Distributed snapshots

- Global states of the system that are consistent with causality
- Such global states can be used for detecting stable properties, e.g.
  - Distributed deadlock detection
  - Distributed garbage collection
  - Distributed termination detection
- Main difficulty
  - System state changes during observation

Distributed snapshots: assumptions

- The system is connected, that is there is a path between every pair of processors
- $C_{i,j}$ channel from p_i to p_j
- Channels are reliable and FIFO
  - Messages sent are eventually received in order

Distributed snapshots: definitions

- Processor state
  - State of a processor (at an instant) is the assignment of a value to each variable of that processor
- Channel state
  - State of $C_{i,j}$ is the ordered list of messages sent by p_i but not yet received at p_j

Distributed snapshots: definitions

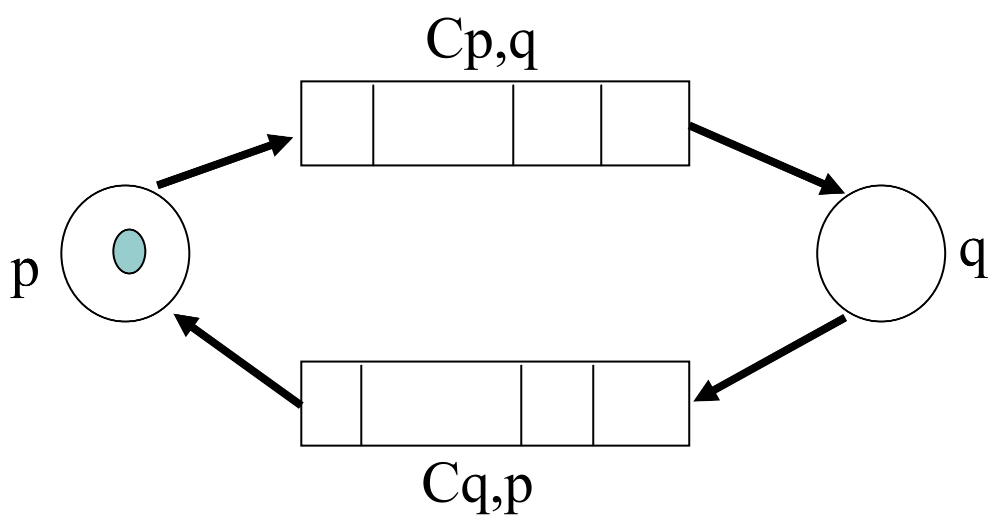
- Global state of the system
  - A pair (S,L) where
    - $S = (s_1,.., s_M)$ denotes the processor states and
    - L = channel states
- Note
  - A global state cannot be taken instantaneously
  - It must be computed in a distributed manner

Distributed snapshots

- The problem
  - Devise a distributed algorithm that computes a *consistent global state*.

- What do we mean by consistent global state?

