CS514: Intermediate Course in Computer Systems

Lecture 11: Oct. 6, 8, 2003

Time and ordering

Time and Ordering

CS514

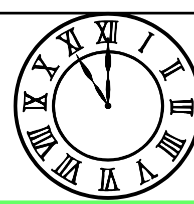
- We tend to casually use temporal concepts
- Example: "membership changes dynamically"
  - Implies a notion of time: *first* membership was X, *later* membership was Y
  - Challenge: relating local notion of time in a single process to a global notion of time
- Will discuss this issue before developing multicast delivery ordering options in more detail

Time in Distributed Systems

CS514

- Three notions of time:
  - Time seen by external observer. A global clock of perfect accuracy
  - Time seen on clocks of individual processes. Each has its own clock, and clocks may drift out of sync
  - Logical notion of time: event *a* occurs before event *b* and this is detectable because information about *a* may have reached *b*

External Time



CS514

- The "gold standard" against which many protocols are defined
  - *Not implementable*: no system can avoid uncertain details that limit temporal precision!
  - Use of external time is also risky: many protocols that seek to provide properties defined by external observers are *extremely* costly and, sometimes, are unable to cope with failures

Time seen on internal clocks

CS514

- Most workstations have reasonable clocks
- Clock synchronization is the big problem (will visit topic later in course): clocks can drift apart and resynchronization, in software, is inaccurate
  - Unpredictable speeds a feature of all computing systems, hence can't predict how long events will take (e.g. how long it will take to send a message and be sure it was delivered to the destination)

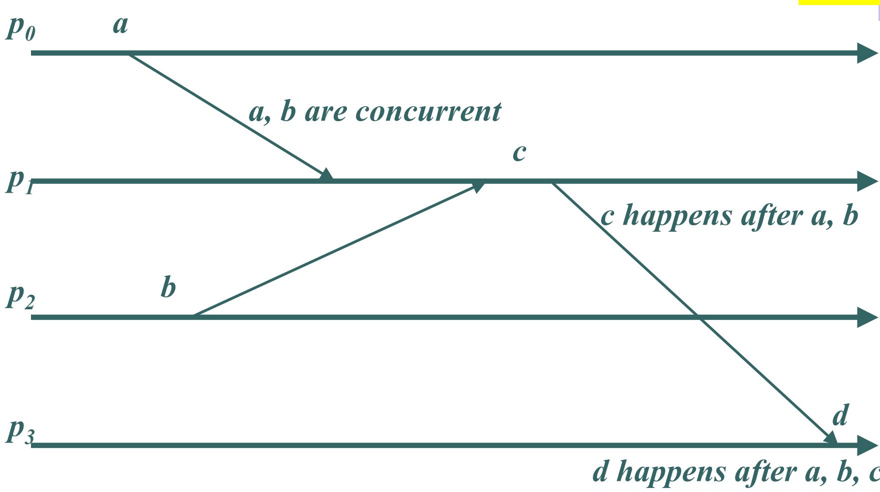
Logical notion of time

CS514

- Has no clock in the sense of "real-time"
- Focus is on definition of the "happens before" relationship: *"a happens before b" if:*
  - both occur at same place and *a* finished before *b* started, or
  - *a* is the send of message *m, b* is the delivery of *m*, or
  - *a* and *b* are linked by a chain of such events

Logical time as a time-space picture

CS514



Notation

CS514

- Use “arrow” to represent happens-before relation
- For previous slide:
  - $a \rightarrow c$, $b \rightarrow c$, $c \rightarrow d$
  - *hence*, $a \rightarrow d$, $b \rightarrow d$
  - *a, b are concurrent*
- Also called the “potential causality” relation

Logical clocks

CS514

- Proposed by Lamport to represent causal order
- Write: LT(e) to denote logical timestamp of an event e, LT(m) for a timestamp on a message, LT(p) for the timestamp associated with process p
- Algorithm ensures that if $a \rightarrow b$, then $LT(a) < LT(b)$

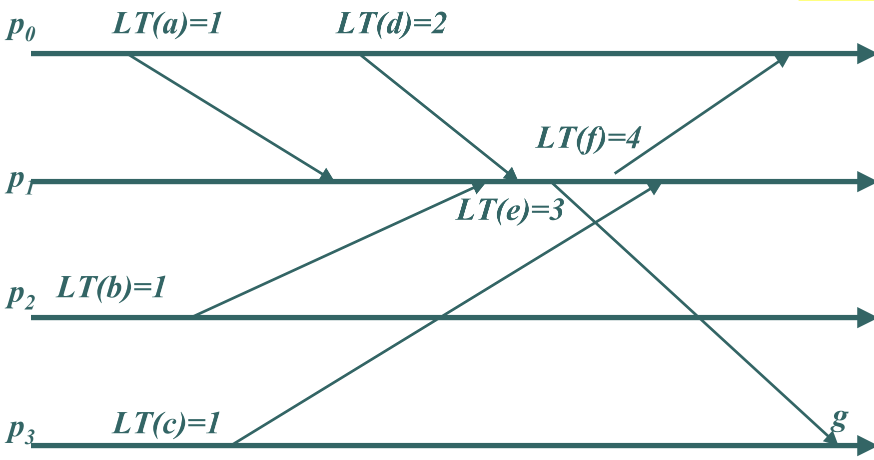
Algorithm

CS514

- Each process maintains a counter, *LT(p)*
- For each event other than message delivery: set *LT(p) = LT(p)+1*
- When sending message m, *LT(m) = LT(p)*
- When process q receives message m, set *LT(q) = max(LT(m), LT(q))+1*

