Introduction and Uses of Predicate Detection

- Industrial process control, distributed debugging, computer-aided verification, sensor networks
- E.g., $\psi$ 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., $\phi \implies \square \phi$
  - Predicate $\phi$ 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
⇒ 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.
⇒ the stable property will necessarily be true.
Unstable Predicates: Challenges in Detection

Challenges:
- unpredictable propagation times, unpredictable process scheduling ⇒
  - multiple executions pass through different global states;
  - predicate may be true in some executions and false in others
- No global time ⇒
  - 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 ⇒ 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 ⇒ define predicate on all the observations of the distributed program
Modalities on Predicates

- **Possibly** $(\phi)$: There exists a consistent observation of the execution such that predicate $\phi$ holds in a global state of the observation.

- **Definitely** $(\phi)$: For every consistent observation of the execution, there exists a global state of it in which predicate $\phi$ holds.

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

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

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

Assume state lattice is available.

Global state $GS = \{s_{k_1}, s_{k_2}, \ldots, s_{k_n}\}$ is abbreviated $GS_{k_1}, k_2, \ldots, k_n$.

Level of a global state $\langle s_{k_i}(\forall i) \rangle$ is $\sum_{i=1}^{n} k_i$.

Possibly ($\phi$): examine the state lattice level-by-level, from the initial state at level 0 up to the final state. If $\phi$ is true, return(1).

Definitely ($\phi$): Sufficient but not necessary that all states at some level satisfy $\phi$.

Example execution and the corresponding state lattice.

The states belonging to $\text{Reach} \phi$ (line (2d)) at any level are either marked by shaded circles or clear circles.

The states belonging to $\text{Reach} \phi$ (line (2f)) at any level are marked by clear circles.

In line (2b), when $\text{lvl} = 11$, $\text{Reach} \phi$ becomes $\emptyset$ and the algorithm exits from the loop.

- state in which predicate is true
- state reachable without predicate being true
Detecting Relational Predicates, on-line, centralized

(variables)

set of global states \( \text{Reach}_\phi, \text{Reach}_\text{Next}_\phi \leftarrow \{GC^{0,0,\ldots,0}\} \)

int \( lvl \leftarrow 0 \)

(1) \textit{Possibly}(\( \phi \))

(1a) while (no state in \( \text{Reach}_\phi \) satisfies \( \phi \)) do

(1b) if (\( \text{Reach}_\phi = \{\text{final state}\} \)) then return false;

(1c) \( lvl \leftarrow lvl + 1; \)

(1d) \( \text{Reach}_\phi \leftarrow \{\text{states at level} \ lvl \}; \)

(1e) return true.

(2) \textit{Definitely}(\( \phi \))

(2a) remove from \( \text{Reach}_\phi \) those states that satisfy \( \phi \)

(2b) \( lvl \leftarrow lvl + 1; \)

(2c) while (\( \text{Reach}_\phi \neq \emptyset \)) do

(2d) \( \text{Reach}_\text{Next}_\phi \leftarrow \{\text{states of level} \ lvl \text{ reachable from a state in} \ \text{Reach}_\phi\}; \)

(2e) remove from \( \text{Reach}_\text{Next}_\phi \) all the states satisfying \( \phi \);

(2f) if \( \text{Reach}_\text{Next}_\phi = \{\text{final state}\} \) then return false;

(2g) \( lvl \leftarrow lvl + 1; \)

(2h) \( \text{Reach}_\phi \leftarrow \text{Reach}_\text{Next}_\phi; \)

(2i) return true.
Centralized Algorithm for Relational Predicates (contd.)

Definitely($\phi$):

- Replacing line (1a) by: “(some state in $\text{Reach}_\phi$ satisfies $\neg\phi$)” will not work!
- The algorithm examines the state lattice level-by-level:
  1. Tracks states at each level in which $\phi$ 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.
- $\text{Reach\_Next}_\phi$ at level $lvl$ contains the set of states at level $lvl$ that are reachable from the initial state without passing through any state satisfying $\phi$.
- return(1) if $\text{Reach\_Next}(\phi)$ becomes $\emptyset$, else return(0).
- In example, at $lvl = 11$, $\text{Reach\_Next}(\phi)$ 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 $GS^1_{min},k_2,\ldots,k_n$ 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 $s_j^{k_j}$.

\[
(\forall j) \ WC(s_{\text{j}}^{k_{\text{j}}})[j] = WC(s_{i}^{k_{i}})[j]
\]

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 $WC(s_i^{k_i})$.