Distributed snapshot: meaning of consistent global state



Two possible states for each processor: s_0, s_1

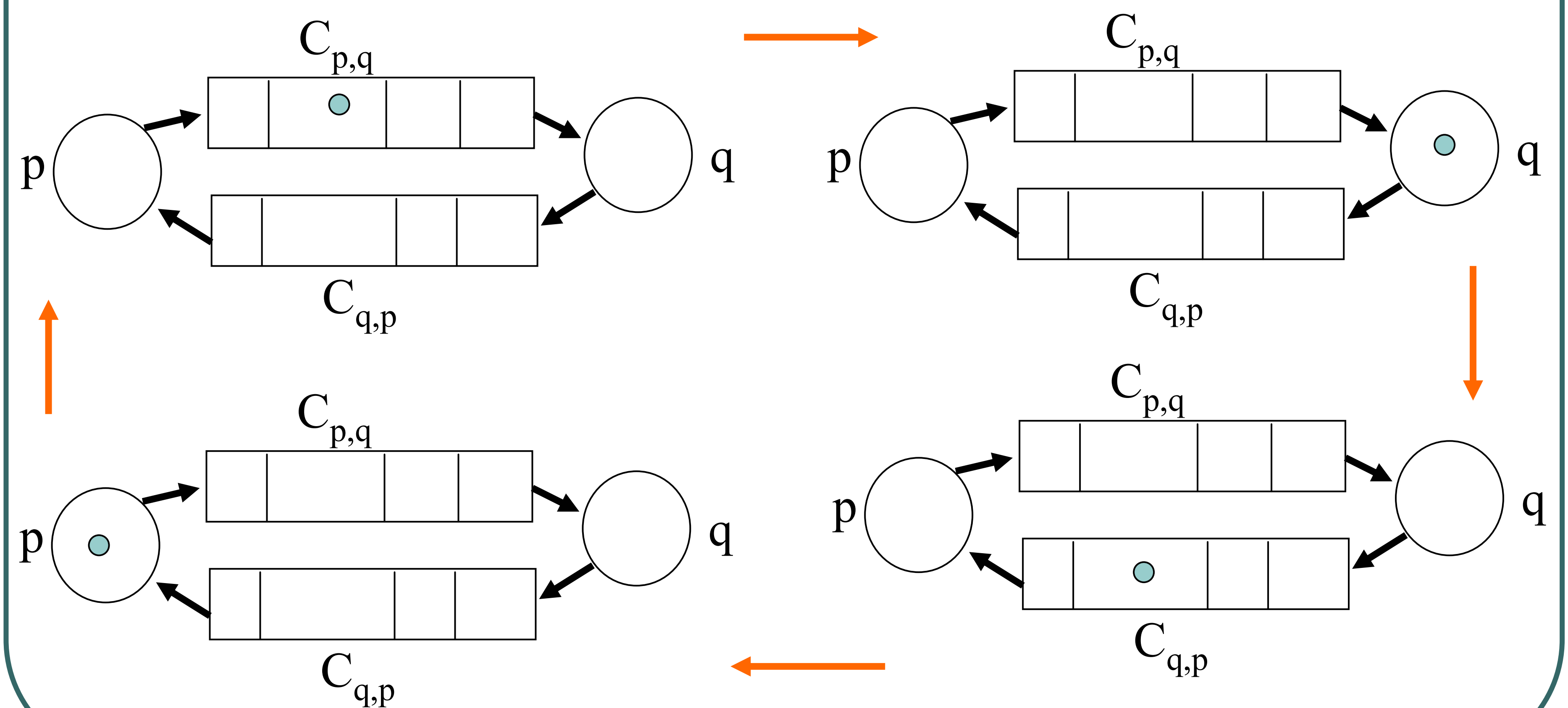
In s_0: the processor hasn't the token

In s_1: the processor has the token

The system contains exactly one token which moves back and forth between p and q. Initially, p has the token.

Events: sending/receiving the token

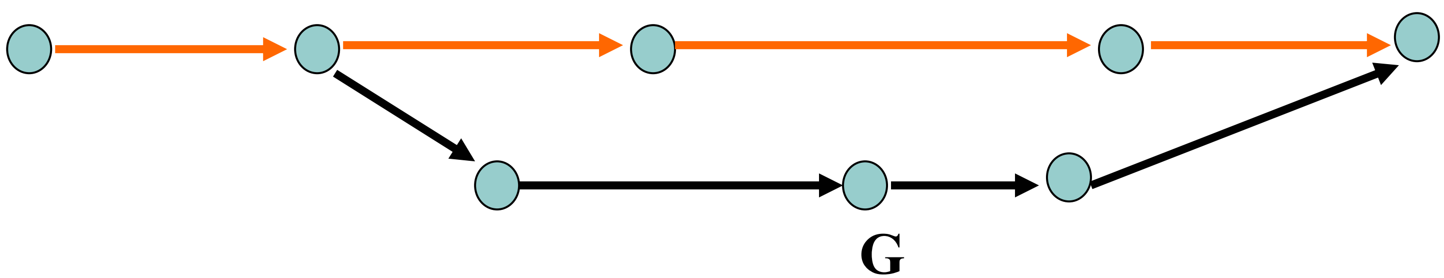
Distributed snapshots: meaning of consistent global state



Distributed snapshots: meaning of consistent global state

A global state G is consistent if it is one that could have occurred

Consider a system with two possible runs (non-determinism!)




Actual transitions

The output of the snapshot algorithm can be G !

Distributed snapshots

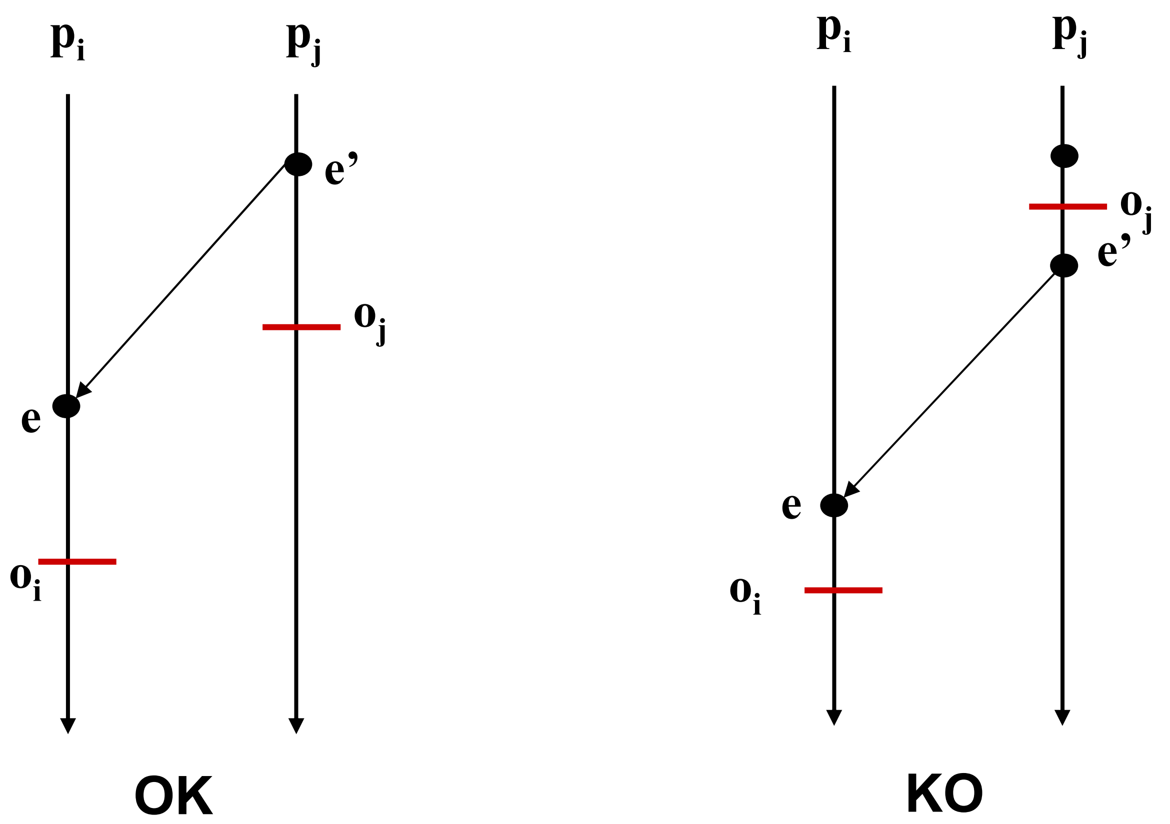
- $S=\{s_1,..,s_M\}$
- o_i: event of observing s_i at p_i
- $O(S)=\{o_1,..,o_M\}$

