Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms

CS 249 Project
Fall 2005
Wing Wong
Outline

- Introduction
- Asynchronous distributed systems, distributed computations, consistency
- Two different strategies to construct global states
  - Monitor passively observes the system (reactive-architecture)
  - Monitor actively interrogates the system (snapshot protocol)
- Properties of global predicates
- Sample applications: deadlock detection and debugging
Introduction

- global state = union of local states of individual processes
- many problems in distributed computing require:
  - construction of a global state and
  - evaluation of whether the state satisfies some predicate $\Phi$
- difficulties:
  - uncertainties in message delays
  - relative speeds of computations
- global state obtained can be obsolete, incomplete, or inconsistent
Distributed Systems

- collection of sequential processes $p_1, p_2, \ldots, p_n$
- unidirectional communication channels between pairs of processes
- reliable channels
- messages may be delivered out of order
- network strongly connected (not necessarily completely)
Asynchronous Distributed Systems

- no bounds on relative process speeds
- no bounds on message delays
- no synchronized local clocks
- communication is the only possible mechanism for synchronization
Distributed Computations

- distributed program executed by a collection of processes
- each process executes a sequence of events
- communication through events $send(m)$ and $receive(m)$, $m$ as message identifier
Distributed Computations

- \[ h_i = e_{i1}^1 e_{i2}^2 \ldots \]
  - **local history** of process \( p_i \)
  - canonical enumeration
  - total order imposed by sequential execution

- \[ h_i^k = e_{i1}^1 e_{i2}^2 \ldots e_i^k \]
  - initial prefix of \( h_i \) containing first \( k \) events

- \[ H = h_1 \cup \ldots \cup h_n \]
  - **global history** containing all events
  - does not specify relative timing between events
Distributed Computations

- to order events, define binary relation “→” to capture “cause-and-effect”:
  1. If $e_i^k, e_i^l \in h_i$ and $k < l$, then $e_i^k \rightarrow e_i^l$,
  2. If $e_i = \text{send}(m)$ and $e_j = \text{receive}(m)$, then $e_i \rightarrow e_j$,
  3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

- $e \rightarrow e'$ if and only if $e$ “causally precedes” $e'$
- concurrent events: neither $e \rightarrow e'$ nor $e' \rightarrow e$, write $e \parallel e'$
- distributed computation $=$ partially ordered set defined by $(H, \rightarrow)$
Distributed Computations

Figure 1. Space-Time Diagram Representation of a Distributed Computation

\[ e_2^1 \rightarrow e_3^6 ; e_2^2 \parallel e_3^6 \]
Global States, Cuts and Runs

- $\sigma_i^k$
  - local state of process $\rho_i$ after event $e_i^k$

- $\Sigma = (\sigma_1, \ldots, \sigma_n)$
  - global state of distributed computation
  - n-tuple of local states

- cut $C = h_1^{c_1} \cup \ldots \cup h_n^{c_n}$ or $(c_1, \ldots, c_n)$
  - subset of global history $H$
Global States, Cuts and Runs

- \((\sigma_1^{c_1}, \ldots, \sigma_n^{c_n})\)
  - global state correspond to cut \(C\)

- \((e_1^{c_1}, \ldots, e_n^{c_n})\)
  - frontier of cut \(C\)
  - set of last events

- run
  - a total ordering \(R\) including all events in global history
  - consistent with each local history
Global States, Cuts and Runs

- cut $C = (5,2,4)$; cut $C' = (3,2,6)$
- a consistent run $R = e_1 e_1 e_2 e_1 e_3 e_3 e_2 e_2 e_1 e_5 e_1 e_4 e_1 e_5 e_3 e_6 e_2 e_1$
Consistency

- cut $C$ is consistent if for all events $e$ and $e'$
  \[(e \in C) \land (e' \rightarrow e) \Rightarrow e' \in C.\]
  - closed under the causal precedence relation
- **consistent global state** corresponds to a consistent cut
- run $R$ is consistent if for all events, $e \rightarrow e'$ implies $e$ appears before $e'$ in $R$
Consistency

