CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Recap… Consistent cuts

- On Monday we saw that simply gathering the state of a system isn't enough
- Often the "state" includes tricky relationships
- Consistent cuts are a way of collecting state that "could" have arisen *concurrently in real-time*

What time is it?

- In distributed system we need practical ways to deal with time
  - E.g. we may need to agree that update A occurred before update B
  - Or offer a "lease" on a resource that expires at time 10:10.0150
  - Or *guarantee* that a time critical event will reach all interested parties within 100ms

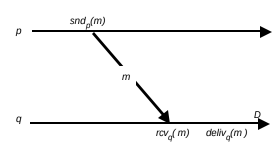
But what does time "mean"?

- Time on a global clock?
  - E.g. with GPS receiver
- … or on a machine's local clock
  - But was it set accurately?
  - And could it drift, e.g. run fast or slow?
  - What about faults, like stuck bits?
- … or could try to agree on time

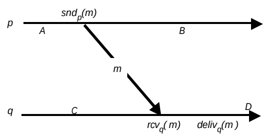
Lamport's approach

- Leslie Lamport suggested that we should reduce time to its basics
  - Time lets a system ask "Which came first: event A or event B?"
  - In effect: time is a means of labeling events so that…
    - If A happened before B, TIME(A) < TIME(B)
    - If TIME(A) < TIME(B), A happened before B

Drawing time-line pictures:

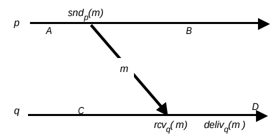


Drawing time-line pictures:



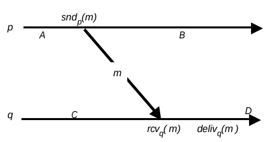
- A, B, C and D are “events”.
  - Could be anything meaningful to the application
  - So are snd(m) and rcv(m) and deliv(m)
- What ordering claims are meaningful?

Drawing time-line pictures:



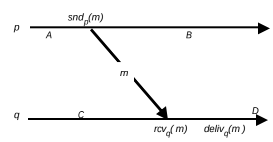
- A happens before B, and C before D
  - “Local ordering” at a single process
  - Write $A \xrightarrow{p} B$ and $C \xrightarrow{q} D$

Drawing time-line pictures:



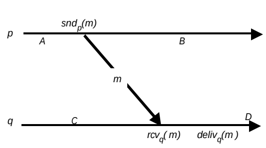
- $snd_p(m)$ also happens before $rcv_q(m)$
  - “Distributed ordering” introduced by a message
  - Write $snd_p(m) \xrightarrow{M} rcv_q(m)$

Drawing time-line pictures:



- A happens before D
  - Transitivity: A happens before $snd_p(m)$, which happens before $rcv_q(m)$, which happens before D

Drawing time-line pictures:



- B and D are concurrent
  - Looks like B happens first, but D has no way to know. No information flowed…

Happens before “relation”

- We’ll say that “A happens before B”, written A→B, if
  1. $A \rightarrow^{P} B$ according to the local ordering, or
  2. A is a *snd* and B is a *rcv* and $A \rightarrow^{M} B$, *or*
  3. A and B are related under the transitive closure of rules (1) and (2)
- So far, this is just a mathematical notation, not a “systems tool”

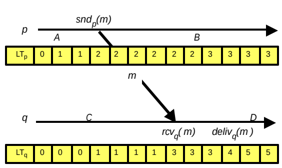
Logical clocks

- A simple tool that can capture parts of the happens before relation
- First version: uses just a single integer
  - Designed for big (64-bit or more) counters
  - Each process *p* maintains LT_p, a local counter
  - A message *m* will carry LT_m

Rules for managing logical clocks

- When an event happens at a process *p* it increments LT_p.
  - Any event that matters to *p*
  - Normally, also *snd* and *rcv* events (since we want receive to occur "after" the matching send)
- When p sends *m*, set
  - $LT_m = LT_p$
- When q receives *m*, set
  - $LT_q = \max(LT_q, LT_m)+1$

Time-line with LT annotations



- $LT(A) = 1$, $LT(snd_p(m)) = 2$, $LT(m) = 2$
- $LT(rcv_q(m))=\max(1,2)+1=3$, etc…

