Chapter 11: Global Predicate Detection

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Introduction and Uses of Predicate Detection

- Industrial process control, distributed debugging, computer-aided verification, sensor networks
- E.g., ψ defined as $x_i + y_j + z_k < 100$
- Different from global snapshots: global snapshot gives one of the values that could have existed during the execution
- Stable predicate: remains true once it becomes true, i.e., $\phi \Longrightarrow \quad \Box\phi$
  - predicate ϕ at a cut C is stable if:

$$(C \models \phi) \implies (\forall C' \mid C \subseteq C', C' \models \phi)$$

  - E.g., deadlock, termination of execution are stable properties

Stable Properties

- Deadlock: Given a Wait-For Graph $G = (V, E)$, a *deadlock* is a subgraph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$ and for each i in V', i remains blocked unless it receives a reply from some process(es) in V'.
  - (local condition:) each deadlocked process is locally blocked, and
  - (global condition:) the deadlocked process will not receive a reply from some process(es) in V'.
- Termination of execution: Model active and passive states, and state transitions between them. Then execution is terminated if:
  - (local condition:) each process is in passive state, and
  - (global condition:) there is no message in transit between any pair of processes.
- Repeated global snapshots is not practical!
- Utilize a 2-phased approach of observing potentially inconsistent global states.

Stable Properties

- Deadlock: Given a Wait-For Graph $G = (V, E)$, a *deadlock* is a subgraph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$ and for each i in V', i remains blocked unless it receives a reply from some process(es) in V'.
  - (local condition:) each deadlocked process is locally blocked, and
  - (global condition:) the deadlocked process will not receive a reply from some process(es) in V'.
- Termination of execution: Model active and passive states, and state transitions between them. Then execution is terminated if:
  - (local condition:) each process is in passive state, and
  - (global condition:) there is no message in transit between any pair of processes.
- Repeated global snapshots is not practical!
- Utilize a 2-phased approach of observing potentially inconsistent global states.

Stable Properties

- Deadlock: Given a Wait-For Graph $G = (V, E)$, a *deadlock* is a subgraph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$ and for each i in V', i remains blocked unless it receives a reply from some process(es) in V'.
  - (local condition:) each deadlocked process is locally blocked, and
  - (global condition:) the deadlocked process will not receive a reply from some process(es) in V'.
- Termination of execution: Model active and passive states, and state transitions between them. Then execution is terminated if:
  - (local condition:) each process is in passive state, and
  - (global condition:) there is no message in transit between any pair of processes.
- Repeated global snapshots is not practical!
- Utilize a 2-phased approach of observing potentially inconsistent global states.

Two-phase Observation of Global States

- In each state observation, all local variables used to define the local conditions, as well as the global conditions, are observed.
- Two potentially inconsistent global states are recorded consecutively, such that the second recording is initiated after the first recording has completed. Stable property true if:
  - The variables on which the local conditions as well as the global conditions are defined have not changed in the two observations, as well as between the two observations.
- Recording 2 snapshots serially via ring/tree/flat-tree based algorithms (Chap 8).

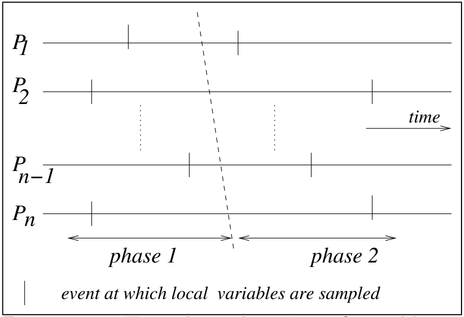


Figure 11.1: Two-phase detection of a stable property.

None of the variables changes between the two observations

$\Rightarrow$ after the termination of the first observation and before the start of the second observation, there is an instant when the variables still have the same value.

$\Rightarrow$ the stable property will necessarily be true.

