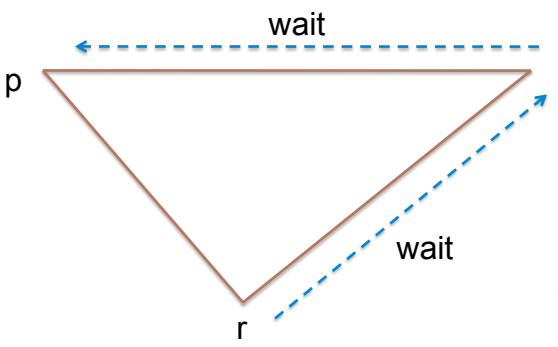
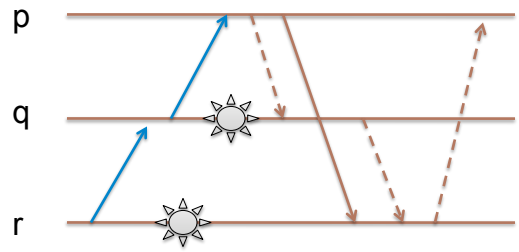
DISTRIBUTED SNAPSHOTS

Global Predicate Evaluation

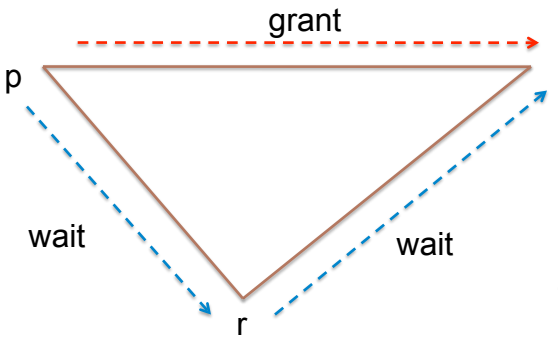
- Passive Monitor + causal delivery
  - Processes send notifications of events
  - Guarantees consistent observations
  - See examples involving delivery rules earlier
- Active Monitor + causal delivery
  - Monitor runs protocol with processes as they are executing the application
  - Distributed snapshot algorithms
  - What we are concerned with now



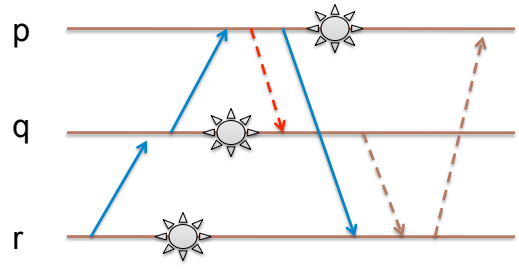
Deadlock Detection and Ghost Deadlocks



96



Deadlock Detection and Ghost Deadlocks



97

Computing Snapshots

- Chandy and Lamport: flooding algorithm
- We'll consider several other methods too
  - Logical timestamps
  - Flooding algorithm without blocking
  - Two-phase commit with blocking
- Each pattern arises commonly in distributed systems

98

Assumptions

- Channels reliable, preserve FIFO order
- State of a channel, Ch(i,j), are those messages that p_i has sent to p_j but p_j has not yet received
- IN(I) = set of channels incoming to p_i
- OUT(I) = set of channels outgoing from p_i
- Each process p_i records local state s_i and state of its incoming channels Ch(j,i), for all j in IN(i)

99

SNAPSHOTS USING CLOCKS

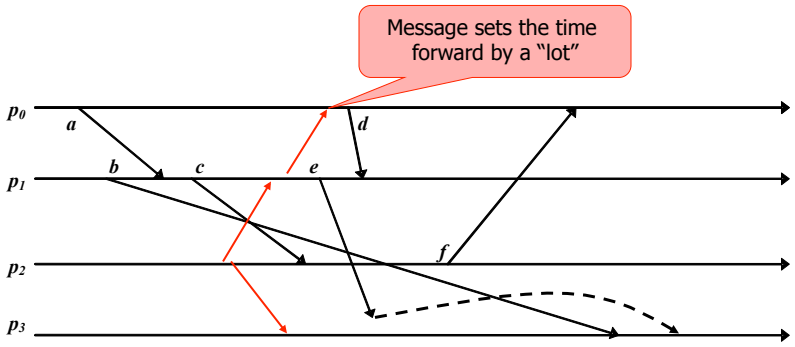
100

Snapshots using Real Time Clocks

- P_0 sends message "take snapshot at real time ts" to all processes
- When RC = ts, all processes:
  - record local state s_i
  - send empty ("snap") messages over outgoing channels
  - start recording messages received over incoming channels
  - for each incoming channel, record channel state until receiving a message with TS(m) ≥ ts
    - guaranteed to receive such a message on every incoming channel because of snap messages sent out