- Definition
  - S is a consistent cut iff $O(S) = \{o_1,..,o_M\}$ is consistent with causality

Distributed snapshots

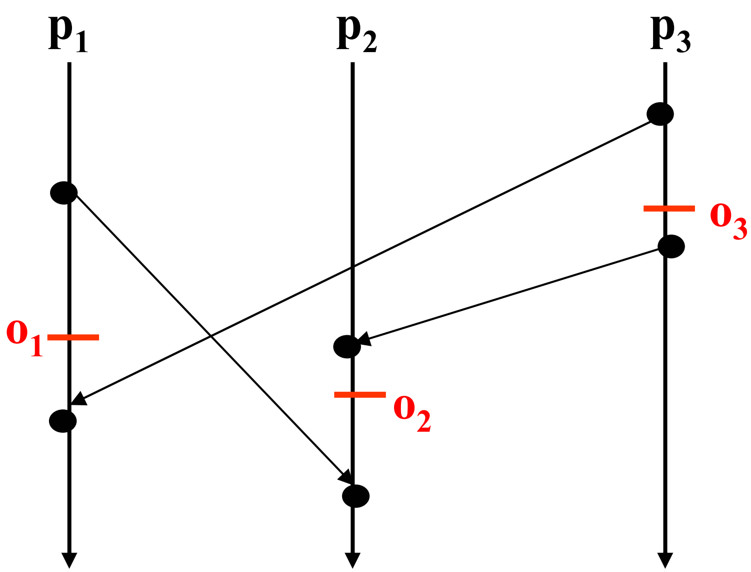
- Definition:
  - $\{o_1,..,o_M\}$ is consistent with causality iff
    $(\forall\ e : e \text{ in } E_i \ \wedge\ e <_H o_i :$
    $(\forall\ e' : e' \text{ in } E_j \wedge\ e' <_H e : e' <_H o_j)\ )$
- Such an observation cannot indicate the receipt of a message without the sending of that message

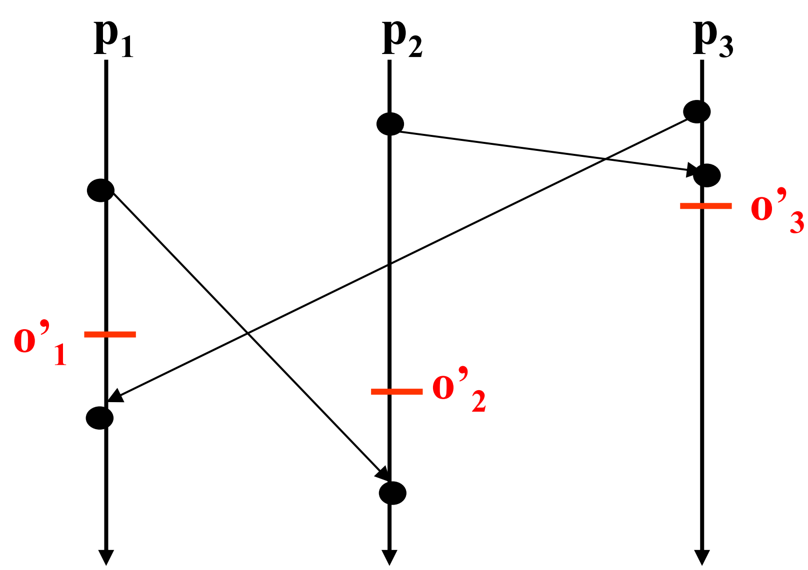
Distributed snapshots: intuition



Distributed snapshots

Exercise





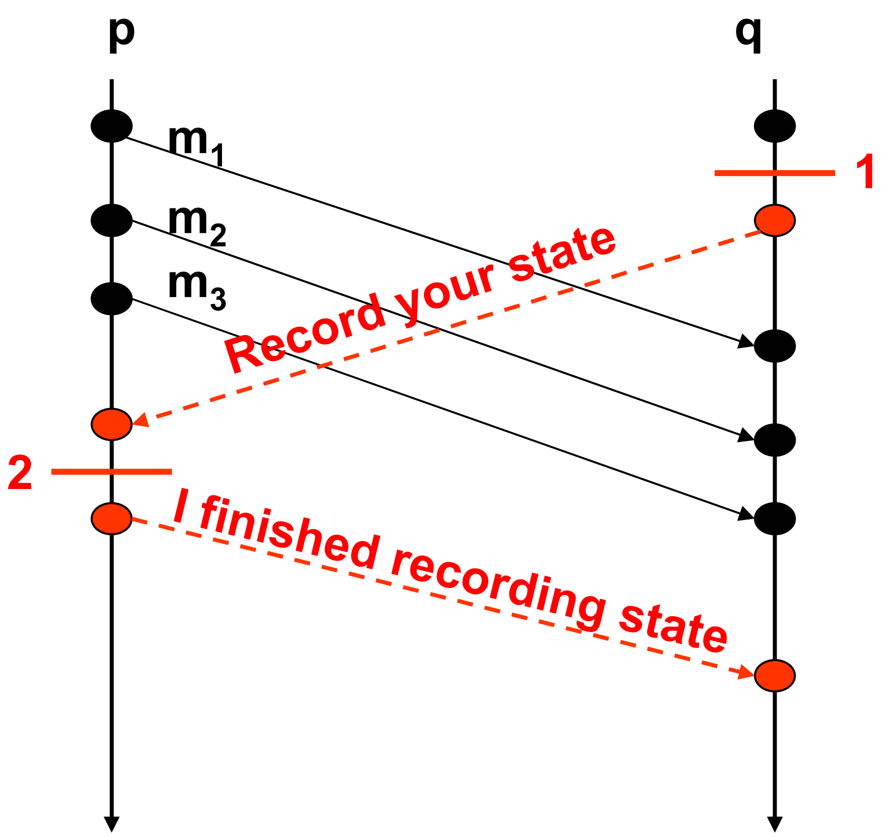
Is $O=\{o_1,o_2,o_3\}$ consistent with causality?