Illustration of logical timestamps

CS514



Concurrent events

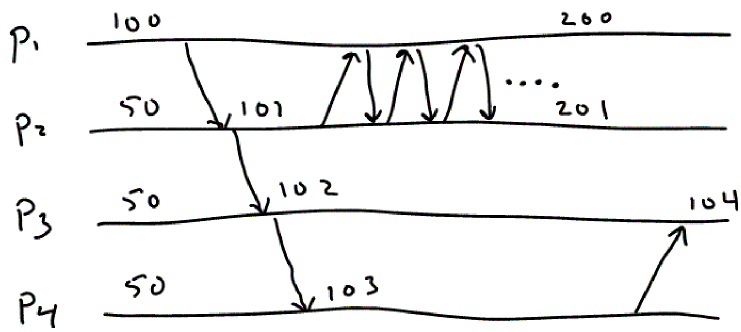
CS514

- If a, b are concurrent, LT(a) and LT(b) may have arbitrary values!
- Thus, logical time lets us determine that *a* potentially happened before *b*, but not that *a* definitely did so!
- Example: processes p and q never communicate. Both will have events 1, 2, ... but even if LT(e)<LT(e') e may not have happened before e'

Example . . .

CS514

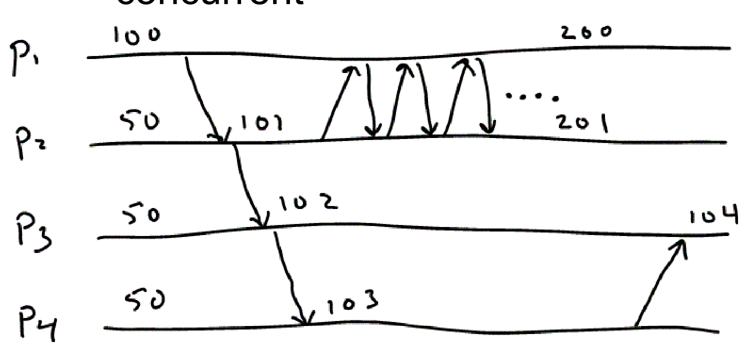
- Logical timestamps fall "out of sync" with lack of message "cross pollination"



Example . . .

CS514

- We can't compare Lamport LTs out-of-context and know if they are causal or concurrent



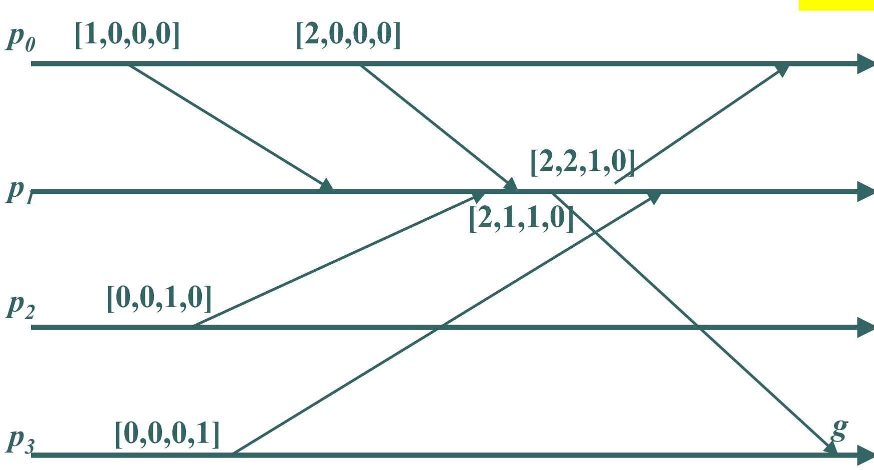
Vector timestamps

CS514

- Extend logical timestamps into a list of counters, one per process in the system
- Again, each process keeps its own copy
- Event e occurs at process p:
  p increments VT(p)[p]
  (p'th entry in its own vector clock)
- q receives a message from p:
  q sets VT(q)=max(VT(q),VT(p))
  (element-by-element)

Illustration of vector timestamps

CS514



Vector timestamps accurately represent the happens-before relationship!

CS514

- Define VT(e)<VT(e’) if,
  - for all i, VT(e)[i]≤VT(e’)[i], and
  - for some j, VT(e)[j]<VT(e’)[j]
- Example: if VT(e)=[2,1,1,0] and VT(e’)=[2,3,1,0] then VT(e)<VT(e’)
- Notice that not all VT’s are “comparable” under this rule: consider [4,0,0,0] and [0,0,0,4]

Vector timestamps accurately represent the happens-before relationship!

CS514

- Now can show that VT(e)<VT(e’) if and only if e $\rightarrow$ e’:
  - If e $\rightarrow$ e’, there exists a chain $e_0 \rightarrow e_1 \dots \rightarrow e_n$ on which vector timestamps increase “hop by hop”
  - If VT(e)<VT(e’) suffices to look at VT(e’)[proc(e)], where proc(e) is the place that e occurred. By definition, we know that VT(e’)[proc(e)] is at least as large as VT(e)[proc(e)], and by construction, this implies a chain of events from e to e’

Examples of VT's and happens-before

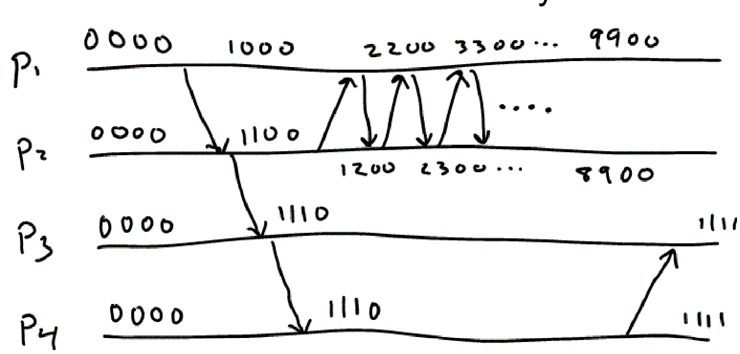
CS514

- Example: suppose that VT(e)=[2,1,0,1] and VT(e')=[2,3,0,1], so VT(e)<VT(e')
- How did e' "learn" about the 3 and the 1?
  - Either these events occurred at the same place as e', or
  - Some chain of send/receive events carried the values!
- If VT's are not comparable, the corresponding events are concurrent!

