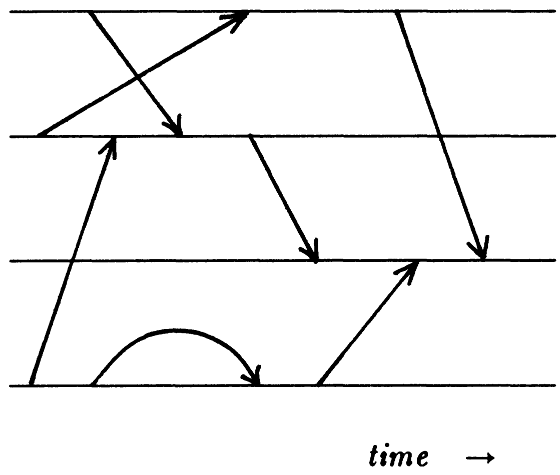

Figure 1. An event graph for a distributed system.

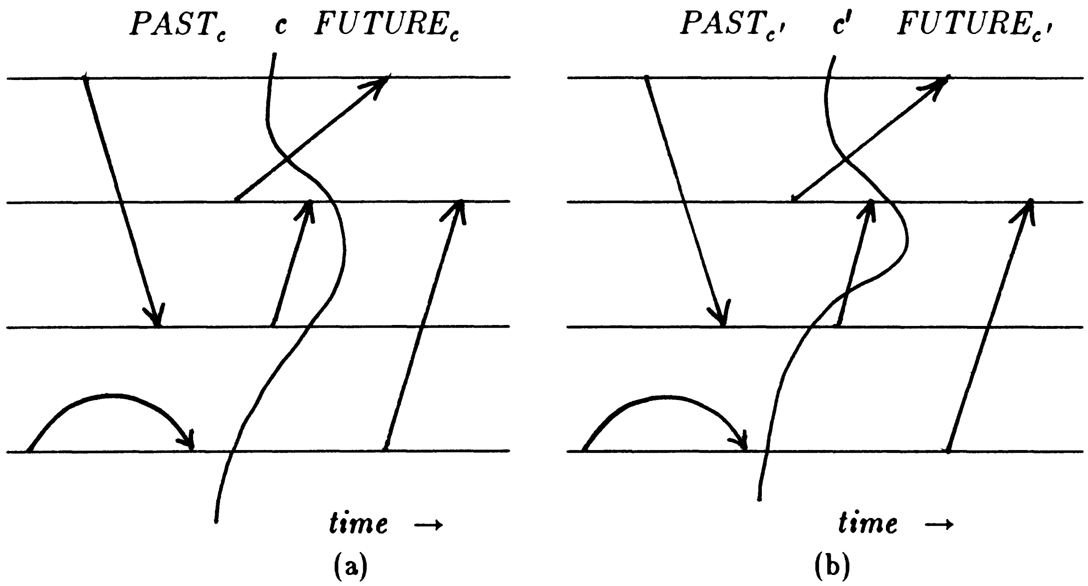

Figure 2. Proper and improper cuts in a distributed system.

Each process then takes a local snapshot of its state and of the states of its incoming channels. The global state of the system at this proper cut can then be constructed from the local distributed snapshots. Stable properties of the system can then be detected from the global state description.

In [Chan84] each process is made responsible for reporting the states of its incoming channels. However, it may be possible to give an algorithm in which each process is made responsible for its outgoing channels. This is due to the fact that there are no failures and no messages are lost. In our model, since messages could be lost, processes can report only what is observed in terms of messages received. Hence it is necessary that the processes are made responsible for their incoming channels.

The successful participation of each process in the algorithm was instrumental in determining the global state in [Chan84]. In a failure-prone environment this may not be possible : processes may fail or messages may be lost in transit. In particular, a proper cut may not be determined due to loss of signals. Hence we need to weaken the definition of a proper cut. Instead of dealing with consistency of the whole system, we now tackle pair-wise consistency between two processes.

We define a **two-process cut** for processes P_i and P_j to be a list (e_i, e_j), where e_i and e_j are events in processes P_i and P_j's computation respectively. Then a function C_{ij}

is defined for a two-process cut to capture all interaction between processes P_i and P_j:

C_{ij} ((e_i^*, e_j^*)) = { e | e is an event on P_i's computation not later than e_i^* or on P_j's computation not later than e_j^* and e is either an autonomous event or a communication event between P_i and P_j }

C_{ij} ((e_i^*, e_j^*)) is called a **point of interaction** between P_i and P_j if for all e, e' on P_i or P_j's computation: $e < e'$ and $e' \in C_{ij}((e_i^*, e_j^*)) \Rightarrow e \in C_{ij}((e_i^*, e_j^*))$.

