Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT

ELSEVIER

Information Processing Letters 87 (2003) 243–249

Information Processing Letters

www.elsevier.com/locate/ipl

Distributed algorithm to detect strong conjunctive predicates

Punit Chandra, Ajay D. Kshemkalyani *

Computer Science Department, University of Illinois at Chicago, Chicago, IL 60607, USA

Received 21 November 2002; received in revised form 24 March 2003

Communicated by K. Iwama

Abstract

This paper presents an on-line distributed algorithm for detection of *Definitely*(ϕ) for the class of conjunctive global predicates. The only known algorithm for detection of *Definitely*(ϕ) uses a centralized approach. A method for decentralizing the algorithm was also given, but the work load is not fairly distributed and the method uses a hierarchical structure. The centralized approach has a time, space, and total message complexity of $O(n^2m)$, where n is the number of processes and m is the maximum number of messages sent by any process. The proposed on-line distributed algorithm uses the concept of intervals rather than events, and assumes p is the maximum number of intervals at any process. The worst-case time complexity across all the processes is $O(\min(pn^2, mn^2))$. The worst-case space overhead across all the processes is $\min(2mn^2, 2pn^2)$.

Keywords: Distributed computing; Predicate detection; Causality; Global state

1. Introduction

Predicate detection in a distributed system is important for various purposes such as monitoring, synchronization and coordination, debugging, and industrial process control. Cooper and Marzullo [2] and Marzullo and Neiger [12] defined two modalities under which a predicate can hold for a distributed execution.

- *Possibly*(ϕ): There exists a consistent observation of the execution such that ϕ holds in global state of the observation.
- *Definitely*(ϕ): For every consistent observation of the execution, there exists a global state of it in which ϕ holds.

Possibly(ϕ) and *Definitely*(ϕ) have also been referred to as the weak and strong modalities for predicate ϕ, respectively, in the literature [5,6]. Marzullo et al. [2,12] proposed an online centralized algorithm to detect *Possibly*(ϕ) and *Definitely*(ϕ) for an arbitrary predicate ϕ. The algorithm works by building a lattice of global states. Although it detects generalized global predicates, the complexity of the algorithm is e^n, where e is the maximum number of events on any process, and n is the number of processes. To reduce the complexity of the algorithm, researchers focused on special classes of global predicates. Conjunctive global predicates is such class. Several researchers have presented polynomial time algorithms for this

* Corresponding author.
E-mail addresses: pchandra@cs.uic.edu (P. Chandra), ajayk@cs.uic.edu (A.D. Kshemkalyani).

doi:10.1016/S0020-0190(03)00295-3

class of global predicates. Garg and Waldecker [5,6] presented centralized algorithms to detect *Possibly*(ϕ) and *Definitely*(ϕ) with message space, storage, and time complexity of $O(n^2m)$, where m is the maximum number of messages sent by any process. Stoller and Schneider [14] presented an algorithm which combines the Garg–Waldecker [5] approach with the approach of Marzullo et al. [2,12], and thus has the best of both the approaches.

Distributed algorithms to detect *Possibly*(ϕ) have been presented by Garg and Chase [4] and Hurfin et al. [7]. Both the algorithms have message space, storage and time complexity of $O(n^2m)$. There does not exist any distributed algorithm to detect *Definitely*(ϕ), which is a much harder problem than detecting *Possibly*(ϕ). In [6], Garg and Waldecker gave a decentralized approach for detecting *Definitely*(ϕ). The decentralized approach divides the set of processes into multiple groups with a checker process for each group. The checker process uses the centralized algorithm to check for a strong conjunctive predicate within its group. It then sends selected information about a partial potential solution to a higher process in the hierarchy. This process is repeated at all levels until the final solution is found at the top of the hierarchy. The problems with this technique stem from the fact that the workload is not uniformly distributed. The checker process still uses a centralized algorithm within its group. Further, due to the hierarchical structure of the algorithm, this can not be considered truly distributed.

We present an on-line distributed algorithm for detection of *Definitely*(ϕ) that avoids the above problems. The algorithm uses the concept of intervals rather than events, and assumes p is the maximum number of intervals at any process. The worst-case space overhead across all the processes is $\min(2pn^2, 2mn^2)$. This is equivalent to $\min(2pn, 2mn)$ per process if the destinations of the mn messages are evenly divided among the n processes. The worst-case space overhead at a process is $\min(2pn, 2mn(n-1))$. The worst-case time complexity across all the processes is $O(\min(pn^2, mn^2))$. This is equivalent to $O(\min(pn, mn))$ per process if the destinations of the mn messages are evenly divided among the n processes. The worst-case time complexity at a process is $O(\min(pn, mn^2))$. The algorithm uses at most $O(\min(pn^2, mn^2))$ number of messages and has a worst-case message space overhead of $O(\min(pn^2, mn^2))$.