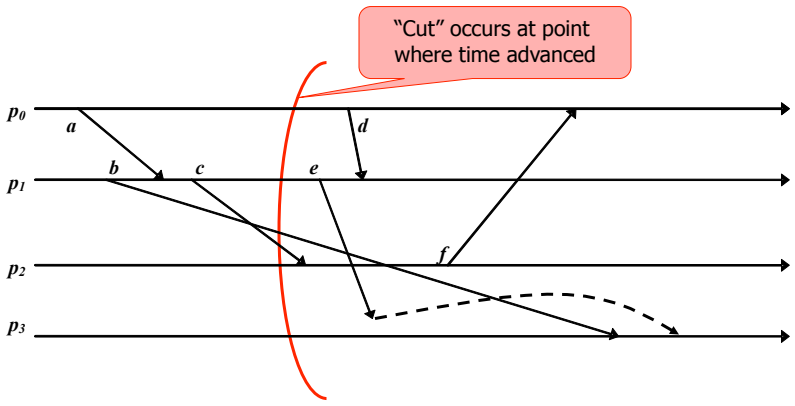
101

Snapshots using Logical Clocks



102

Snapshots using Logical Clocks



103

Snapshots using Logical Clocks

- All processes recording snapshot:
  - record local state s_i
  - set their logical clock to some big value
    - Think of clock as (epoch-number, counter)?
  - send empty ("snap") messages over outgoing channels
  - start recording messages received over incoming channels
  - for each incoming channel, record channel state until receiving a message with big incoming clock value

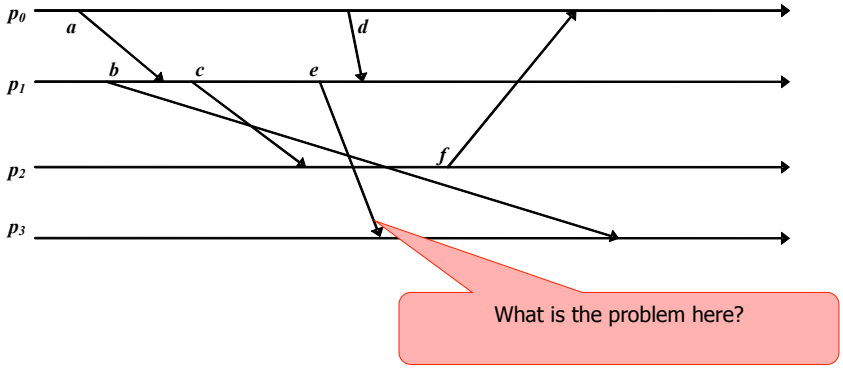
104

How does this work?

- Recall that with Lamport's clocks, if e is causally prior to e' then LC(e) < LC(e')
- Our scheme creates a snapshot for each process at instant it reaches logical time t
- Easy to see that these events are concurrent: a possible instant in real-time
- Depends upon FIFO channels

