CS 138: Ordering and Global State

This material is partially covered in Chapter 14 of Coulouris, Dollimore, Kindberg, and Blair.

Administrivia

- HW2 is out today, due on the 15th (1 week)
- Review session will be on Monday, March 21st, 5:30pm
- Midterm will be on Tuesday, March 22nd, with material up to Raft (next two classes)

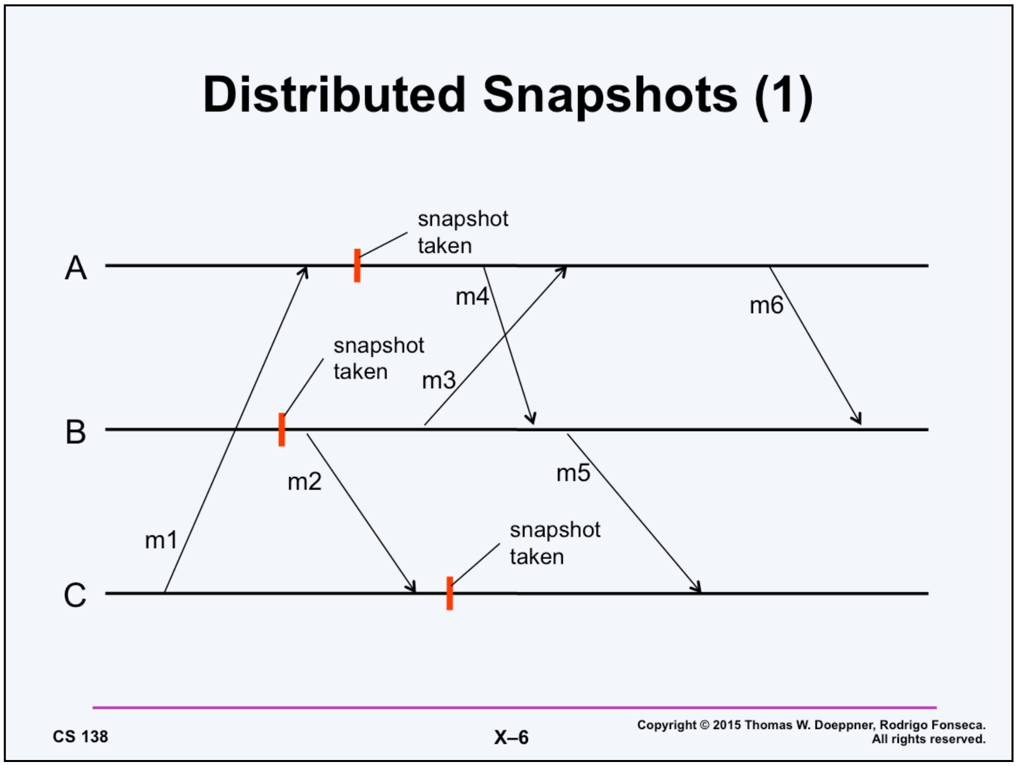
Global State

Failure Happens

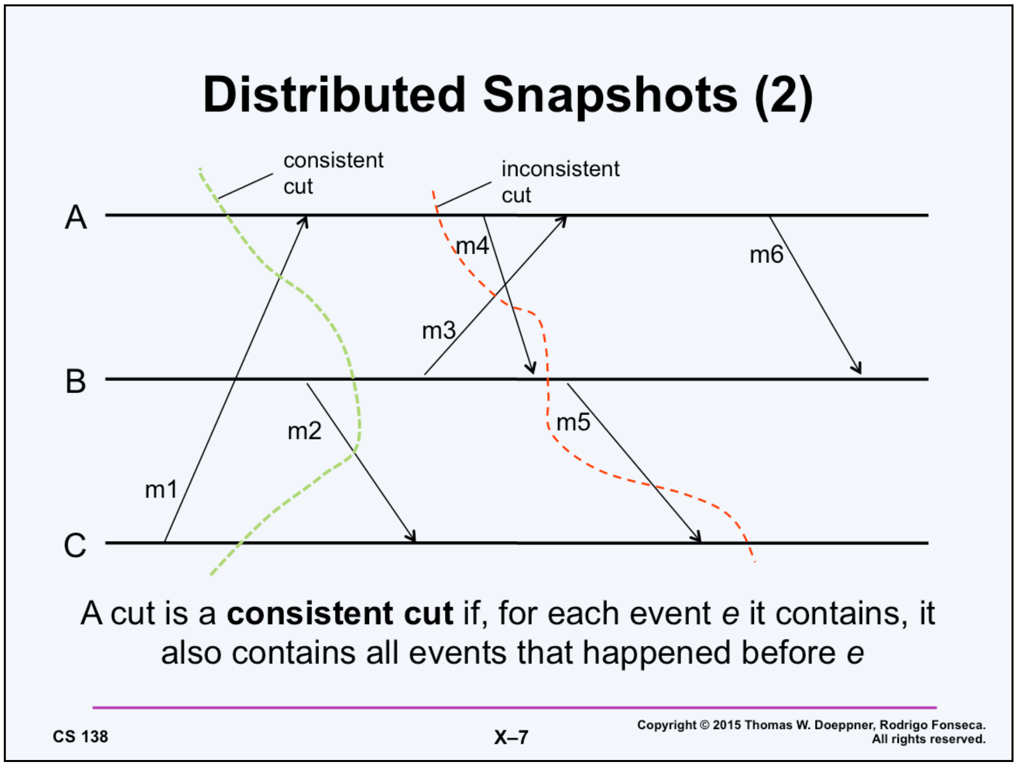
- What to do about it?
  - you of course have everything backed up
  - so, restore the backups

Global State

- Your system consists of 100 nodes
  - each produces a snapshot of itself periodically
  - does some collection of these snapshots constitute a meaningful notion of "global state"?



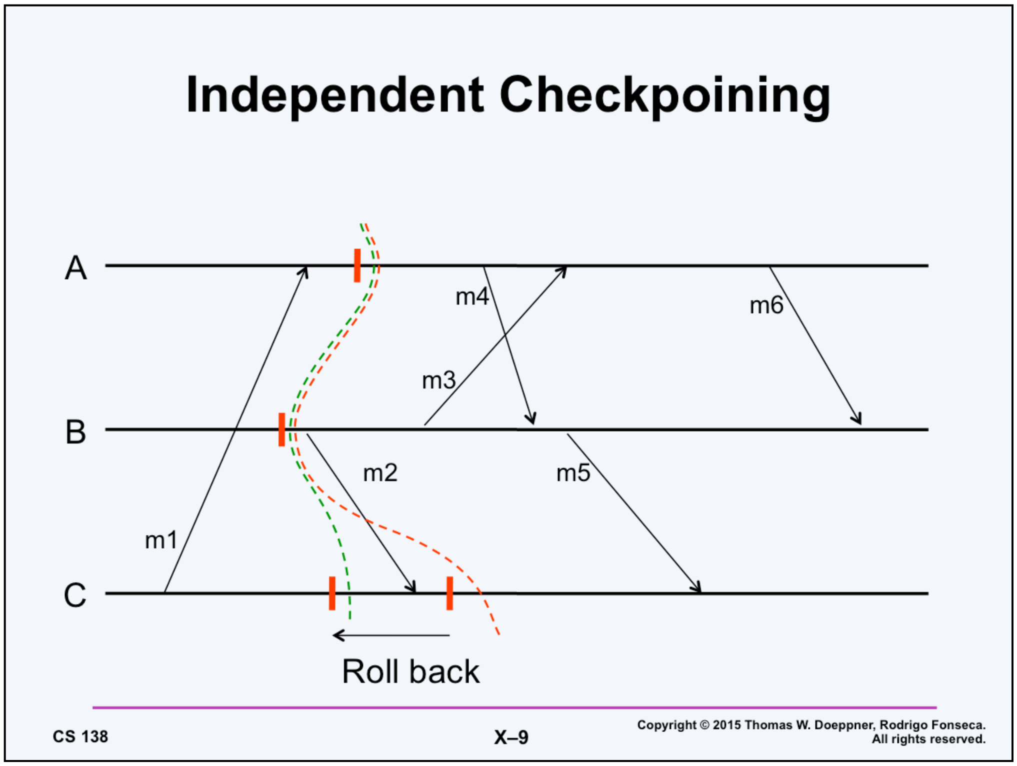
Suppose snapshots of each machine’s state were taken at the moments shown in the slide. If all machines crashed and their states were restored with the contents of their respective snapshots, would the system as a whole be in a state that it might have been in before the crash?