Same example . . .

CS514

- 1000 is causal to 1111 and 8900
- 1111 is not causal to 8900
- Now we can determine causality and concurrency!



Notice that vector timestamps require a static notion of system membership

CS514

- For vector to make sense, must agree on the number of entries
- But vector timestamps are useful within process groups because the groups synchronize on “views”
- Vector timestamp really looks like:
  - View-ID, [p1, p2, p3, . . . pN]

Vector Compression Tricks

CS514

- Size of the vector is the main overhead
- Subsequent timestamps in a “burst” of multicasts can be omitted
  - Receiver understands that the vector entry of the sender can be incremented
- Reset timestamp values with each view
- Sort vector so that “receive-only” processes are last
  - Truncate vector by eliminating trailing zeros
- Send the difference between current and last vector

What about "real-time" clocks?

CS514

- Accuracy of clock synchronization is ultimately limited by uncertainty in communication latencies
- These latencies are "large" compared with speed of modern processors (typical latency may be 35us to 500us, time for thousands of instructions)
- Limits use of real-time clocks to "coarse-grained" applications

What about GPS-based synchronization?

CS514

- Inexpensive GPS clocks accurate to ±1microsecond are available (<$500)
  - 2 microseconds is 1/15000th of a typical cross-country latency
  - But about the same latency as a small packet over a short gigabit link
- So increasingly can't be used to indicate message order in some environments
  - Also, GPS clock may fail . . .

Interpretations of temporal terms

CS514

- Understand now that "a happens before b" means that information can flow from a to b
- Understand that "a is concurrent with b" means that there is no information flow between a and b
- What about the notion of an "instant in time", over a set of processes?

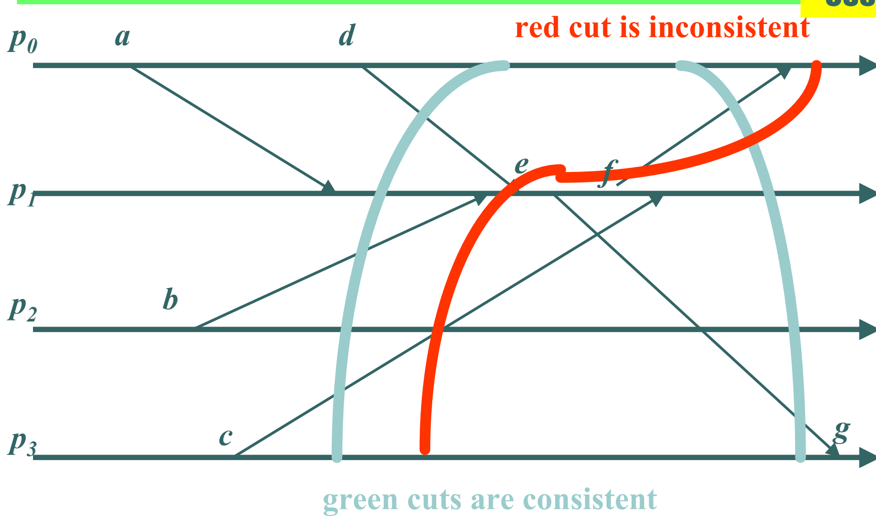
Chandy and Lamport: Consistent cuts

CS514

- Draw a line across a set of processes
- Line cuts each execution
- Consistent cut has property that the set of included events is closed under happens-before relation:
  - If the cut "includes" event b, and event a happens before b, then the cut also includes event a
  - In practice, this means that every "delivered" message was sent within the cut

Illustration of consistent cuts

CS514



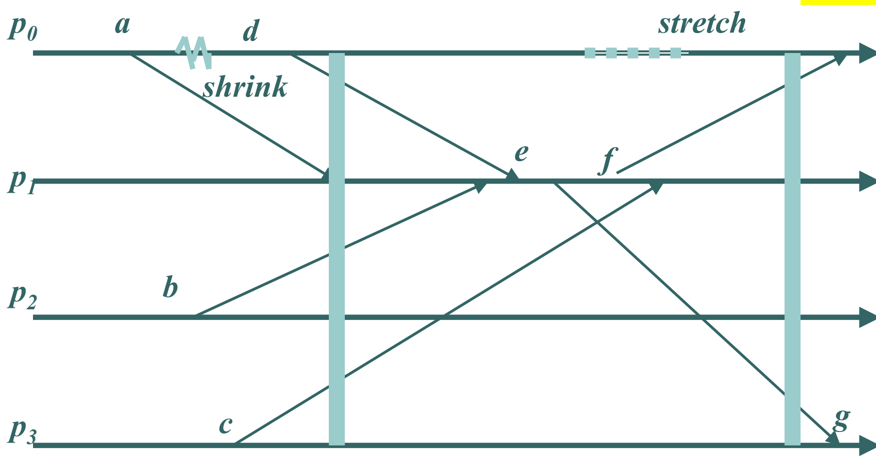
Intuition into consistent cuts

CS514

- A consistent cut is a state that *could* have arisen during execution, depending on how processes were scheduled
- An inconsistent cut could not have arisen during execution
- One way to see this: think of process timelines as rubber bands. Scheduler stretches or compresses time but can't deliver message before it was sent