- run $R = e^1e^2\ldots$ results in a sequence of global states $\Sigma^0\Sigma^1\Sigma^2$
- $\Sigma^i$ is obtained from $\Sigma^{i-1}$ by some process executing event $e^i$, or $\Sigma^{i-1}$ leads to $\Sigma^i$
- denote the transitive closure of the leads-to relation by $\sim>_R$
- $\Sigma'$ is reachable from $\Sigma$ in run $R$ iff $\Sigma \sim>_R \Sigma'$
Lattice of Global States

- **lattice** = set of all consistent global states, along with leads-to relation
- $\Sigma^{k_1 \ldots k_n}$ = shorthand for global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$
- $k_1 + \ldots + k_n$ = level of lattice

Figure 3. A Distributed Computation and the Lattice of its Global States
Lattice of Global States

- **path** = sequence of global states of increasing level (downwards)
- each path corresponds to a consistent run
- a possible path: \( \Sigma^00 \Sigma^01 \Sigma^11 \Sigma^21 \Sigma^31 \Sigma^32 \Sigma^42 \Sigma^43 \Sigma^44 \Sigma^54 \Sigma^64 \Sigma^65 \)

Figure 3. A Distributed Computation and the Lattice of its Global States
Observing Distributed Computations (reactive-architecture)

- processes notify monitor process $p_0$ whenever they execute an event
- monitor constructs observation as the sequence of events corresponding to the notification messages
- problem:
  - observation may be inconsistent due to variability in notification message delays
Observing Distributed Computations

\[ R = e_1^3 e_1^1 e_2^1 e_2^1 e_3^3 e_3^1 e_3^4 e_3^4 e_1^5 e_1^5 e_3^6 e_3^6 e_2^6 e_2^6 \]

\[ O_1 = e_2^1 e_1^1 e_3^1 e_3^3 e_3^4 e_2^2 e_3^1 e_2^3 e_1^4 e_1^5 \ldots \]

\[ O_2 = e_1^1 e_3^1 e_2^1 e_2^2 e_3^3 e_3^4 e_3^2 e_3^5 e_3^6 \ldots \]

\[ O_3 = e_3^1 e_2^1 e_1^1 e_3^2 e_3^3 e_3^4 e_3^4 e_1^2 e_2^5 e_3^6 \ldots \]
Observing Distributed Computations

- any permutation of run $R$ is a possible observation

- we need:
  - delivery rule at monitor process to restore message order

- we have First-In-First-Out (FIFO) delivery using sequence number for all source-destination pair $p_i, p_j$:
  - $send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$
Delivery Rule 1

- assume:
  - global real-time clock
  - message delays bound by $\delta$
- process includes timestamp (real-time clock value) when notifying $p_0$ of local event $e$

**DR1:** At time $t$, deliver all received messages with timestamps up to $t - \delta$ in increasing timestamp order
Delivery Rule 1

- let $RC(e)$ denotes value of global clock when $e$ is executed

- real-time clock satisfies **Clock Condition:**
  
  $e \rightarrow e' \Rightarrow RC(e) < RC(e')$

- but **logical clocks** also satisfies clock condition...
Logical Clocks

- event orderings based on increasing clock values
- $LC(e_i)$ denotes value of logical clock when $e_i$ is executed by $p_i$
- each sent message $m$ contains timestamp $TS(m)$
- update rules by $p_i$ at occurrence of $e_i$:

$$LC(e_i) := \begin{cases} 
LC + 1 & \text{if } e_i \text{ is an internal or send event} \\
\max\{LC, TS(m)\} + 1 & \text{if } e_i = receive(m) 
\end{cases}$$
Logical Clocks