Is $O'=\{o'_1,o'_2,o'_3\}$ consistent with causality?

Distributed snapshots: messages sent but not yet received

- Let m be a message, we write
  - s(m): sending event of message m
  - r(m): receipt event of message m
- Let S be an observed global state
  - $O(S)=\{o_1,o_2,\ldots,o_M\}$
  - If $s(m) <_{p_i} o_i$ and $o_j <_{p_j} r(m)$ then message m is sent but not received relatively to observation O(S)

Distributed snapshots: messages sent but not yet received



Processor **q** observes its state at 1
Then, **q** sends **special message** to **p** asking **p** to record its state

Processor **p** records its state at 2
then **p** sends a **special message** to **q** saying that it recorded its state

When **q** receives the **special message** from **p**, processor **q** determines that messages **m_1**, **m_2**, **m_3** were sent but were not yet received

In the resulting global state, the state of **C_{pq}** must be **[m1, m2, m3]**

Distributed snapshots

- Use of non-consistent global state can lead to wrong conclusions

- E.g. a false deadlock can be claimed!

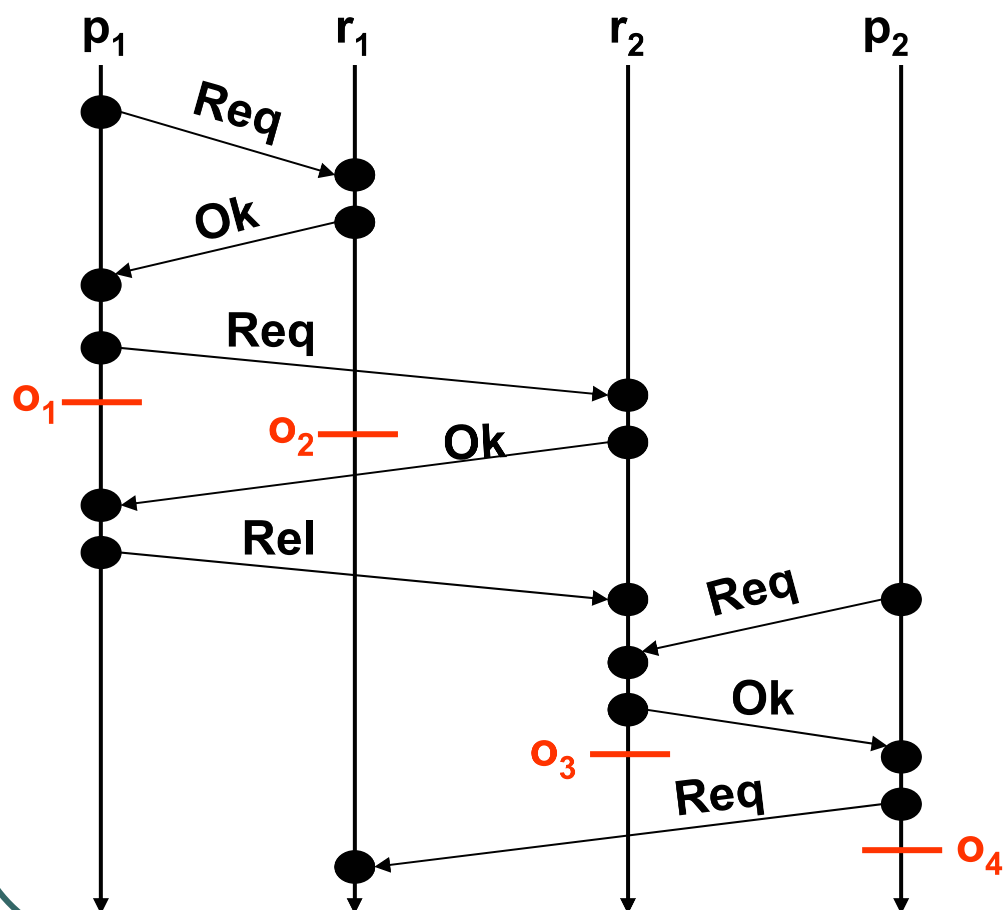
False deadlock illustrated

- Assume
  - $\mathbf{r}_1$ and $\mathbf{r}_2$ two resources
  - Each resource is either *available* or *un-available*
  - $\mathbf{p}_1$ and $\mathbf{p}_2$ two processors that access resources
- To access a resource **r**, a processor **p** sends Request (Req) to **r** and **p** starts to wait until it receives Ok from **r**

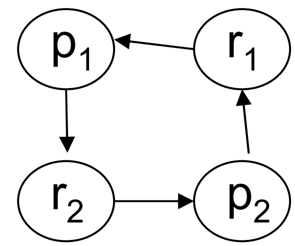
False deadlock illustrated

- When a resource **r** in status *available* receives Req from **p**,
  - **r** sends Ok to **p**
  - **r** becomes *un-available*
  - **r** starts to wait until it receives Release (Rel) from **p**
- A deadlock situation occurs when there is a cycle in the wait-for-graph amongst processors and resources

False deadlock illustrated



Inference from
o_1: p_1 waits for r_2
o_2: r_1 waits for p_1
o_3: r_2 waits for p_2
o_4: p_2 waits for r_1



From $O=\{o_1,o_2,o_3,o_4\}$ the deadlock detector concludes there is a deadlock!