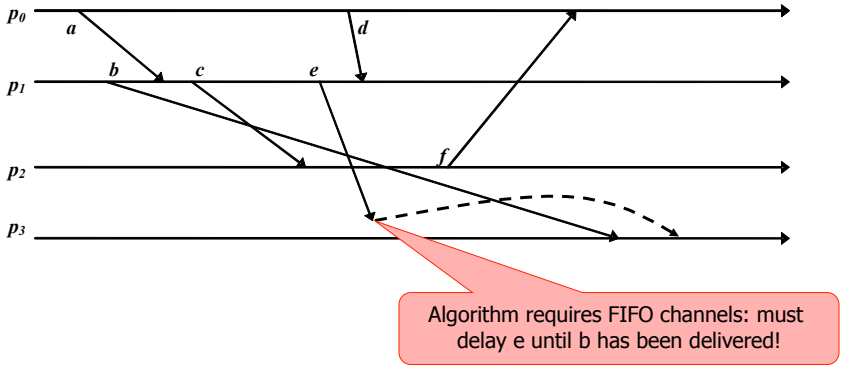
105

Logical Clocks for Snapshots



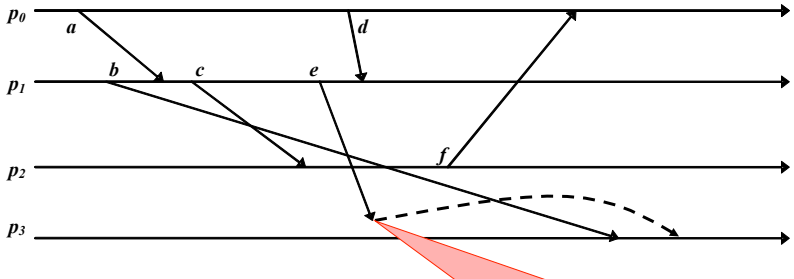
106

Logical Clocks for Snapshots



107

Logical Clocks for Snapshots



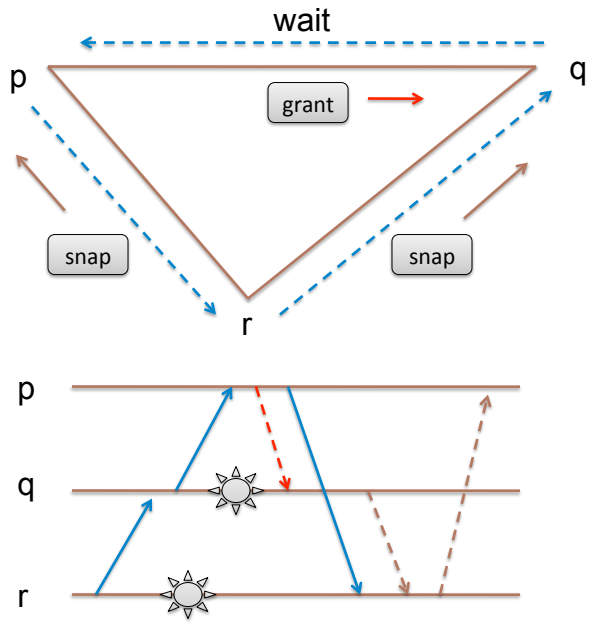
Note: This is FIFO ordering, not causal ordering! Causal ordering will be an implicit result of the protocol.

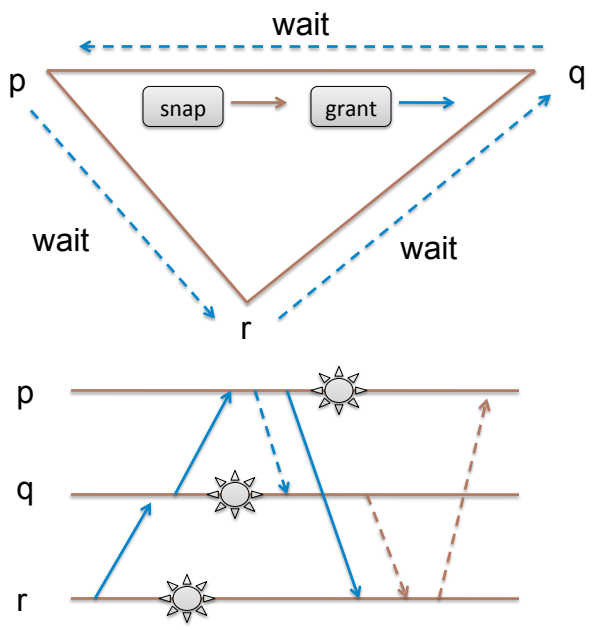
SNAPSHOTS USING FLOODING (CHANDY & LAMPORT)

109

Snapshots using Flooding

- To make a cut, observer sends out "snap" messages
- On receiving "snap" the first time, process A
  - Writes down its state
  - Creates empty channel state record for all incoming channels
  - Sends "snap" on all outgoing channels
  - Waits for "snap" on all incoming channels
  - A's piece of the snapshot is its state and the channel contents once it receives "snap" from all neighbors
- Note: also assumes FIFO channels