Figure 4. Logical Clocks
Delivery Rule 2

- replace real-time clock by logical clock
- need **gap-detection** property:
  - given events $e$, $e'$ where $LC(e) < LC(e')$, determine if some event $e''$ exists such that $LC(e) < LC(e'') < LC(e')$
  - message is “**stable**” at $p$ if no future messages with timestamps smaller than $TS(m)$ can be received by $p$
Delivery Rule 2

- with FIFO, when $p_0$ receives $m$ from $p_i$ with timestamp $TS(m)$, can be certain no other message $m'$ from $p_i$ with $TS(m') \leq TS(m)$

- message $m$ at $p_0$ guaranteed stable when $p_0$ has received at least one message from all other processes with timestamps $> TS(m)$

**DR2**: Deliver all received messages that are stable at $p_0$ in increasing timestamp order
Strong Clock Condition

- DR1, DR2 assume $RC(e) < RC(e')$ (or $LC(e) < LC(e')$) $\Rightarrow e \rightarrow e'$
- recall RC and LC guarantee clock condition: $e \rightarrow e' \Rightarrow RC(e) < RC(e')$
- DR1, DR2 can unnecessarily delay delivery
- want timing mechanism $TC$ that gives Strong Clock Condition:

\[ e \rightarrow e' \equiv TC(e) < TC(e') \]
Timing Mechanism 1 - Causal Histories

- **causal history** as “clock” value
  - set of all events that causally precede event $e$:
    \[
    \theta(e) = \{ e' \in H \mid e' \rightarrow e \} \cup \{ e \}
    \]
  - smallest consistent cut that includes $e$
  - **projection** of $\theta(e)$ on process $p_i$: $\theta_i(e) = \theta(e) \cap h_i$
Timing Mechanism 1 - Causal Histories

Figure 6. Causal History of Event $e_1^4$

$\theta(e_1^4) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_2^1, e_2^3, e_3^1, e_3^2, e_3^3\}$
Timing Mechanism 1 - Causal Histories

To maintain causal histories:

- $\theta$ initially empty
- If $e_i$ is an internal or send event
  - $\theta(e_i) = \{e_i\} \cup \theta(\text{previous local event of } p_i)$
- If $e_i$ = receive of message $m$ by $p_i$ from $p_j$
  - $\theta(e_i) = \{e_i\} \cup \theta(\text{previous local event of } p_i) \cup \theta(\text{corresponding send event at } p_j)$
Timing Mechanism 1 - Causal Histories

new send event:

new receive event:

new event $e_1^5$

new event $e_2^3$
Timing Mechanism 1 - Causal Histories

- can interpret clock comparison as set inclusion:
  \[ e \rightarrow e' \equiv \theta(e) \subset \theta(e') \]
  (why not set membership, \([e \rightarrow e' \equiv e \in \theta(e')]\)?)

- unfortunately, causal histories grow too rapidly
Timing Mechanism 2 - Vector Clocks

- note:
  - projection $\theta_i(e) = h_i^k$ for some unique $k$
  - $e_i^r \in \theta_i(e)$ for all $r < k$
  - can use single number $k$ to represent $\theta_i(e)$
  - $\theta(e) = \theta_1(e) \cup \ldots \cup \theta_n(e)$

- represent entire causal history by n-dimensional **vector clock** $VC(e)$, where for all $1 \leq i \leq n$
  - $VC(e)[i] = k$, if and only if $\theta_i(e) = h_i^k$
Timing Mechanism 2 - Vector Clocks

Figure 7. Vector Clocks
Timing Mechanism 2 - Vector Clocks

- To maintain vector clock:
  - each process $p_i$ initializes $VC$ to contain all zeros
  - update rules by $p_i$ at occurrence of $e_i$:
    
    $VC(e_i)[i] := VC[i] + 1$ if $e_i$ is an internal or send event
    $VC(e_i) := \max\{VC, TS(m)\}$ if $e_i = receive(m)$
    $VC(e_i)[i] := VC[i] + 1$