What's wrong with O?

Distributed snapshots: the algorithm

- For simplicity
  - We assume only one snapshot computation
- Principle
  - Uses two types of messages
    - SnapshotRequest
      - Serve to start a snapshot computation
    - SnapshotToken
      - Serve to ``propagate'' a snapshot computation

    - Each processor records its state and the state of its incoming channels

Distributed snapshots: the algorithm

- A processor that receives SnapshotRequest
  - Records its state
  - Sends a SnapshotToken to each of its outgoing channels
  - Starts to record state of incoming channels
- Recording of the state of an incoming channel is finished when a SnapshotToken is received along it

Distributed snapshots: the algorithm

- A processor **p** that receives SnapshotToken on channel $\mathbf{C_{qp}}$
  - If this is the first time that SnapshotToken is received then
    - **p** records its state
    - **p** records the state of $\mathbf{C_{qp}}$ as empty
    - **p** sends one SnapshotToken to each outgoing channel
    - **p** starts to record the state of all its other incoming channels
  - Otherwise, **p** records the state of $\mathbf{C_{qp}}$ as the list of messages received from **q** since the time **p** recorded its state

Distributed snapshots: the algorithm

- Local termination of a snapshot
  - A processor learns that its participation to a snapshot computation is terminated when it has recorded its state and received a SnapshotToken on each of its incoming channels
- When does an initiating processor learn that the global snapshot computation is terminated?
  - We'll see later

Distributed snapshots: the algorithm (pseudo-code)

- Constants
  - Incoming: set of processors from which I receives
  - Outgoing: set of processors that receive from me
- Variables
  - MyVersion: Integer
  - StateForSnapshotOf: $\{1,..,M\} \rightarrow$ States
  - CurrentSnapshotVersionOf: $\{1,..,M\} \rightarrow$ Integer
  - TokensReceivedForSnapshotOf: $\{1,..,M\} \rightarrow$ Integer
  - ChannelState: $\{1,..,M\} \rightarrow (\{1,..,M\} \rightarrow$ MessageLists$)$

Distributed snapshots: the algorithm (pseudo-code)

- Variable initialization
  - MyVersion := 0
  - StateForSnapshotOf(i) := ?, $1 \leq i \leq M$
  - CurrentSnapshotVersionOf(i) :=0, $1 \leq i \leq M$
  - TokensReceivedForSnapshotOf(i) :=?, $1 \leq i \leq M$
  - (ChannelState(i))(j) := ?, $1 \leq i,j \leq M$

  - ?: we don't care!

Distributed snapshots: the algorithm (pseudo-code)

```
Wait for SnapshotRequest or SnapshotToken(r,v)
on SnapshotRequest do
    MyVersion := MyVersion +1
    StateForSnapshotOf(self) := relevant local state
    CurrentSnapshotVersionOf(self) := MyVersion
     for every q in Outgoing do
          send(q,SnapshotToken(self, MyVersion))
     od
     TokenReceivedForSnapshotOf(self) := 0
od
...
```

Distributed snapshots: the algorithm (pseudo-code)

```
Wait for SnapshotRequest or SnapshotToken(r,v)
    …
  on SnapshotToken(r,v) from q do
    if CurrentSnapshotVersionOf(r) < v then
      StateForSnapshotOf(r) := relevant local state
      CurrentSnapshotVersionOf(r) := v
      (ChannelState(r))(q) := nil
      for every k in Outgoing do
        send(k,SnapshotToken(r,v))
      od
      TokenReceivedForSnapshotOf(r) := 1
       /* prepare for incoming channel state computation*/
    else
     …
  od
```

Distributed snapshots: the algorithm (pseudo-code)

```
Wait for SnapshotRequest or SnapshotToken(r,v)
    …
 on SnapshotToken(r,v) from q do
   if CurrentSnapshotVersionOf(r) < v then
     …
   else
     if CurrentSnapshotVersionOf(r) = v then
       TokensReceivedForSnapshotOf(r) := TokensReceivedForSnapshotOf(r) + 1
        (ChannelState(r))(q) := list of msg received from q since I recorded my state
        if TokensReceivedForSnapshotOf(r) = | Incoming | then
           my local participation is finished for snapshot computation (r,v)
 od
```

Distributed snapshots the algorithm's properties

- Shows an important technique for designing distributed algorithms: Diffusing computation
  - The computation starts at one processor, then the computation diffuses through the whole set of processors
- Shows how to distinguish between objects that have same name, but actually different
  - The use of version numbers
- Shows how to flush communication channels
  - The use of SnapshotToken messages
- Can record a global state that never occurred

Distributed snapshots: adding termination detection

- The algorithm for the snapshot computation can be easily adapted to construct a spanning tree of the system
- Spanning tree of the system
  - Each node maintains a variable MyParent [1..M]
  - A processor that starts a snapshot computation is the root of the associated spanning tree
  - The processor from which I receive SnapshotToken(r,v) for the first time is my parent in spanning tree associated to the snapshot computation identified by (r,v)

Distributed snapshots: adding termination detection

- Using the spanning tree associated to a snapshot computation, we can let the processor that started that snapshot computation detect when the computation is terminated

Distributed snapshots: adding termination detection

- Idea
  - When I receive a SnapshotToken(r,v) for the first time, I send SnapshotToken(r,v) to all outgoing channels except to the one leading to MyParent(r)
  - When I have received SnapshotToken(r,v) on all my incoming channels
    - If r≠ self then I send SnapshotToken(r,v) to MyParent(r)
    - Otherwise, I learn that the global computation is finished