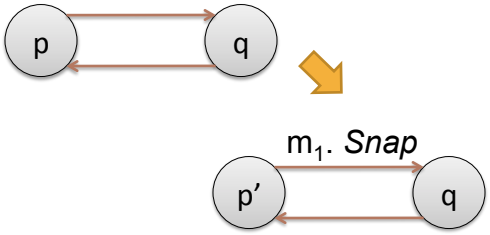
112

Property of Snapshot

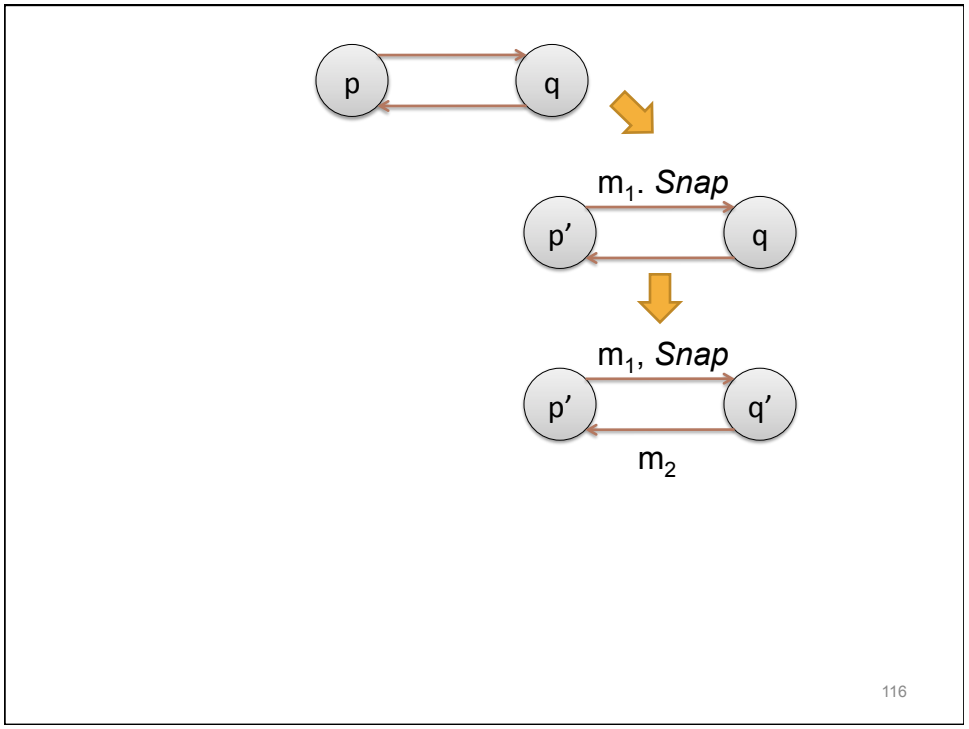
- Observed global state may not have been reached