2. System model and background

We assume an asynchronous distributed system in which n processes communicate by reliable message passing. Messages may be delivered out of order on the channels. A poset event structure model $(E, \prec)$, where $\prec$ is an irreflexive partial ordering representing the causality relation [10] on the event set E, is used as the model for a distributed system execution. Three kinds of events are considered: send, receive, and internal events. E is partitioned into local executions at each process. Let N denote the set of all processes. Each E_i is a totally ordered set of events executed by process P_i. We assume vector clocks are available [3, 13]. Each process maintains a vector clock V of size $n = |N|$ integers, by using the following rules.

(1) Before an internal event at process P_i, the process P_i executes $V_i[i] = V_i[i] + 1$.
(2) Before a send event at process P_i, the process P_i executes $V_i[i] = V_i[i] + 1$. It then sends the message timestamped by V_i.
(3) When process P_j receives a message with timestamp T from process P_i, P_j executes $(\forall k \in [1, \ldots, n])$ $V_j[k] = \max(V_j[k], T[k])$; $V_j[j] = V_j[j] + 1$ before delivering the message. The timestamp of an event is the value of the vector clock when the event occurs.

A *conjunctive predicate* is of the form $\bigwedge_i \phi_i$, where ϕ_i is a predicate defined on variables local to process P_i. Let $\phi_{i,j}$ denote $\phi_i \wedge \phi_j$. Let us define durations of interest at each process as the durations in which the local predicate is true. Such an interval at process P_i is identified by the (totally ordered) subset of adjacent events of E_i for which the predicate is true. We use $V_i^-(X)$ and $V_i^+(X)$ to denote the vector timestamp for interval X at process P_i at the start and the end of an interval, respectively.

We assume that intervals X and Y occur at P_i and P_j, respectively, and are denoted as X_i and Y_j, respectively. We also assume that there are a maximum of p intervals at any process. For any two intervals X and X' that occur at the same process, if X ends before

X' begins, then we say that X is a *predecessor* of X' and X' is a *successor* of X.

For intervals X and Y, Lamport defined the following relation [11]: $X \hookrightarrow Y$ iff $\exists x \in X, \exists y \in Y, x \prec y$. The relation $\hookrightarrow$ is used by our algorithm to detect *Definitely*(ϕ). In terms of vector timestamps, $X_i \hookrightarrow Y_j$ if $V_i^-(X_i)[i] \leqslant V_j^+(Y_j)[i]$.

3. Algorithm to detect *Definitely*(ϕ)

The vector timestamps of the start of and of the end of an interval form a data type *Log*, as shown in Fig. 1. When an interval completes at process P_i, the interval's *Log* is added to a local queue Q_i selectively, based on a criterion explained later. The processes collectively run a token-based algorithm to process the queues.

The following two results given on p. 297 of [8] and in [9] are used in the context of detecting *Definitely*(ϕ).

Theorem 1. *Definitely*($\phi_{i,j}$) *holds if and only if* $X_i \hookrightarrow Y_j$ *and* $Y_j \hookrightarrow X_i$.

Theorem 2. *For a conjunctive predicate* ϕ, *Definitely*(ϕ) *holds if and only if Definitely*($\phi_{i,j}$) *is true for all process pairs* P_i *and* P_j *in* N.

In order for a distributed algorithm to process the queued intervals efficiently, we first show an important result about when two given intervals may potentially be a part of the solution.

Lemma 1. *For intervals* X_i *and* Y_j *at the head of* Q_i *and* Q_j, *respectively, if* $X_i \not\hookrightarrow Y_j$ *then interval* Y_j *should be dequeued from the queue* Q_j.