- $VC(e_i)[i] \equiv$ number of events $p_i$ has executed up to and including $e_i$
- $VC(e_i)[j] \equiv$ number of events of $p_j$ that causally precede event $e_i$ of $p_i$
Timing Mechanism 2 - Vector Clocks

causal histories

vector clocks

new send event:

new receive event:
Vector Clock Comparison

- Define “less than” relation:
  \( V < V' \equiv (V \neq V') \land (\forall 1 \leq k \leq n: V[k] \leq V'[k]) \)
Properties of Vector Clocks

1. **Strong Clock Condition:**
   \[ e \rightarrow e' \equiv VC(e) < VC(e') \]

2. **Simple Strong Clock Condition:**
   Given event \( e_i \) of \( p_i \) and event \( e_j \) of \( p_j \), \( i \neq j \)
   \[ e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i] \]
Properties of Vector Clocks

3. Test for Concurrency: given event $e_i$ of $p_i$ and event $e_j$ of $p_j$
   
   $e_i \parallel e_j \equiv (\text{VC}(e_i)[i] > \text{VC}(e_j)[i]) \land (\text{VC}(e_j)[j] > \text{VC}(e_i)[j])$

4. Pairwise Inconsistent: given event $e_i$ of $p_i$ and $e_j$ of $p_j$, $i \neq j$
   
   if $e_i$, $e_j$ cannot belong to the frontier of the same consistent cut
   
   $(\text{VC}(e_i)[i] < \text{VC}(e_j)[i]) \lor (\text{VC}(e_j)[j] < \text{VC}(e_i)[j])$
Properties of Vector Clocks

5. **Consistent Cut:**
   - frontier contains no pairwise inconsistent events
     \[ VC(e_i^c)[i] \geq VC(e_j^c)[i], \; \forall 1 \leq i, j \leq n \]

6. **Counting # of events causally precede** \( e_i \):
   - \[ #(e_i) = (\Sigma_{j=1}^{n} VC(e_i)[j]) - 1 \]
   - # events = 4+1+3-1 = 7
Properties of Vector Clocks

7. **Weak Gap-Detection**: given event $e_i$ of $p_i$ and $e_j$ of $p_j$, if $\text{VC}(e_i)[k] < \text{VC}(e_j)[k]$ for some $k \neq j$, there exists event $e_k$ such that $\neg(e_k \rightarrow e_i) \land (e_k \rightarrow e_j)$
Causal Delivery and Vector Clocks

- assume processes increment local component of $VC$ only for events notified to monitor $p_0$
- $p_0$ maintains set $M$ for messages received but not yet delivered
- suppose we have:
  - message $m$ from $p_j$
  - $m'$ = last message delivered from process $p_k$, $k \neq j$
Causal Delivery and Vector Clocks

To deliver $m$, $p_0$ must verify:

1. no earlier message from $p_j$ is undelivered (i.e. $TS(m)[j] - 1$ messages have been delivered from $p_j$)

2. no undelivered message $m''$ from $p_k$ s.t. 
   $send_k(m') \rightarrow send_k(m'') \rightarrow send_j(m)$, $\forall k \neq j$
   (i.e. whether $TS(m')[k] \geq TS(m)[k]$ for all $k$)
Causal Delivery and Vector Clocks

- $p_0$ maintains array $D[1...n]$ where $D[i] = TS(m_i)[i]$, $m_i$ being last message delivered from $p_i$
- e.g. on right, delivery of $m$ is delayed until $m''$ is received and delivered
Delivery Rule 3

- **Causal Delivery:**
  - for all messages $m, m'$, sending processes $p_i, p_j$ and destination process $p_k$

\[
\text{send}_i(m) \rightarrow \text{send}_j(m') \Rightarrow \text{deliver}_k(m) \rightarrow \text{deliver}_k(m')
\]