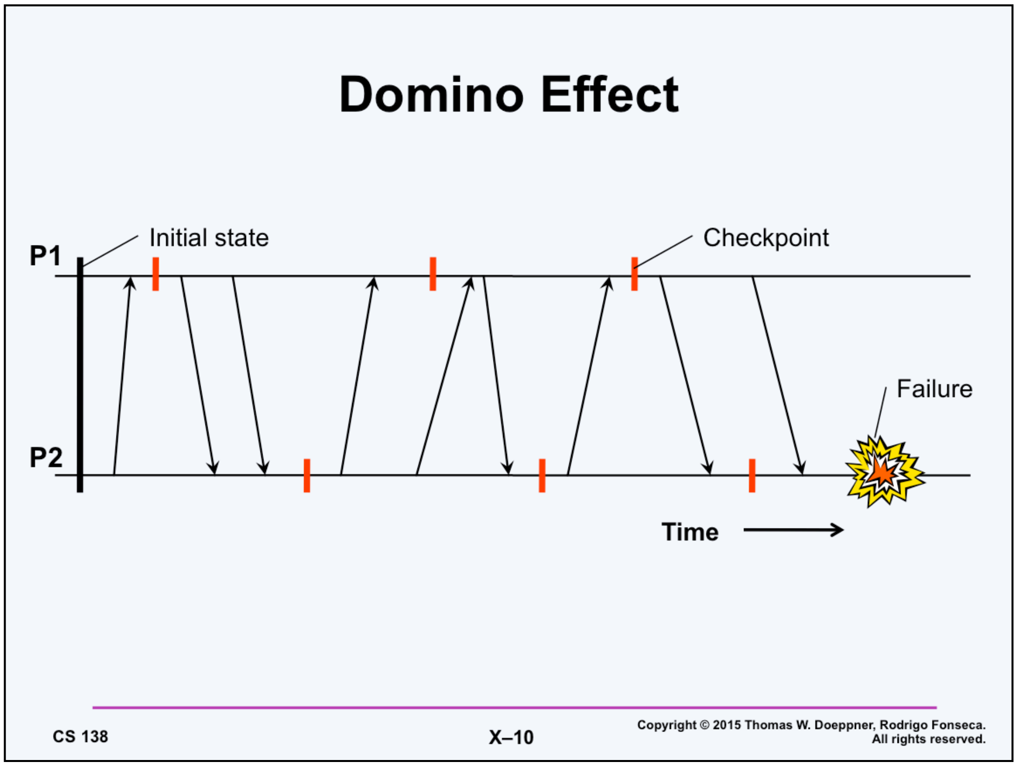
For a distributed snapshot to represent a possible state of the distributed system, we must make certain **that if in one process's snapshot we have the receipt of a message, then some other process's snapshot must contain the sending of the message**. A "consistent cut" is a distributed snapshot that has this property. (Note that we'll also look at a stronger notion in which the snapshots must all be concurrent: none may have a causal relationship with any of the others.)

Checkpointing

- Produce a distributed snapshot
  - how?
- Independent checkpointing
  - each process checkpoints itself periodically when convenient
  - to produce distributed snapshot
    - start with most recent checkpoints
    - roll back until consistent global checkpoint is achieved



Suppose snapshots of each machine's state were taken at the moments shown in the slide. If all machines crashed and their states were restored with the contents of their respective snapshots, would the system as a whole be in a state that it might have been in before the crash?



This slide (adapted from Figure 8-25 from Tanenbaum and Van Steen) illustrates a problem with independent checkpoints — rolling them back to achieve a consistent global checkpoint might result in rolling back to the distributed system's initial state.

Coping

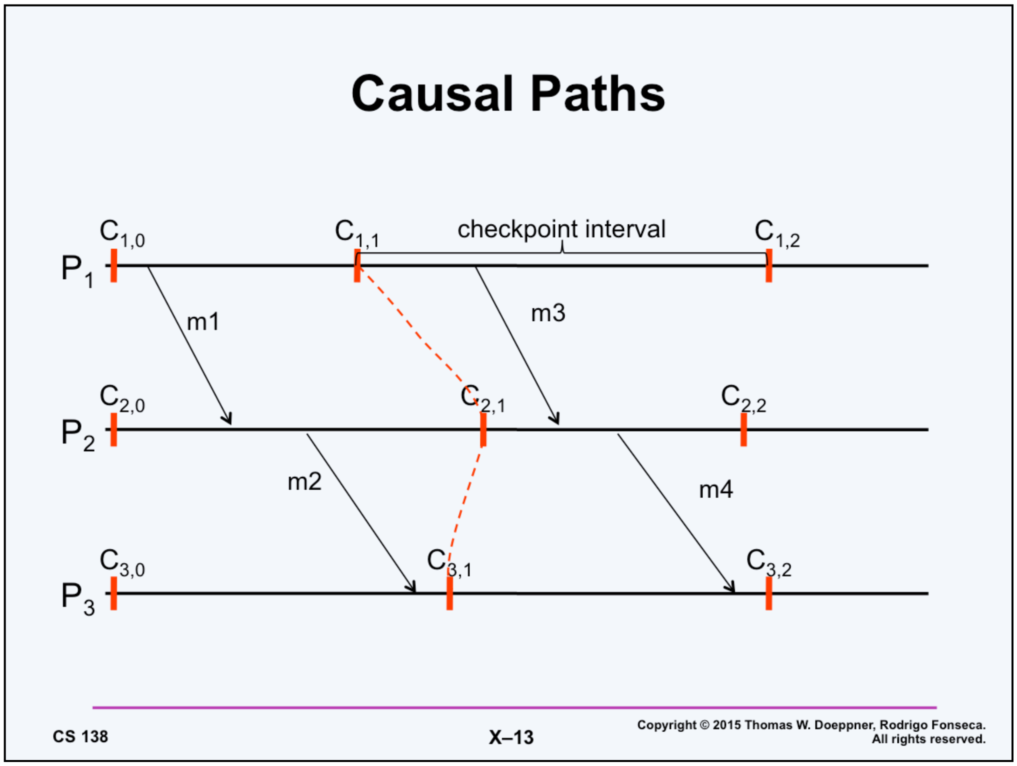
- Take independent, periodic checkpoints, plus a few more

  or

- Produce a global snapshot on demand

Independent Checkpoints

- Goal
  - all checkpoints are “useful”
    - no need to roll back
- What are the conditions for checkpoints to for a consistent cut?

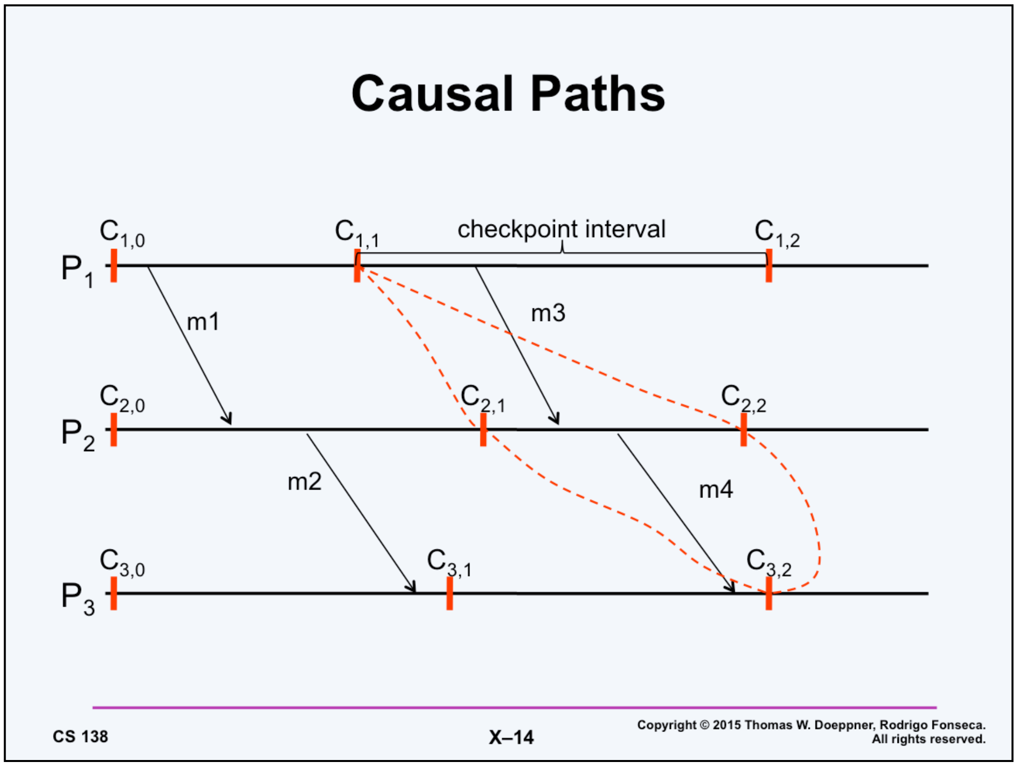


The state of snapshots that form a consistent cut for a global snapshot.

