Chapter 10: Time and Global States

- Introduction
- Clocks, events and process states
- Synchronizing physical clocks
- Logical time and logical clocks
- Global states
- Distributed debugging
- Summary

Time is an important issue in DS

- Need to measure accurately
  - E.g. auditing in e-commerce
- Algorithms depending on
  - E.g. consistency, make
- Correctness of distributed systems frequently hinges upon the satisfaction of global system invariants.
  - Examples of global invariants
    - Absence of deadlocks
    - Write access to a distributed database never granted to more than one process
    - Objects are only subject to garbage collection when no further reference to them exists

Clock in computer

- A device that count oscillations occurring in a crystal at a definite frequency
- Hardware time: $H(t)$
  - The counts of oscillation since an original point
- Software time: $C(t) = \alpha H(t) + \beta$
  - Timestamp of an event

Clock skew and clock drift

- Clock drift
  - Crystal oscillate at different rate
  - Clock drift can not be avoided
- Clock skew
  - The instantaneous difference between the readings of any two clocks

Astronomical Time & International Atomic Time

- Rotation of earth on its axis and about the sun
  - Days, Years, etc
  - Second is $1/86400$ astronomical time
- Standard second
  - Atomic oscillator Cs$^{133}$
  - Drift rate: one part in $10^{13}$
Chapter 10: Time and Global States

- Introduction
- Clocks, events and process states
- Synchronizing physical clocks
- Logical time and logical clocks
- Global states
- Distributed debugging
- Summary

Coordinated Universal Time (UTC)

- Shift between astronomical time and atomic time
  - The period of the Earth’s rotation about its axis is gradually getting longer
  - Tidal friction, atmospheric effects, etc
- Leap second
  - Atomic time which is inserted a leap second occasionally to keep in step with astronomical time
- Broadcast UTC to the World
  - E.g., by GPS or WWW

External & Internal synchronization

- $C_i$: $p_i$’s clock
- $I$: an interval of real time
- External synchronization
  - For a synchronization bound $D > 0$, and for a source $S$ of UTC time, $|S(t) - C_i(t)| < D$, for $i = 1, 2, \ldots N$ and for all real times $t$ in $I$
  - Clocks $C_i$ are accurate to within the bound $D$

External & Internal synchronization (2)

- Internal synchronization
  - For a synchronization bound $D > 0$, $|C_i(t) - C_j(t)| < D$ for $i, j = 1, 2, \ldots N$, and for all real times $t$ in $I$
  - Clocks $C_i$ agree within the bound $D$
- If the system is external synchronized with a bound $D$ then the same system is internal synchronized with a bound $2D$

General synchronization issues

- Correctness of a hardware clock $H$
  - A bounded drift rate $\rho$, e.g. $10^{-6}$ seconds/second, $t$ and $t'$ are real time
    $$(1 - \rho) \leq \frac{H(t') - H(t)}{(t' - t)} \leq (1 + \rho)$$
- Correctness of a software clock
  - Monotonicity: $t' > t \Rightarrow C(t') > C(t)$
    - Set clock back
    - Errors in the make process
    - Change the clock rate

Synchronization in a synchronous system

- Protocol
  - Sender: send $M(t)$
  - Receiver: set time to $t + T_{trans}$
- Bounds are know in synchronous system
  - $\min < T_{trans} < \max$ (constant)
- So, set $T_{trans} = (\min + \max) / 2$
  - Receiver’s clock = $t + (\min + \max) / 2$
Synchronization in a synchronous system (2)

- Clock skew between sender and receiver
  \[(\text{max} - \text{min}) / 2\]

\[\begin{array}{cccc}
  t & t + \text{min} & t + T_{\text{trans}} & t + \text{max} \\
\end{array}\]

Cristian’s method of synchronizing clocks

- Applying circumstance
  - C/S Round-trip time is short compared with the required accuracy

- Protocol
  - \(m_r, m_t(t), T_{\text{round}}\)
  - Estimated time: \(t + m_t + T_{\text{round}}/2\)