- **DR3 (Causal Delivery):** Deliver message $m$ from process $p_j$ as soon as
  - $D[j] = TS(m)[j] - 1$, and
  - $D[k] \geq TS(m)[k], \forall k \neq j$

- $p_0$ set $D[j]$ to $TS(m)[j]$ after delivery of $m$
Causal Delivery and Hidden Channels

- should apply to closed systems
- incorrect conclusion with hidden channels (communication channel external to the system)

Figure 9. External Environment as a Hidden Channel
Active Monitoring
- Distributed Snapshots

- monitor $p_0$ requests states of other processes and combine into global state
- assume channels implement FIFO delivery
- **channel state** $\chi_{i,j}$ for channel $p_i$ to $p_j$: messages sent by $p_i$ not yet received by $p_j$
Distributed Snapshots

- notations:
  \( \text{IN}_i \) = set of processes having direct channels to \( p_i \)
  \( \text{OUT}_i \) = set of processes to which \( p_i \) has a channel

- for each execution of the snapshot protocol, process \( p_i \) record its local state \( \sigma_i \) and the states of its incoming channels (\( \chi_{j,i} \) for all \( p_j \in \text{IN}_i \))
Distributed Snapshots

- Snapshot Protocol (Chandy-Lamport)
  1. $p_0$ starts the protocol by sending itself a “take snapshot” message
  2. when receiving the “take snapshot” message for the first time from process $p_f$
     - $p_i$ records local state $\sigma_i$ and relays the “take snapshot” message along all outgoing channels
     - channel state $\chi_{f,i}$ is set to empty
     - $p_i$ starts recording messages on other incoming channels
Distributed Snapshots

Snapshot Protocol (Chandy-Lamport)

3. when receiving the “take snapshot” message beyond the first time from process $p_s$:
   - $p_i$ stops recording messages along channel from $p_s$
   - channel state $\chi_{s,i}$ are messages that have been recorded
Distributed Snapshots

- dash arrows indicate “take snapshot” messages
- constructed global state: $\Sigma^{23}$; $x_{1,2}$ empty; $x_{2,1} = \{m\}$

Figure 10. Application of the Chandy-Lamport Snapshot Protocol
Properties of Snapshots

- Let $\Sigma^s = \text{global state constructed}$
  $\Sigma^a = \text{global state when protocol initiated}$
  $\Sigma^f = \text{global state when protocol terminated}$
- $\Sigma^s$ is guaranteed to be consistent
- actual run that the system followed may not pass through $\Sigma^s$
- but $\exists$ a run $R$ such that $\Sigma^a \sim_R \Sigma^s \sim_R \Sigma^f$
Properties of Snapshots

- \( r = e_2^1 e_1^1 e_1^2 e_1^3 e_2^2 e_1^4 e_2^4 e_1^5 e_2^5 e_1^6 \)
  \( = \Sigma^{00} \Sigma^{01} \Sigma^{11} \Sigma^{21} \Sigma^{31} \Sigma^{32} \Sigma^{42} \Sigma^{43} \Sigma^{44} \Sigma^{54} \Sigma^{55} \Sigma^{65} \)

- \( \Sigma^a = \Sigma^{21} \)

- \( \Sigma^f = \Sigma^{55} \)

- \( r \) does not pass through \( \Sigma^s (= \Sigma^{23}) \)
Properties of Snapshots

- but $\Sigma^{21} \rightarrow \Sigma^{23} \rightarrow \Sigma^{55}$
Properties of Global Predicates

Now we have two methods for global predicate evaluation:

- monitor passively observing runs
- monitor actively constructing snapshots

Utility of either approach depends (in part) on properties of the predicate
Stable Predicates

- communication delays $\Rightarrow \Sigma^s$ can only reflect some past state of the system
- **stable** predicate: once become true, remain true
- e.g. deadlock, termination, loss of all tokens, unreachable storage
- if $\Phi$ is stable, then
  
  $$(\Phi \text{ is true in } \Sigma^s) \Rightarrow (\Phi \text{ is true in } \Sigma^f) \text{ and } (\Phi \text{ is false in } \Sigma^s) \Rightarrow (\Phi \text{ is false in } \Sigma^a)$$
Stable Predicates

