Global State Detection Based on Peer-to-Peer Interactions

Punit Chandra and Ajay D. Kshemkalyani

Computer Science Department, Univ. of Illinois at Chicago,
Chicago, IL 60607, USA
{pchandra, ajayk}@cs.uic.edu

Abstract. This paper presents an algorithm for global state detection based on peer-to-peer interactions. The interactions in distributed systems can be analyzed in terms of the peer-to-peer pairwise interactions of intervals between processes. This paper examines the problem: "If a global state of interest to an application is specified in terms of the pairwise interaction types between each pair of peer processes, how can such a global state be detected?" Devising an efficient algorithm is a challenge because of the overhead of having to track the intervals at different processes. We devise a distributed on-line algorithm to efficiently manage the distributed data structures and solve this problem. We prove the correctness of the algorithm and analyze its complexity.

1 Introduction

The pairwise interaction between processes is an important way of information exchange even in pervasive systems and large distributed systems such as peer-to-peer networks [12, 13] that do collaborative computing. We observe that the pairwise interactions of processes form a *basic building block* for information exchange. This paper advances the state-of-the-art in analyzing this building block by integrating it into the analysis of the dynamics of (i) global information exchange, and (ii) the resulting global states of a distributed system [5]. The study of global states and their observations, first elegantly formalized by Chandy and Lamport [5], is a fundamental problem in distributed computing [5, 9].

Many applications in a distributed peer-to-peer system inherently identify local durations or *intervals* at processes during which certain application-specific local predicates defined on local variables are true in a system execution [1]. Hence, we require a way to *specify* how durations at different processes are related to one another, and also a way to *detect* whether the specified relationships hold in an execution. The formalism and axiom system formulated in [7] identified a complete *orthogonal* set $\Re$ of 40 causality-based fine-grained temporal interactions (or relationships) between a pair of intervals to *specify* how durations at two peer processes are related to one another. The following problem **DOOR** for the <u>D</u>etection <u>o</u>f <u>O</u>rthogonal <u>R</u>elations was formulated and addressed in [1]. *"Given a relation $r_{i,j}$ from $\Re$ for each pair of processes i and j, devise*

Table 1. Space, message and time complexities. Note: M = maximum queue length at P_0, the central server. $p \geq M$, as all the intervals may not be sent to P_0.

Centralized algorithm	Average time complexity at P_0	Total number of messages	Space at P_0 (= total message space)	Average space at P_i, $i \in [1, n]$
Fine_Rel [3,4]	$O(n^2M)$ or $O(n[min(4m, np)])$	$O(min(4m, np))$	$O(min[(4n-2)np, 10nm])$	$O(n)$

Distributed Algorithms	Average time complexity/proc.	Total number of messages	Total average message space	Total space
Algorithm [1]	$O(min(np, 4mn))$	$O(n \cdot min(np, 4mn))$	$O(n^2 \cdot min(np, 4mn))$	$O(min(2np(2n-1), 10n^2m))$
this algorithm	$O(min(np, 4mn))$	$O(min(np, 4mn))$	$O(n^2 \cdot min(np, 4mn))$	$O(min(2np(2n-1), 10n^2m))$

a distributed on-line algorithm to identify the intervals, if they exist, one from each process, such that each relation $r_{i,j}$ is satisfied by the (i, j) process pair."

A solution satisfying the set of relations $\{r_{i,j}(\forall i, j)\}$ identifies a global state of the system [5]. Thus, the problem can be viewed as one of detecting a global state that satisfies the specified interval-based conditions per pair of peers.

Devising an efficient on-line algorithm to solve problem DOOR is a challenge because of the overhead of having to track the intervals at different processes. A distributed on-line algorithm to solve this problem was outlined in [1] without any formal discussion, without any analysis of its theoretical basis, and without any correctness proofs. A centralized but on-line algorithm was given in [3, 4]. In this paper, we devise a more efficient distributed on-line algorithm to solve this problem, and then prove its correctness. The algorithm uses $O(\min(np, 4mn))$ number of messages, where n is the number of processes, m is the maximum number of messages sent by any process, and p is the maximum number of intervals at any process. The total space complexity across all the processes is $\min(4n^2p - 2np, 10n^2m)$, and the average time complexity at a process is $O(\min(np, 4mn))$. The performance of the centralized algorithm [3, 4] and the algorithm in [1] are compared with the performance of the algorithm in this paper, in Table 1. The proposed algorithm uses an order of magnitude $O(n)$ messages fewer than the earlier algorithm [1], although that comes at the cost of somewhat larger messages.

2 System Model and Preliminaries

We assume an asynchronous distributed peer-to-peer system in which n processes communicate solely by reliable message passing over logical FIFO channels. $(E, \prec)$, where $\prec$ is an irreflexive partial ordering representing the causality or the "happens before" relation [10] on the event set E, is used as the model for a distributed system execution. E is partitioned into local executions at each process. Each E_i is a linearly ordered set of events executed by process P_i. We use N to denote the set of all processes.