Unstable Predicates: Challenges in Detection

Challenges:

- unpredictable propagation times, unpredictable process scheduling $\Rightarrow$
  - multiple executions pass through different global states;
  - predicate may be true in some executions and false in others
- No global time $\Rightarrow$
  - monitor finds predicate true in a state but predicate may never have been true (at any instant)
  - even a true predicate may be undetected due to intermittent monitoring

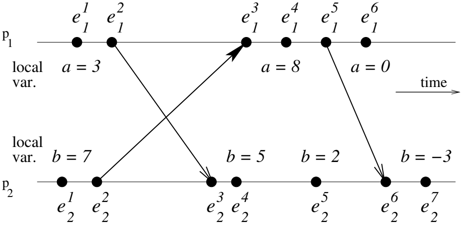
Observations:

- examine all states in an execution $\Rightarrow$ define predicate on observation of entire execution
- Multiple observations of same program may pass thru' different global states; predicate may be true in some observations but not others $\Rightarrow$ define predicate on all the observations of the distributed program

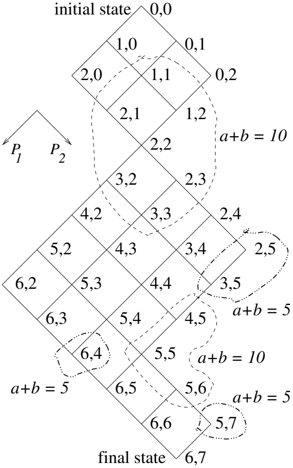
Modalities on Predicates

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

$(0,0), e_2^1, (0,1), e_1^1, (1,1), e_2^2, (1,2), e_1^2, (2,2), e_2^3, (2,3), e_2^4, (2,4), e_1^3, (3,4), e_1^4, (4,4), e_2^5, (4,5), e_1^5, (5,5), e_1^6, (6,5), e_2^6, (6,6), e_2^7, (6,7)$



(a)

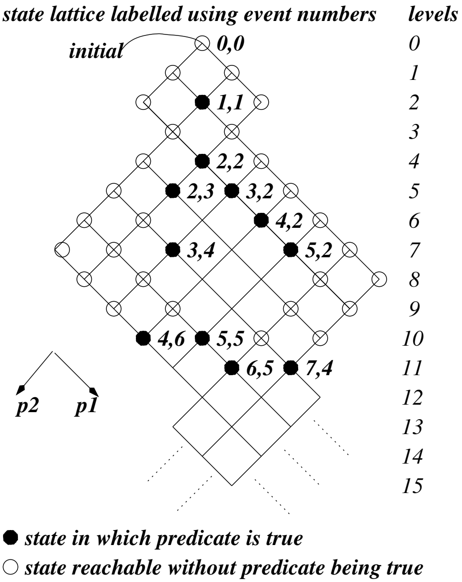


(b)

Complexity: $O(m^n)$, where there are m events at each of the n processes.

- *Definitely*$(a + b = 10)$
- *Possibly*$(a + b = 5)$

Centralized Algorithm for Relational Predicates



Detecting Relational Predicates, on-line, centralized

(variables)
set of global states $Reach_\phi, Reach\_Next_\phi \longleftarrow \{GC^{0,0,\ldots 0}\}$
int $lvl \longleftarrow 0$

(1) *Possibly*(ϕ)
(1a) **while** (no state in $Reach_\phi$ satisfies ϕ) **do**
(1b) **if** ($Reach_\phi = \{final\ \ state\}$) **then return** *false*;
(1c) $lvl \longleftarrow lvl + 1$;
(1d) $Reach_\phi \longleftarrow$ {states at level lvl };
(1e) **return** *true*.

(2) *Definitely*(ϕ)
(2a) remove from $Reach_\phi$ those states that satisfy ϕ
(2b) $lvl \longleftarrow lvl + 1$;
(2c) **while** ($Reach_\phi \neq \emptyset$) **do**
(2d) $Reach\_Next_\phi \longleftarrow$ {states of level lvl reachable from a state in $Reach_\phi$};
(2e) remove from $Reach\_Next_\phi$ all the states satisfying ϕ;
(2f) **if** $Reach\_Next_\phi =$ {final state} **then return** *false*;
(2g) $lvl \longleftarrow lvl + 1$;
(2h) $Reach_\phi \longleftarrow Reach\_Next_\phi$;
(2i) **return** *true*.

Centralized Algorithm for Relational Predicates (contd.)

Definitely(ϕ):

- Replacing line (1a) by: "(some state in $Reach_\phi$ satisfies $\neg\phi$)" will not work!
- The algorithm examines the state lattice level-by-level:
  1. Tracks states at each level in which ϕ is not true
  2. The tracked states at a level have to be reachable from states at previous level satisfying (1) and this property (2) recursively.
- $Reach\_Next_\phi$ at level lvl contains the set of states at level lvl that are reachable from the initial state *without* passing through any state satisfying ϕ.
- return(1) if $Reach\_Next(\phi)$ becomes $\emptyset$, else return(0).
- In example, at $lvl = 11$, $Reach\_Next(\phi)$ becomes empty and $Definitely(\phi)$ is detected.