- deadlock detection through snapshots (p.29, 30)

```plaintext
process p(i): 1 \leq i \leq n
var pending: queue of [message, integer] init empty;
working: boolean init false;
blocking: array [1..n] of boolean init false;

while true do
  while working or (size(pending) > 0) do
    receive m from p(i);
    case m.type of
      request:
        blocking[] := true;
        pending := pending + [m, i];
        response:
          [m, i] := NextState(m, i);
          working := (m.type = request);
          send m to p(i);
          if (m.type = response) then blocking[] := false;
          if s = 0 then
            % this is the first snapshot message
            send [type: snapshot, data: blocking] to p(0);
            send [type: snapshot] to p(1),...p(i-1),p(i+1),...p(n)
            s := (s + 1) mod n;
          esac
        end;
      od:
    while not working and (size(pending) > 0) do
      [m, i] := head(pending);
      pending := tail(pending);
      [m, i] := NextState(m, i);
      working := (m.type = request);
      send m to p(i);
      if (m.type = response) then blocking[] := false;
    od
  od
end p(i);
```

Figure 12. Deadlock Detection through Snapshots: Server Side

```plaintext
process p(0):
var wfg: array [1..n] of array [1..n] of boolean;
j, k: integer; m: message;

while true do
  wait until deadlock is suspected;
  send [type: snapshot] to p(1),...p(n);
  for k := 1 to n do
    receive m from p(i);
    wfg[i][j] := m.data;
    if (cycle in wfg) then system is deadlocked
  od
end p(0);
```

Figure 13. Deadlock Detection through Snapshots: Monitor Side
Stable Predicates

- deadlock detection using reactive protocol (p.31, 32)

```plaintext
process p(i); 1 ≤ i ≤ n

var pending: queue of [message, integer] init empty;
working: boolean init false;
m: message; j: integer;

while true do
    while working or (size(pending) > 0) do
        receive m from p(j);
        case m.type of
            request:
                send [type: requested, of: i, by: j] to p(0);
                pending := pending + [m, j];
                response:
                    [m, j] := NextState(m, j);
                    working := (m.type = request);
                    send m to p(0);
                if (m.type = response) then
                    send [type: responded, to: j, by: i] to p(0);
            esac
        while not working and (size(pending) > 0) do
            [m, j] := first(pending);
            [m, j] := NextState(m, j);
            working := (m.type = request);
            send m to p(0);
            if (m.type = response) then
                send [type: responded, to: j, by: i] to p(0)
            esac
        od
    od
end p(i);
```

```plaintext
process p(0):

var wfg: array [1..n, 1..n] of boolean init false;
m: message; j: integer;

while true do
    receive m from p(0);
    if (m.type = responded) then
        wfg[m,by, m.to] := false
    else
        wfg[m.of, m.by] := true;
    if (cycle in wfg) then
        system is deadlocked
    od
end p(0);
```

Figure 14. Deadlock Detection using Reactive Protocol: Server Side

Figure 15. Deadlock Detection using Reactive Protocol: Monitor Side
Nonstable Predicates

- e.g. debugging, checking if queue lengths exceed some thresholds
- Two problems:
  1. condition may not persist long enough for it to be true when the predicate is evaluated
  2. if a predicate $\Phi$ is found true, do not know whether $\Phi$ ever held during the actual run
Nonstable Predicates

- e.g. monitoring condition $(x = y)$
  - 7 states where $(x = y)$ holds
  - but no longer hold after state $\Sigma^{54}$

- e.g. $(y - x) = 2$
  - condition hold only in $\Sigma^{31}$ and $\Sigma^{41}$
  - monitor might detect $(y - x) = 2$ even if actual run never goes through $\Sigma^{31}$ or $\Sigma^{41}$