Illustration of consistent cuts

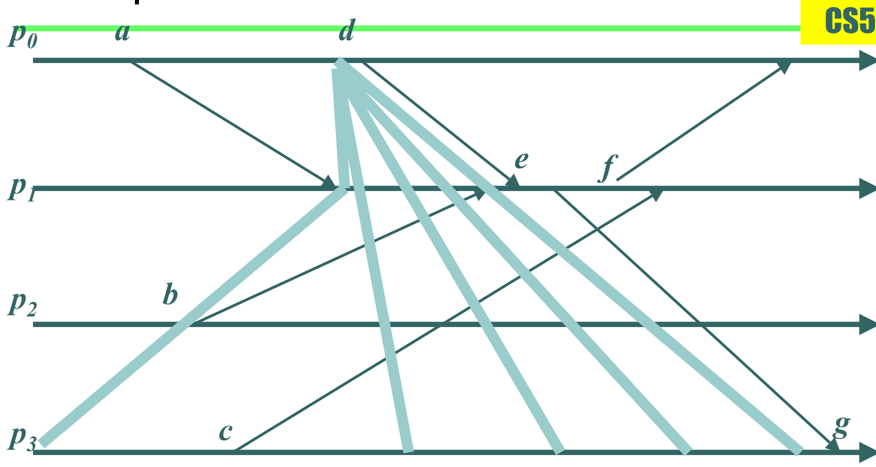
CS514



green cuts are consistent

There may be many consistent cuts through any point in the execution

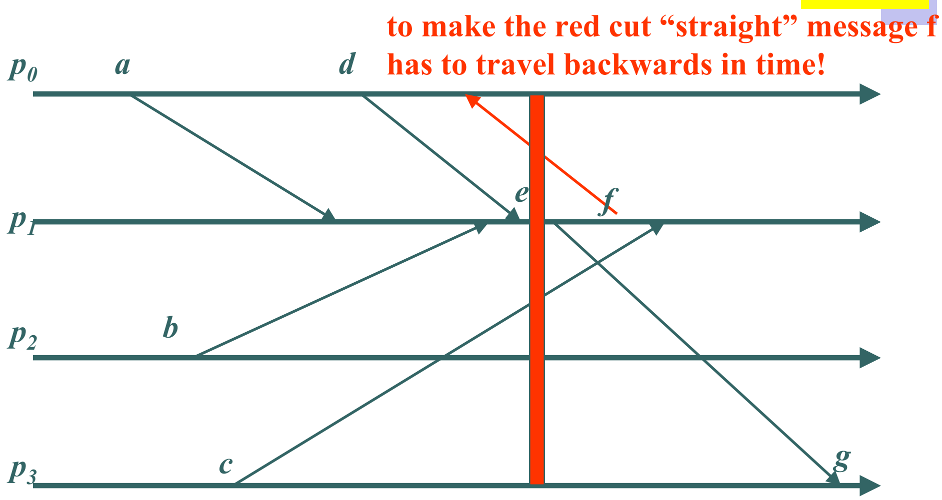
CS514



possible cuts define a range of potentially instantaneous system states

Illustration of consistent cuts

CS514



Moving from model to practice

CS514

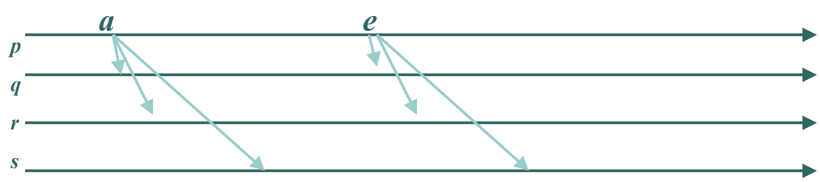
- Now we have basic elements of a model
  - It lets us talk about executions, gives meaning to temporal terms like “now”, “when”, “before”, “after”, “at the same time”, “concurrently”
- Move on to solve problems using this model and timestamps

Ordering properties: FIFO

CS514

- *Fifo* or *sender ordered* multicast: ***fbcast***

  Messages are delivered in the order they were sent (by any single sender)

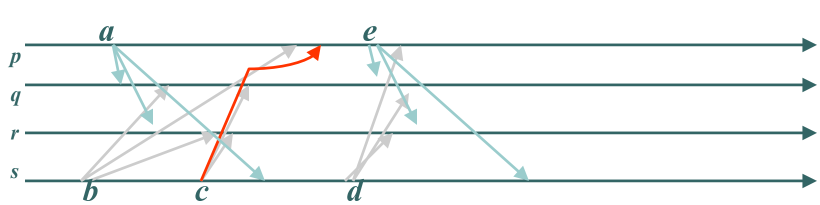


Ordering properties: FIFO

CS514

- *Fifo* or *sender ordered* multicast: ***fbcast***

  Messages are delivered in the order they were sent (by any single sender)



delivery of c to p is delayed until after b is delivered

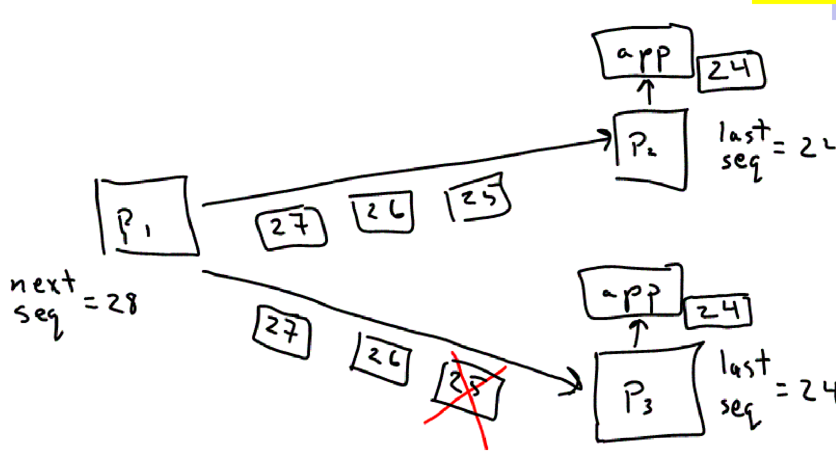
Implementing FIFO order

CS514

- Basic reliable multicast algorithm has this property
  - Without failures all we need is to run it on FIFO channels (like TCP, except "wired" to our GMS
- Multithreaded applications: must carefully use locking or order can be lost as soon as delivery occurs!