According to the definition, if an event is included in a consistent cut, then all events that happen before that event have to be included as well.

One necessary condition for this to happen, then, is **that there is no path between the different snapshots**, like in the slide.

C11, C21, and C31 form a consistent global snapshot.



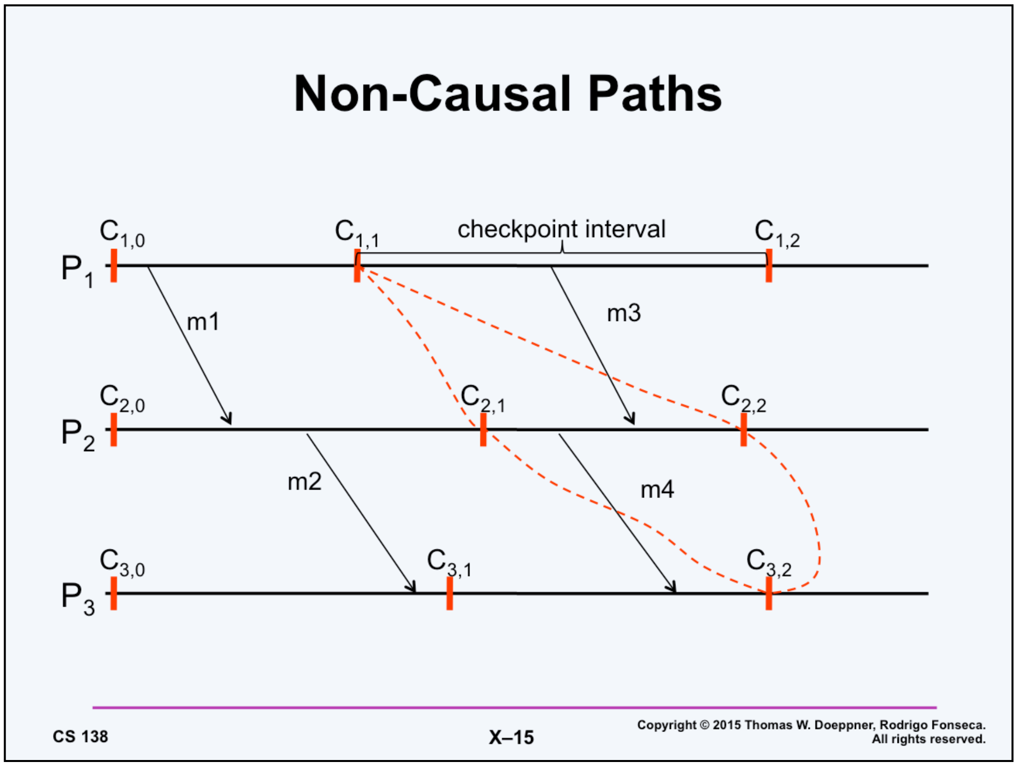
Conversely, if there is a path between the snapshots, then there can't be a consistent global snapshot.

We'd like to come up with an easy method for characterizing when it's the case that a consistent global snapshot cannot be formed.

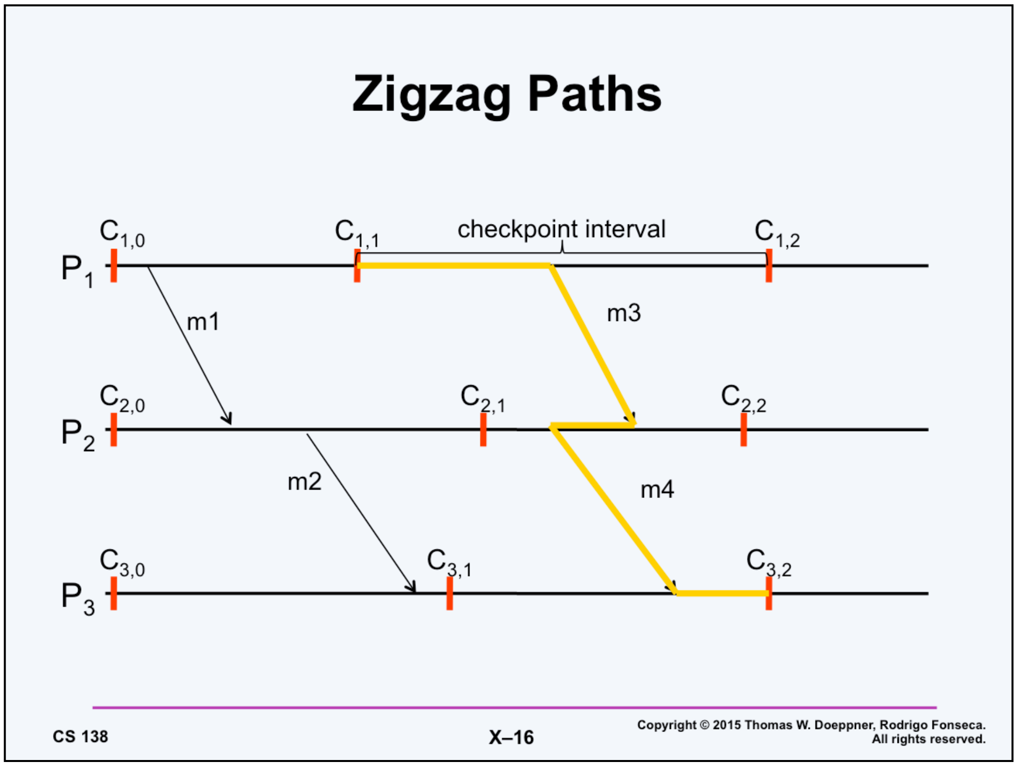
To this end, let's **define a "checkpoint interval"** to be the interval of time in a process that starts with a checkpoint on the process and goes to, but does not include, the next checkpoint on that process. As an hypothesis, one that is borne out in the slide, it seems that if there is a causal path of messages from the checkpoint interval of one checkpoint to the checkpoint interval just prior to another, then the two checkpoints cannot be in the same consistent global snapshot.

The reasoning in support of the hypothesis is to consider any global snapshot containing two checkpoints such that there is a causal sequence of messages from the checkpoint interval of one to the checkpoint interval just prior to the other. For each message in the sequence, the sending of the message must either come before or after the checkpoint of the process doing the send; similarly for the receipt of the message. Since the sending of the first message in the sequence comes after its process's checkpoint and the receipt of the last comes before its process's checkpoint, there must be at least one message in the sequence whose send comes after the sending process's checkpoint and whose receipt comes before the receiving process's checkpoint. Thus the global snapshot is not consistent.

This establishes that the presence of such a causal path **is sufficient** to rule out a global snapshot's being consistent. Is this a necessary condition?



It's not a sufficient condition for there to be no consistent snapshot.
Conversely, the fact that there is no path is not necessary for there to be a snapshot.
This slide is almost identical to the previous one, except that m3 is received after m4 is sent.
Thus there is no causal path from $C_{1,1}$'s checkpoint interval to the interval just prior to $C_{3,2}$.



Let's generalize the notion of a causal path to a "zigzag path" from $C_{1,1}$ to $C_{3,2}$. It's just like a causal path, except that we allow one message to follow another if the receipt of the first is in the same checkpoint interval as the sending of the second. It can be shown (see "Necessary and Sufficient Conditions for Consistent Global Snapshots," Robert H. B. Netzer and Jian Xu, IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 2, February 1995) that if there is such a zigzag path between two snapshots, they cannot be part of a consistent global snapshot.