Distributed snapshots: adding termination detection

```
Wait for SnapshotRequest or SnapshotToken(r,v)
on SnapshotRequest do
    MyVersion := MyVersion +1
    StateForSnapshotOf(self) := relevant local state
    CurrentSnapshotVersionOf(self) := MyVersion
     for every q in Outgoing do
          send(q,SnapshotToken(self, MyVesion))
     od
     TokenReceivedForSnapshotOf(self) := 0
      MyParent(self) := self
od
...
```

Distributed snapshots: adding termination detection

```
Wait for SnapshotRequest or SnapshotToken(r,v)
  ...
 on SnapshotToken(r,v) from q do
   if CurrentSnapshotVersionOf(r) < v then
     MyParent(r) := q
     StateForSnapshotOf(r) := relevant local state
     CurrentSnapshotVersionOf(r) := v
     (ChannelState(r))(q) := nil
     for every k in Outgoing and k ≠ C_{self,q} do
       send(k,SnapshotToken(r,v))
     od
     TokenReceivedForSnapshotOf(self) := 1
      /* prepare for incoming channel state computation*/
   else
     ...
 od
```

Distributed snapshots: adding termination detection

```
Wait for SnapshotRequest or SnapshotToken(r,v)
  on SnapshotToken(r,v) from q do
     if CurrentSnapshotVersionOf(r) < v then
        …
     else
        if CurrentSnapshotVersionOf(r) = v then
           TokensReceivedForSnapshotOf(r) := TokensReceivedForSnapshotOf(r) + 1
            (ChannelState(r))(q) := list of msg received from q since I recorded my state
            if TokensReceivedForSnapshotOf(r) = | Incoming | then
               if r = self then
                  global termination
               else
                  send(MyParent(r),SnapshotToken(r,v))
  od
```

Distributed snapshots

- Discussion
  - Concurrent observations (see book)
- Exercise
  - Explain why the proposed algorithm computes consistent global states

Modeling a distributed computation

- How to prove that a given distributed algorithm is correct?

- A formal model of a distributed computation serves this purpose
- Idea
  - Model a distributed computation as a sequence of global state transitions where each global state transition is caused by the execution of one event

Modeling a distributed computation

- Initial state of the system
  - $\mathbf{G^0 = (S^0, L^0)}$ where
    - $\mathbf{S^0}$ is the vector of processor initial states
    - $\mathbf{L^0}$ is the channel initial states, each channel is empty

- Given a global state **G**, a processor **p**
  - We write $\mathbf{s_p|_G}$ for state of **p** in global state **G**
  - We write $\mathbf{State(c)|_G}$ for state of channel **c** in **G**

Modeling a distributed computation

- Events
  - An event **e** is a 5-tuple **(p, s, s', m, c)** where
    - **p** is a processor
    - **s**, **s'** are possible states of p
    - **m** is a message or Null
    - **c** is a channel or Null
  - Interpretation of an event **e = (p, s, s', m, c)**
    - Takes **p** from **s** to **s'**
    - Possibly sends or receives **m** on **c**

Modeling a distributed computation

- Let
  - **G=(S,L)** be a global state
  - **e =(p, s, s', m, c)** an event

- Event **e** *can occur* in **G** if
  - There is a global state **G'** such that executing **e** in **G** changes the system state from **G** to **G'**
    - Notation: $\mathbf{G} \rightarrow_{\mathbf{e}} \mathbf{G'}$

Modeling a distributed computation

- If $\mathbf{G} \rightarrow_{\mathbf{e}} \mathbf{G'}$ then $\mathbf{G'}$ is defined by
  - $\mathbf{s_p|_{G'} = s'}$
  - For every $\mathbf{q \neq p}$, $\mathbf{s_q|_{G'} = s_q}$
  - If **c** is an outgoing channel from **p**, then
    - $\mathbf{State(c)|_{G'} = [m_1, m_2, m_3, \ldots, m_k, m]}$
  - If **c** is an incoming channel to **p** and $\mathbf{State(c)|_G = [m, m_1, m_2, \ldots, m_k]}$ then
    - $\mathbf{State(c)|_{G'} = [m_1, m_2, m_3, \ldots, m_k,]}$
  - For every $\mathbf{c' \neq c}$, $\mathbf{State(c')|_{G'} = State(c')|_G}$
- Note
  - If **c = Null**, then for every **c'**, $\mathbf{State(c')|_{G'} = State(c')|_G}$

Modeling a distributed computation

- Let
  - **G** be a global state
  - $\mathbf{seq = <e_0, e_1, \ldots, e_j>}$ be a sequence of events
- **seq** *can occur* in **G** if there is a sequence of global states $\mathbf{<G=G_0, G_1, \ldots, G_{j+1}=G'>}$ such that $\mathbf{G_i} \rightarrow_{\mathbf{ei}} \mathbf{G_{i+1}}$ for i=0, …, j
  - Notation: $\mathbf{G} \rightarrow_{\mathbf{seq}} \mathbf{G'}$

Modeling a distributed computation

- Let **G**, **G'** be two global states
- We say that **G'** is reachable from **G** if there is a sequence of events **seq** such that $\mathbf{G} \rightarrow_{\mathbf{seq}} \mathbf{G'}$
- We define a computation of the system as a sequence of events **seq** such that
  - $\mathbf{G^0} \rightarrow_{\mathbf{seq}} \mathbf{G'}$ for some **G'**

Modeling a distributed computation

- Characteristics of this model
  - Each global state transition is caused by the execution of one event that can occur in the current global state
- But several events can occur at a given global state
  - In this case, one of them is randomly selected
  - i.e. non-deterministic computations