We now define a ϕ-cut to be a list of events $(e_1^*, \ldots, e_n^*)$ where some of the e_i^*'s may be *unknown* and denoted as ϕ. We say that a ϕ-cut is **proper** if $\forall\, i,j, i \neq j$, $e_i^* \neq \phi$ and $e_j^* \neq \phi \Rightarrow C_{ij}((e_i^*, e_j^*))$ is a point of interaction. A proper ϕ-cut corresponds to a point of computation via the functions C_{ij}'s.

We define $first\_after_c\ (e_i)$ for a channel $c=(i,j)$ to be the first receive by P_j of a message sent by P_i after e_i along the channel c; if no such message is received then $first\_after_c(e_i)$ is defined to be ∞. This gives our first result which is a natural extension of Theorem 3 in [Chan84].

Theorem 1: A necessary and sufficient condition for a ϕ-cut to be proper is :
$\forall$ i,j, i $\neq$ j : for the channel $c=(i,j)$
$e_i^* \neq \phi$ and $e_j^* \neq \phi \Rightarrow first\_after_c(e_i^*) > e_j^*$.

Proof: Follows from the definitions of proper ϕ-cut and $first\_after_c$. □

4. Determining Global State

As in [Chan84] the global state $S(T)$ at a point of computation T is given by

$$S(T) = (P(T), C(T))$$

where $P(T)$ is the set of process states and $C(T)$ is the set of channel states at T. For a channel (i,j) the state in $C(T)$ is the sequence *seq* of messages that are still in the channel (i,j). In [Chan84], these are precisely the messages sent by P_i but not received by P_j. Since in our model messages could be lost, *seq* is potentially a subsequence of the sequence of messages that are sent by P_i and not received by P_j. In particular, *seq* satisfies

$$receiveseq_{ji}, seq \subset sendseq_{ij}, \qquad (*)$$

where $receiveseq_{ji}$ is the sequence of messages received by P_j from P_i, $sendseq_{ij}$ is the sequence of messages sent by P_i to P_j and "," is the juxtaposition operator.

In view of failures of processes or channels, the sets $P(T)$ and $C(T)$ may not contain all the entries corresponding to the processes and channels in the system, but in absence of any mishap at process P_i or process P_j or in the channel (i,j), we have an equality in the above relation as in [Chan84].

Thus, to determine the global state of the system at a point of computation, we need to determine the corresponding proper ϕ-cut $(e_1^*, \ldots, e_n^*)$. Then the state of a process P_j is the state immediately after e_j^* if $e_j^* \neq \phi$, else it is unknown. Further, we need to determine a sequence of messages, seq, that satisfies (*) for each incoming channel (i,j) of P_j if $e_j^* \neq \phi$. Note that since P_j is responsible for its incoming channels, if $e_j^* = \phi$, the states of these channels are undefined. In the next section, we describe an algorithm to determine the above.

5. The Algorithm and its Correctness

Figure 3 gives an algorithm to determine the global state of the system at a proper ϕ-cut. As described earlier, the ϕ-cut is determined by using special messages called *signals* which are propagated through the system. The signals are special in that they do not affect the underlying computation of the system; however, like normal messages, they can be lost in transit. We also allow processes to send $time-out$ messages to themselves; recall that time-outs are not lost unless the process activating them fails itself.

We associate a unique identifier with each instance of the algorithm. Without loss of generality, we assume that a given process, say P_i, initiates the algorithm. Thus, a numbering scheme suffices to identify the instances. Each signal carries with it the instance identifier. For reasons discussed later, we also stipulate that each normal message sent by a process also carries as a header, the number of the latest instance of the algorithm it participated in. Hence we use the notation $(message-type, instance-number)$ to denote messages sent by a process. If message-type is msg_k or $signal_k$ then it respectively denotes the receipt of a normal message or a signal from process P_k; message-type $time-out_k$ denotes receipt of a time-out corresponding to the process P_k. A local variable $inst_j$ is maintained to record the latest instance of the algorithm process P_j participated in.