Zigzag Path Definition

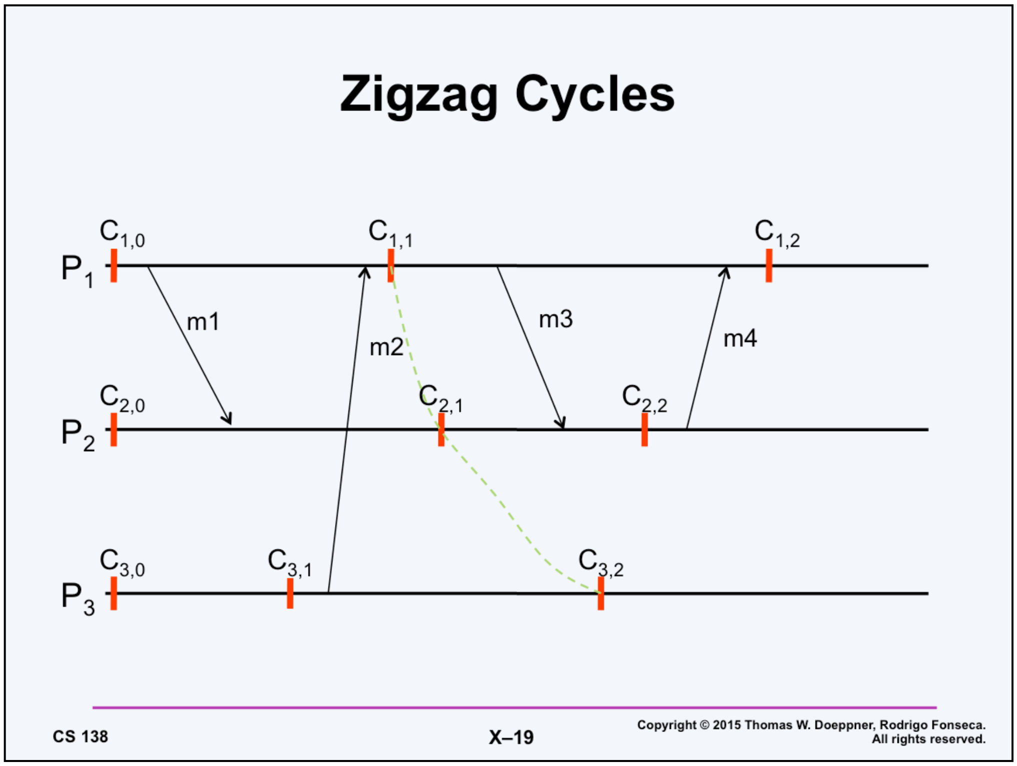
- A zigzag path exists from $C_{p,i}$ to $C_{q,k}$ iff there are messages m_1, m_2, … m_n such that
  - m_1 is sent by process p after $C_{p,i}$
  - if m_h ($1 \le h \le n$) is received by process r, then m_{h+1} is sent by r in the same or a later checkpoint interval (although m_{h+1} may be sent before or after m_h is received), and
  - m_n is received by process q before $C_{q,k}$

This definition is from the aforementioned paper by Netzer and Xu.

Theorem

- **A set of checkpoints S, each from a different process, can belong to the same consistent global snapshot iff no checkpoint in S has a zigzag path to any other checkpoint (including itself) in S**

The theorem is from Netzer and Xu. A proof may be found at ftp://ftp.cs.brown.edu/pub/techreports/93/cs93-32.pdf.



Note that there is a zigzag path from $C_{1,1}$ to $C_{2,2}$; thus the two checkpoints cannot both be in a consistent global snapshot. However, if $C_{1,2}$ took place just before, rather than just after P_1 received m4, then $C_{1,2}$ and $C_{2,2}$ could both be in a consistent global snapshot. In other words, $C_{2,2}$ is *potentially* a useful checkpoint until m4 is sent.

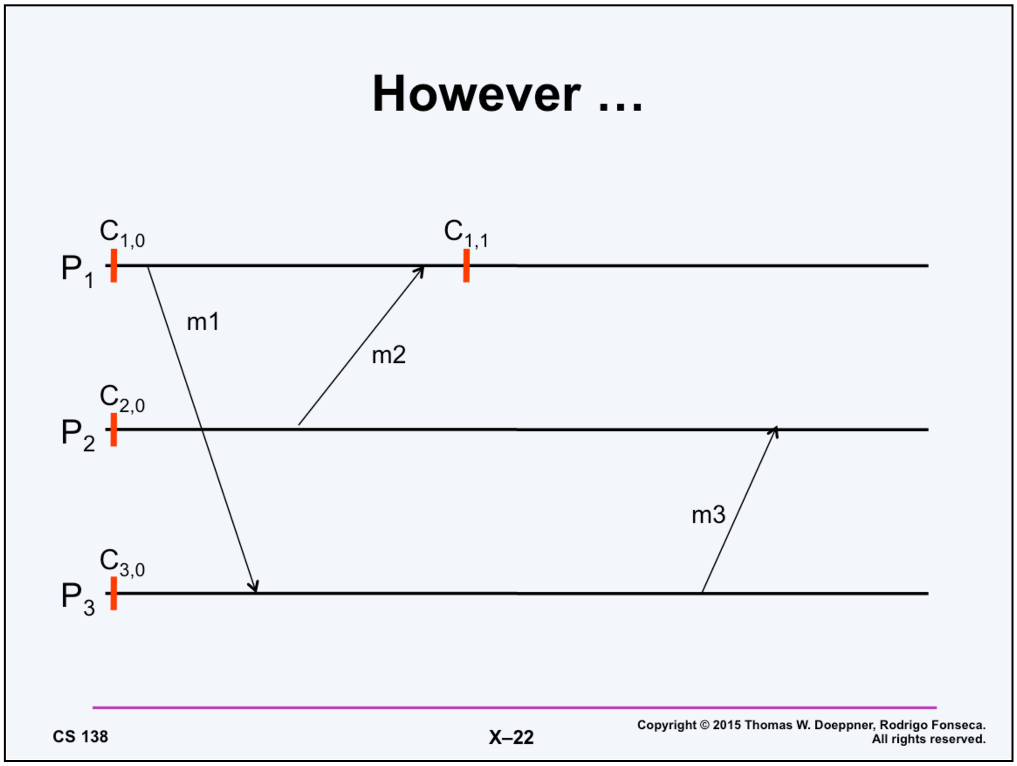
However, if $C_{1,2}$ takes place as shown on the slide (and thus there is a zigzag cycle from $C_{2,2}$ to itself, consisting of messages m3 and m4), then $C_{2,2}$ can henceforth never be in a consistent global snapshot (and is thus no longer useful). If failures occurred after $C_{1,2}$, there would have to be a rollback to the consistent global snapshot shown on the slide.

Corollary

- A checkpoint is *useful* if it potentially belongs to some consistent global checkpoint
- Corollary: A checkpoint is useful iff it is part of no zigzag cycle

Adaptive Checkpointing

- On receipt of a message, receiver checks if message completes a zigzag cycle
  - if so, a new checkpoint is taken before the message is processed
  - thus, no cycle



Does m2 complete a zigzag cycle? As it turns out, it does, but P_1 will have to wait, potentially an unbounded amount of time, to learn if message m3 will occur.

Coping …

- On receipt of message, check for a causal path to a checkpoint preceding the send
  – the path plus the just-received message form a zigzag cycle
- Doesn't catch all zigzag cycles
  – testing shows it catches most of them

Finding Causal Paths

- Use vector clocks
  - components are counts of checkpoints in each process
  - details may be an exercise …

Producing a Consistent Global Snapshot on Demand

- Process A wants all other processes to send it snapshots that together form a consistent cut (and thus a global snapshot)
- Can this be done?