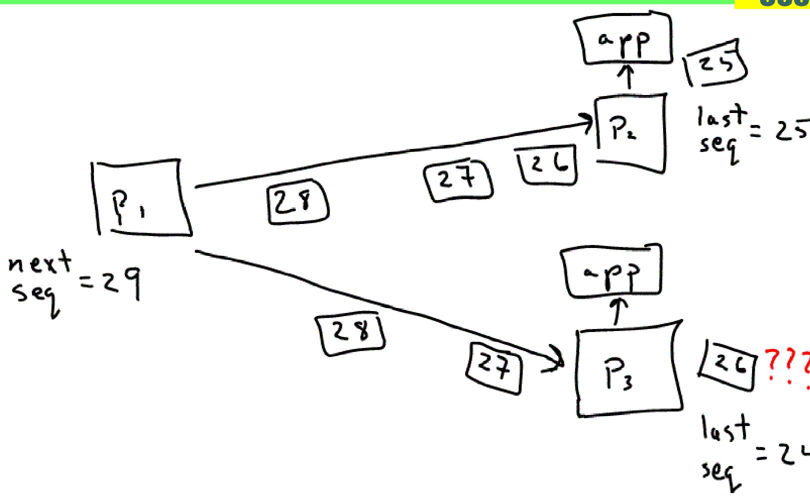
Implementing FIFO order: Sender sequence numbers

CS514



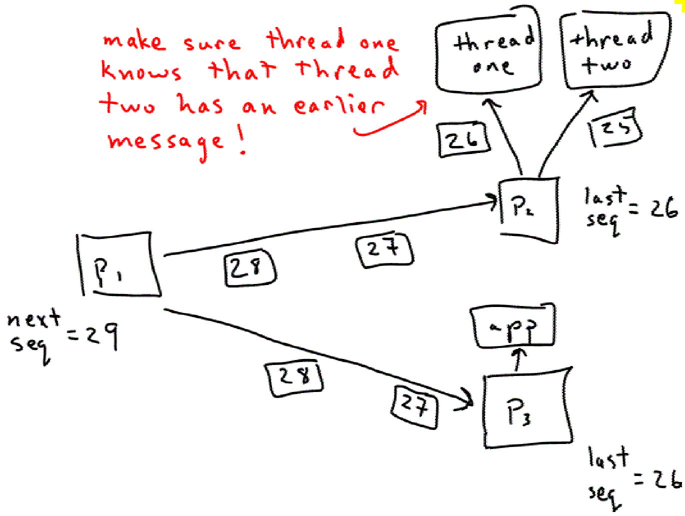
Immediate in-sequence message delivery

CS514



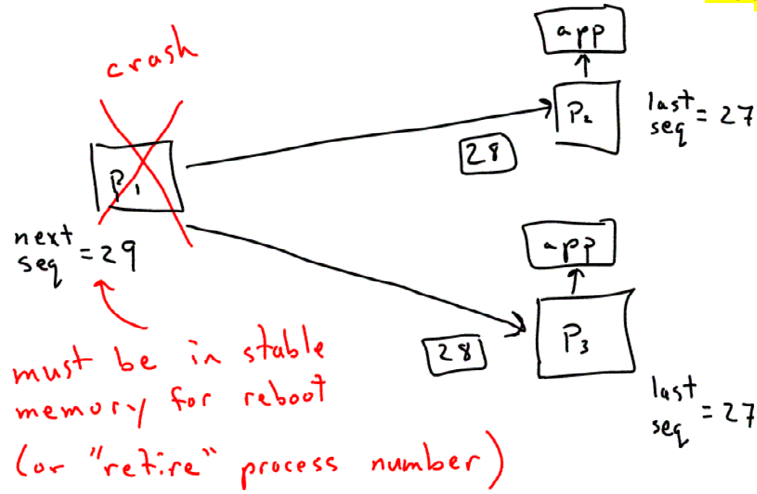
Careful with thread locking

CS514



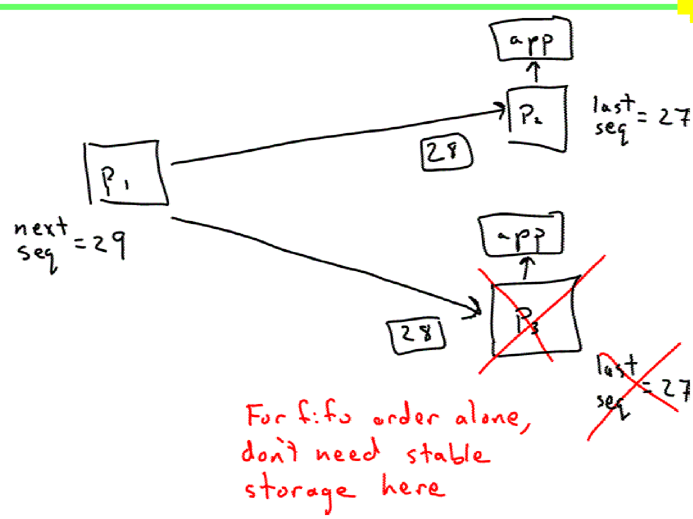
Sender crash

CS514



Receiver crash

CS514

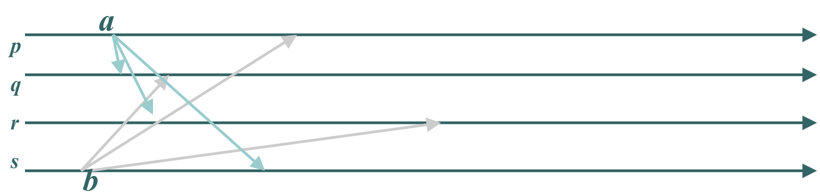


Ordering properties: Causal

CS514

- *Causal* or *happens-before* ordering: ***cbcast***

  If send(a) → send(b) then deliver(a) occurs before deliver(b) at common destinations