- The latest global state $GS^1_{max},k_2,\ldots,k_n$ 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 $P_j$.

\[
(\forall j) \ WC(s_{\text{j}}^{k_{\text{j}}})[i] < WC(s_{i}^{k_{i}})[i] \leq WC(s_{j}^{k_{j+1}})[i]
\]

Hence, the highest level of the state lattice, in which local state $s_i^{k_i}$ participates, is $\sum_{j=1}^{n} WC(s_j^{k_j})[j]$. 
Constructing State Lattice using Queues for Intervals

Fig 11.4: Queues $Q_1 \ldots Q_n$ for each of the $n$ processes
Conjunctive Predicates

- $\phi$ can be expressed as the conjunction $\land_{i \in N} \phi_i$, where $\phi_i$ is local to process $i$.

- If $\phi$ 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 $\phi$ is true in $C'$.

- If $\phi$ 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 $(a = 3 \land b = 2)$ and Definitely $(a = 3 \land b = 7)$ and true.
Detecting Conjunctive Predicates

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

![Fig 11.5](image)

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**($\phi$): $\min(X) \prec \max(Y) \land \min(Y) \prec \max(X)$
- **Possibly**($\phi$): $\max(X) \prec \min(Y) \lor \max(Y) \prec \min(X)$

For multiple processes:

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

![Diagram](https://via.placeholder.com/150)

(a) **Definitely**($\phi$)  
(b) **Possibly**($\phi$)  

**Fig 11.6**
Centralized Algorithm for Possibly/Definitely

queue of Log: $Q_1, Q_2, \ldots Q_n \leftarrow \bot$

set of int: $updatedQueues, newUpdatedQueues \leftarrow \{\}$

On receiving interval from process $P_z$ at $P_0$:

1. Enqueue the interval onto queue $Q_z$
2. if (number of intervals on $Q_z$ is 1) then
   3. $updatedQueues \leftarrow \{z\}$
   4. while ($updatedQueues$ is not empty)
      5. $newUpdatedQueues \leftarrow \{\}$
      6. for each $i \in updatedQueues$
         7. if ($Q_i$ is non-empty) then
            8. $X \leftarrow$ head of $Q_i$
            9. for $j = 1$ to $n$
               10. if ($Q_j$ is non-empty) then
                    11. $Y \leftarrow$ head of $Q_j$
                    12. if ($min(X) \not\prec max(Y)$) then // Definitely
                         13. $newUpdatedQueues \leftarrow \{j\} \cup newUpdatedQueues$
                    14. if ($min(Y) \not\prec max(X)$) then // Definitely
                         15. $newUpdatedQueues \leftarrow \{i\} \cup newUpdatedQueues$
                    16. if ($max(X) \prec min(Y)$) then // Possibly
                         17. $newUpdatedQueues \leftarrow \{i\} \cup newUpdatedQueues$
                    18. if ($max(Y) \prec min(X)$) then // Possibly
                         19. $newUpdatedQueues \leftarrow \{j\} \cup newUpdatedQueues$
               10' if ($max(X) \prec min(Y)$) then // Possibly
                    16' Delete heads of all $Q_k$ where $k \in newUpdatedQueues$
                    17' $updatedQueues \leftarrow newUpdatedQueues$
            12' if (all queues are non-empty) then
                18' solution found. Heads of queues identify intervals that form the solution.

Centralized Algorithm for *Possibly/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, $O(n \cdot \min(pn, 4M))$. 

A. Kshemkalyani and M. Singhal (Distributed Computing)
State-based Algorithm for \( \text{Possibly}(\phi) \) (conjunctive \( \phi \))

\( \text{Possibly}(\phi) \) iff consistent global state where (mutually concurrent) \( \forall i, \forall j, \ s_i \not\prec s_j \land s_j \not\prec s_i \)

- Whenever \( \phi_i \) becomes true, \( P_i \) sends vector timestamp to \( P_0 \)
- \( i \)th row of \( GS \) matrix tracks \( P_i \)'s timestamp;
- \( \text{Valid}[i] \) indicates whether that local state of \( P_i \) can be part of the solution
- if \( \text{Valid}[i] \) is false, then new state from \( P_i \) is considered

\[
\begin{align*}
\text{GS}[j,j] & & \text{GS}[j,k] \\
\text{GS}[k,j] & & \text{GS}[k,k]
\end{align*}
\]

Fig 11.7: (a) \( P_j \)'s old state is invalid. (b) \( P_k \)'s old state is invalid.
State-based Algorithm for **Possibly**($\phi$) (conjunctive $\phi$)

\begin{verbatim}
integer: GS[1...n, 1...n]; //ith row tracks vector time of Pi
boolean: Valid[1...n]; //Valid[j] = 0 implies P_j state GS[j, ·] to be advanced
queue of array of integer: Q_1, Q_2, ..., Q_n ←⊥; //Q_i stores timestamp info from P_i