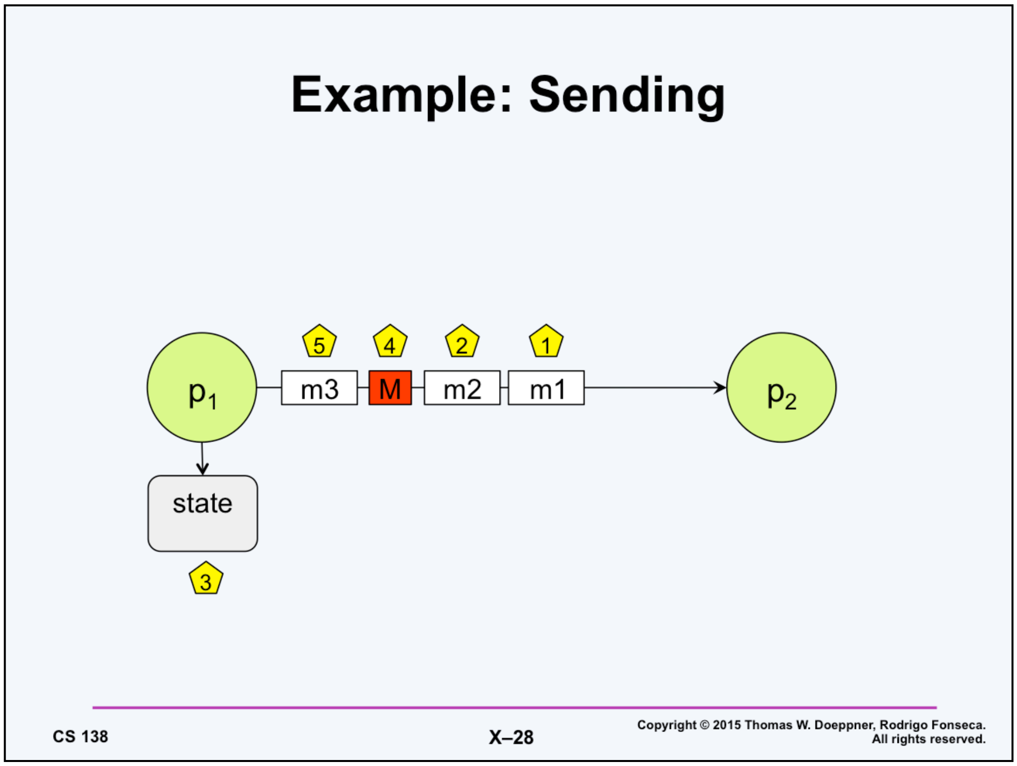
Distributed Snapshot Algorithm

- Chandy & Lamport, 1985
  - algorithm to select a *consistent cut*
  - any process may initiate a snapshot at any time
  - processes can continue normal execution
    - send and receive messages
  - assumes:
    - no failures of processes & channels
    - strong connectivity
      - at least one path between each process pair
    - unidirectional, FIFO channels
    - reliable delivery of messages

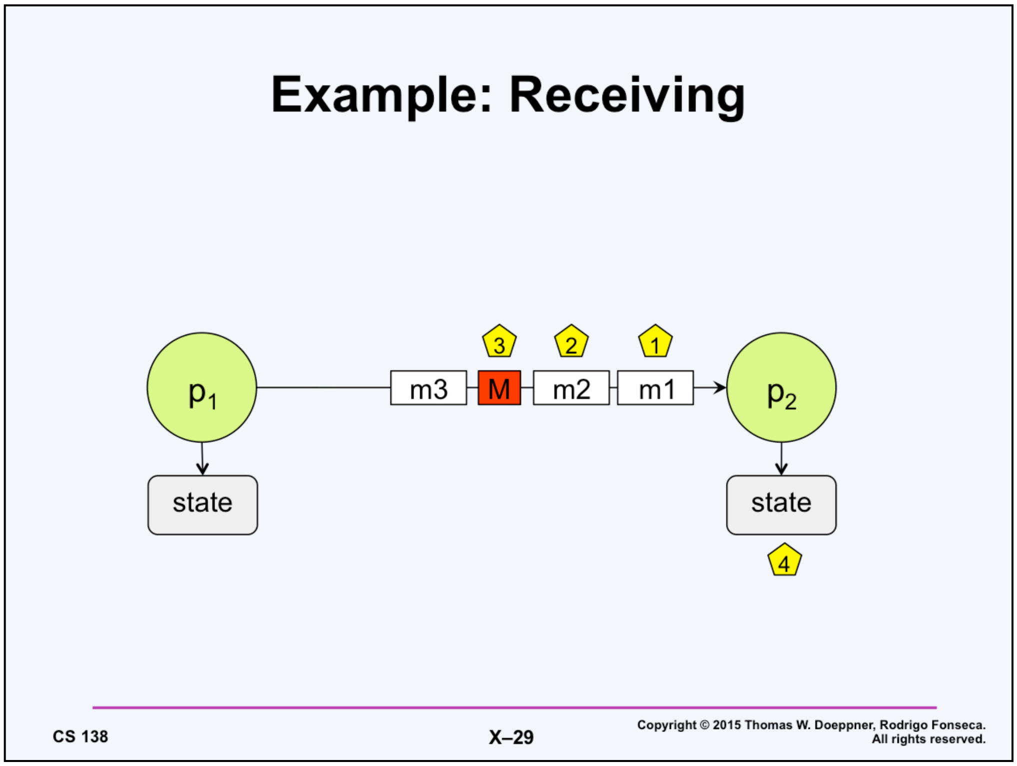
Chandy and Lamport's snapshot algorithm is used to select consistent cuts. The algorithm is distributed: it works by recording the local states of each process and then by merging them. The assumptions the algorithm makes are summarized above.

Approach

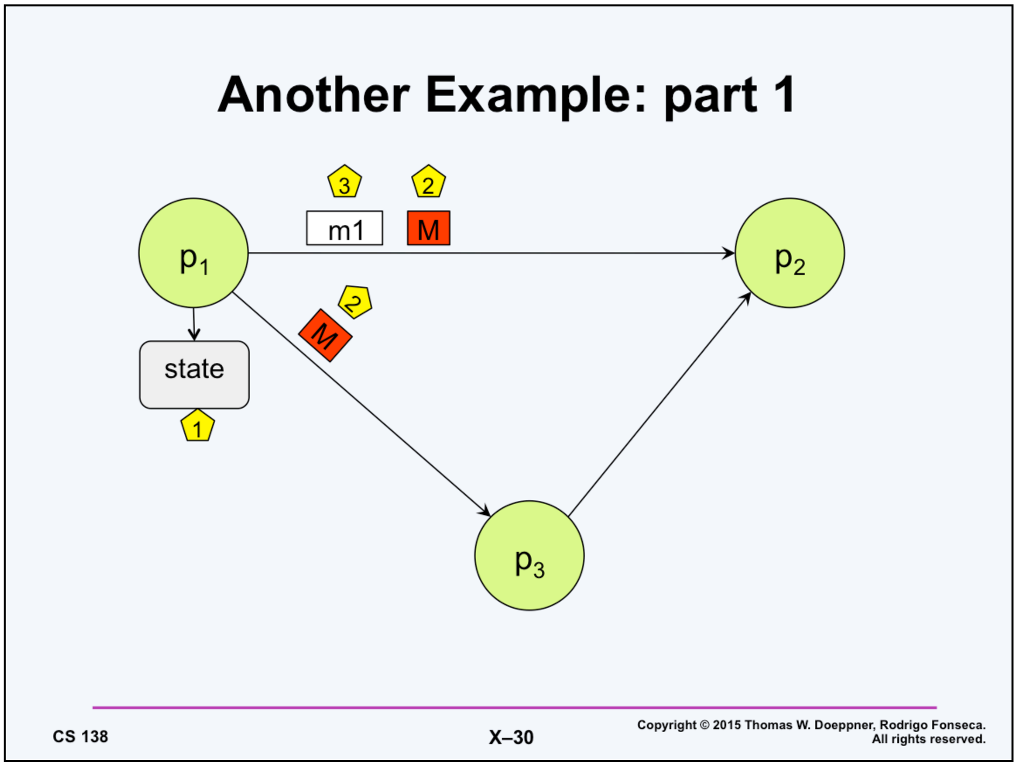
- Snapshot consists of saved states of all nodes along with messages in transit
- For each pair of directly connected nodes A and B
  - must record messages sent before A saved its state but received after B saved its state
  - nodes send out special *marker* messages immediately after saving their states