We assume vector clocks [6, 11]. The durations of interest at each process can be the durations during which some local predicate of interest is true. Such a

Table 2. Dependent relations for interactions between intervals are given in the first two columns [7]. Tests for the relations are given in the third column.

Relation r	Expression for $r(X,Y)$	Test for $r(X,Y)$
R1	$\forall x \in X \forall y \in Y, x \prec y$	$V_y^-[x] > V_x^+[x]$
R2	$\forall x \in X \exists y \in Y, x \prec y$	$V_y^+[x] > V_x^+[x]$
R3	$\exists x \in X \forall y \in Y, x \prec y$	$V_y^-[x] > V_x^-[x]$
R4	$\exists x \in X \exists y \in Y, x \prec y$	$V_y^+[x] > V_x^-[x]$
S1	$\exists x \in X \forall y \in Y, x \not\preceq y \bigwedge y \not\preceq x$	$\exists x^0 \in X$: $V_y^-[y] \not\leq V_x^{x^0}[y] \wedge V_x^{x^0}[x] \not\leq V_y^+[x]$
S2	$\exists x_1, x_2 \in X \exists y \in Y, x_1 \prec y \prec x_2$	$\exists y^0 \in Y : V_x^+[y] \not< V_y^{y^0}[y] \wedge V_y^{y^0}[x] \not< V_x^-[x]$

Table 3. The 40 orthogonal relations in $\Re$ [7]. The upper part gives the 29 relations assuming dense time. The lower part gives 11 additional relations for nondense time.

Interaction	Relation $r(X,Y)$						Relation $r(Y,X)$					
Type	R1	R2	R3	R4	S1	S2	R1	R2	R3	R4	S1	S2
$IA(=IQ^{-1})$	1	1	1	1	0	0	0	0	0	0	0	0
$IB(=IR^{-1})$	0	1	1	1	0	0	0	0	0	0	0	0
$IC(=IV^{-1})$	0	0	1	1	1	0	0	0	0	0	0	0
$ID(=IX^{-1})$	0	0	1	1	1	1	0	1	0	1	0	0
$ID'(=IU^{-1})$	0	0	1	1	0	1	0	1	0	1	0	1
$IE(=IW^{-1})$	0	0	1	1	1	1	0	0	0	1	0	0
$IE'(=IT^{-1})$	0	0	1	1	0	1	0	0	0	1	0	1
$IF(=IS^{-1})$	0	1	1	1	0	1	0	0	0	1	0	1
$IG(=IG^{-1})$	0	0	0	0	1	0	0	0	0	0	1	0
$IH(=IK^{-1})$	0	0	0	1	1	0	0	0	0	0	1	0
$II(=IJ^{-1})$	0	1	0	1	0	0	0	0	0	0	1	0
$IL(=IO^{-1})$	0	0	0	1	1	1	0	1	0	1	0	0
$IL'(=IP^{-1})$	0	0	0	1	0	1	0	1	0	1	0	1
$IM(=IM^{-1})$	0	0	0	1	1	0	0	0	0	1	1	0
$IN(=IM'^{-1})$	0	0	0	1	1	1	0	0	0	1	0	0
$IN'(=IN'^{-1})$	0	0	0	1	0	1	0	0	0	1	0	1
$ID''(=(IUX)^{-1})$	0	0	1	1	0	1	0	1	0	1	0	0
$IE''(=(ITW)^{-1})$	0	0	1	1	0	1	0	0	0	1	0	0
$IL''(=(IOP)^{-1})$	0	0	0	1	0	1	0	1	0	1	0	0
$IM''(=(IMN)^{-1})$	0	0	0	1	0	0	0	0	0	1	1	0
$IN''(=(IMN')^{-1})$	0	0	0	1	0	1	0	0	0	1	0	0
$IMN''(=(IMN'')^{-1})$	0	0	0	1	0	0	0	0	0	1	0	0

duration, also termed as an *interval*, at process P_i is identified by the corresponding events within E_i. Each interval can be viewed as defining an event of higher granularity at that process [8], as far as the local predicate is concerned. Such higher-level events, one from each process, can be used to identify a global state. Intervals are denoted using X and Y. An interval X at P_i is denoted by X_i.

It was shown in [7] that there are 29 or 40 causality-based mutually orthogonal ways in which any two durations can be related to each other, depending on whether dense or nondense time is assumed. These orthogonal interaction types were identified by using the six relations given in the first two columns of Table 2. Relations R1 (strong precedence), R2 (partially strong precedence), R3 (partially weak precedence), R4 (weak precedence) define *causality conditions*. S1 and S2 define *coupling conditions*. The set of 40 relations is denoted as $\Re$.