Constructing the State Lattice

To assemble global state $GS = \{s_1^{k_1}, s_2^{k_2}, \ldots, s_n^{k_n}\}$, i.e., $GS^{k_1,k_2,\ldots k_n}$, from the corresponding local states, how long does a local state need to be kept in its queue?

- The earliest global state $GS_{min}^{k_1,k_2,\ldots k_n}$ containing $s_i^{k_i}$ is identified as follows. The j^{th} component of $VC(s_i^{k_i})$ is the local value of P_j in its local snapshot state $s_j^{k_j}$.

$$(\forall j)\ VC(s_j^{k_j})[j] = VC(s_i^{k_i})[j] \tag{1}$$

  Thus, the lowest level of the state lattice, in which local state $s_i^{k_i}$ (k^{th} local state of P_i) participates, is the sum of the components of $VC(s_i^{k_i})$.

- The latest global state $GS_{max}^{k_1,k_2,\ldots k_n}$ containing $s_i^{k_i}$ is identified as follows. The i^{th} component of $VC(s_j^{k_j})$ should be the largest possible value but cannot exceed or equal $VC(s_i^{k_i})[i]$ for consistency of $s_i^{k_i}$ and $s_j^{k_j}$. $VC(s_i^{k_i})$ is identified as per Equation 2; note that the condition on $VC(s_j^{k_j+1})[i]$ is applicable if $s_j^{k_j}$ is not the last state at P_j.

$$(\forall j)\ VC(s_j^{k_j})[i] < VC(s_i^{k_i})[i] \leq VC(s_j^{k_j+1})[i] \tag{2}$$

  Hence, the highest level of the state lattice, in which local state $s_i^{k_i}$ participates, is $\sum_{j=1}^{n} VC(s_j^{k_j})[j]$.

Constructing State Lattice using Queues for Intervals

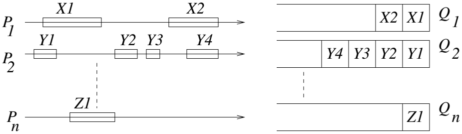
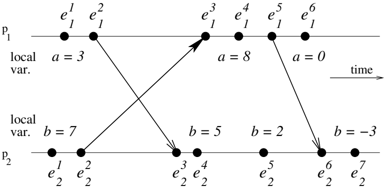
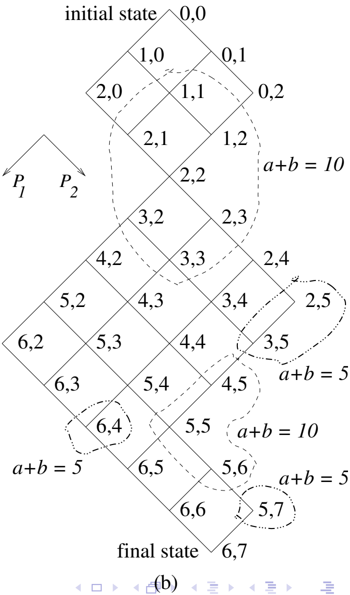


Fig 11.4: Queues $Q_1 \dots Q_n$ for each of the n processes

Conjunctive Predicates

- ϕ can be expressed as the conjunction $\wedge_{i \in N} \phi_i$, where ϕ_i is local to process i.
- If ϕ is false in any cut C, then there is at least one process i such that the local state of i in cut C will never form part of any other cut C' such that ϕ is true in C'.
- If ϕ is false in some cut C, we can advance the local state of at least one process to the next event, and then evaluate the predicate in the resulting cut
- This gives a $O(mn)$ time algorithm, where m is the number of events at any process.
- In example, *Possibly*$(a = 3 \wedge b = 2)$ and *Definitely*$(a = 3 \wedge b = 7)$ and true.





Detecting Conjunctive Predicates

- Global state-based approach: $O(mn)$ time
- Interval-based approach: interval X represents duration in which ϕ_i true at i. Standard min and max semantics

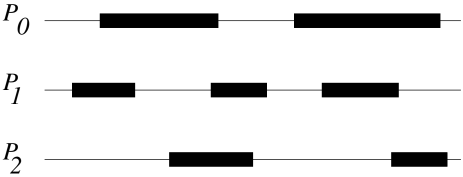


Fig 11.5

Optimization: If no send or receive between start of interval and the end of next interval at that process, the intervals have exact same relation w.r.t. other intervals at other processes.