Modeling a distributed computation

- How to prove that a given distributed algorithm is correct in this model
  - We must show that **every** global state reachable from the initial global state satisfies the correctness properties
- Stable properties
  - A global state predicate $\mathbf{\Phi}$ such that if $\mathbf{\Phi}$(G) is true then $\mathbf{\Phi}$(G') is true for every G' reachable from G

Relationships between models

- So far, three models for a distributed computation that can be grouped into two classes
- Observational models
  - Model 1
    - Set of events related by Happens-before relation (H-DAG)
  - Model 2
    - Consistent cuts
  - Framework for analyzing what happened in an execution
  - Concurrency of events is explicitly captured

Relationships between models

- Predictive model
  - Model 3
    - Global state transitions (interleaving)
  - Given an initial global state $\mathbf{G}^0$ and a distributed algorithm **A**, one can say what might occur when **A** is executed starting from $\mathbf{G}^0$
  - Captures consistent cuts and causality
  - Concurrency is not explicitly captured, instead non-determinism is used

Relationships between models

- Given
  - **A** a distributed algorithm
  - **H-DAG(A)**, an execution of **A**
  - **E** all events in **H-DAG(A)**
  - ***Z(E)*** all consistent cuts of **H-DAG(A)**
- There is an execution DAG, ***G(E)*** that corresponds to **H-DAG(A)**
  - **G(E)** shows all possible executions, in the Model 3, that can be derived from **H-DAG(A)**

Relationships between models

- Construction of *G(E)*
  - Since events in each processor are totally ordered, they can be numbered sequentially
  - Let **Number(e)** be the number assigned to event **e**

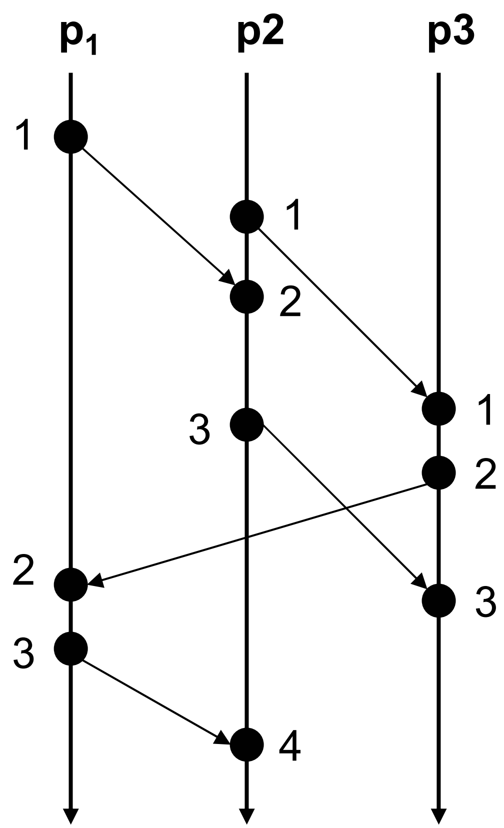
Relationships between models

- Construction of *G(E)*
  - Let $z_1 = (z_{11}, z_{12}, \ldots, z_{1M})$ and $z_2 = (z_{21}, z_{22}, \ldots, z_{2M})$ be two consistent cuts
  - $z_1 <_H z_2$ if
    - $Number(z_{1j}) \leq Number(z_{2j})$ for $j=1,2, \ldots, M$
  - z_2 *immediately follows* z_1 if there is a j such that
    - $Number(z_{2k}) = Number(z_{1k})$ for $k=1,2, \ldots, M$ and $k \neq j$
    - $Number(z_{2j}) = Number(z_{1j}) + 1$