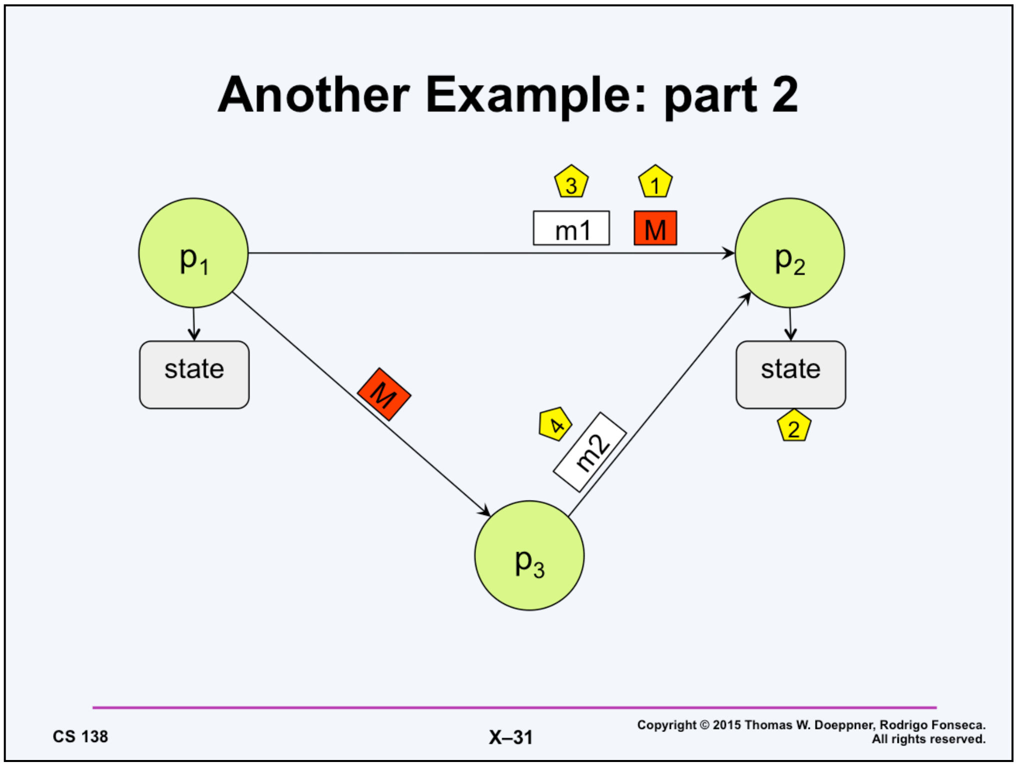
Here we have a two-node system. P_1 sends out two messages and then decides to initiate a snapshot. It saves its state, then sends out a marker message on the channel to p_2. It then sends out another message.



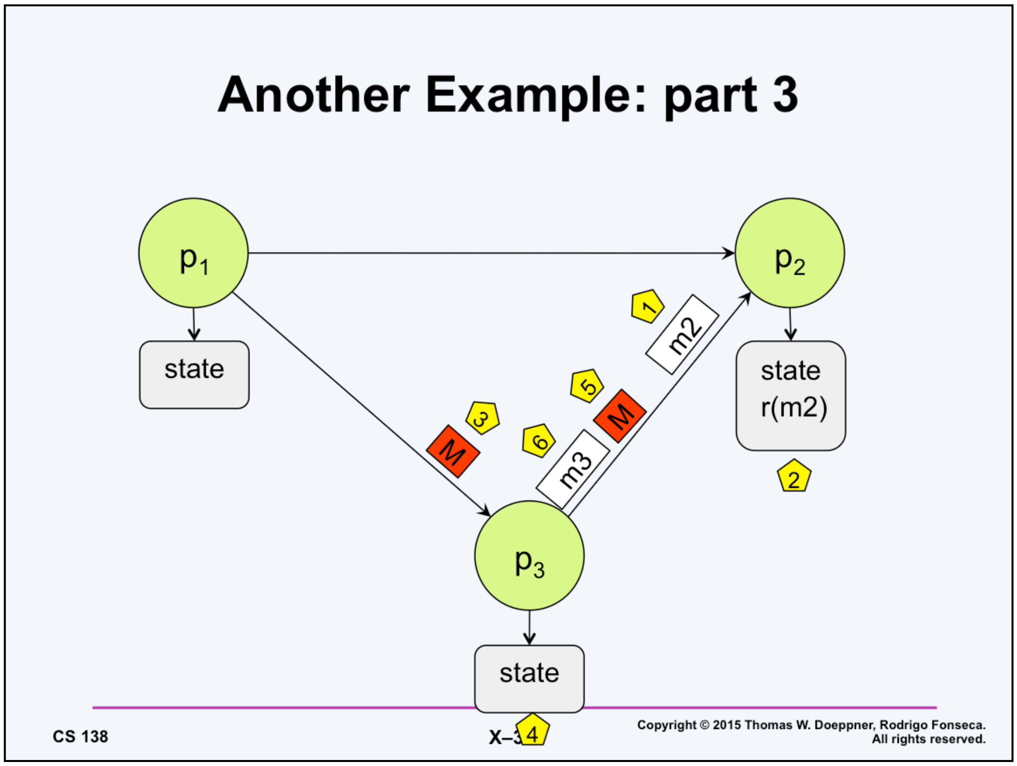
P_2 receives the first two messages, then receives the marker message. This is its indication to save its state, so it does so. Since the third message was sent after p_1 saved its state, there is no need for p_2 to record it.



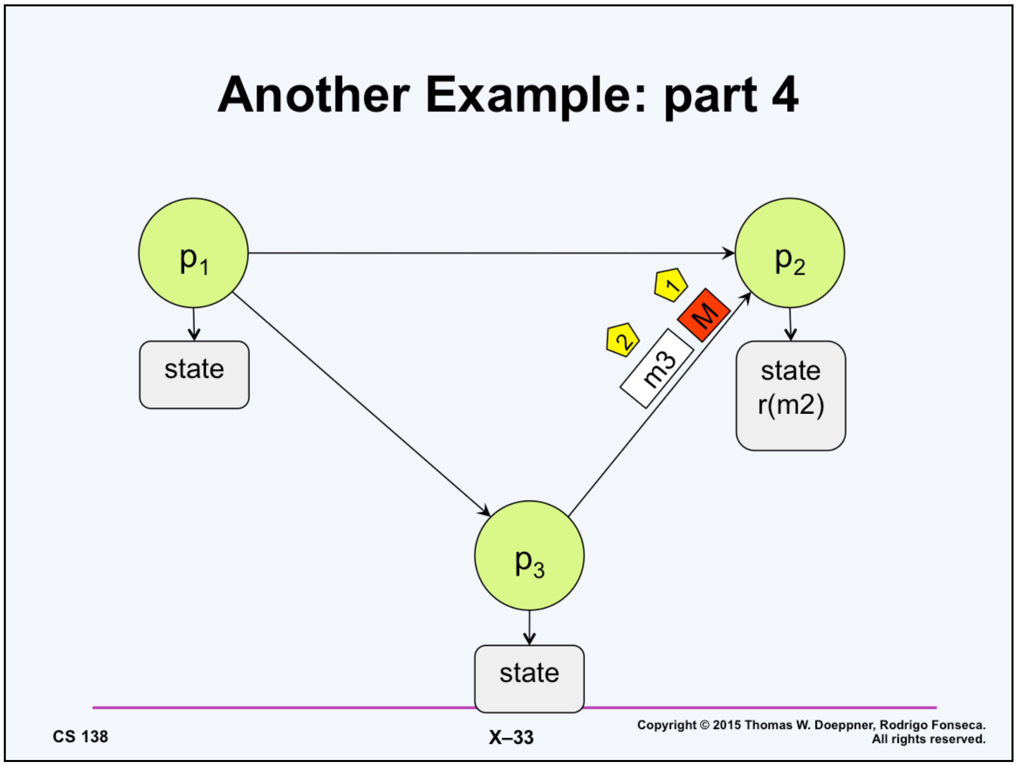
In this example, p_1 decides to start a snapshot, so it first saves its state, then sends out marker messages on all of its outgoing channels. Having done this, it sends message m1 on the channel to p_2.



P_2 receives the marker message, which lets it know that it should take a snapshot. So it saves its state. It then receives message m1, but doesn't record it in the snapshot (since it was sent after p_1 saved its state). The marker message to p_3 is still in transit. In the meantime, p_3 sends message m2 to p_2.



P_2 receives message m2. Since it was sent before p_3 recorded its state (which it hasn't done yet), p_2 records the message as part of the channel's state. P_3 finally receives the marker message from p_1, so it records its state, and then sends a marker message on the channel to p_2 to let p_2 know that it has finally recorded its state.



Finally, p_2 receives p_3's marker message, thus letting it know that p_3 has saved its state. Thus when m3 arrives, p_2 does not record it.