Detecting Conjunctive Predicates, over Multiple Processes

For two processes:

- $Definitely(\phi)$: $min(X) \prec max(Y) \bigwedge min(Y) \prec max(X)$
- $\overline{Possibly(\phi)}$: $max(X) \prec min(Y) \bigvee max(Y) \prec min(X)$

For multiple processes:

- $Definitely(\phi)$: $\bigwedge_{i,j \in N} Definitely(\phi_i \bigwedge \phi_j)$
- $Possibly(\phi)$: $\bigwedge_{i,j \in N} Possibly(\phi_i \bigwedge \phi_j)$

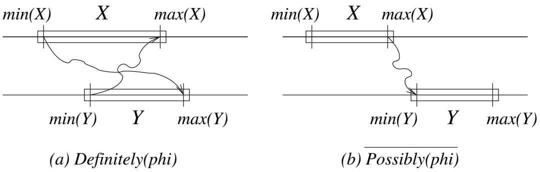


Fig 11.6

Centralized Algorithm for *Possbly*/*Definitely*

queue of *Log*: $Q_1, Q_2, \ldots Q_n \longleftarrow \perp$

set of int: *updatedQueues*, *newUpdatedQueues* $\longleftarrow \{\}$

<u>On receiving interval from process P_z at P_0:</u>

(1) Enqueue the interval onto queue Q_z

(2) **if** (number of intervals on Q_z is 1) **then**

(3) *updatedQueues* $\longleftarrow \{z\}$

(4) **while** (*updatedQueues* is not empty)

(5) *newUpdatedQueues* $\longleftarrow \{\}$

(6) **for** each $i \in$ *updatedQueues*

(7) **if** (Q_i is non-empty) **then**

(8) $X \longleftarrow$ head of Q_i

(9) **for** $j = 1$ **to** n

(10) **if** (Q_j is non-empty) **then**

(11) $Y \longleftarrow$ head of Q_j

(12) **if** ($min(X) \not\prec max(Y)$) **then** // *Definitely*

(13) *newUpdatedQueues* $\longleftarrow \{j\} \cup$ *newUpdatedQueues*

(14) **if** ($min(Y) \not\prec max(X)$) **then** // *Definitely*

(15) *newUpdatedQueues* $\longleftarrow \{i\} \cup$ *newUpdatedQueues*

(12') **if** ($max(X) \prec min(Y)$) **then** // *Possibly*

(13') *newUpdatedQueues* $\longleftarrow \{i\} \cup$ *newUpdatedQueues*

(14') **if** ($max(Y) \prec min(X)$) **then** // *Possibly*

(15') *newUpdatedQueues* $\longleftarrow \{j\} \cup$ *newUpdatedQueues*

(16) Delete heads of all Q_k where $k \in$ *newUpdatedQueues*

(17) *updatedQueues* $\longleftarrow$ *newUpdatedQueues*

(18) **if** (all queues are non-empty) **then**

(19) solution found. Heads of queues identify intervals that form the solution.

Centralized Algorithm for *Possbly*/*Definitely*: Complexity

- Lines (12)-(15) for *Definitely*; lines (12')-(15') for *Possibly*
- sets *updatedQueues* and *newupdatedQueues* (temp)
- For each comparison, if desired modality is not satisfied, at least one of the two intervals gets deleted, by first being placed in *newUpdatedQueues*
- If every queue is non-empty and the queue-heads cannot be pruned, then the queue heads form a solution
- Termination: if a solution exists, detected in lines (18)-(19).
- Complexity at P_0 (ito n, p, M – # msgs sent in execution):
  - Message complexity: $min(pn, 4M)$ control messages, each of size $2n$
  - Space complexity: $min(pn, 4M) \cdot 2n$
  - Time complexity: When an interval is compared with others, $O(n)$ steps. Hence, $O(n \cdot min(pn, 4M))$.

State-based Algorithm for *Possibly*(ϕ) (conjunctive ϕ)

Possibly(ϕ) iff consistent global state where
(mutually concurrent) $\forall i, \forall j,\ s_i \not\prec s_j \wedge s_j \not\prec s_i$

- Whenever ϕ_i becomes true, P_i sends vector timestmp to P_0
- ith row of GS matrix tracks P_i's tmstp;
- *Valid*[i] indicates whether that local state of P_i can be part of the solution
- if *Valid*[i] is false, then new state from P_i is considered

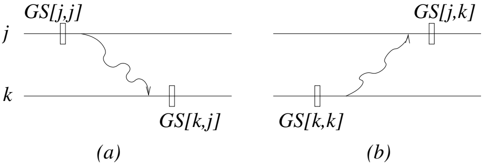


Fig 11.7: (a) P_j's old state is invalid. (b) P_k's old state is invalid.