Proof. From the definition of $\hookrightarrow$, we get that $V_i^-(X)[i] \nleqslant V_j^+(Y)[i]$. For any interval X' which succeeds interval X, $V_i^-(X)[i] < V_i^-(X')[i]$, thus $V_i^-(X')[i] \nleqslant V_j^+(Y)[i]$, which implies $X' \not\hookrightarrow Y$. So Y can never be a part of the solution and should be deleted from the queue. □

Lemma 2. *If Definitely*($\phi_{i,j}$) *does not hold for interval pair* X_i *and* Y_j *at the head of* Q_i *and* Q_j, *respectively, then either interval* X_i *or interval* Y_j *can be removed from its queue* Q_i *or* Q_j, *respectively.*

Proof. As *Definitely*($\phi_{i,j}$) is not true, from Theorem 1 either $X \not\hookrightarrow Y$ or $Y \not\hookrightarrow X$. Hence by Lemma 1, either X or Y is deleted corresponding to these cases. □

Based on Theorems 1 and 2, we state our problem in terms of detecting *Definitely*($\phi_{i,j}$) for pairs of processes, along the lines of detecting pairwise orthogonal relations [1].

Problem statement. In a distributed execution, identify a set of intervals $\mathcal{I}$ containing one interval from each process, such that (i) the local predicate ϕ_i is true in $I_i \in \mathcal{I}$, and (ii) for each pair of processes P_i and P_j, *Definitely*($\phi_{i,j}$) holds, i.e., $I_i \hookrightarrow I_j$ and $I_j \hookrightarrow I_i$.

Before presenting the algorithm, we justify why the *Log* of an interval is stored in the local queue conditionally, as shown in Fig. 1. An interval Y at P_j is deleted if on comparison with some interval X on P_i, $X \not\hookrightarrow Y$, i.e., $V_i^-(X)[i] \nleqslant V_j^+(Y)[i]$. Thus the interval ($Y$) being deleted or retained depends on its value of $V_j^+(Y)[i]$. The value $V_j^+(Y)[i]$ changes only when a message is received. Hence an interval needs to be stored only if a receive has occurred since the last time a *Log* of a local interval was queued.

The token-based algorithm uses three types of messages (see Fig. 2) that are sent among the processes. Request messages of type *REQUEST*, reply messages of type *REPLY*, and token messages of type *TOKEN*, are denoted *REQ*, *REP*, and T, respectively. In the algorithm (see Fig. 3), only the token-holder process can send *REQ*s and receive *REP*s. The process (P_i) having the token sends *REQ*s to all other processes (line 3f). $Log_i.start[i]$ and $Log_i.end[j]$ for

---

type *Log*
  start: array$[1 \ldots n]$ of integer;
  end: array$[1 \ldots n]$ of integer;

type Q: queue of *Log*;

When an interval begins:
$Log_i.start = V_i^-$.
When an interval ends:
$Log_i.end = V_i^+$
if (a receive event has occurred since the last time
  a *Log* was queued on Q_i) **then**
 Enqueue Log_i on to the local queue Q_i.

Fig. 1. Tracking intervals locally at process P_i.

type *REQUEST*	//used by P_i to send a request to each P_j
start: integer;	//contains $Log_i.start[i]$ for the interval at the queue head of P_i
end: integer;	//contains $Log_i.end[j]$ for the interval at the queue head of P_i, when sending to P_j
type *REPLY*	//used to send a response to a received request
updated: set of integer;	//contains the indices of the updated queues
type *TOKEN*	//used to transfer control between two processes
updatedQueues: set of integer;	//contains the index of all the updated queues

Fig. 2. Data types used by messages.

(1) **Process P_i initializes local state**
(1a) Q_i is empty.

(2) **Token initialization**
(2a) A randomly elected process P_i holds the token T.
(2b) $T.updatedQueues = \{1, 2, \ldots, n\}$.

(3) *RcvToken*: **When P_i receives a token T**
(3a) Remove index i from $T.updatedQueues$
(3b) **wait until** (Q_i is nonempty)
(3c) $REQ.start = Log_i.start[i]$, where Log_i is the log at head of Q_i
(3d) **for** $j = 1$ to n
(3e) $REQ.end = Log_i.end[j]$
(3f) Send the request *REQ* to process P_j
(3g) **wait until** (REP_j is received from each process P_j)
(3h) **for** $j = 1$ to n
(3i) $T.updatedQueues = T.updatedQueues \cup REP_j.updated$
(3j) **if** ($T.updatedQueues$ is empty) **then**
(3k) Solution detected. Heads of the queues identify intervals that form the solution.
(3l) **else**
(3m) **if** ($i \in T.updatedQueues$) **then**
(3n) dequeue the head from Q_i
(3o) Send token to P_k where k is randomly selected from the set $T.updatedQueues$.