113

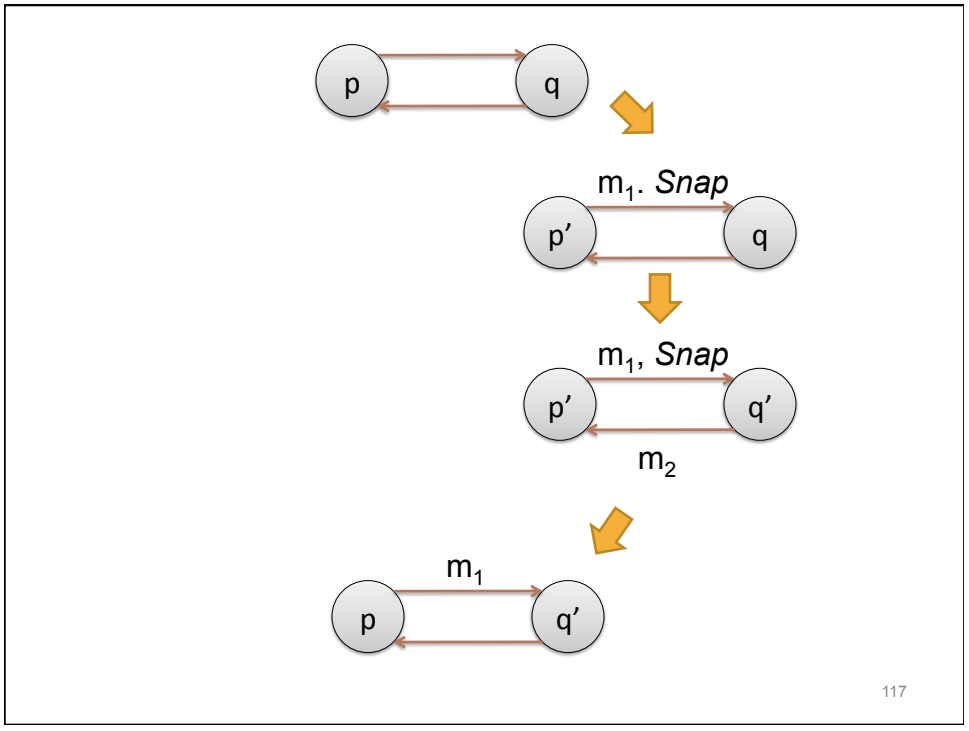
114



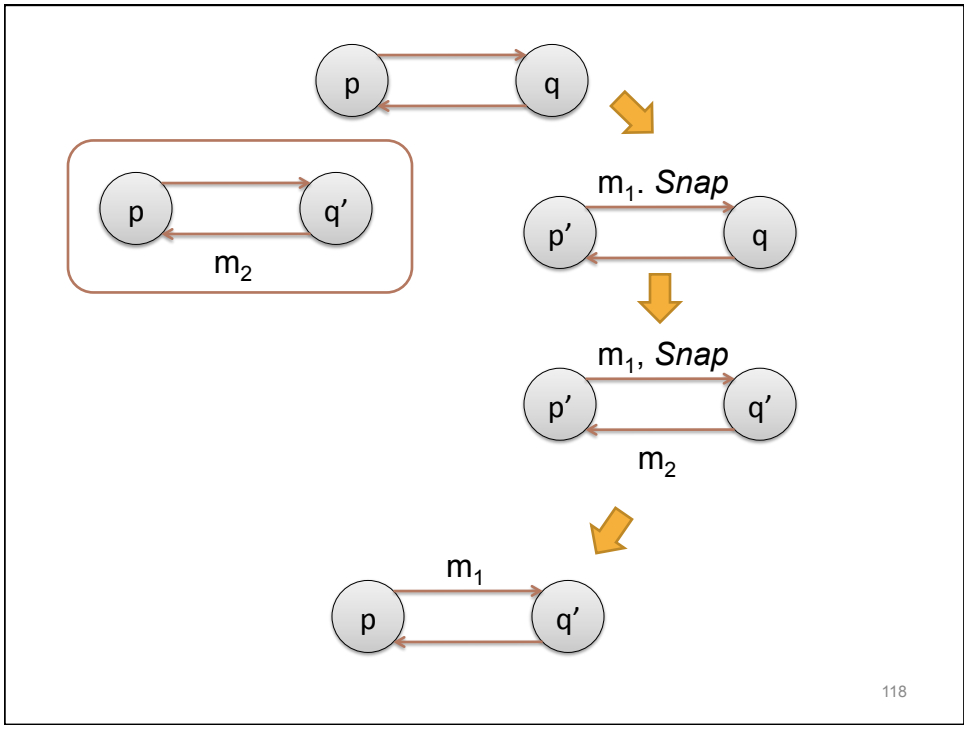
115

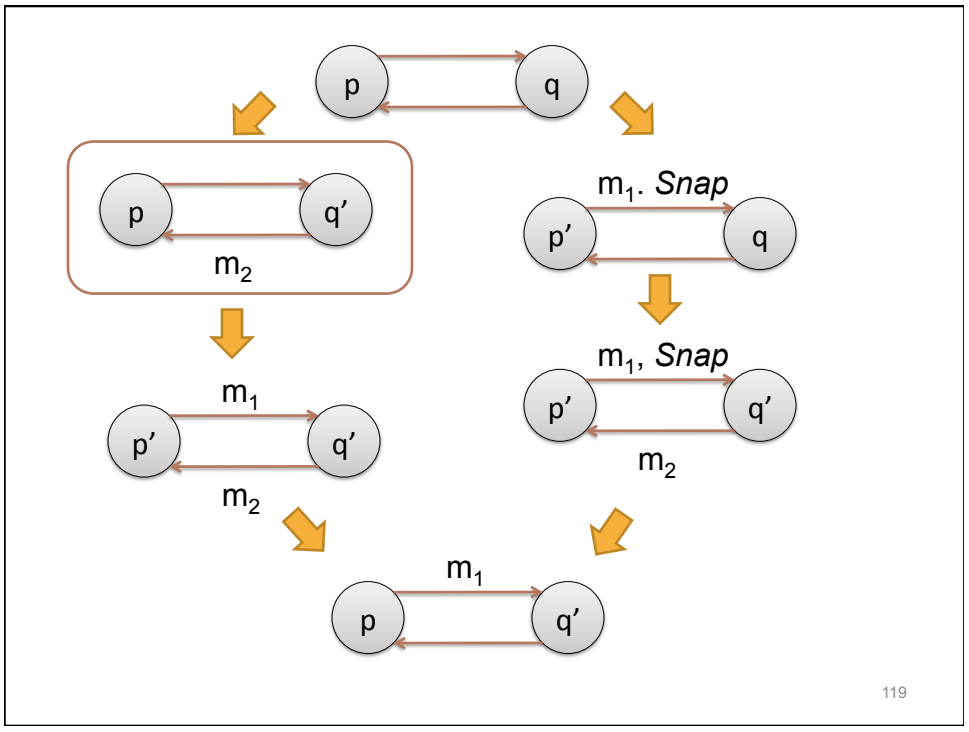


116



117


p
q
p
q'
m2
m1. Snap
p'
q
m1, Snap
p'
q'
m2
m1
p
q'
118


p
q
p
q'
m2
m1. Snap
p'
q
m1
p'
q'
m2
m1, Snap
p'
q'
m2
m1
p
q'
119

Property of Snapshot

- Observed global state may not have been reached
- But it is a **reachable** global state given starting state and ending state
- Snapshot protocol useful for checking **stable** predicates (e.g. deadlock, termination detection, garbage collection)

120

SNAPSHOTS USING 2PC

121

2-phase Commit Protocol

- In this, the initiator sends to all neighbors:
  - "Please halt"
  - A halts computation, sends "please halt" to all downstream neighbors
  - Waits for "halted" from all of them
  - Replies "halted" to upstream caller
- Now initiator sends "snap"
  - A forwards "snap" downstream
  - Waits for replies
  - Collects them into its own state
  - Send's own state to upstream caller and resumes

122

Why does this work?

- Forces the system into an idle state
- In this situation, nothing is changing…
- Usually, *sender* in this scheme records unacknowledged outgoing channel state
- Alternative: upstream process tells receiver how many incoming messages to await, receiver does so and includes them in its state.