The Berkeley algorithms

- Internal synchronization
  1. The master polls the slaves’ clocks
  2. The master estimates the slaves’ clocks by round-trip time
     - Similar to Christian’s algorithm
  3. The master averages the slaves’ clock values
     - Cancel out the individual clock’s tendencies to run fast or slow
  4. The master sends back to the slaves the amount that the slaves’ clocks should adjust by
     - Positive or negative value
     - Avoid further uncertainty due to the message transmission time
  5. Slave adjust its clock

Design aims of Network Time Protocol

- External synchronization
  - Enable clients across the Internet to be synchronized accurately to UTC

- Reliability
  - Can survive lengthy losses of connectivity
  - Redundant server & redundant path between servers

Design aims of Network Time Protocol (2)

- Scalability
  - Enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers

- Security
  - Protect against interference with the time service
Network Time Protocol Architecture

- Arrows denote synchronization control, numbers denote strata.
- Reconfigure when servers become unreachable

Synchronization measures

- **Multicast mode**
  - Intend for use on a high speed LAN
  - Assuming a small delay
  - Low accuracy but efficient

- **Procedure-call mode**
  - Similar to Christian’s
  - Higher accuracy than multicast

- **Symmetric mode**
  - The highest accuracy

Symmetric mode synchronization

Assuming

\( t, t' \): actual transmission time of \( m, m' \);
\( o \): actual B’s clock skew relative to A.

We have

\[ T_{i-2} = T_{i-3} + t + o, \quad T_i = T_{i-1} + t' - o \]

Symmetric mode synchronization (2)

\[ T_{i-2} = T_{i-3} + t + o, \quad T_i = T_{i-1} + t' - o \]

Then

\[ d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1} \]

**Addition**:

\[ o = (T_{i-2} - T_{i-3} + T_i - T_{i-1}) / 2 \]

where \( o_i = (T_{i-2} - T_{i-3} + T_i - T_{i-1}) / 2 \)

we have \( o = o_i + (t' - t) / 2 \)

Symmetric mode sync. implementation

- NTP servers retain 8 most recent pairs \(<o_i,d_i>\)
- The value \( o_i \) of that corresponds to the minimum value \( d_i \) is chosen to estimate \( o \)
- A NTP server exchanges with several peers in addition to with parent
  - Peers with lower stratum numbers are favoured
  - Peers with the lowest synchronization dispersion are favoured

Symmetric mode synchronization

Accuracy analysis

Due \( t, t' \geq 0 \), then

\[ o_i - (t' + t) / 2 \leq o \leq o_i + (t' + t) / 2 \]

Then

\[ o_i - d_i / 2 \leq o \leq o_i + d_i / 2 \]

\( o_i \) is the estimated time

\( d_i \) is the measure of the accuracy
Chapter 10: Time and Global States

- Introduction
- Clocks, events and process states
- Synchronizing physical clocks
- Logical time and logical clocks
- Global states
- Distributed debugging
- Summary

Happen-before relation

- **HB1**: If \( \exists \) process \( p_i : e \rightarrow e' \), then \( e \rightarrow e' \)
- **HB2**: For any message \( m \),
  - \( \text{send}(m) \rightarrow \text{receive}(m) \)
- **HB3**: If \( e, e' \) and \( e'' \) are events such that
  - \( e \rightarrow e' \) and \( e' \rightarrow e'' \), then \( e \rightarrow e'' \)

Causal ordering

Events occurring at three processes

- **Example**
  - \( a \parallel e \)

Shortcomings

- Not suitable to processes collaboration that does not involve messages transmission
- Capture potential causal ordering

Happen-before relation

Lamport timestamps algorithm

- **LC1**
  - \( L_i \) is incremented before each event is issued at process \( p_i : L_i := L_i + 1 \)
- **LC2**:
  - (a) When a process \( p_i \) sends a message \( m \), it piggybacks on \( m \) the value \( t = L_i \)
  - (b) On receiving \( (m, t) \), a process \( P_j \) computes \( L_j := \max(L_j, t) \) and then applies LC1 before timestamping the event \( \text{receive}(m) \)

Lamport timestamps algorithm

- \( e \rightarrow e' \Rightarrow L(e) < L(e') \)
- \( L(e) < L(e') \Rightarrow e \rightarrow e' \) or \( e \parallel e' \)
Totally ordered logical clocks