Given a set of orthogonal relations, one between each pair of processes, that need to be detected, each of the 29 (40) possible independent relations in the dense (nondense) model of time can be tested for using the bit-patterns for the dependent relations, as given in Table 3 [7]. The tests for the relations $R1 - R4$, $S1$, and $S2$ using vector timestamps are given in the third column of Table 2. V_i^- and V_i^+ denote the vector timestamp at process P_i at the start and at the end of an interval, respectively. V_i^x denotes the vector timestamp of event x_i at process P_i. When the process is not specified explicitly, we assume that interval X occurs at P_i and interval Y occurs at P_j. For any two intervals X and X' that occur at the same process, if $R1(X, X')$, then we say that X is a *predecessor* of X' and X' is a *successor* of X.

Our goal is to efficiently apply the tests in Table 2 in a distributed manner across all the processes. Each process P_i, $1 \leq i \leq n$, maintains information about the timestamps of the start and end of its local intervals, and certain other local information, in a local queue Q_i. The n processes collectively run some distributed algorithm to process the information in the local queues and solve problem **DOOR**. In order for distributed algorithms to process the queued intervals efficiently, we first give some results about when two given intervals may potentially satisfy a given interaction type we want to detect.

3 Conditions for Satisfying Given Interaction Types

The discussion in this section is based on [1, 3, 4] which gave other (less efficient) solutions to solve problem DOOR.

Definition 1. Prohibition function $\mathcal{H} : \Re \rightarrow 2^{\Re}$ *is defined as* $\mathcal{H}(r_{i,j}) = \{R \in \Re \mid if\ R(X,Y)\ is\ true\ then\ r_{i,j}(X,Y')\ is\ false\ for\ all\ Y'\ that\ succeed\ Y\}$.

Definition 2. *The "allows" relation* $\leadsto$ *is a relation on* $\Re \times \Re$ *such that* $R' \leadsto R''$ *if the following holds. If* $R'(X,Y)$ *is true then* $R''(X,Y')$ *can be true for some* Y' *that succeeds* Y.

Lemma 1. *If* $R \in \mathcal{H}(r_{i,j})$ *then* $R \not\leadsto r_{i,j}$ *else if* $R \notin \mathcal{H}(r_{i,j})$ *then* $R \leadsto r_{i,j}$.

Proof: If $R \in \mathcal{H}(r_{i,j})$, using Definition 1, it can be inferred that $r_{i,j}$ is false for all Y' that succeed Y. This does not satisfy Definition 2. Hence $R \not\leadsto r_{i,j}$. If $R \notin \mathcal{H}(r_{i,j})$, it follows that $r_{i,j}$ can be true for some Y' that succeeds Y. This satisfies Definition 2 and hence $R \leadsto r_{i,j}$. □

Table 4 gives $\mathcal{H}(r_{i,j})$ for each of the 40 interaction types in $\Re$.

Theorem 1. *For* $R', R'' \in \Re$ *and* $R' \neq R''$, *if* $R' \leadsto R''$ *then* $R'^{-1} \not\leadsto R''^{-1}$.

Lemma 2. *If the relationship* $R(X,Y)$ *between intervals* X *and* Y *(belonging to process* P_i *and* P_j*, resp.) is contained in the set* $\mathcal{H}(r_{i,j})$*, and* $r_{i,j} \neq R$*, then interval* X *can be removed from the queue* Q_i.

Proof: From the definition of $\mathcal{H}(r_{i,j})$, we get that $r_{i,j}(X,Y')$ cannot exist, where Y' is any successor interval of Y. Further, $r_{i,j} \neq R$. So we have that X can never be a part of a solution and can be deleted from the queue. □

Table 4. $\mathcal{H}(r_{i,j})$ for the 40 orthogonal relations in $\Re$. The upper part is for 29 relations under dense time. The lower part is for 11 additional relations under nondense time.

Interaction Type $r_{i,j}$	$\mathcal{H}(r_{i,j})$	$\mathcal{H}(r_{j,i})$
$IA\ (= IQ^{-1})$	ϕ	$\Re - \{IQ\}$
$IB\ (= IR^{-1})$	$\{IA, IB, IF, II, IP, IO, IU, IX, IUX, IOP\}$	$\Re - \{IQ\}$
$IC\ (= IV^{-1})$	$\{IA, IB, IF, II, IP, IO, IU, IX, IUX, IOP\}$	$\Re - \{IQ\}$
$ID\ (= IX^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ\}$
$ID'\ (= IU^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ\}$
$IE\ (= IW^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ\}$
$IE'\ (= IT^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ\}$
$IF\ (= IS^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ\}$
$IG\ (= IG^{-1})$	$\Re - \{IQ, IR, IJ, IV, IK, IG\}$	$\Re - \{IQ, IR, IJ, IV, IK, IG\}$
$IH\ (= IK^{-1})$	$\Re - \{IQ, IR, IJ, IV, IK, IG\}$	$\Re - \{IQ, IR, IJ\}$
$II\ (= IJ^{-1})$	$\Re - \{IQ, IR, IJ, IV, IK, IG\}$	$\Re - \{IQ, IR, IJ\}$
$IL\ (= IO^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$
$IL'\ (= IP^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$
$IM\ (= IM^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$
$IN\ (= IM'^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$
$IN'\ (= IN'^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$
$ID''\ (= (IUX)^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ\}$
$IE''\ (= (ITW)^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ\}$
$IL''\ (= (IOP)^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$
$IM''\ (= (IMN)^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$
$IN''\ (= (IMN')^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$
$IMN''\ (= (IMN'')^{-1})$	$\Re - \{IQ, IR, IJ\}$	$\Re - \{IQ, IR, IJ\}$