State-based Algorithm for *Possibly*(ϕ) (conjunctive ϕ)

integer: $GS[1 \ldots n, 1 \ldots n]$; //$i$th row tracks vector time of P_i
boolean: $Valid[1 \ldots n]$; //$Valid[j] = 0$ implies P_j state $GS[j, \cdot]$ to be advanced
queue of array of integer: $Q_1, Q_2, \ldots Q_n \longleftarrow \perp$; //$Q_i$ stores timestamp info from P_i

while $(\exists j \mid Valid[j] = 0)$ **do** //P_j's state $GS[j, \cdot]$ is not consistent with others
    if ($Q_j = \perp$ and P_j has terminated) **then**
        return(0);
    else
        await Q_j becomes non-empty;
        $GS[j, 1 \ldots n] \longleftarrow head(Q_j)$; //Consider next state of P_j for consistency
        $dequeue(head(Q_j))$;
        $Valid[j] \longleftarrow 1$;
    for $k = 1$ **to** n **do** //Check P_j's state w.r.t. P_k's state (for every P_k)
        if $k \neq j$ **and** $Valid[k] = 1$ **then**
            if $GS[j, j] \leq GS[k, j]$ **then** //P_j's state is inconsistent with P_k's state
                $Valid[j] \longleftarrow 0$; //next state of P_j needs to be considered
            else if $GS[k, k] \leq GS[j, k]$ **then** //P_k's state inconsistent with P_j's state
                $Valid[k] \longleftarrow 0$; //next state of P_k needs to be considered
return(1).

State-based Algorithm for *Possibly*(ϕ) (conjunctive ϕ)

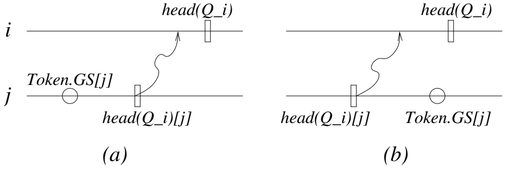
Let m be # local states at any process; let M be # messages sent in the execution

- Termination: when $Valid[j] = 1$ for all j
- Time complexity: $O(n^2m)$
- Space complexity: $O(n^2m)$
- Message complexity: $2M$ control messages, each of size n

Distributed state-based algorithm for $Possibly(\phi)$

(conjunctive ϕ)

- $Token.GS[1..n]$ gives the timestamp of the latest cut under consideration as a candidate solution.
- $Token.Valid[1..n]$. $Token.Valid[i] = 0$ implies all P_i local states up to $Token.GS[i]$ cannot be part of the solution. So from Q_i, consider the earliest local state such that local timestamp is greater than $Token.GS[i]$.
- $Token.GS[i], .Valid[i]$ entries are set accordingly.
- Consistency checks made between $head(Q_i)[j]$ and $Token.GS[j]$ (for all j), to determine whether the various $Token.Valid$ entries should be 1 or 0.
- Token passed to any process for which $Token.Valid = 0$.



P_i tests whether P_j's candidate local state $Token.GS[j]$ is consistent with $head(Q_i)[i]$, which is assigned to $Token.GS[i]$. The two possibilities are illustrated. (a) Not consistent. (b) Consistent.

Distributed State-based Algorithm for *Possibly*(ϕ)

struct token {
  integer: $GS[1 \ldots n]$; //Earliest possible global state as a candidate solution
  boolean: $Valid[1 \ldots n]$; } *Token*; //$Valid[j] = 0$ indicates P_j's state $GS[j]$ is invalid
queue of array of integer: $Q_i \longleftarrow \perp$

Initialization. *Token* is at a randomly chosen process.