Assumption

\( T_i \): local timestamp of \( e \) that is an event occurring at \( p_i \)
\( T_j \): local timestamp of \( e' \) that is an event occurring at \( p_j \)

Define the timestamps of \( e, e' \) are \((T_i, i)\), \((T_j, j)\)

Define:
\[(T_i, i) < (T_j, j) \text{ if } T_i < T_j \text{, or } T_i = T_j \text{ and } i < j\]

Vector Clocks - algorithm

- Each process \( p_i \) keeps a vector clock \( V_i \)
- VC1: Initially, \( V[i][j] = 0 \), for \( i, j = 1,2,\ldots, N \)
- VC2: Just before \( p_i \) timestamps an event, it sets \( V[i][j] := V[i][j] + 1 \)
- VC3: \( p_i \) includes the value \( t = V_i \) in every message it sends
- VC4: When \( p_i \) receives a timestamp \( t \) in a message, it sets \( V[j][j] := \max(V[j][j], t[j]) \), for \( j=1,2,\ldots,N \)

Vector Clocks - example

- Compare vector timestamps
  \(- V = V' \iff V[i][j] = V'[i][j] \text{ for } i = 1,2,\ldots, N\)
  \(- V \leq V' \iff V[i][j] \leq V'[i][j] \text{ for } i = 1,2,\ldots, N\)
  \(- V < V' \iff V[i][j] < V'[i][j] \text{ for } i = 1,2,\ldots, N\)
  \(- \text{Otherwise } V < V' \)
  \(- V(e) < V(e') \iff e 
  \rightarrow e', V(e) \neq V(e') \iff e \mid e'\)

Chapter 10: Time and Global States

- Introduction
- Clocks, events and process states
- Synchronizing physical clocks
- Logical time and logical clocks
- Global states
- Distributed debugging
- Summary

Global System Invariants and States

- Correctness of distributed systems frequently hinges upon the satisfaction of global system invariants
- Examples of global invariants
  - Absence of deadlocks
  - Write access to a distributed database never granted more than one process
  - Objects are only subject to garbage collection when no further reference to them exists
- Thus, the ability to construct a global state and evaluate a predicate over such a state is a core problem in distributed computing.
Global System Invariants and States

• Distributed garbage collection
  – Based on reference counting
  – Should include the state of communication channels
• Distributed deadlock detection
  – Look for “waits-for” relationship
• Distributed termination detection
  – Look for state in which all processes are passive
• Distributed debugging
  – Need collect values of distributed variables at the same time
• Distributed monitoring:
  – Notify an administrator in case of failures

Global System Invariants and States

• Global state predicates
  – A function that maps from the set of global states of processes in the system \( \mathcal{S} \) to \{True, False\}

• Definition (Global Predicate Evaluation, GPE). The problem of detecting whether the global state of a distributed system satisfies some predicate.

Global System Invariants and States

A global state obtained through remote observations could be

• obsolete: represent an old state of the system.
• Solution: build the global state more frequently
• incomplete: not “capture” every moment of the system
• Solution: build all possible global states
• inconsistent: informally, a global state is inconsistent if it could never have been constructed by an idealized observer external to the system.

Global states and consistent cuts

• Formal definitions:
  – The system is composed of a set of processes
    \( \mathcal{S} = \{ p_i | i = 1, 2, \ldots, N \} \)
  – Event(e):
    – occurrence of a single action
    – locally numbered using the canonical numeration
    – local events that change the local state
    – each process executes:
      • send(m), receive(m) events that match based on message identifier \( m \)
Global states and consistent cuts

- The local history of a process $p_i$ is a (possibly infinite) sequence of events:
  - $\text{history}(p_i) = h_i = \langle e_{i0}, e_{i1}, e_{i2}, \ldots e_{in} \rangle$
- The partial history up to event $e^k_i$ is denoted $h^k_i$, and is represented by prefix of a process’s history of the first $k$ events of $h_i$.
  - $h^k_i = \langle e_{i0}, e_{i1}, \ldots e_{ik} \rangle$
- $\rightarrow_i$ occur before in $p_i$, e.g. $e \rightarrow_i e'$
  - Total order of events in $p_i$
- Global history of processes set $\mathcal{H} = h_1 \cup h_2 \cup \ldots \cup h_N$