Intuitively, the proper ϕ-cut determined by an instance of the algorithm is the list of events $(e_1^*, \ldots, e_n^*)$, where each e_i^* is the event on process P_i's computation immediately preceding the receipt of the first signal received by P_i during the execution. Since signal messages could be lost, we also need to piggyback the signals on normal messages by means of an instance identifier tag. Otherwise the determined ϕ-cuts could be improper. For example, consider the sequence of events in Figure 4. (We use dashed lines to represent signals, solid lines to represent normal messages and 'X' to denote failures.) Process P_i selects e_i^* and sends signals to processes P_j and P_k. The signal to P_j is lost.

Initiator P_i:

$inst_i \leftarrow inst_i + 1$;
select as e_i^* the latest event in its computation;
set the process state to be the current state;
send out $(signal_i, inst_i)$ to every other process;
start timers corresponding to each process.

$\forall j, P_j$:

On receipt of a message m:
if $(m = (signal_k, inst_k)) \vee (m = (msg_k, inst_k)) \rightarrow$
  if $inst_k > inst_j \rightarrow$
    $inst_j \leftarrow inst_k$;
    select as e_j^* the latest event in its computation;
    set process state to be the current state;
    set the state of channel (k,j) to be the null sequence;
    send $(signal_j, inst_j)$ to every other process;
    start timers corresponding to each process;
  | $inst_k = inst_j \rightarrow$
    set the state of channel (k,j) to be the sequence of messages
    received after e_j^* and before m;
    abort the timer corresponding to process k;
  | $inst_k < inst_j \rightarrow$ skip;
  fi;
| $m = (time\text{-}out_k, inst_j) \rightarrow$
  set the state of channel (k,j) to be the sequence of messages
  received after e_j^* and before m;
  discard all messages from P_k after the time-out until a $(signal_k, inst_k)$
  or $(msg_k, inst_k)$ is received with $inst_k \geq inst_j$;
fi;

Figure 3. The Algorithm

Subsequently, P_i sends message m to P_j which reaches safely. Process P_k relays the signal to P_j which then chooses e_j^*. But then we have an improper ϕ-cut since $m = first\_after_c(e_i^*) < e_j^*$ for the channel $c = (i, j)$. The algorithm overcomes this anomaly since m carries the instance number as a header and process P_j selects e_j^* before accepting m.

We now show the correctness of our algorithm.

Theorem 2: The algorithm in Figure 3 is correct, i.e.,

1. the algorithm terminates,
2. the cut $(e_1^*, \ldots, e_n^*)$ obtained is a proper ϕ-cut and
3. if in the ϕ-cut $(e_1^*, \ldots, e_n^*)$ $e_j^* \neq \phi$ then the algorithm determines the state of the process P_j and its incoming channels at that proper ϕ-cut.

Proof:

1. At most one signal is sent per channel and one timer is activated per signal. Either the signal will reach the receiver or the sender will time out. Eventually, the

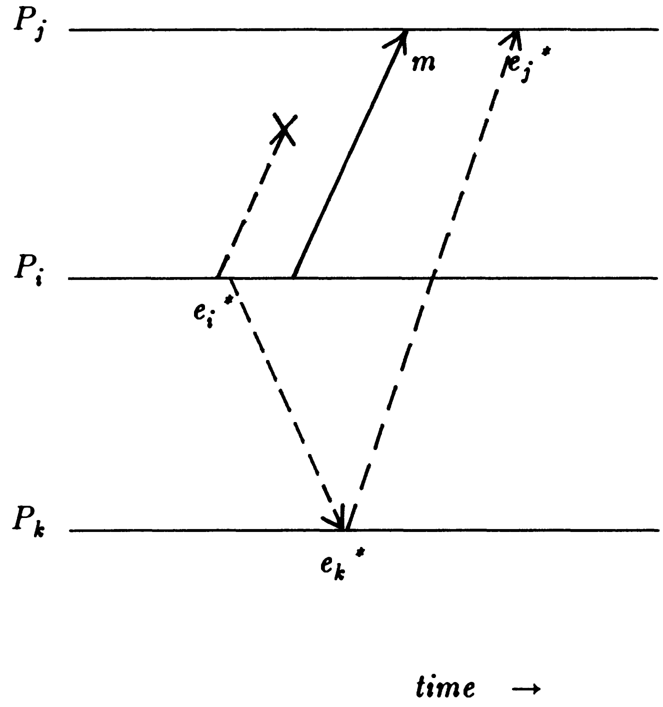

Figure 4. Improper ϕ-cuts due to loss of signals

initiator will get the signals echoed back to it or it will time out. Thus the algorithm terminates.