On receiving *Token* at P_i
(1) **while** ($Token.Valid[i] = 0$) **do** // $Token.GS[i]$ is the latest state of P_i known to be inconsistent
(2) **await** (Q_i to be nonempty); //with other candidate local state of P_j, for some j
(3) **if** $((head(Q_i))[i] > Token.GS[i])$ **then**
(4) $Token.GS[i] \longleftarrow (head(Q_i))[i]$; // earliest possible state of P_i that can be part of solution
(5) $Token.Valid[i] \longleftarrow 1$; //is written to *Token* and its validity is set.
(6) **else dequeue** $head(Q_i)$;
(7) **for** $j = 1$ **to** n $(j \neq i)$ **do** // for each other process P_j: based on P_i's local state, determine whether
(8) **if** $j \neq i$ **and** $(head(Q_i))[j] \geq Token.GS[j]$ **then** // P_j's candidate local state (in *Token*)
(9) $Token.GS[j] \longleftarrow (head(Q_i))[j]$; // is consistent. If not, P_j needs to consider a
(10) $Token.Valid[j] \longleftarrow 0$; // later candidate state with a timestamp $> head((Q_i)[j]$
(11) **dequeue** $head(Q_i)$;
(12) **if** for some k, $Token.Valid[k] = 0$ **then**
(13) **send** *Token* to P_k;
(14) **else return(1)**.

Distributed State-based Algorithm for *Possibly*(ϕ): Complexity

- Termination: algorithm finds a solution when $Token.Valid[j]$ is 1, for all j (line (14)). If a solution is not found, the code hangs in line (2). The code can be modified to terminate unsuccessfully in line (2) by modeling an explicit 'process terminated' state.
- Time complexity: $O(mn^2)$ across all processes
- Space complexity: $O(mn^2)$ across all processes
- Message complexity: Token makes $O(mn)$ hops; token size is $2n$ integers

Distributed Interval-based Algorithm for *Definitely*(ϕ)

Define $I_i \hookrightarrow I_j$ as: $\min(I_i) \prec \max(I_j)$.
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 $I_j \hookrightarrow I_i$.

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:
$Log_i.start \longleftarrow V_i$.
When an interval ends:
$Log_i.end \longleftarrow V_i$
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.

Distributed Interval-based Algorithm for *Definitely*(ϕ)

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

- Token-holder P_i sends REQ msg containing $Log_i.start[i]$ and $Log_i.end[j]$ pertaining to its interval X_i, to P_j (all other processes P_j)
- Each P_j then checks if its interval Y_j satisfies the Definitely condition.
- If not, one or both intervals are deleted. This is conveyed to P_i using REPLY messages.
- If $T.updatedQueues$ is empty, intervals at each queue head form solution
- Otherwise, token is forwarded to some process whose id is in $T.updatedQueues$
- If a solution exists, it is detected; if a solution is detected, it is correct.

Distributed Interval-based Algorithm for *Definitely*(ϕ)

(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) $T.updatedQueues \longleftarrow \{1, 2, \ldots, n\}$.
(3) *RcvToken* : When P_i receives a token T
(3a) Remove index i from $T.updatedQueues$
(3b) **wait until** (Q_i is nonempty)
(3c) $REQ.start \longleftarrow Log_i.start[i]$, where Log_i is the log at head of Q_i
(3d) **for** $j = 1$ to n
(3e) $REQ.end \longleftarrow Log_i.end[j]$
(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) $T.updatedQueues \longleftarrow T.updatedQueues \cup REP_j.updated$
(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) $REP.updated \longleftarrow \phi$
(4c) $Y \longleftarrow$ head of local queue Q_j
(4d) $V_i^-(X)[i] \longleftarrow REQ.start$ and $V_i^+(X)[j] \longleftarrow REQ.end$
(4e) Determine $X \hookrightarrow Y$ and $Y \hookrightarrow X$
(4f) **if** ($Y \not\hookrightarrow X$) **then** $REP.updated \longleftarrow REP.updated \cup \{i\}$
(4g) **if** ($X \not\hookrightarrow Y$) **then**
(4h) $REP.updated \longleftarrow REP.updated \cup \{j\}$
(4i) Dequeue Y from local queue Q_j
(4j) Send reply REP to P_i.

Distributed Interval-based Algorithm for *Definitely*(ϕ): Complexity

p: max no. of intervals at any process;
m: max no. of messages sent per process

- Space complexity: Worst case across all processes is $O(\min(n^2p, n^2m))$. This is also worst case space at any process. Total # Logs across all processes is $\min(2n^2p, 2n^2m)$.
- Time complexity: Worst case across all processes is $O(\min(pn^2, mn^2))$. Worst case at a process is $O(\min(pn, mn^2))$
- Message complexity: Total # Logs across all processes is $O(\min(np, mn))$. This is the worst case number of messages. Worst case message space overhead is $O(n \min(np, mn))$.