Global states and consistent cuts

A cut $C$ is consistent:

For all events $e \in C$, $f \rightarrow e \Rightarrow f \in C$

\[ \langle e_{i0}^0, e_{i2}^0 \rangle, \langle e_{i1}^1, e_{ij}^1 \rangle \]

Global states and consistent cuts

A consistent global state:

- correspond to a consistent cut
- The state of $s_i$ in cut $C$ is that of $p_i$ immediately after the last event processed by $p_i$ in $C$ (i.e., $e^{j\pi}$)

Execution of a distributed system

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$

Global states and consistent cuts

A linearization (consistent) run

An ordering of the events in a global history $H$ that is consistent with this happened-before relation $\rightarrow$ on $H$.

Pass only consistent global state

may have more than one linearization run

$S'$ is reachable from a state $S$

there is a linearization that pass through $S$ and then $S'$
Monitoring Distributed Computation

- GPE can be stated as evaluating a predicate $\Phi$ that is a function of the global state
- Assumptions (relaxed later):
  - There is a single process $p_0$ called monitor which is responsible for evaluating $\Phi$
  - We assume that $p_0$ is distinct from $p_1 \ldots p_n$
  - Events executed on behalf of monitoring do not alter canonical enumeration of its events.
- Two possible approaches:
  - Proactive monitoring (take a “snapshot”)
  - Passive monitoring (receive notifications)

Monitoring Distributed Computation

- Multi-tier system based on RPC, deadlock detection
- Processes use RMIs:
  - Client sends a request for method execution; blocks.
  - Server receives request.
  - Server executes method; may invoke other methods in other servers, acting as a client.
  - Server sends reply to client
  - Clients receives reply; unblocks.
- Such a system can deadlock, as RMIs are blocking. It is important to be able to detect when a deadlock occurs.

Monitoring Distributed Computation

- Detection based on a Wait-For Graph (WFG)
  - nodes $\equiv$ processes
  - Given two processes $p_i, p_j$, an edge from $p_i$ to $p_j$ is added if $p_j$ has received a request from $p_i$ but has not replied yet.
  - WFG can be constructed solely on local states
  - If WFG contains a cycle, there is a deadlock
  - $\Phi$ = “WFG contains a cycle”

Monitoring Distributed Computation

- Consider cuts $C'$ and $C$ in the previous figure.
  - They could have been generated by a “request for snapshot” message sent to all nodes and received at different times.
  - The graph associated with $C'$ contains a deadlock:
    - $(1 \rightarrow 3), (3 \rightarrow 2), (2 \rightarrow 1)$
  - It is easy to see that this is impossible.

Monitoring Distributed Computation

- In the space-time diagram, a cut $C$ is consistent if all the arrows start on the left of the cut and finish on the right of the cut.
- Predicates can be evaluated in consistent cuts, because they correspond to potential global states that could have taken place during an execution.
A Passive Approach to GPE

• How it works
  – The sequence of events corresponding to the order in which notification messages arrive at the monitor is called an observation.
  – At each (meaningful) event, a node sends a message to the monitor describing its local state.
  – The monitor collects messages and builds an observation of the global state.

A Passive Approach to GPE

• Observations vs Runs
  • Definition (Consistent Observation) An observation is consistent if it corresponds to a consistent run.
    – An observation can correspond to:
      • A consistent run
      • A run which is not consistent
      • No run at all

A Passive Approach to GPE

• Problem: Observations may not correspond to a run, or a consistent run, due to the asynchrony messages may arrive in any order.
• Solution: we will adopt the following approach:
  – Messages will be re-ordered in order to guarantee that observation correspond to consistent run, and thus consistent global states.
  – To be ordered, each message m carries a timestamp TS(m) containing “ordering” information.
  – The act of providing the process with a message in the desired order is called delivery; the event deliver(m) is thus distinct from receive(m).
  – The rule describing which messages can be delivered among those received is called delivery rule.