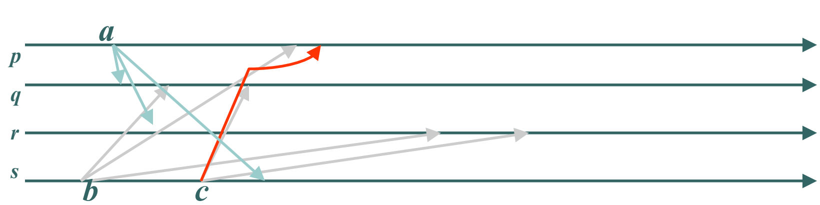


Ordering properties: Causal

CS514

- *Causal* or *happens-before* ordering: ***cbcast***

  If send(a) → send(b) then deliver(a) occurs before deliver(b) at common destinations



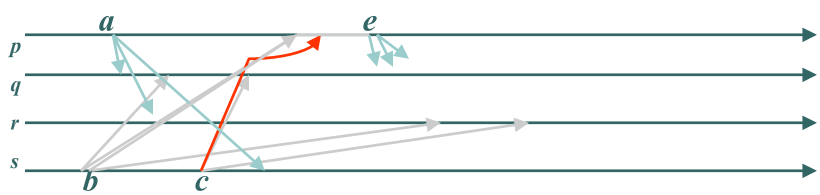
delivery of c to p is delayed until after b is delivered

Ordering properties: Causal

CS514

- *Causal* or *happens-before* ordering: ***cbcast***

  If send(a) → send(b) then deliver(a) occurs before deliver(b) at common destinations



delivery of c to p is delayed until after b is delivered

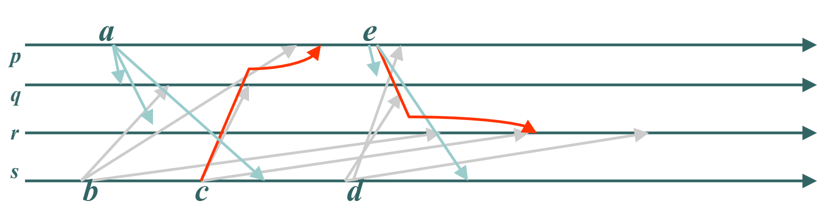
e is sent (causally) after b

Ordering properties: Causal

CS514

- *Causal* or *happens-before* ordering: ***cbcast***

  If send(a) → send(b) then deliver(a) occurs before deliver(b) at common destinations



delivery of c to p is delayed until after b is delivered

delivery of e to r is delayed until after b&c are delivered

Implementing causal order

CS514

- Start with a FIFO multicast
- Frank Schmuck showed that we can always strengthen this into a causal multicast by adding vector time (no additional messages needed)
  - If group membership were static this is easily done, small overhead
  - With dynamic membership, at least abstractly, we need to identify each VT index with the corresponding process, which seems to double the size

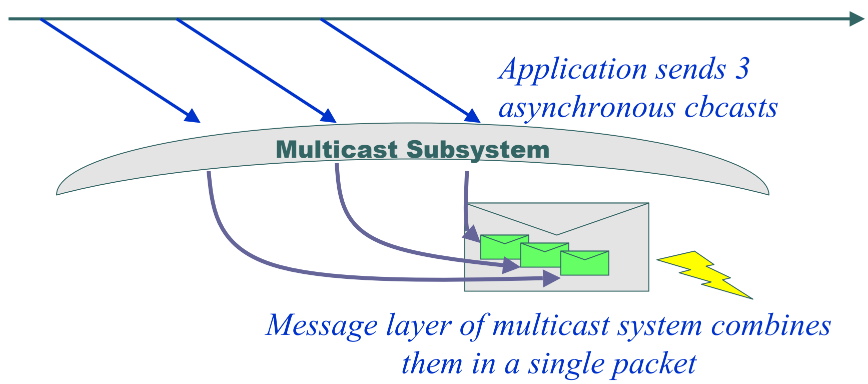
Insights about c/fbcast

CS514

- These two primitives are *asynchronous*:
  - Sender doesn't get blocked and can deliver a copy to itself without "stopping" to learn a safe delivery order
  - If used this way, the multicast can seem to sit in the output buffers a long time, leading to surprising behavior
  - But this also gives the system a chance to concatenate multiple small messages into one larger one

Concatenation

CS514



State Machine Concept

CS514

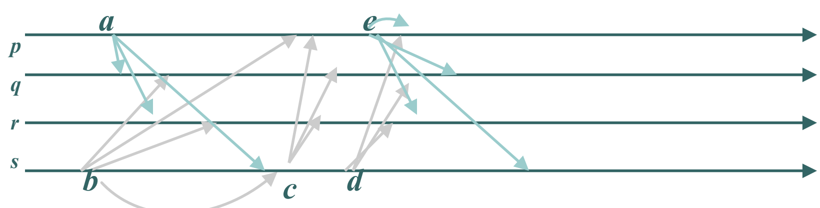
- Sometimes, we want a replicated object or service that advances through a series of "state machine transitions"
  - Every process stays exactly in sync
  - Used for Byzantine failure modes
- Clearly will need all copies to make the same transitions
- Leads to a need for totally ordered multicast

Ordering properties: Total

CS514

- *Total* or *locally total* multicast: ***abcast***

  Messages are delivered in same order to all recipients (including the sender)



all deliver a, b, c, d, then e

Basic Idea

CS514

- With causal ordering, causal messages are received by all group members in the same order
- But the order of concurrent messages is not defined
- Total ordering arbitrarily defines an “agreed” ordering of concurrent messages
  - E.g. LT+process_id

Two basic approaches

CS514

- Token ring
  - A token containing the global sequence number is passed around
  - Spread, Totem, Transis
- Sequencer
  - A single node sequences all messages
    - ISIS
- Either way, the trick is to only give one node control at a time