Figure 16. Global States Satisfying Predicates $(x = y)$ and $(y - x) = 2$
Nonstable Predicates

- very little value to detect nonstable predicate

**Figure 16. Global States Satisfying Predicates** ($x = y$) and ($y - x = 2$)
Nonstable Predicates

- With observations, can extend predicates:
- **Possibly**($\Phi$): There exist a consistent observation $O$ of the computation such that $\Phi$ holds in a global state of $O$
- **Definitely**($\Phi$): For every consistent observation $O$ of the computation, there exists a global state of $O$ in which $\Phi$ holds
- e.g. **Possibly**($(y - x) = 2$), **Definitely**($x = y$)
Nonstable Predicates

- use of extended predicate in debugging:
  if $\Phi = \text{some erroneous state}$, then
  \textbf{Possibly}(\Phi) indicates a bug, even if it is
  not observed during an actual run

- if predicate $\Phi$ is stable, then
  \textbf{Possibly}(\Phi) $\equiv \textbf{Definitely}(\Phi)$
Detecting Possibly and Definitely $\Phi$

- detection based on the lattice of consistent global states
- If any global state in the lattice satisfies $\Phi$, then $\textbf{Possibly}(\Phi)$ holds
- $\textbf{Definitely}(\Phi)$ requires all possible runs to pass through a global state that satisfies $\Phi$
Detecting Possibly and Definitely $\Phi$

- Possibly $((y - x) = 2)$
- Definitely $y = x$ (why?)

Figure 16. Global States Satisfying Predicates $(x = y)$ and $(y - x) = 2$
Detecting Possibly and Definitely $\Phi$

- set of global state $current$ with progressively increasing levels
- any member of $current$ satisfies $\Phi$ => Possibly($\Phi$) true

```plaintext
procedure Possibly($\Phi$);
    var current: set of global states;
    $\ell$: integer;
    begin
        % Synchronize processes and distribute $\Phi$
        send $\Phi$ to all processes;
        current := global state $\Sigma^{0,0}$;
        release processes;
        $\ell$ := 0;
        % Invariant: current contains all states of level $\ell$ that are reachable from $\Sigma^{0,0}$
        while (no state in current satisfies $\Phi$) do
            if current = final global state then return false
            $\ell$ := $\ell + 1$;
            current := states of level $\ell$
        od
        return true
    end
```

Figure 17. Algorithm for Detecting Possibly($\Phi$).
Detecting Possibly and Definitely $\Phi$

- iteratively construct set of global states of level $l$ without passing through a state that satisfies $\Phi$
- set empty $\Rightarrow$ Definitely($\Phi$) true
- set contains the final state $\Rightarrow \neg$Definitely($\Phi$) true

```plaintext
procedure Definitely(\Phi);
    var current, last: set of global states;
    $\ell$: integer;
    begin
        % Synchronize processes and distribute $\Phi$
        send $\Phi$ to all processes;
        last := global state $\Sigma^{0-0}$;
        release processes;
        remove all states in last that satisfy $\Phi$;
        $\ell$ := 1;
        % Invariant: last contains all states of level $\ell - 1$ that are reachable
        % from $\Sigma^{0-0}$ without passing through a state satisfying $\Phi$
        while (last $\neq \{ \}$) do
            current := states of level $\ell$ reachable from a state in last;
            remove all states in current that satisfy $\Phi$;
            if current = final global state then return false
            $\ell$ := $\ell + 1$;
            last := current
        od
        return true
    end;
```

Figure 18. Algorithm for Detecting Definitely($\Phi$).
Conclusions

- many distributed system problems require recognizing certain global conditions
- two approaches to constructing global states:
  - reactive-architecture based
  - snapshot based
- timing mechanism that captures causal precedence relation
- applying to distributed deadlock detection and debugging
- solutions can be adapted to deal with nonstable predicates, multiple observations and failures