Snapshot Rules

- ***Marker receiving rule for process p_i***

  On p_i's receipt of a *marker* message over channel c:

  if **(p_i has not yet recorded its state)**

  it records its state

  it records the state of c as the empty sequence

  it turns on recording of messages arriving over other channels

  else

  p_i records the state of c as the set of messages it has received over c since it saved its state and before it received the marker over c

- ***Marker sending rule for process p_i***

  After p_i has recorded its state, for each outgoing channel c:

  p_i sends one marker message over c (before it sends any other messages over c)

The algorithm can be defined using two *state marking rules***.** The *marker sending rule* requires processes to send a marker along each of their outgoing links after they have saved their local state. The *marker receiving rule* requires a process that hasn't yet recorded its state to do so, after the receipt of the first marker. It then starts noting the messages that it receives on the other incoming links. If a process receives a message after it has already saved its local state, then it records the *state of that channel* as the messages that it received on that channel since it saved its state.

The algorithm can be initiated by any server assuming that the server has received an imaginary marker from some imaginary incoming link. Several processes can initiate the process concurrently. The algorithm works fine as long as the markers can be uniquely differentiated.

Termination

- Process P has completed its part of the algorithm when it has processed markers on all input channels
- It sends its saved local state and channel histories to the initiator
  – the intent is that collection of local states form consistent cut
    - channel histories are the messages in transit at time of cut

Analysis

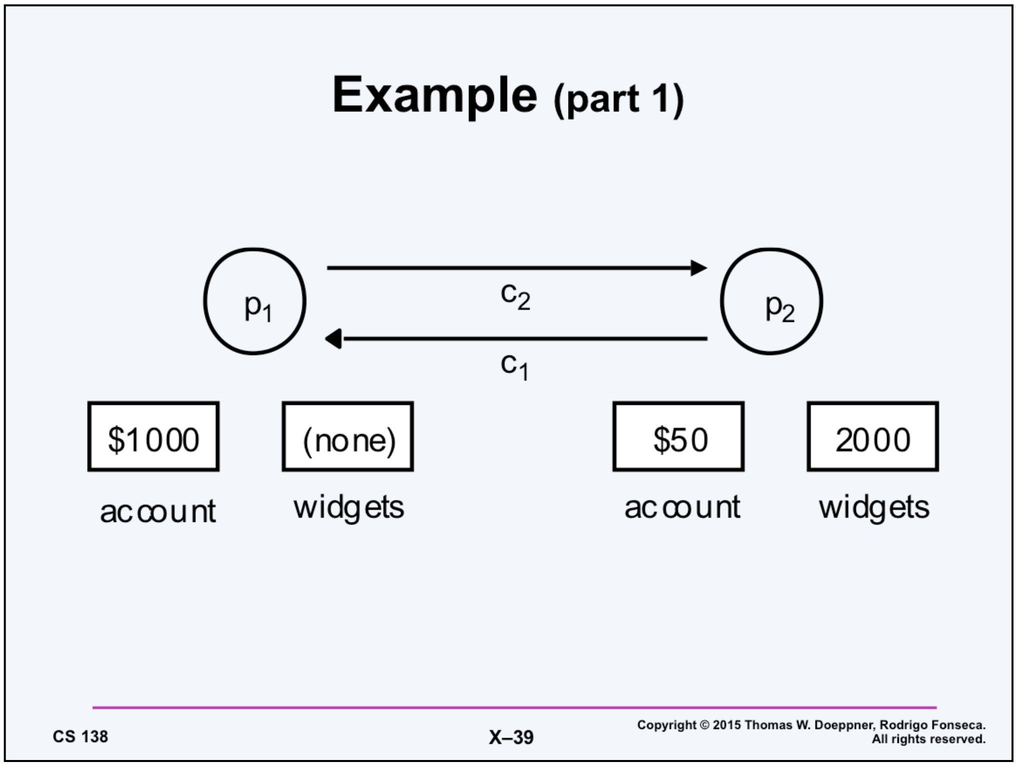
- Does it find a consistent cut?
  - if so, then for any P_a and P_b, if m is a message sent from P_a to P_b, then if recv(m) is in the cut, so is send(m)
    - i.e., if recv(m) occurred before P_b recorded its state, then send(m) occurred before P_a recorded its state
  - stronger statement: if for any P_a and P_b, if e_a and e_b are events in P_a and P_b, such that e_a happens before e_b ($e_a \rightarrow e_b$), then if e_b is in the cut, so is e_a
    - i.e., if e_b occurred before P_b recorded its state, then e_a occurred before P_a recorded its state

Proof

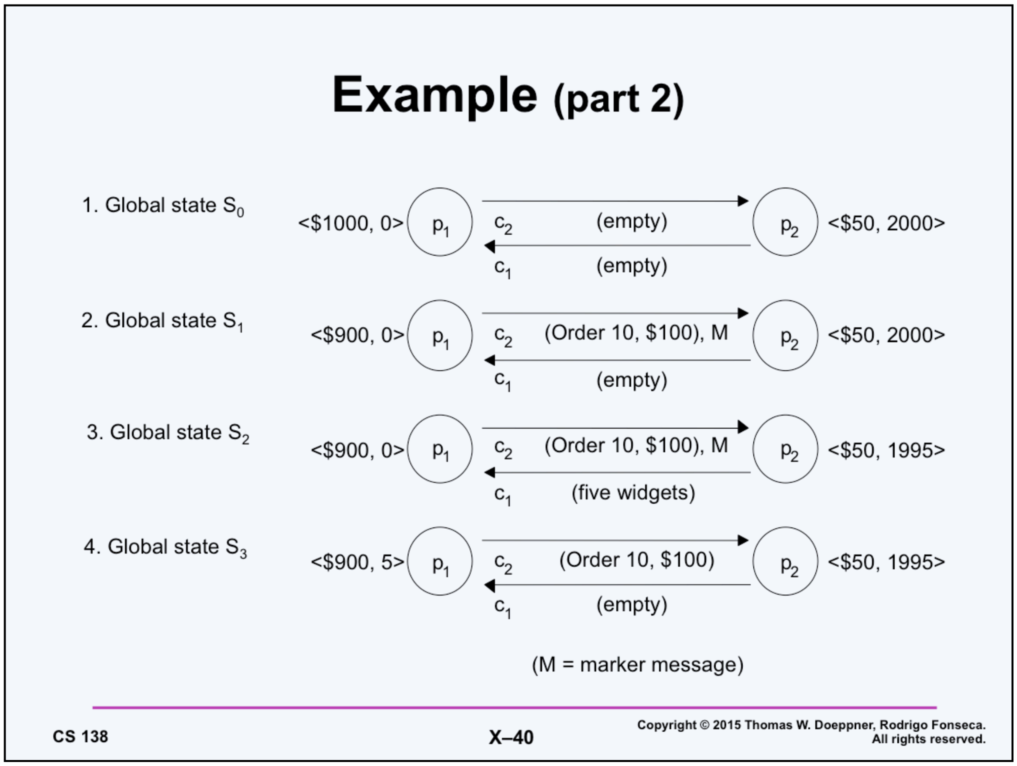
- Assume no: P_a recorded its state before e_a occurred (e_b is in the cut, but e_a is not)
  - since $e_a \rightarrow e_b$, there was some sequence of messages $m_1, m_2, \ldots, m_h$ that brought on $e_a \rightarrow e_b$
  - since P_a recorded its state before e_a occurred, it sent marker messages out on all its outgoing channels before transmitting m_1
  - since the channels are FIFO, a marker reached P_b before m_h
  - but then P_b would have recorded its state before e_a
  - but then e_b would not have been in the cut
    - contradiction