A Passive Approach to GPE

• Problem: Observations may not correspond to a run because a channel between a pair of process may re-order messages in any possible way.
• Definition: FIFO Delivery - First-in, First-Out
  Two messages sent by pi to pj must be delivered in the same order in which they have been sent:
  – \( \lor m, m' : send_i(m) \rightarrow send_i(m') \)
  – \( \lor deliver_j(m) \rightarrow deliver_j(m') \)

A Passive Approach to GPE

• Problem: If we use FIFO delivery, all the observations taken by p0 will be runs; but there is no guarantee that they will be consistent runs.
• Solution: a very simple mechanism, based on strong assumptions, and then we refine it by relaxing those assumptions.
• Initial assumptions
  – All processes have access to a real-time clock RC; let denote with RC(e) the real-time at which e is executed.
  – All messages are delivered within a time.
  – The timestamp of a message m sent through an event e = send(m) is TS(m) = RC(e).

A Passive Approach to GPE

• Implementation
  – Each process maintains a local sequence number incremented at each message sent.
  – The timestamp of a message corresponds to the local sequence number of the sender at the time of sending.
  – DR0 (FIFO Delivery Rule):
    • If the last message delivered by pi from pj has timestamp s, pi may deliver “any” message m received from pj with TS(m) = s + 1.

A Passive Approach to GPE

• FIFO Delivery is not sufficient
• Problem If we use FIFO delivery, all the observations taken by pi will be runs; but there is no guarantee that they will be consistent runs.
• Solution: a very simple mechanism, based on strong assumptions, and then we refine it by relaxing those assumptions.
• Initial assumptions
  – All processes have access to a real-time clock RC; let denote with RC(e) the real-time at which e is executed;
  – All messages are delivered within a time.
  – The timestamp of a message m sent through an event e = send(m) is TS(m) = RC(e).
A Passive Approach to GPE

- **DR1: Real-time delivery rule**
  - At time $t$, delivery all received messages $m$ such that $TS(m) \leq t - \delta$ in increasing timestamp order.
- **Observation $O$ constructed by $p_0$ using $DR1$ is guaranteed to be consistent.**
- $e \rightarrow e' \Rightarrow RC(e) < RC(e')$
- Note that $RC(e) < RC(e') \neq e \rightarrow e'$, but this rule is sufficient to obtain consistent observations, as two events $e \rightarrow e'$ are never delivered in the incorrect order.

Proactive monitoring---Snapshot Protocol

- **Problem:**
  - **Goal:** To build the global state after an explicit request of the monitor.
  - **How:** By taking “pictures” (snapshot) of the local state when instructed
  - **Challenge:** To build a consistent global state.
- **Solution: Chandy and Lamport Snapshot Algorithm**
  - This particular protocol enables to reason about “channel states”

Proactive monitoring---Snapshot Protocol

- **Channel State**
  - For each channel from $p_i$ to $p_j$, its state $x_{i,j}$ are those messages that $p_i$ has sent but $p_j$ has not received yet.
  - Note: channel state can be obtained by storing appropriate information in the local state, but it is complicated.
- **Recorded information**
  - Each process will record its local state $S_i$ and the content of its incoming channels $x_{j,i}$.
  - When one process record a state $S_i$, make all other processes record states that have been caused by $S_i$.

The “snapshot” algorithm - assumptions

- Neither channels nor processes fail
- Unidirectional channels, FIFO message delivery
- Complete connection among all processes
- Any process may initiate a global snapshot at any time
- Process may continue execution and send and receive normal message while snapshot takes place

The “snapshot” algorithm - method

- **principle of operation**
  - broadcast marker
  - upon receipt of marker record own state, and record any incoming message from another process until that process has recorded its state (these messages then belong to the channel between the processes)
  - processes may record their state at different points in time, but the differential is always accounted for by the state of the channel in between
- **marker sending rule**
  - a) record own state
  - b) broadcast marker
  - a) and b) must proceed any other local actions or message send/receive events
- **marker receiving rule**
  - if $p_i$ has not yet recorded own state (first marker is being received)
    - record own state
    - start recording all messages received on all incoming channels
  - if $p_i$ has already recorded own state
    - record state of channel on which marker was received
    - stop recording that channel
Marker receiving rule for process $pi$
On $pi$’s receipt of a marker message over channel $c$:
if ($pi$ has not yet recorded its state) it
records its process state now;
records the state of $c$ as the empty set;
turns on recording of messages arriving over other incoming
channels;
else $pi$ records the state of $c$ as the set of messages it has received
over $c$ since it saved its state.
end if

Marker sending rule for process $pi$
After $pi$ has recorded its state, for each outgoing channel $c$:
$pi$ sends one marker message over $c$ (before it sends any other message over $c$).