2. Let i and j be such that both e_i^* and e_j^* are not ϕ. If the signal sent by P_i along $c=(i,j)$ reached P_j then by the assumption of the channels being fifo, $first\_after_c(e_i^*) > e_j^*$. If the signal was lost in transit then on the receipt of $first\_after_c(e_i^*)$, P_j,if not already having done it, selects e_j^* before accepting the message. Hence, in either case, $first\_after_c(e_i^*) > e_j^*$ and thus by Theorem 1, the ϕ-cut determined is proper.

3. State of process P_j is the state immediately after e_j^* if $e_j^* \neq \phi$. Part 2 guarantees that the ϕ-cut is proper. Thus P_j's state is determined properly. It is easily verified that the states of P_j's incoming channels are determined correctly; in particular, the state of the channel (i,j), denoted as cs_{ij}, say, given by the algorithm satisfies

$$receiveseq_{ji}, cs_{ij} \subset sendseq_{ij}$$

where $e_j^* \neq \phi$. □

6. Discussion

We have used an ad-hoc mechanism of time-outs to force termination of our algorithm. This is necessary because we make no assumptions about process speeds. Since processes do not share any memory, a process P_i has no way of discerning whether another process P_j has failed or whether the channel (j,i) is down or P_j is incorrigibly slow (relative to P_i). Figures 5a, 5b and 5c show these three situations respectively. Reader may note that in all the three cases the behavior of the process P_i is identical from e_i^* to time-out.

On the other hand, since the mechanism is ad-hoc we have to deal with the problem of premature time-outs. We have resolved it by allowing a process P_i to discard messages from a process P_j if P_i has timed out while expecting a signal from P_j. It resumes accepting messages from P_j only if it receives a signal from the latter for the same instance of the algorithm or any later instance. Since messages could be lost in our model of the system, the discarded messages, such as m in Figure 5, are deemed as *lost*.

Note that discarding messages does not introduce any inconsistency in the global state. In fact, it is desirable to do so. For example, suppose that for the instance of the algorithm corresponding to Figure 5c, both e_i^* and e_j^* are not ϕ. Process P_j reports sending of message m in its state whereas process P_i reports the receipt of m in neither its

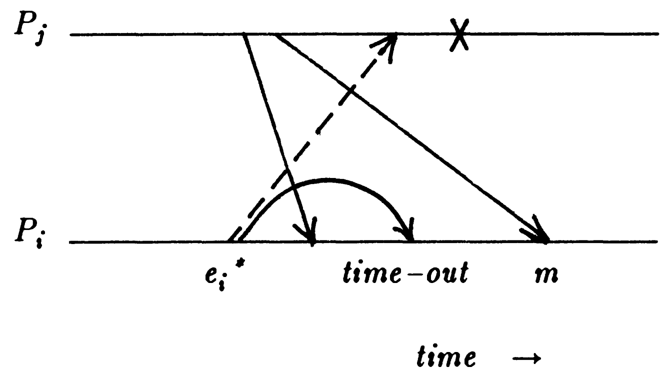

Figure 5a. P_j has failed.

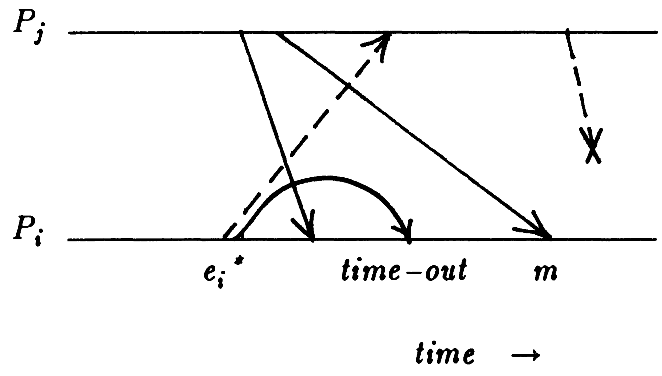

Figure 5b. Link (j,i) is down.