Lemma 3. *If the relationship between a pair of intervals X and Y (belonging to processes P_i and P_j respectively) is not equal to $r_{i,j}$, then interval X or interval Y is removed from its queue Q_i or Q_j, respectively.*

Proof: We use contradiction. Assume relation $R(X,Y)$ $(\neq r_{i,j}(X,Y))$ is true for intervals X and Y. From Lemma 2, the only time neither X nor Y will be deleted is when $R \notin \mathcal{H}(r_{i,j})$ and $R^{-1} \notin \mathcal{H}(r_{j,i})$. From Lemma 1, it can be inferred that $R \leadsto r_{i,j}$ and $R^{-1} \leadsto r_{j,i}$. As $r_{i,j}^{-1} = r_{j,i}$, we get $R \leadsto r_{i,j}$ and $R^{-1} \leadsto r_{i,j}^{-1}$. This is a contradiction as by Theorem 1, R being unequal to $r_{i,j}$, $R \leadsto r_{i,j} \Rightarrow R^{-1} \not\leadsto r_{i,j}^{-1}$. Hence $R \in \mathcal{H}(r_{i,j})$ or $R^{-1} \in \mathcal{H}(r_{j,i})$, and thus at least one of the intervals will be deleted.

4 A Distributed Peer-to-Peer Algorithm

Each process P_i, where $1 \leq i \leq n$, maintains the following data structures. (1) V_i : array[1..n] of integer. This is the *Vector Clock* [6, 11]. (2) I_i : array[1..n] of integer. This is a *Interval Clock* [1, 3, 4] which tracks the latest intervals at

```
1. When an internal event or send event occurs at process $P_i$, $V_i[i] = V_i[i] + 1$.
2. Every message contains the vector clock and Interval Clock of its send event.
3. When process $P_i$ receives a message $msg$, then $\forall$ $j$ do,
       if ($j == i$) then $V_i[i] = V_i[i] + 1$,
       else $V_i[j] = \max(V_i[j], msg.V[j])$.
4. When an interval starts at $P_i$ (local predicate $\phi_i$ becomes true), $I_i[i] = V_i[i]$.
5. When process $P_i$ receives a message $msg$, then $\forall$ $j$ do,
       $I_i[j] = \max(I_i[j], msg.I[j])$
```

Fig. 1. The protocol for maintaining vector clock and *Interval Clock*

```
type Event_Interval = record
    interval_id : integer;
    local_event: integer;
end

type Log = record
    start: array[1..n] of integer;
    end: array[1..n] of integer;
    p_log: array[1..n] of Process_Log;
end

type Process_Log = record
    event_interval_queue: queue of Event_Interval;
end
Start of an interval:
$Log_i.start = V_i^-$.
On receiving a message during an interval:
if (change in $I_i$) then
    for each $k$ such that $I_i[k]$ was changed
        insert $(I_i[k], V_i[i])$ in $Log_i.p\_log[k].event\_interval\_queue$
End of interval:
$Log_i.end = V_i^+$
if (a receive or send occurs between start of previous and end of present interval) then
    Enqueue $Log_i$ on to the local queue $Q_i$.
```

Fig. 2. The data structures and the protocol for constructing *Log* at P_i $(1 \leq i \leq n)$

```
For $S2(X, Y)$:

1. (1a)  for each $event\_interval \in Log_j.p\_log[i].event\_interval\_queue$
   (1b)      if ($event\_interval.interval\_id < Log_i.start[i]$)
   (1c)          then remove $event\_interval$
2. (2a)  $temp = \infty$
   (2b)  if ($Log_j.start[i] \geq Log_i.start[i]$) then $temp = Log_j.start[j]$
   (2c)  else for each $event\_interval \in Log_j.p\_log[i].event\_interval\_queue$
   (2d)          $temp = \min(temp, event\_interval.local\_event)$
3. if ($Log_i.end[j] \geq temp$) then $S2(X, Y)$ is true.

For $S1(Y, X)$:

1. Same as step 1 of scheme to determine $S2(X, Y)$.
2. Same as step 2 of scheme to determine $S2(X, Y)$.
3. if ($Log_i.end[j] < temp$) and ($temp > Log_j.start[j]$) then $S1(Y, X)$ is true.
```

Fig. 3. The protocol to test for $S1(X, Y)$ and $S2(Y, X)$

processes. $I_i[j]$ is the timestamp $V_j[j]$ when the predicate ϕ_j of interest at P_j last became true, as known to P_i. (3) Log_i: contains the information about an interval, needed to compare it with other intervals. Fig. 1 shows how to update the vector clock and *Interval Clock*.