(4) *RcvReq*: **When a *REQ* from P_i is received by P_j**
(4a) **wait until** (Q_j is nonempty)
(4b) $REP.updated = \emptyset$
(4c) $Y =$ head of local queue Q_j
(4d) $V_i^-(X)[i] = REQ.start$ and $V_i^+(X)[j] = REQ.end$
(4e) Determine $X \hookrightarrow Y$ and $Y \hookrightarrow X$
(4f) **if** ($Y \not\hookrightarrow X$) **then** $REP.updated = REP.updated \cup \{i\}$
(4g) **if** ($X \not\hookrightarrow Y$) **then**
(4h) $REP.updated = REP.updated \cup \{j\}$
(4i) Dequeue Y from local queue Q_j
(4j) Send reply *REP* to P_i.

Fig. 3. Distributed algorithm to detect *Definitely*(ϕ).

the interval at the head of the queue Q_i are piggybacked on the request *REQ* sent to process P_j (lines 3c–3e). On receiving a *REQ* from P_i, process P_j compares the piggybacked interval X with the interval Y at the head of its queue Q_j (line 4e). As per Lemma 1, the comparisons between intervals on process P_i and P_j can result in these outcomes.

(1) *Definitely*$(\phi_{i,j})$ is satisfied.
(2) *Definitely*$(\phi_{i,j})$ is not satisfied and interval X can be removed from the queue Q_i. The process index i is stored in *REP*.*updated* (line 4f).
(3) *Definitely*$(\phi_{i,j})$ is not satisfied and interval Y can be removed from the queue Q_j. The interval at the head of Q_j is dequeued and process index j is stored in *REP*.*updated* (lines 4g, 4h).

Note that outcomes (2) and (3) may occur together. After the comparisons, P_j sends *REP* to P_i. Once the token-holder process P_i receives a *REP* from all other processes, it stores the indices of all the updated queues in the set T.*updatedQueues* (lines 3h, 3i). A solution, identified by the set $\mathcal{I}$ formed by the interval I_k at the head of each queue Q_k, is detected if the set *updatedQueues* is empty. Otherwise, if index i is contained in T.*updatedQueues*, process P_i deletes the interval at the head of its queue Q_i (lines 3m, 3n). As the set T.*updatedQueues* is non-empty, the token is sent to a process selected randomly from the set (line 3o). We now prove the correctness of the algorithm.

Lemma 3. *If Definitely*$(\phi_{i,j})$ *is not true for a pair of intervals* X_i *and* Y_j*, then either* i *or* j *is inserted into* T.*updatedQueues.*

Proof. From Lemma 2, if *Definitely*$(\phi_{i,j})$ is not satisfied, then either X_i or Y_j should get deleted. In the algorithm of Fig. 3, the test on either (line 4f) or (line 4g) will be true. Hence, either i or j is inserted in *REP*.*updated* which is later merged into T.*updatedQueues* (line 3i). □

Lemma 4. *An interval is deleted from queue* Q_i *at process* P_i *if and only if the index* i *is inserted into* T.*updatedQueues.*

Proof. When comparing two intervals X and Y at P_j, Y is deleted (line 4i) if and only if j is inserted into *REP*.*updated* (line 4h) (which is later merged into T.*updatedQueues*) as (lines 4h, 4i) are the part of the same **if** block. Similarly X is deleted (line 3n) if and only if $i \in T$.*updatedQueues* (lines 3m, 3n). □

Theorem 3. *When a solution* $\mathcal{I}$ *is detected by the algorithm, the solution is correct, i.e., for each pair* $P_i, P_j \in N$*, the intervals* $I_i = head(Q_i)$ *and* $I_j = head(Q_j)$ *are such that* $I_i \hookrightarrow I_j$ *and* $I_j \hookrightarrow I_i$ (*and hence by Theorems* 1 *and* 2*, Definitely*(ϕ) *must be true*).

Proof. It is sufficient to prove that for the solution detected, which happens at the time T.*updatedQueues* is empty (line 3j), (i) *Definitely*$(\phi_{i,j})$ is satisfied for all pairs (i, j), and (ii) none of the queues is empty. To prove (i) and (ii), note that when T.*updatedQueues* is empty (line 3j), the token must have visited each process at least once because only the token-holder's index can be removed from T.*updatedQueues.* Further, note that each (i, j) pair has been tested at least once for *Definitely*$(\phi_{i,j})$ when the solution is detected.