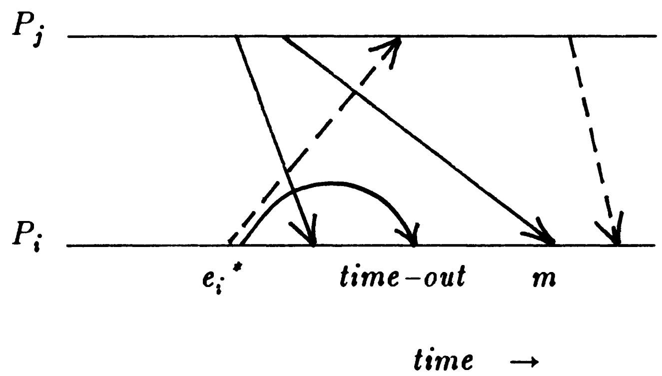

Figure 5c. P_j is slow.

state nor in the state of the channel (j,i). A third process P_k having gathered the global picture has no way of ascertaining deterministically whether m did reach P_i or was lost in transit. This ambiguity would be intolerable, for instance, if the message m was crucial to P_k. On the other hand, if P_i guarantees to refuse such messages, process P_k can safely decide its future course by assuming that these messages were lost.

7. Detecting Stable Properties

In [Chan84] the distributed snapshots taken at a proper cut are put together to obtain a consistent global state of the system. Stable properties, if exist, can be detected from this global picture. In our model, this detection may not always be possible. For example, if, due to failures, the system is partitioned into components at a proper ϕ-cut, an instance of the algorithm originating in one component can not detect stable properties, even if they exist, in another component.

What we can claim, however, is: a proper ϕ-cut defines the processes that are **reachable**† in this dynamically changing system at that ϕ-cut. These are the processes that have a non-ϕ entry in the ϕ-cut. Then a stable property can be detected at a proper ϕ-cut only if the processes involved in that stable property are reachable at that ϕ-cut.

The gathering of local snapshots to construct a global picture can be done in a number of ways. For example, the process and channel states can be reported to the initiator or to a specially designated process - a 'system manager' or broadcasting them to every other process. In the next section, we use our algorithm for detecting deadlocks in a distributed system where an echo strategy is used: a process propogates signals to every process other than its parent, the process from which it received the first signal. On receiving the signals back from all the other processes, it sends a signal back to the parent; the process and channel states are piggybacked onto the signal, along with those received from other processes. The initiator, being at the root of such an echo-tree, finally gets the local snapshots of all the processes reachable at the proper ϕ-cut.

8. A Deadlock Detection Algorithm

As an application of the fault-tolerant distributed snapshots algorithm proposed above, we describe a simple algorithm to detect deadlocks in a distributed system where process and channel failures are permitted. We consider a distributed system where processes are contending for common resources. A process may need to wait in its computation if it requires a resource that is being used by another process. Mutual waiting can lead to deadlocks in such systems. For the sake of illustration, we consider simple cyclic

† The definition of reachability is an operational one: the ϕ-cut defines processes that are reachable, not vice-versa.

deadlocks where a process waits for exactly one other process to release or grant a resource needed to proceed with its computation. We refer the reader to [Brac84] where more general deadlock situations like N-out-of-M request deadlocks are considered. Recall that deadlock amongst a group of processes is a stable property since it persists for all future computations of the system.

[Holt72] models such a system of processes by means of a Wait-For-Graph (WFG). This is a directed graph (V, E) where V is the set of nodes corresponding to the processes in the system and an arc (P_i, P_j) in E corresponds to a request that process P_i issued to process P_j to grant some resource. The presence of deadlocks can then be detected by checking the WFG for cycles.

For any system, whether centralized or distributed, where processes are contending for common resources, the WFG is a dynamically changing graph. But in the centralized case, the global state of the system can easily be determined at any time by freezing the system. A consistent snapshot of the WFG can then be taken which can be checked for cycles. In a distributed system, even in one with no failures, it is impossible to freeze all the system processes at the same time. Situations may arise where a process which was not deadlocked when it started a deadlock detection algorithm can deadlock during the execution of the algorithm and would be discovered as such. On the other hand, an algorithm may terminate without discovering a deadlock that occurred during its execution. Hence, in a failure-free distributed system, a deadlock detection algorithm can ensure only the following:

1. If a process is deadlocked at the time the algorithm is initiated, a deadlock will be detected by that invocation of the algorithm and
2. If the algorithm detects a process to be in a deadlock then that process is deadlocked at the time the algorithm terminates.