Log_i is constructed and stored on the local queue Q_i using the data structures and protocol shown in Fig. 2. The *Log* is used to determine the relationship between two intervals. The tests in Table 2 are used to find which of $R1 - R4$, $S1$, and $S2$ are true. Fig. 3 shows how to implement the tests for $S1$ and $S2$. The data structures in Fig. 2, 3 were proposed and used in the design of the previous algorithms [1, 3, 4] to address problem DOOR. However, the algorithm in this paper is fully distributed and more efficient.

The algorithm identifies a set of intervals $\mathcal{I}$, if they exist, one interval I_i from each process P_i, such that the relation $r_{i,j}(I_i, I_j)$ is satisfied by each (i, j) process pair. If no such set of intervals exists, the algorithm does not return any interval set. The algorithm uses a token T. The data structure for the token (T) is given in Figure 4. $T.log[i]$ contains the *Log* corresponding to the interval at the head of

```
type $T$ = token
    $log$: array [1..n] of $Log$;
//Contains the $Log$s of the intervals (at the queue heads) which may be in soln.
    $C$: array [1..n] of boolean;
//$C[i]$ is true if and only if $Log[i]$ at the head of $Q_i$ can be a part of soln.
end

(1) Initial state for process $P_i$
(1a)    $Q_i$ has a dummy interval

(2) Initial state for the token
(2a)    $\forall i : T.C[i] = false$
(2b)    $T$ does not contain any $Log$
(2c)    A randomly elected process $P_i$ holds the token

(3) On receiving token $T$ at $P_i$
(3a)    while $(T.C[i] = false)$
(3b)        Delete head of the queue $Q_i$
(3c)        if ($Q_i$ is empty) then wait until $Q_i$ is non-empty
(3d)        $T.C[i] = true$
(3e)        $X$ = head of $Q_i$
(3f)        for $j = 1$ to $n$
(3g)            if $(T.C[j] = true)$ then
(3h)                $Y = T.log[j]$
(3i)                Determine $R(X,Y)$ using the tests given in Fig. 3 and Table 2
(3j)                if $(r_{i,j} \neq R(X,Y))$ and $(R(X,Y) \in \mathcal{H}(r_{i,j}))$ then $T.C[i] = false$
(3k)                if $(r_{j,i} \neq R(Y,X))$ and $(R(Y,X) \in \mathcal{H}(r_{j,i}))$ then
(3l)                    $T.C[j] = false$
(3m)                    $T.log[j] = \perp$
(3n)    $T.log[i] = Log_i$
(3o)    if $(\forall k : T.C[k] = true)$ then
(3p)        solution found. $T$ has the solution $Log$s.
(3q)    else
(3r)        $k = i + 1$
(3s)        while $(T.C[k] \neq false)$
(3t)            $k = (k + 1)\ mod\ n$
(3u)        Send $T$ to $P_k$
```

Fig. 4. Distributed algorithm to solve problem DOOR

queue Q_i. $T.C[i] = true$ implies that the interval at the head of queue Q_i may be a part of the final solution and the corresponding log Log_i is stored in the token. If $T.C[i] = false$ then the interval at the head of queue Q_i is not a part of the solution, its corresponding log is not contained in the token, and the interval can be deleted. The algorithm is given in Figure 4. The process (P_i) receives a token only if $T.C[i] = false$, which means the interval at the head of queue Q_i is not a part of the solution and hence is deleted. The next interval X on the queue Q_i is then compared with each other interval Y whose log Log_j is contained in $T.log[j]$ (in which case $T.C[j] = true$, lines 3e-3i). According to Lemma 3, the comparison between intervals X and Y can result in three cases. (1) $r_{i,j}$ is satisfied. (2) $r_{i,j}$ is not satisfied and interval X can be removed from the queue Q_i. (3) $r_{i,j}$ is not satisfied and interval Y can be removed from the queue Q_j. In the third case, the log Log_j corresponding to interval Y is deleted and $T.C[j]$ is set to false (lines 3l-3m). In the second case, $T.C[i]$ is set to false (line 3j) so that in the next iteration of the **while** loop, the interval X is deleted (lines 3a-3b). Note that both cases (2) and (3) can be true as a result of a comparison. The above process is repeated until the interval at the head of the queue Q_i satisfies the required relationships with each of the interval Logs remaining in the token (T). The process (P_i) then adds the log Log_i corresponding to the interval at the head of queue Q_i to the token $T.Log[i]$ and sets $T.C[i]$ equal to true. A solution is detected when $T.C[k]$ is true for all indices k (lines 3n-3p), and is given by all the n log entries of all the processes, $T.Log[1,\ldots,n]$. If the above condition (line 3o) is not satisfied then the token is sent to some process P_j whose log Log_j is not contained in the token $T.Log[j]$ (in which case $T.C[j] = false$, lines 3r-3u). Note that the wait in (line 3c) can be made non-blocking by restructuring the code using an interrupt-based approach.