More Analysis

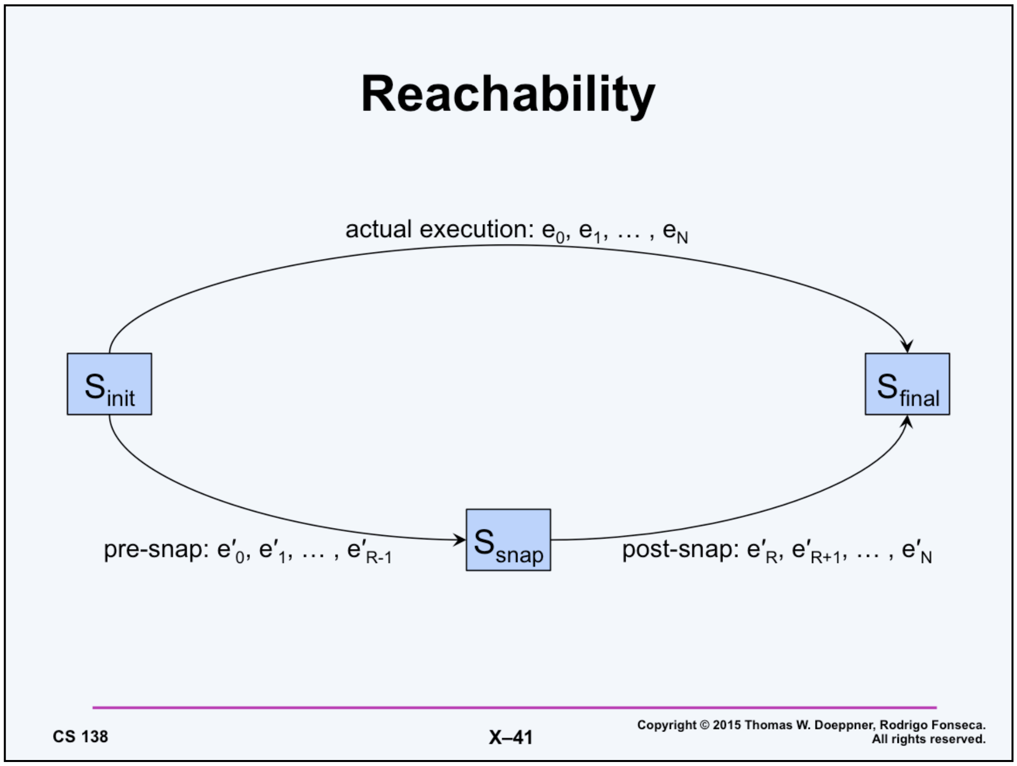
- Snapshot taken isn't necessarily a state that actually happened!
  - but it could have happened …
- If distributed system deadlocks, no distributed snapshot



The slide is Figure 14.11 from Coulouris, Dollimore, Kindberg, and Blair. We have two processes that trade in widgets. Process p_1 has already send an order for 5 widgets (at $10 each) to p_2, which is about to send a response message containing the widgets.



Process p1 begins the snapshot algorithm and records its state: <$1000, 0>. The actual global state at this moment is shown in row 1. In row 2, p1 has sent a marker message along with an order for 10 more widgets. Before p2 receives either message, it responds, in line 3, to the earlier message with 5 widgets, and this response is processed by p3. Finally, in line 4, p2 receives the marker message and records its state: <$50, 1995>. Thus the recorded global snapshot is << $1000, 0>, <$50, 1995>>, a state that never actually occurred.

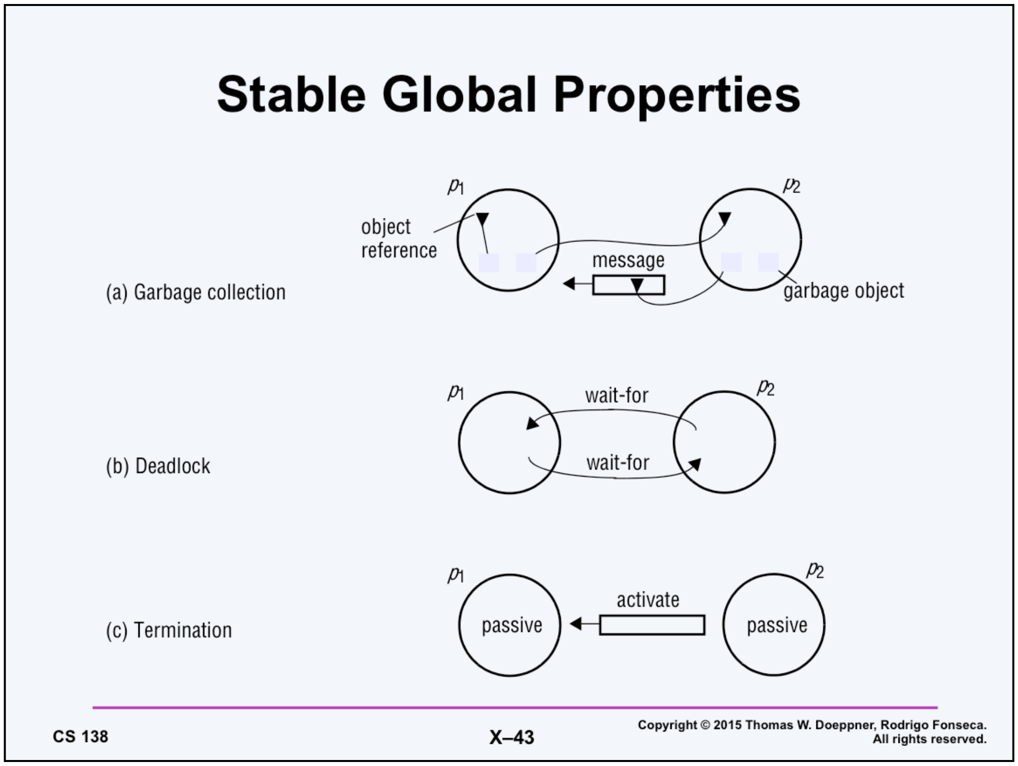


This slide is adapted from Figure 14.13 of Coulouris, Dollimore, Kindberg, and Blair. S_{init} is the global state the system was in just prior to the beginning of execution of the snapshot algorithm. S_{final} is the global state the system was in when the algorithm terminated. S_{snap} is the global state represented by the snapshot. The system went through the sequence of events e_0, e_1, … in going from S_{init} to S_{final}. We claim that S_{snap} is reachable from S_{init} by the sequence of events e'_0, e'_1, … e'_{R-1}, and S_{final} is reachable from S_{snap} by the sequence of events e'_R, e'_{R+1}, … e'_N. Furthermore, e'_0, e'_1, … , e'_{R-1}, e'_R, e'_{R+1}, … , e'_N is a permutation of e_0, e_1, … , e_N. Each of the events in e_0, e_1, … , e_N took place in some process either before or after it recorded its state (i.e., produced a snapshot).

Suppose that e_i is a post-snapshot event at one process and e_{i+1} is a pre-snapshot event at another. It cannot be that $e_i \rightarrow e_{i+1}$, since this would mean that the first is the sending of a message and the second is the receiving of the same message. Since e_i is a post-snapshot event, a marker message would have had to precede the message, making the second event a post-snapshot event, contrary to our assumption. Thus there is no causal relation between the two events and they may be swapped without violating happened-before relationships. By swapping such pairs of events, we can move all pre-snapshot events to the front of the sequence and all post-snapshot events to the rear. The pre-snapshot events are thus e'_0, e'_1, … , e'_{R-1}, and the post-snapshot events are thus e'_R, e'_{R+1}, … , e'_N.

Global Properties

- Safety
  - bad things will not happen
  - e.g., mutual exclusion is a safety property
- Liveness
  - good things will happen
  - e.g., termination is a liveness property
- Stable properties
  - once true — always true
- Transient properties
  - once true — who knows?



This slide is Figure 14.8 from Coulouris, Dollimore, Kindberg, and Blair.

Case (a) above shows a *distributed garbage collection* example where process p1 has two objects that have references to them (one local and one remote); and process p2 has one garbage object and another whose reference is **in** *transit* to p1. This example demonstrates that we also need to take into account the state of the communication channel when we talk about the global properties of a system.

Case (b) demonstrates a *distributed deadlock* scenario, where two processes are blocked waiting to hear from each other. If this is the case, then the processes will not be able to make any progress. Detecting deadlocks requires forming a waits-for graph and checking whether the graph has any cycles.

Case (c) shows a *distributed termination detection* scenario. The problem here is to determine that a distributed algorithm has terminated. This involves more than just checking whether each process has halted. We need to also consider any *activation messages* that may be on their way to their destinations.

Transient Properties

- Distributed debugging
  - assert($\forall a \neq b$ ($|x_a - y_b| < 10$))
    - x_a and y_a reside in process a

How To …

- State collection
  - each process sends snapshots to central server
  - contain vector timestamps
- Central server checks for transient property ϕ
  - looks at global states that could have resulted from initial state, given vector timestamps
- ***possibly ϕ***
  - if ϕ holds in at least one of them
- ***definitely ϕ***
  - for all possible (causally consistent) orderings, ϕ holds at some point

Computing possibly ϕ and definitely ϕ are possible, though certainly time consuming. See Coulouris, Dollimore, Kindberg, and Blair, Section 14.6.