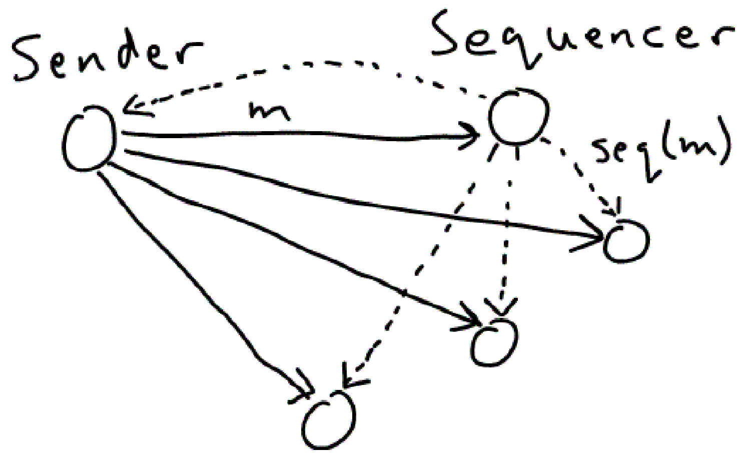
Sequencer

CS514

- All multicast messages passed through a sequencer
- Sequencer assigns a total sequence number, informs all nodes
- Either:
  - The sender multicasts the message
    - Followed by sequence multicast from sequencer
  - The sequencer multicasts the message

Sequencer

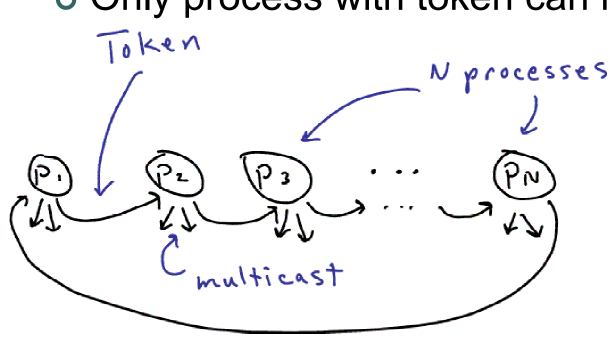
CS514



Token ring approach

CS514

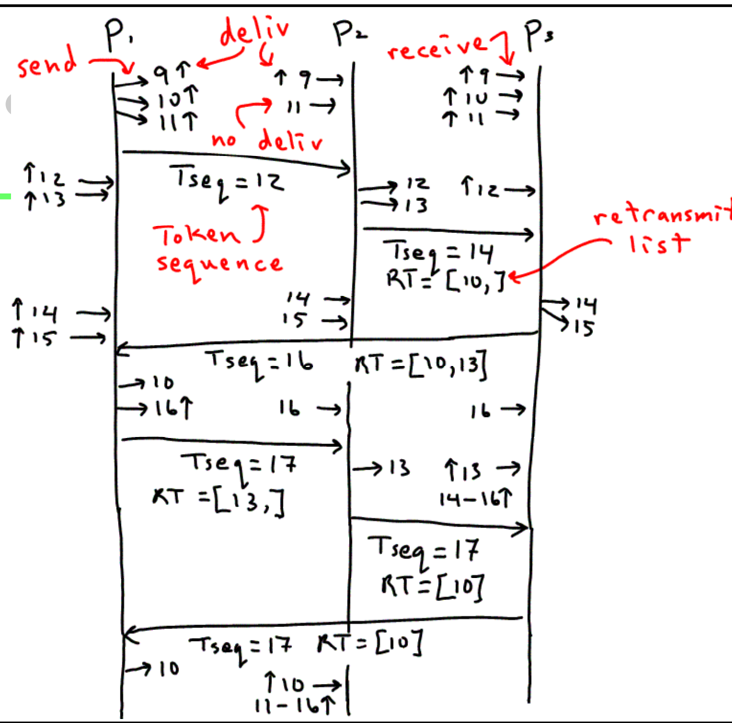
- Token with total sequence number passed around processes
- Only process with token can multicast