- So a snapshot can be safely computed and there is nothing unaccounted for in the channels

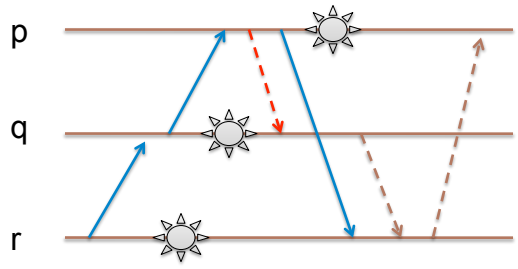
123

Another 2-Phase Protocol

- Suppose we use a two-phase property detection algorithm
- In first phase, asks (for example), "what is your current state"
- You reply "waiting for a reply from B" and give a "wait counter"
- If a second round of the same algorithm detects the same condition with the same wait-counter values, the condition is stable…

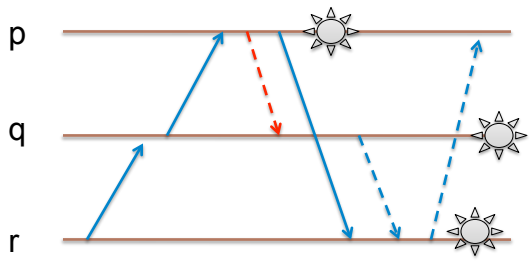
124

A ghost deadlock



125

A ghost deadlock



126

Look twice and it goes away…

- But we could see “new” wait edges mimicking the old ones
- This is why we need some form of counter to distinguish same-old condition from new edges on the same channels…
- Easily extended to other conditions

127

Hidden assumptions

- Use of FIFO channels is a problem
  - Many systems use some form of datagram
  - Many systems have multiple concurrent senders on same paths
- These algorithms assume knowledge of system membership
  - Hard to make them fault-tolerant
  - Recall that a slow process can seem "faulty"

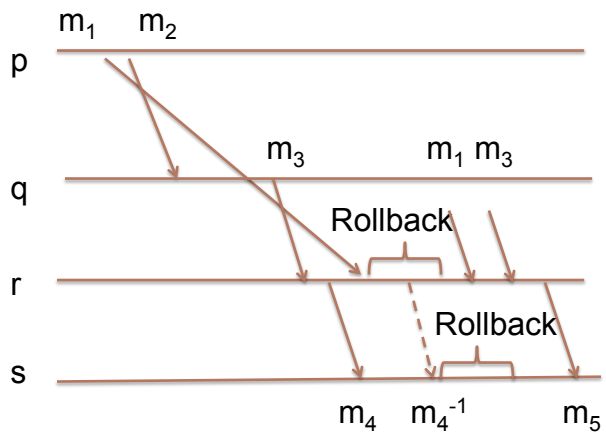
High costs

- With flooding algorithm, n^2 messages
- With 2-phase commit algorithm, system pauses for a long time