(1) while (∃j | Valid[j] = 0) do //P_j’s state GS[j, ·] is not consistent with others
(2)   if (Q_j = ⊥ and P_j has terminated) then
(3)     return(0);
(4)   else
(5)     await Q_j becomes non-empty;
(6)     GS[j, 1...n] ← head(Q_j); //Consider next state of P_j for consistency
(7)     dequeue(head(Q_j));
(8)     Valid[j] ← 1;
(9)   for k = 1 to n do //Check P_j’s state w.r.t. P_k’s state (for every P_k)
(10)  if k ≠ j and Valid[k] = 1 then
(11)    if GS[j, j] ≤ GS[k, j] then //P_j’s state is inconsistent with P_k’s state
(12)      Valid[j] ← 0; //next state of P_j needs to be considered
(13)    else if GS[k, k] ≤ GS[j, k] then //P_k’s state inconsistent with P_j’s state
(14)      Valid[k] ← 0; //next state of P_k needs to be considered
(15)  return(1).
\end{verbatim}
State-based Algorithm for $Possibly(\phi)$ (conjunctive $\phi$)

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: $O(n^2 m)$
- Space complexity: $O(n^2 m)$
- Message complexity: $2M$ control messages, each of size $n$
Distributed state-based algorithm for \( \text{Possibly}(\phi) \) (conjunctive \( \phi \))

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

\[
\begin{align*}
\text{Token.GS}[j] & \quad \text{head}(Q_i)[j] \\
\text{head}(Q_i) & \quad \text{head}(Q_i) \\
\text{head}(Q_i) & \\
\end{align*}
\]

(a) Not consistent. (b) Consistent.

\( \text{Token.GS}[j] \) is consistent with \( \text{head}(Q_i)[i] \), which is assigned to \( \text{Token.GS}[i] \). The two possibilities are illustrated.
Distributed State-based Algorithm for \textit{Possibly}(\phi)

\begin{verbatim}
struct token {
    integer: GS[1...n]; //Earliest possible global state as a candidate solution
    boolean: Valid[1...n]; } Token; //Valid[j] = 0 indicates P_j's state GS[j] is invalid
queue of array of integer: Q_i ←⊥

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] ← (head(Q_i))[i]; // earliest possible state of P_i that can be part of solution
(5)        Token.Valid[i] ← 1; //is written to Token and its validity is set.
(6)    else dequeue head(Q_i);
(7)    for j = 1 to n (j ≠ i) do  // for each other process P_j: based on P_i's local state, determine whether
(8)        if j ≠ i and (head(Q_i))[j] ≥ Token.GS[j] then  // P_j's candidate local state (in Token)
(9)            Token.GS[j] ← (head(Q_i))[j]; // is consistent. If not, P_j needs to consider a
(10)        Token.Valid[j] ← 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).
\end{verbatim}
Distributed State-based Algorithm for Possibly($\phi$): 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 \( I_i \leftarrow I_j \) as: \( \min(I_i) \prec \max(I_j) \).

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

```
type Log
    start: array[1...n] of integer;
    end: array[1...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.
```
### Distributed Interval-based Algorithm for \textit{Definitely}(\phi)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REQUEST</strong></td>
<td></td>
</tr>
<tr>
<td>\texttt{start} : integer;</td>
<td>contains Log\textsubscript{i}.	exttt{start}[j] for the interval at the queue head of P\textsubscript{i}, when sending to P\textsubscript{j}</td>
</tr>
<tr>
<td>\texttt{end} : integer;</td>
<td>contains Log\textsubscript{i}.	exttt{end}[j] for the interval at the queue head of P\textsubscript{i}, when sending to P\textsubscript{j}</td>
</tr>
<tr>
<td><strong>REPLY</strong></td>
<td></td>
</tr>
<tr>
<td>\texttt{updated} : set of integer;</td>
<td>contains the indices of the updated queues</td>
</tr>
<tr>
<td><strong>TOKEN</strong></td>
<td></td>
</tr>
<tr>
<td>\texttt{updatedQueues} : set of integer;</td>
<td>contains the index of all the updated queues</td>
</tr>
</tbody>
</table>

- Token-holder P\textsubscript{i} sends \texttt{REQ} msg containing Log\textsubscript{i}.	exttt{start}[i] and Log\textsubscript{i}.	exttt{end}[j] pertaining to its interval X\textsubscript{i}, to P\textsubscript{j} (all other processes P\textsubscript{j})
- Each P\textsubscript{j} then checks if its interval Y\textsubscript{j} satisfies the Definitely condition.
- If not, one or both intervals are deleted. This is conveyed to P\textsubscript{i} using \texttt{REPLY} messages.
- If T.\texttt{updatedQueues} is empty, intervals at each queue head form solution
- Otherwise, token is forwarded to some process whose id is in T.\texttt{updatedQueues}
- If a solution exists, it is detected; if a solution is detected, it is correct.
Distributed Interval-based Algorithm for Definitely($\phi$)

(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) $T\.updatedQueues \leftarrow \{1, 2, \ldots, n\}$.

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

\[ p: \text{ max no. of intervals at any process; } \]
\[ m: \text{ max no. of messages sent per process} \]

- **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 \( O(n \min(np, mn)) \).