5 Correctness Proof

Lemma 4. *After P_i executes the loop in lines (3f-3m), if $T.C[i] = true$ then the relationship $r_{i,j}$ is satisfied for interval X_i at the head of queue Q_i and each interval Y_j at the head of queue Q_j satisfying $T.C[j] = true$.*

Proof: The body of the loop (lines 3j-l) implements Lemma 2 by testing for $R(X_i, Y_j) \in \mathcal{H}(r_{i,j})$ and $R(Y_j, X_i) \in \mathcal{H}(r_{j,i})$, when R is not equal to $r_{i,j}$. If $r_{i,j}$ is not satisfied between interval X and interval Y then by Lemma 3, X or Y is deleted, i.e., (line 3j) or (lines 3k, 3l) is executed and hence $T.C[i]$ or $T.C[j]$ is set to false. This implies if both $T.C[i]$ and $T.C[j]$ are true then the relationship $r_{i,j}(X, Y)$ is true.

It remains to show that for all j for which $T.C[j]$ is true when the loop in (3f-3m) completes, Y_j which is in $T.Log[j]$ is the same as $head(Q_j)$. This follows by observing that (i) $T.Log[j]$ was the same as $head(Q_j)$ when the token last visited and left P_j, and (ii) $head(Q_j)$ is deleted only when $T.C[j]$ is false and hence the token visits P_j. □

Theorem 2. *When a solution $\mathcal{I}$ is detected by the algorithm in Figure 4, the solution is correct, i.e., for each $i, j \in N$ and $I_i, I_j \in \mathcal{I}$, the intervals $I_i = head(Q_i)$ and $I_j = head(Q_j)$ are such that $r_{i,j}(I_i, I_j)$.*

Proof: It is sufficient to prove that for the solution detected, which happens at the time $T.C[k] = true$ for all k (lines 3o,p), (i) $r_{i,j}(I_i, I_j)$ is satisfied for all pairs (i, j), and (ii) none of the queues is empty. To prove (i) and (ii), note that at this time, the token must have visited each process at least once because only the token-holder P_i can set $T.C[i]$ to true. Consider the latest time t_i when process P_i was last visited by the token (and $T.C[i]$ was set to true and $T.Log[i]$ was set to

$head(Q_i)$). Since t_i until the solution is detected, $T.C[i]$ remains true, otherwise the token would revisit P_i again (lines 3s-u) – leading to a contradiction. Linearly order the process indices in array $Visit[1, \ldots, n]$ according to the increasing order of the times of the last visit of the token. Then for k from 2 to n, we have that when the token was at $P_{Visit[k]}$, the intervals corresponding to $T.Log[k]$ and $T.Log[m]$, for all $1 \leq m < k$, were tested successfully for relation $r_{k,m}$ and $T.C[k]$ and $T.C[m]$ were true after this test. This shows that the intervals from every pair of processes got tested, and by Lemma 4, that $r_{k,m}(X_k, Y_m)$ was satisfied for $X_k = head(Q_k)$ and $Y_m = head(Q_m)$ at the time of comparison.

As shown above, since t_k until the solution is detected, $T.C[k]$ remains true and the token does not revisit P_k. Hence, from t_k until the solution is detected, none of the intervals tested at t_k using $T.Log$ got dequeued from their respective queues and $r_{k,m}(X_k, Y_m)$ continues to be satisfied for $X_k = head(Q_k)$ and $Y_m = head(Q_m)$ when the solution is detected. □

Let $\mathcal{I}(h)$ denote the set of intervals at the head of each queue, that are compared during the processing triggered by hop h of the token. Each $\mathcal{I}(h)$ identifies a system state (not necessarily consistent). Observe that for any $\mathcal{I}(h)$ and $\mathcal{I}(h+1)$ and any P_i, interval $I_i(h+1)$ in $\mathcal{I}(h+1)$ is equal to or an immediate successor of interval $I_i(h)$ in $\mathcal{I}(h)$. We thus say that all the $\mathcal{I}$ are linearly ordered, and $\mathcal{I}(h)$ *precedes* $\mathcal{I}(h')$, for all $h' > h$. Let $\mathcal{I}(S)$ denote the set of intervals that form the *first* solution, assuming it exists. Let $\mathcal{I}(b)$ denote the first value of $\mathcal{I}(h)$ that contains one (or more) of the intervals belonging to $\mathcal{I}(S)$. Let $\mathcal{I}(init)$ denote the initial set of intervals. Clearly, $\mathcal{I}(init)$ precedes $\mathcal{I}(b)$ which precedes $\mathcal{I}(S)$.

Lemma 5. *In any hop h of the token, no interval $X_i \in \mathcal{I}(S)$ gets deleted.*

Proof: This can be shown by considering two possibilities.