To prove (i), it follows from Lemma 3 that *Definitely*$(\phi_{i,j})$ is satisfied for all pairs (i, j) when T.*updatedQueues* $= \emptyset$. For any (i, j) pair, consider the latest time $t_{i,j}$ when the given (i, j) pair was tested. To prove (ii), it remains to show that between $t_{i,j}$ and the time that the solution is declared when T.*updatedQueues* $= \emptyset$, none of these intervals compared at $t_{i,j}$ is dequeued. If one of these intervals were to get dequeued, then by Lemma 4, that process index (say, i) would get inserted in T.*updatedQueues* and the token would have to re-visit that process P_i, resulting in another test for (i, j), a contradiction. The result follows. □

Theorem 4. *If a solution* $\mathcal{I}$ *exists, i.e., for each pair* $P_i, P_j \in N$*, the intervals* I_i, I_j *belonging to* $\mathcal{I}$ *are such that* $I_i \hookrightarrow I_j$ *and* $I_j \hookrightarrow I_i$ (*and hence from Theorems* 1 *and* 2*, Definitely*(ϕ) *must be true*)*, then the solution is detected by the algorithm.*

Proof. Consider the n intervals, one at each process, that form the *first* solution. We prove using contradiction that none of these intervals gets deleted. Assume that interval I_j is the *first* interval forming a part of the solution to get deleted. We then have $I_i' \not\hookrightarrow I_j$, which implies that some predecessor interval I_i of I_i' must form part of the solution with I_j and thus satisfy $(I_i \hookrightarrow I_j \wedge I_j \hookrightarrow I_i)$. But this implies I_i *already* got deleted in some earlier test, because intervals at each process are examined and deleted serially in the order of their occurrence. This is a contradiction. Hence, no interval forming a part of the solution gets deleted.

Observe from (line 3a) that for each hop of the token, the size of T.*updatedQueues* decreases by 1 if no interval is deleted from any queue in the ensuing *REQ-REP* phase (refer Lemma 4). It follows that in at most each $(n-1)$ consecutive hops, the interval at the head of the queue of some process must get replaced by the immediate successor interval at that process, otherwise T.*updatedQueues* becomes empty and a solution gets detected. This guarantees progress and within a finite number of steps, the interval from each process forming a part of the first solution will be at the head of the corresponding queue. As no such interval gets deleted, within $|T.updatedQueues|$ hops of the token after this state, the solution gets detected. □.

3.1. Complexity analysis

The complexity analysis can be done in terms of two parameters—the maximum number of messages sent per process (m) and the maximum number of intervals per process (p).

3.1.1. Space complexity at P_1 to P_n

(1) *In terms of* m: From Fig. 1, observe that the *Log* for an interval is stored on the queue only if a receive has occurred since the last time a *Log* for an interval was stored on the queue (at the same process). As there are a total of nm messages exchanged between all processes, a total of nm interval *Log*s are stored across all the queues, though not necessarily at the same time.
   - As the vector timestamp at the start (V^+) and at the end (V^-) of each interval is stored in each *Log* and there are a total of mn *Log*s stored on the various process queues, the worst-case space overhead across all processes is $mn \cdot 2n = 2mn^2$.
   - For a process, the worst case occurs when it receives m messages from all the other $n-1$ processes. The number of *Log*s stored on the process queue is $m(n-1)$, one *Log* for each receive event. As each *Log* contains two vector timestamps, the worst-case space at the process is $m(n-1)\cdot 2n = \mathrm{O}(mn^2)$.

   Note that the worst case just discussed is for a single process. The worst-case system-wide space overhead always remains $2mn^2$.

(2) *In terms of* p: The total number of *Log*s stored at each process is p because in the worst case, the *Log* for each interval may need to be stored. As each *Log* has size $2n$, the worst-case overhead is $2np$ integers over all *Log*s per process, and the worst-case space complexity across all processes is $2n^2p = \mathrm{O}(n^2p)$.

As the total number of *Log*s stored on all the processes is $\min(np, mn)$, the worst-case space overhead across all the processes is $\min(2n^2p, 2n^2m)$. This is equivalent to $\min(2np, 2nm)$ per process if the mn message destinations are divided equally among the processes (implying that each process has up to $\min(p, m)$ *Log*s). The worst-case space overhead at a process is $\min(2np, 2n(n-1)m)$.

3.1.2. Time complexity

The two components contributing to time complexity are *RcvReq* and *RcvToken*.