VIRTUAL TIME

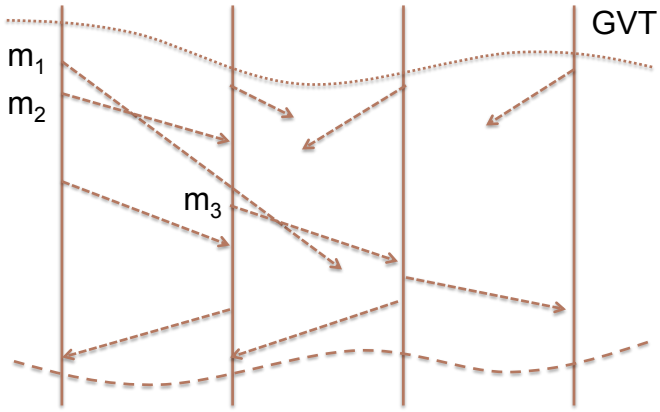
130

Virtual-1



131

Virtual-2



132

WALL CLOCK TIME

133

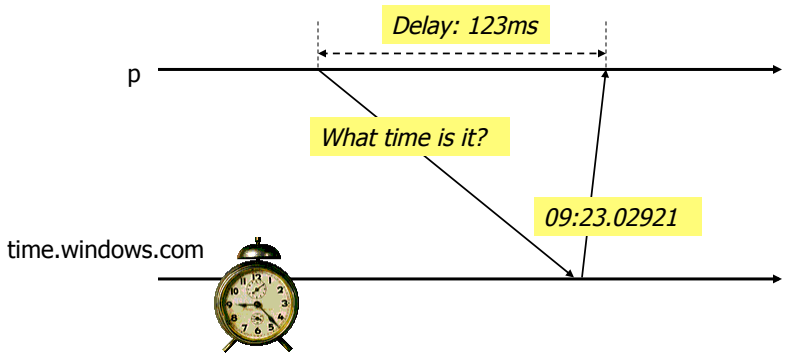
Introducing "wall clock time"

- There are several options
  - "Extend" a logical clock or vector clock with the clock time
    - "B and D were concurrent, although B occurred first"
    - But unless clocks are closely synchronized such statements could be erroneous!
  - Clock synchronization algorithm
    - reconcile differences between clocks

Synchronizing clocks

- Without help, clocks will often differ by many milliseconds
  - Problem : delay in downloading network time
  - "Uplink" vs "downlink" delays
- Outright failures of clocks are rare…

Synchronizing clocks



- *Suppose p synchronizes with time.windows.com and notes that 123 ms elapsed while the protocol was running… what time is it now?*

136

Synchronizing clocks

- Options?
  - Guess that the delay was evenly split
  - Ignore the delay
  - Factor in only "certain" delay
    - Ex: GPS and we *know* that the link takes at most 5ms
- In general can't do better than uncertainty in the link delay

137

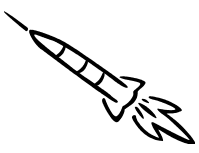
Consequences?

- Clocks not perfectly synchronized.
  - We say that clocks are "inaccurate"
- Clocks can drift during periods between synchronizations
  - Relative drift between clocks is their "precision"

138

Thought question

- We are building an anti-missile system
- Radar tells the interceptor where it should be and what time to get there
- Do we want the radar and interceptor to be as accurate as possible, or as precise as possible?



Real systems?

- Typically
  - "master clock" owner
  - periodically broadcasts the time
- Processes then update their clocks
  - But drift between updates
  - Low accuracy
  - Often precision will be poor compared to message round-trip times

Clock synchronization

- To optimize for precision we can
  - Set all clocks from a GPS source or some other time "broadcast" source
    - Limited by uncertainty in downlink times
  - Or run a protocol between the machines
    - Many have been reported in the literature
    - Precision limited by uncertainty in message delays
    - Some can even overcome arbitrary failures in a subset of the machines!