1. An interval X_i in $\mathcal{I}(S)$ get compared with some interval Y_j that is also in $\mathcal{I}(S)$. In this case, both conditions are false in lines 3j and 3k as $r_{i,j}(X_i, Y_j)$ is satisfied, and *this comparison* does not cause either of the intervals to be deleted from $T.Log[]$. Also, $T.C[i]$ and $T.C[j]$ remain true.
2. An interval X_i gets compared with some $Y_j \in \mathcal{I}(h) \setminus \mathcal{I}(S)$. Observe that Y_j is a predecessor of the interval Y'_j at P_j such that $Y'_j \in \mathcal{I}(S)$ and thus $r_{i,j}(X_i, Y'_j)$ holds. We have that $R(X_i, Y_j) \neq r_{i,j}$. We now show that Y_j gets deleted and X_i does not get deleted.
   – Applying Theorem 1, we have $R(X_i, Y_j) \rightsquigarrow r_{i,j}(X_i, Y'_j)$ which implies $R^{-1} \not\rightsquigarrow r_{j,i}$. From Lemma 1, we conclude that $R^{-1} \in \mathcal{H}(r_{j,i})$. There are two cases to consider. (a) The token is at P_i. By Lemma 2 which is implemented in lines 3k-m, the comparison results in Y_j being deleted from $T.Log$ and subsequently from Q_j (line 3b when the token reaches P_j). (b) The token is at P_j. By Lemma 2 which is implemented in lines 3j and 3a-3e, the comparison results in Y_j being deleted from Q_j.
   – As $R \rightsquigarrow r_{i,j}$ therefore from Lemma 1, we conclude that $R \notin \mathcal{H}(r_{i,j})$. By Lemma 2 which is implemented in lines 3j and 3k-m (depending on whether P_i or P_j has the token), this comparison does not result in X_i being deleted. □

Lemma 6. *In any hop h of the token, at least one interval $Y_j \in \mathcal{I}(h) \setminus \mathcal{I}(S)$ gets deleted.*

Proof: Line 3b deletes interval $(head(Q_i))$ when the token arrives at P_i. □

Theorem 3. *If a solution $\mathcal{I}$ exists, i.e., for each $i, j \in N$, the intervals I_i, I_j belonging to $\mathcal{I}$ are such that $r_{i,j}(I_i, I_j)$, then the solution is detected by the algorithm in Figure 4.*

Proof: From state $\mathcal{I}(init)$ in which $T.Log$ was initialized to contain no log, onwards until state $\mathcal{I}(b)$, Lemma 6 is true. Hence, in each hop of the token, the interval at the head of the queue of some process must get replaced by the immediate successor interval at that process. This guarantees progress and that state $\mathcal{I}(b)$ is reached.

From state $\mathcal{I}(b)$ onwards until state $\mathcal{I}(S)$, both Lemmas 5 and 6 apply. Lemma 6 guarantees progress (some interval not belonging to $\mathcal{I}(S)$ gets replaced by its immediate successor interval at every hop). Lemma 5 guarantees that no interval belonging to the solution set of $\mathcal{I}(S)$ gets deleted. Once $T.Log$ contains all the intervals of $\mathcal{I}(S)$ and hence $T.C[k]$ is true for all k, the solution is detected. □

Thus, Theorems 2 and 3 show that the algorithm detects a solution if and only if it exists.

6 Complexity Analysis

The complexity analysis sketched below and summarized in Table 1 is in terms of two parameters – the maximum number of messages sent per process (m) and the maximum number of intervals per process (p). Details are given in [2].

Space complexity at P_1 to P_n:

1. In terms of m: It is necessary to store *Log* for four intervals for every message – two for the send event and two for the receive event. Refer [2, 3, 4] for the reasoning. As there are a total of nm number of messages exchanged between all processes, a total of $4nm$ interval *Log*s are stored across all the queues, though not necessarily at the same time.
   - The *total space overhead* across all processes is $2.mn.n + 4mn.2n = 10mn^2$. The term $2.mn.n$ arises because for each of the mn messages sent, each of the other n processes eventually (due to transitive propagation of *Interval Clock*) may need to insert a *Event_Interval* tuple (size 2) in its *Log*. This can generate $2.mn.n$ overhead in *Log*s across the n processes. The term $4mn.2n$ arises because the vector timestamp at the start and at the end of each interval is also stored in each *Log*. It now follows that the average number of *Log*s per process is $4m$ and the average space overhead per process is $10mn$. Also note that the average size of a *Log* is $O(n)$.

- For a process, the *worst case* occurs when it receives m messages from each of the other $n-1$ processes. Assuming the process sends m messages, a total of $m(n-1)+m$ messages are sent or received by this process. In this case, the number of *Log*s stored on the process queue is $2mn$, two *Log*s for each receive event or send event (see [2,3,4] for justification). The total space required at the process is $2mn.2n+2m(n-1) = O(mn^2)$. The term $2mn.2n$ arises from the fact that each *Log* contains two vector timestamps and there are a total of $2mn$ *Log*s stored on the process queue. The term $2m(n-1)$ arises because an *Event_Interval* is added for each receive and there are a total of $m(n-1)$ messages received on the process.