The “snapshot” algorithm - example

The Chandy-Lamport Algorithm: Theorem:
• Terminates: Proof sketch:
  – Assumption: a process that has received a marker message
    records its state within a finite time and sends marker
    messages via each outgoing channel within a finite time
  – If there is a communication path from
    $pi$ to $pj$, then $pj$ will
    record its state a finite time after
    $pi$ recordes its status.
  – Since the communication graph is strongly connected, all
    process in the graph will have recorded their state and the
    state of incoming channels a finite time after some process
    initiated snapshot taking.

• Construct reachability relationship
  – Reachability Theorem: Let $Sys = e_0, e_1, \ldots$, the linearization
    of a system execution. Let
    • $S_{init}$ the initial global state of the system immediately
      before Chandy-Lamport snapshot-taking was initiated by
      the first process,
    • $S_{snap}$ the recorded snapshot state, and
    • $S_{final}$ the global system state after the algorithm
      terminated.
    – Then there is a permutation $Sys' = e'_0, e'_1, \ldots$ of $Sys$ such that
      • $S_{init}$, $S_{snap}$ and $S_{final}$ occur in $Sys'$ and
      • $S_{snap}$ is reachable from $S_{init}$ and
      • $S_{final}$ is reachable from $S_{snap}$.
The Chandy-Lamport Algorithm: Theorem

**Proof:**
- Split events in $\text{Sys}$ in
  - **pre-snap** events: occurred before the respective process in which this event occurred recorded its state
  - **post-snap** events: all other events

---

Theorem: The Chandy-Lamport Algorithm

- **how to order events to obtain $\text{Sys}'$**
  - Assume $e_j$ is post-snap event at one process, and $e_{j+1}$ pre-snap in a different process
  - $e_j \rightarrow e_{j+1}$, since otherwise they would be send and receive of the same message, and then they would be either both post-snap or both pre-snap
  - $e_j$ and $e_{j+1}$ may be swapped in $\text{Sys}'$
  - Swap adjacent events, if necessary and possible, until $\text{Sys}'$ is so that all pre-snap events precede all post-snap events
  - Let $e'_j, e'_r, e'_s$ denote the prefix of pre-snap events in $\text{Sys}'$, hence the set of events prior to state recording for each process, hence all events leading from $S_{\text{init}}$ up to the state being recorded as $S_{\text{snap}}$
  - Since we have disturbed neither $S_{\text{init}}$ nor $S_{\text{final}}$ we have established the reachability relationship amongst these states

---

Chapter 10: Time and Global States

- **Introduction**
- **Clocks, events and process states**
- **Synchronizing physical clocks**
- **Logical time and logical clocks**
- **Global states**
- **Distributed debugging**
- **Summary**

---

Distributed debugging

- **example**
  - Safety condition of a distributed system: $|x_j-x_k| \leq \delta$
- **approach**
  - A monitor
    - Collect states of other distributed processes
    - $x_j=100, x_{k-1}=105$ (1.0)
    - $x_j=95, x_{k-1}=90$ (2.0)
    - $x_j=100, x_{k-1}=105$ (3.0)
    - $x_j=95, x_{k-1}=90$ (4.3)
    - $x_j=95, x_{k-1}=90$ (2.3)

---

Distributed debugging

**Properties**
- Typically interested in invariant safety properties
  - The system does not reach a deadlock state
  - The difference between variables $x$ and $y$ is always non-zero
  - The valves $v_1$ and $v_2$ may never be open at the same time
- Chandy-Lamport snapshot algorithm can at best prove violation of these properties
- Interested in a monitoring algorithm that records system traces in order to decide whether safety properties were, or may have been, violated in a given system run
  - Let $H$ be the execution history of a system and $\Phi$ a state predicate
    - $\text{pos } \Phi$: there is a consistent global state $S$ through which a linearization of $H$ passes such that $\Phi(S)$ is true.
    - $\text{def } \Phi$: for all linearizations $L$ of $H$ there is a consistent global state $S'$ through which $L$ passes such that $\Phi(S')$ is true.
  - For Chandy-Lamport: $\Phi(S_{\text{snap}}) \Rightarrow \text{pos } \Phi$
**Distributed debugging**