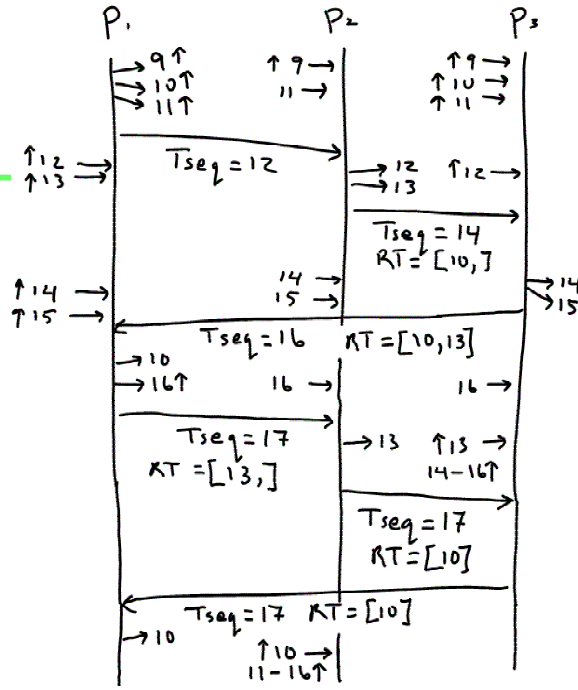
Token has other uses

CS514

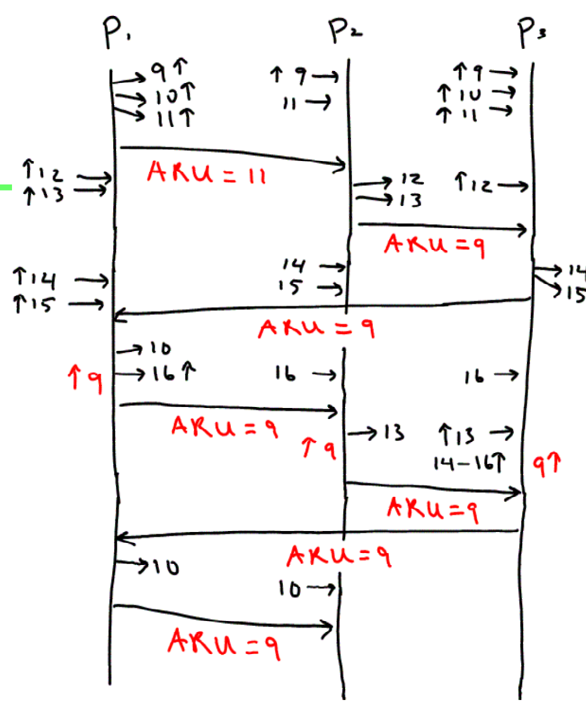
- Used for message reliability
  - Token has a “retransmit list”
- Used for “Safe” message delivery
  - Message not delivered until all processes have received it
  - Ken calls “dynamically uniform”
  - Big performance penalty!
- Used for flow control
  - Token says how many messages have been sent

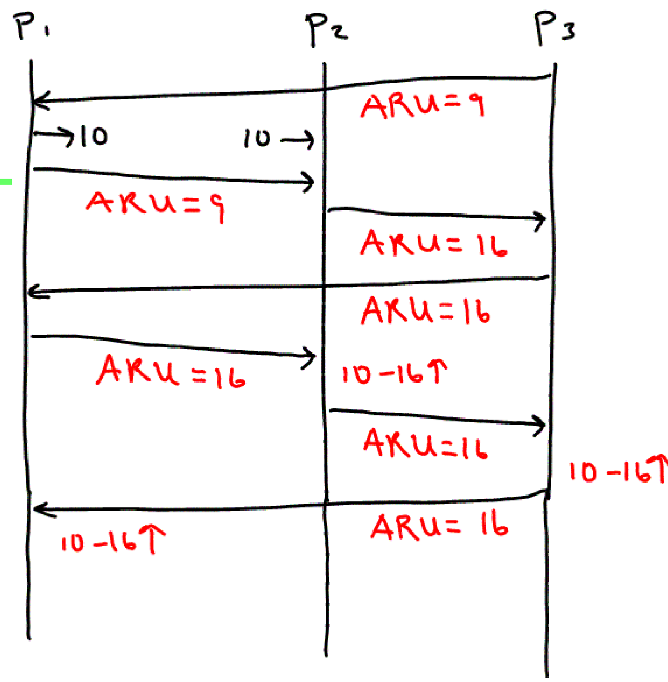


CS514


P1
P2
P3
9↑
10↑
11↑
↑ 9
11
↑9
↑10
↑11
↑12
↑13
Tseq = 12
12
13
↑12
Tseq = 14
RT = [10,]
↑14
↑15
14
15
14
15
Tseq = 16
RT = [10,13]
10
16↑
16
16
Tseq = 17
RT = [13,]
13
↑13
14–16↑
Tseq = 17
RT = [10]
Tseq = 17
RT = [10]
10
↑10
11–16↑

CS514

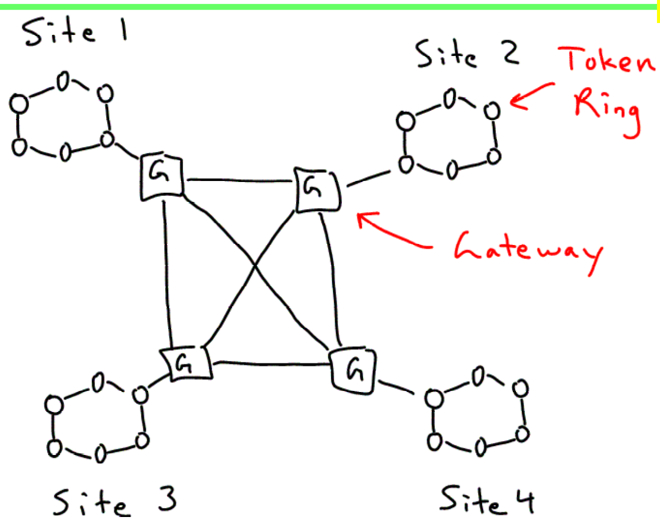

P1
P2
P3
9↑
10↑
11↑
↑ 9
11
↑9
↑10
↑11
↑12
↑13
AKU = 11
12
13
↑12
AKU = 9
↑14
↑15
14
15
14
15
AKU = 9
10
↑9
16↑
16
16
AKU = 9
↑9
13
↑13
14–16↑
9↑
AKU = 9
AKU = 9
10
10
AKU = 9



CS514

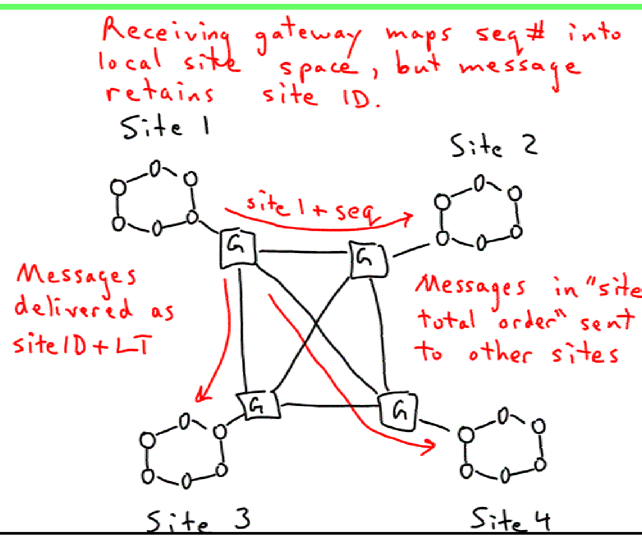
Wide area total ordering (Totem)

CS514



Wide area total ordering (Totem)

CS514



Wide area total ordering (Totem)

CS514