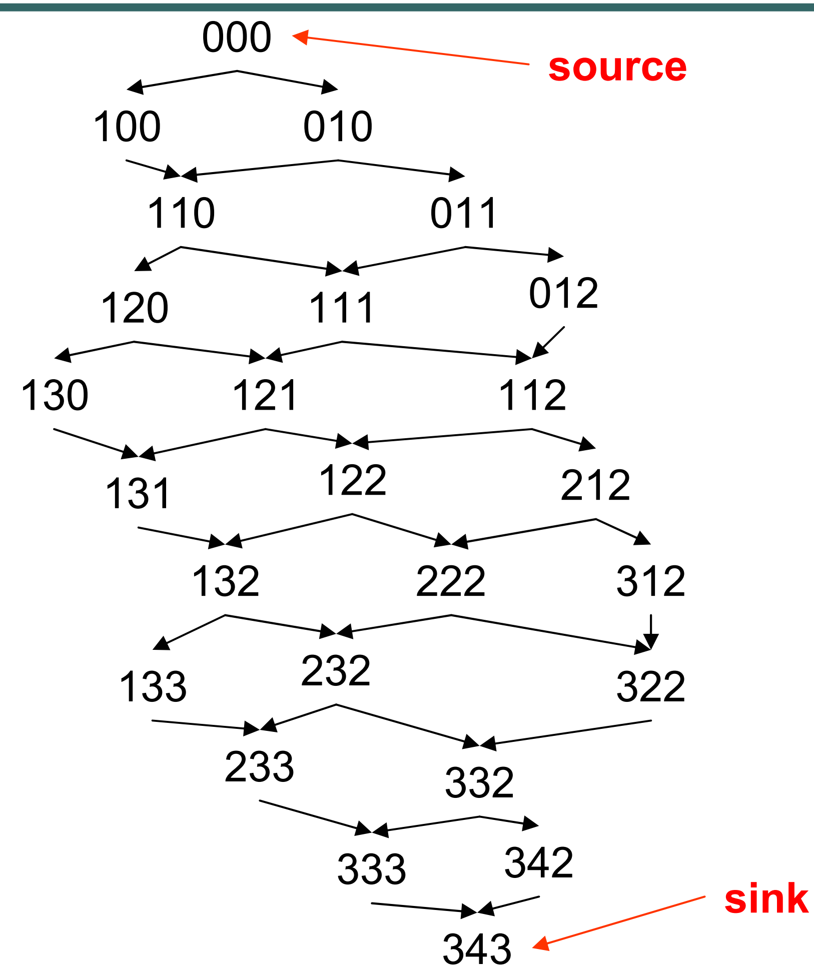
Relationships between models

- Construction of *G(E)*
  - **G(E)** is a directed graph where
    - The nodes are elements of **Z(E)**
    - There is an arc from z_1 to z_2 if z_2 immediately follows z_1
  - Each node $z=(z_1, \ldots, z_M)$ of ***G(E)*** is labeled by a vector $(n_1, n_2, \ldots, n_M)$ where n_i is the number of z_i
  - Each arc in of ***G(E)*** represents the execution of one event (a global state transition)

Execution DAG illustrated



H-DAG



Execution-DAG

Execution DAG predicates

- Predicates on execution DAG
- Useful for
  - Expressing desirable properties
  - Debugging
  - Proving correctness

Execution DAG predicates

- Let
  - G(E) be an execution DAG
  - Φ be a global state predicate
  - z be a node of G(E)
- We define Concur(z) as the set of all nodes z’ of G(E) such that there is no path from z to z’ or from z’ to z
  - E.g. 112 and 131 of the example execution DAG

Execution DAG predicates

- POSSIBLY predicate
  - POSSIBLY(Φ,G(E)) is true if
    - There is a node z of G(E) such that $\Phi(z)$ is true
- ALLPATH predicate
  - ALLPATH(Φ,G(E)) is true if
    - For every path P from the source to the sink of G(E), there is z in P such that $\Phi(z)$ is true

Execution DAG predicates

- DEFINITELY predicate
  - DEFINITELY(Φ,G(E)) is true if
    - There is a node z of G(E) such that Φ(z') is true for every z' in Concur(z)
- ALWAYS predicate
  - ALWAYS(Φ,G(E)) is true if
    - For every node z of G(E) , Φ(z) is true

Summarizing future executions

- Ways to describe future states of an execution given the current state
- Definition
  - Let G_0 be a global state. A **run r** on G_0 is defined as a finite or an infinite sequence of events <e_0, e_1, e_2, …> that *can occur* in G_0
- Notation:
  - r[i]: sequence formed by the first i+1 events of r, $0 \leq i \leq |r|-1$
  - $R(G_0)$ denotes all possible runs on G_0

Summarizing future executions

- *Eventually* predicate
  - Let Φ be a global state predicate
  - Let Ψ be a predicate on $\mathbf{R}(G_0)$
  - *Eventually*(Φ, G_0, Ψ) is true iff
    - For every $\mathbf{r}$ in $\mathbf{R}(G_0)$ such that $\Psi(r)$ is true,
      - There exists i in $\{0, 1, 2, \ldots, |r|-1\}$
      - There exists a global state G such that $G_0 \rightarrow_{r[i]} G$ and $\Phi(G)$ is true

Summarizing future executions

- *Always* predicate
  - Let Φ be a global state predicate
  - Let Ψ be a predicate on $\mathbf{R}(G_0)$
  - *Always*(Φ, G_0, Ψ) is true iff
    - For every $\mathbf{r}$ in $\mathbf{R}(G_0)$ such that $\Psi(r)$ is true,
      - For every i in $\{0, 1, 2, \ldots, |r|-1\}$
      - There exists a global state G such that $G_0 \rightarrow_{r[i]} G$ and $\Phi(G)$ is true

Failures in a distributed system

- A distributed system like any other computing system is built on
  - Hardware and software components that can fail from time to time
- A faulty component can behave in an unpredictable manner

Failures in a distributed system

- But, in many cases we must design distributed systems in a way that enables them to achieve their well-defined goals despite the presence of failure of some of its components
  - Hospital
  - Banking
  - Transportation, …
- The system must be fault-tolerant

Failures in a distributed system

- Fault-tolerance requires
  - Failure semantics
    - Description of the way a faulty component behaves
  - Mechanisms/techniques for handling well-defined failures
- Any interaction between two components of a distributed system can be seen as involving
  - A server: provides service to clients
  - A client: requests service from the server

Failures in a distributed system: failure semantics (F. Cristian 91)

- Omission failure
  - Occurs when a server omits to respond to a request
- Response failure
  - Occurs when a server responds incorrectly to a request
    - Wrong state transition at the server
    - Wrong value is returned to the client
- Timing failure
  - Occurs when a server responds either too earlier or too late

Failures in a distributed system: failure semantics

- Crash failure
  - Occurs when a server repeatedly omits to respond to requests until it is **restarted**
  - Cannot be accurately detected
  - Depending upon the state of the server when it restarts (if any restart), several subclasses of crash failure are given

Failures in a distributed system: failure semantics

- Amnesia-crash
  - Crash failure and
  - In addition, when the server restarts, it restarts from its initial state, the state by the time of failure is lost
- Pause-crash
  - Crash failure and
  - In addition, when the server restarts, it restarts from the state before the crash

Failures in a distributed system: failure semantics

- Halting-crash
  - Crash failure and
  - The server never restart
- Fail-stop
  - A fail-stop server is one that either functions correctly or stops. And when stopped this fact is accurately detected by its clients
  - Very nice but not realistic

Failures in a distributed system: failure semantics

- Byzantine failure
  - A Byzantine server can exhibits any of
    - Crash, Timing, Response and Omission failure
  - A Byzantine server can send contradictory messages to different recipients
  - The most general failure semantics, captures malicious behaviors
  - Often assumed for ultra-reliable systems