In a failure-prone environment, even condition 1 is too strong to ensure. For instance, if a process is involved in a deadlock with another process, the algorithm will not detect this deadlock if during its execution, the two processes are in different network partitions due to communication failures. Instead, we have the following weaker condition:

1′. If a process is deadlocked at the time it invokes the algorithm, a deadlock will be detected if it involves the processes that were reachable during that invocation of the algorithm.

Based on the above observations, our deadlock detection algorithm is as follows:

Each process P_i maintains the name of the process to which it has sent a request in a variable called $Request_i$. $Request_i$ is updated whenever P_i is granted the resource for its request or makes a new request. Whenever a process suspects that it is deadlocked, it initiates the distributed snapshots algorithm. Each local snapshot of a process consists of a description of $Request_i$.

Each process, on getting signals back from other processes in the network or timing out, sends a signal back to its parent and also piggybacks its local snapshot onto it along with the snapshots received from other processes. Finally, the initiator of the algorithm receives snapshots of the processes reachable at that proper ϕ-cut. It can then check for presence of request-cycles among the received *Request* values.

The correctness of the algorithm follows from the fact that the ϕ-cut is proper. Condition $1'$ now follows since the consistent local snapshots of reachable processes are available. Condition 2 follows from stability of deadlocks as a property of future computations.

In the algorithm we have presented above, a single process, viz. the initiator, gathers the consistent global state of the system and computes the stable property of deadlock from that state. Although conceptually simple, only the gathering of local state is distributed. Hence it carries with it some of the disadvantages of centralized systems like uneven distribution of computational and communication load, unreliability etc. For this reason, it is desirable to distribute the computation of stable properties as well. In [Brac84] such a distributed algorithm for deadlock detection is presented. The underlying model is failure-free, however. We are currently investigating extension of that algorithm to failure-prone environments.

9. Conclusion

We have extended the Chandy-Lamport algorithm to determine global states of a distributed system to operate in a realistic, failure-prone environment. We accomplish this by weakening the definition of proper or consistent cuts: instead of consistency of the whole system as such, we deal with pairwise consistency between those processes that are in some sense 'alive' and able to participate in the algorithm. The algorithm can be used to detect the stable properties which involve only those processes that were reachable during its execution. As an illustration, we have presented an algorithm for deadlock detection in failure-prone distributed systems.

Acknowledgement

We thank Dale Skeen for detailed comments on the paper and also for pointing out an error in an earlier version of the algorithm.

References

[Brac84]: Bracha, G. and S. Toueg., "A Distributed Algorithm for Generalized Deadlock Detection," *Proc. Third ACM Symposium on Principles of Distributed Computing,* Vancouver, Canada, August 1984.

[Chan84]: Chandy, K.M. and L. Lamport., "Detecting Stability in Distributed Systems," manuscript, 1984. A variant of this paper is to appear in *ACM Transactions on Computer Systems,*1985.

[Eswa76]: Eswaran, K.P., J.N. Gray, R.A. Lorie and I.L. Traiger., "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM,* Vol. 19, No. 11, Nov. 1976, pp. 624-633.

[Holt72]: Holt, R. C., "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys,* Vol. 4, No. 3, Sept. 1972, pp. 179-196.

[Jeff83]: Jefferson, D.R., "Virtual Time," *International Conference on Parallel Processing,* August 1983, pp. 384-394.

[Lamp78]: Lamport, L., "Time, Clocks and Ordering of Events in a Distributed System," *Communications of the ACM,* Vol. 21, No. 7, July 1978, pp. 558-565.

[Pres83]: Presotto, D.L., "PUBLISHING: A Reliable Broadcast Communication Mechanism," Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, Report No. UCB/CSD 83/165, Dec. 1983.

[Russ80]: Russell, D.L., "State Restoration in Systems of Communicating Processes," *IEEE Transactions on Software Engineering,* SE-6, No. 2, March 1980, pp. 183-194.

[Schn82]: Schneider, F.B., "Synchronization in Distributed Programs," *ACM Transactions of Programming Languages and Systems,* Vol. 4, No. 2, April 1982, pp. 179-195.