RcvReq: In the worst case, the number of *REQ*s received by a process is equal to the number of *Log*s on all other processes, because a *REQ* is sent only once for each *Log*. The total number of *Log*s over all the queues is $\min(np, mn)$ (see Section 3.1.1), hence the number of interval pairs compared per process is $\min((n-1)p, m(n-1))$. As it takes $\mathrm{O}(1)$ time to execute *RcvReq*, the worst-case time complexity per process for *RcvReq* is $\mathrm{O}(\min(np, mn))$. As the processes execute *RcvReq* in parallel, this is also the total time complexity for *RcvReq*.

RcvToken: The token makes at most $\min(np, mn)$ hops serially and each hop requires $\mathrm{O}(n)$ time complexity. Hence the worst-case time complexity for *RcvToken* across all processes is $\mathrm{O}(\min(pn^2, mn^2))$. In the worst case, a process receives the token each time its queue head is deleted, and this can happen as many times as the number of *Log*s at the process. As the number of *Log*s at a process is $\min(p, m(n-1))$, the worst-case time complexity per process is $\mathrm{O}(\min(pn, mn^2))$.

The worst-case time complexity across all the processes is $O(\min(pn^2, mn^2))$. This is equivalent to $O(\min(pn, mn))$ per process if the mn message destinations are divided equally among the processes (implying that each process has up to $\min(p, m)$ *Log*s). The worst-case time complexity at a process is $O(\min(pn, mn^2))$.

3.1.3. Message complexity

For each *Log*, either no messages are sent, or $n - 1$ *REQ*s, $n - 1$ *REP*s and one token T are sent.

- As the total number of *Log*s over all the queues is $\min(np, mn)$, hence the worst-case number of messages over all the processes is $O(n \min(np, mn))$.
- The size of each T is equal to $O(n)$, while the size of each *REP* and each *REQ* is $O(1)$. Thus for each *Log*, the message space overhead is $O(n)$ if any messages are sent for that *Log*. Hence the worst-case message space overhead over all the processes is equal to $O(n \min(np, mn))$.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9875617.

References

[1] P. Chandra, A.D. Kshemkalyani, Detection of orthogonal interval relations, in: Proc. 9th International Conference on High Performance Computing, in: Lecture Notes in Comput. Sci., Vol. 2552, Springer, Berlin, 2002, pp. 323–333.

[2] R. Cooper, K. Marzullo, Consistent detection of global predicates, in: Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, May 1991, pp. 163–173.

[3] C.J. Fidge, Timestamps in message-passing systems that preserve partial ordering, Australian Comput. Sci. Comm. 10 (1) (1988) 56–66.

[4] V. Garg, C. Chase, Distributed algorithms for detecting conjunctive predicates, in: Proc. 15th IEEE International Conference on Distributed Computing Systems, 1995, pp. 423–430.

[5] V.K. Garg, B. Waldecker, Detection of weak unstable predicates in distributed programs, IEEE Trans. Parallel Distrib. Systems 5 (3) (1994) 299–307.

[6] V.K. Garg, B. Waldecker, Detection of strong unstable predicates in distributed programs, IEEE Trans. Parallel Distrib. Systems 7 (12) (1996) 1323–1333.

[7] M. Hurfin, M. Mizuno, M. Raynal, M. Singhal, Efficient distributed detection of conjunctions of local predicates, IEEE Trans. Software Engrg. 24 (8) (1998) 664–677.

[8] A.D. Kshemkalyani, Temporal interactions of intervals in distributed systems, J. Comput. System Sci. 52 (2) (1996) 287–298.

[9] A.D. Kshemkalyani, A fine-grained modality classification for global predicates, IEEE Trans. Parallel Distrib. Systems (2003), to appear; Technical Report UIC-EECS-00-10, University of Illinois at Chicago, 2000.

[10] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Comm. ACM 21 (7) (1978) 558–565.

[11] L. Lamport, On interprocess communication, Part I: Basic Formalism; On interprocess communication, Part II: Algorithms, Distrib. Comput. 1 (1986) 77–85; 86–101.

[12] K. Marzullo, G. Neiger, Detection of global state predicates, in: Proc. 5th Workshop on Distributed Algorithms, in: Lecture Notes in Comput. Sci., Vol. 579, Springer, Berlin, October 1991, pp. 254–272.

[13] F. Mattern, Virtual time and global states of distributed systems, in: Parallel and Distributed Algorithms, North-Holland, 1989, pp. 215–226.

[14] S. Stoller, F. Schneider, Faster possibility detection by combining two approaches, in: Proc. 9th International Workshop on Distributed Algorithms, in: Lecture Notes in Comput. Sci., Vol. 972, Springer, Berlin, 1995, pp. 318–332.