Note that the worst case just discussed is for a single process; the total space overhead always remains $O(mn^2)$ and on an average, the space complexity for each process is $O(nm)$.

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. The total number of *Log*s stored at all the processes is np. Consider the cumulative space requirement for *Log* over all the intervals at a single process.
   - Each *Log* stores the start (V^-) and the end (V^+) of an interval, which requires a maximum of $2np$ integers over all *Log*s per process.
   - Additionally, an *Event_Interval* is added to the *Log* for every component of *Interval Clock* which is modified due to the receive of a message. Since a change in a component of *Interval Clock* implies the start of a new interval on another process, the total number of times the component of *Interval Clock* can change is equal to the number of intervals on all the processes. Thus the total number of *Event_Interval* which can be added to the *Log* of a single process is $(n-1)p$. This gives a total of $2(n-1)p$ integers (corresponding to *Event_Interval*s) at each process.

   The total space required for *Log*s corresponding to all p intervals on a single process is $2(n-1)p + 2np$. So the total space is $4n^2p - 2np = O(n^2p)$.

Thus, the total number of *Log*s stored on all the processes is $\min(np, 4mn)$ and the total space overhead for all the processes is $\min(4n^2p - 2np, 10n^2m)$.

Time Complexity: The time complexity is the product of the number of steps required to determine an orthogonal relationship from $\Re$ between a pair of intervals, and the number of interval pairs considered.

- The first part of the product on average takes $O(1)$ time (to determine a relationship [2,3,4]). Note this part does not depend on the algorithm being centralized or distributed.
- To analyze the second part of the product, consider Figure 4. The maximum number of times an interval at the head of its queue is compared locally with intervals contained in the token is $(n-1)$. The reason being that when an interval comes to the head of a queue (Q_i), it may be compared with $n-1$ other intervals (contained in the token), one corresponding to each other process, but the next time that token reaches the process (P_i) is when $C[i] = false$ and hence the interval is dequeued. Thus the total number of

interval pairs compared is $(n-1)$ times the total number of *Log*s over all the queues. The total number of *Log*s over all the queues was shown above to be equal to $\min(np, 4mn)$, hence the total number of interval pairs compared is $(n-1)\min(np, 4mn)$.

As on average it takes $O(1)$ time to determine a relationship, the average time complexity of the algorithm is equal to $O(n \cdot \min(np, 4mn))$. Hence the average time complexity per process is $O(\min(np, 4mn))$.

Message Complexity: The token is sent to P_j whenever $C[j]$ is false, and $C[j]$ is false if the interval at the head of the queue Q_j has to be deleted. Thus, the maximum number of times the token is sent is equal to the total number of intervals across all the queues, which is equal to $\min(np, 4nm)$. Hence, the total number of messages sent is $\min(np, 4nm)$. The maximum number of *Log*s stored on a token in $n-1$ and the size of each *Log* on the average can be seen to be $O(n)$. Thus, the total message space overhead is $O(n^2 \min(np, 4mn))$.

References

1. P. Chandra, A. D. Kshemkalyani, Detection of orthogonal interval relations. Proc. 9th Intl. High Performance Computing Conference (HiPC), LNCS 2552, Springer, 323-333, 2002.
2. P. Chandra, A. D. Kshemkalyani, Distributed detection of temporal interactions. Tech. Report UIC-ECE-02-07, Univ. of Illinois at Chicago, May 2002.
3. P. Chandra, A. D. Kshemkalyani, Detecting global predicates under fine-grained modalities. Proc. 8th Asian Computing Conference (ASIAN), LNCS 2896, Springer-Verlag, 91-109, December 2003.
4. P. Chandra, A. D. Kshemkalyani, Causality-based predicate detection across space and time. IEEE Transactions on Computers, 54(11): 1438-1453, November 2005.
5. K. M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems, 3(1): 63-75, 1985.
6. C. J. Fidge, Timestamps in message-passing systems that preserve partial ordering. Australian Computer Science Communications, 10(1): 56-66, February 1988.
7. A. D. Kshemkalyani, Temporal interactions of intervals in distributed systems. Journal of Computer and System Sciences, 52(2): 287-298, April 1996.
8. A. D. Kshemkalyani, A framework for viewing atomic events in distributed computations. Theoretical Computer Science, 196(1-2), 45-70, April 1998.
9. A. D. Kshemkalyani, M. Raynal, M. Singhal, An introduction to global snapshots of a distributed system. IEE/IOP Distributed Systems Engineering Journal, 2(4): 224-233, December 1995.
10. L. Lamport, Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7): 558-565, July 1978.
11. F. Mattern, Virtual time and global states of distributed systems. Parallel and Distributed Algorithms, North-Holland, 215-226, 1989.
12. D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, Z. Xu, Peer-to-peer computing. Hewlett Packard Technical Report HPL-2002-57, 2002.
13. J. Rissom, T. Moors, Survey of research towards robust peer-to-peer networks: Search methods. Technical Report UNSW-EE-P2P-1-1, Univ. of New South Wales, 2004.