- **In example:**
  - **process behaviour:**
    - small local changes are reported to monitor, but not to other process
    - large local changes cause message to monitor, and also message to other process
  - **inconsistent cut C1 violates $\Phi$**
  - **consistent cut C2 satisfies $\Phi$**

**Observing consistent global states**

- In order for the monitor to infer consistency of the constructed state information, the processes maintain vector clocks and piggyback their vector clock value with every message to $M$
- Let $S$ a global state that $M$ has constructed from the state messages received, and $V(s_j)$ the vector time stamp received from process $i$.
- $S$ is **consistent** iff $V(s_j)[i] \geq V(s_k)[i]$ i.e., the number of $i$’s events known at $k$ when it sent $s_k$ is no more than the number of events at $i$ when it sent $s_i$
- In the example, this condition is clearly violated for $V(s_i) = (1,0)$ and $V(s_k) = (2,1)$. Hence $C1$ is inconsistent and does not constitute a violation of $\Phi$.

**Distributed debug introduction**

- Monitor construct the reachability lattice by the consistent global state identification algorithm
  - Find consistent global states
  - Establish the reachability relation between states
  - $S_{ij}$ is in level $(i+j)$
  - Show all the linearizations corresponding to a history

**Observing consistent global states**

- **Apply a given global state predicate $\phi$ on the states**
  - **Possibly $\phi$:** there is a consistent global state $s$ through which a linearization of $H$ passes such that $\phi(s)$ is true
  - **Definitely $\phi$:** for all linearizations $L$ of $H$, there is a consistent global state set $S$ through which $L$ passes such that $\phi(S)$ is true

**Observing consistent global states**

- We use the passive approach in which processes send notifications of events relevant to the monitor $p_0$;
  - Events are tagged with vector clocks;
  - The monitor collects all the events and builds the lattice of global states.
- **How?**
  - To detect Possibly(): if there exists one global state in which $\phi$ is true, then return true, otherwise false.
  - To detect Definitely(): mark nodes where $\phi$ is true with a value 1, the other nodes with value 0. If the cost of the shortest path between the initial state and the final state is larger than 0, return true, otherwise false.
Evaluating possibly $\phi$ and definitely $\phi$

- Evaluating possibly $\phi$
  - There is a downwards way in which there is a state evaluated to True by $\phi$
- Evaluating definitely $\phi$
  - There is no downwards way in which there is not a state evaluated to True by $\phi$
- Example
  - If $\phi$ evaluates to True in the state at level 5, then definitely $\phi$
  - If $\phi$ evaluates to false in the state at level 5, then possibly $\phi$

Evaluating definitely $\phi$

It is possible to evaluate a predicate over an entire computation using an observation obtained by a passive monitor.

- Possibly $\phi$: There exists 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.

Examples: Possibly $(y - x) = 2$, Definitely $(x = y)$

Debugging
If Possibly $\phi$ is true, and $\neg \phi$ identifies some erroneous state of the computation, then there is a bug, even if it is not observed during an actual run.

Chapter 10: Time and Global States

- Introduction
- Clocks, events and process states
- Synchronizing physical clocks
- Logical time and logical clocks
- Global states
- Distributed debugging
- Summary
Summary

• **Synchronize physical clocks**
  – Christian’s algorithm
  – Berkeley algorithm
  – Network Time Protocol

• **Logical time**
  – Happen-before relation
  – Lamport timestamp algorithm
  – Vector clock

Summary ...continued

• **Global states**
  – Consistent cut, consistent state
  – Snapshot algorithm
  – Construct reachability relationship by snapshot

• **Global debugging**
  – The monitor collects distributed events with vector timestamp
  – Construct reachability relationship
  – Examine possibly $\phi$ and definitely $\phi$