Logical clocks

- If A happens before B, $A \rightarrow B$, then $LT(A)<LT(B)$
- But converse might not be true:
  - If $LT(A)<LT(B)$ can't be sure that $A \rightarrow B$
  - This is because processes that don't communicate still assign timestamps and hence events will "seem" to have an order

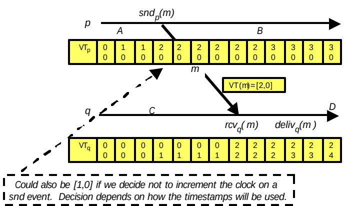
Can we do better?

- One option is to use *vector* clocks
- Here we treat timestamps as a list
  - One counter for each process
- Rules for managing vector times differ from what did with logical clocks

Vector clocks

- Clock is a vector: e.g. $VT(A)=[1, 0]$
  - We'll just assign p index 0 and q index 1
  - Vector clocks require either agreement on the numbering, or that the actual process id's be included with the vector
- Rules for managing vector clock
  - When event happens at p, increment $VT_p[index_p]$
    - Normally, also increment for snd and rcv events
  - When sending a message, set $VT(m)=VT_p$
  - When receiving, set $VT_q=\max(VT_q, VT(m))$

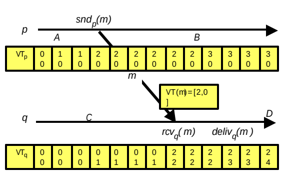
Time-line with VT annotations



Rules for comparison of VTs

- We'll say that $VT_A = VT_B$ if
  - $\forall_i$, $VT_A[i] = VT_B[i]$
- And we'll say that $VT_A < VT_B$ if
  - $VT_A = VT_B$ but VT_A ? VT_B
  - That is, for some i, $VT_A[i] < VT_B[i]$
- Examples?
  - [2,4] = [2,4]
  - [1,3] < [7,3]
  - [1,3] is "incomparable" to [3,1]

Time-line with VT annotations



- VT(A)=[1,0]. VT(D)=[2,4]. So VT(A)<VT(D)
- VT(B)=[3,0]. So VT(B) and VT(D) are incomparable

Vector time and happens before

- If A→B, then VT(A)<VT(B)
  - Write a chain of events from A to B
  - Step by step the vector clocks get larger
- If VT(A)<VT(B) then A→B
  - Two cases: if A and B both happen at same process p, trivial
  - If A happens at p and B at q, can trace the path back by which q "learned" $VT_A[p]$
- Otherwise A and B happened concurrently

Consistent cuts

- If we had time, we could revisit these using logical and vector clocks
- In fact there are algorithms that find a consistent cut by
  - Implementing some form of clock
  - Asking everyone to record their state at time now+δ (for some large δ)
- And this can be made to work well...

Replication

- Another use of time arises when we talk about replicating data in distributed systems
- The reason is that:
  - We replicate data by multicasting updates over a set of replicas
  - They need to apply these updates in the same order
  - And order is a temporal notion

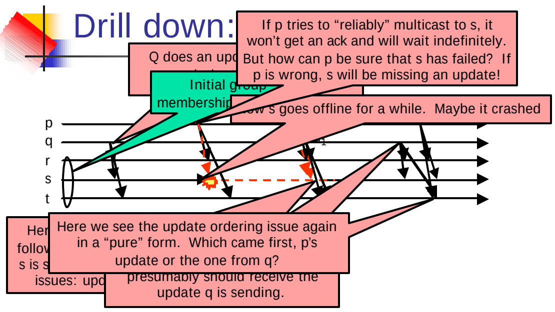
... and replication is powerful!

- Replicate data or a service for high availability
- Replicate data so that group members can share loads and improve scalability
- Replicate locking or synchronization state
- Replicate membership information in a data center so that we can route requests
- Replicate management information or parameters to tune performance

Let’s look at time vis-à-vis updates

- Maybe logical notions of time can help us understand when one update comes before another update
- Then we can think about building replicated update algorithms that are optimized to run as fast as possible while preserving the needed ordering

Drill down:



Questions to ask about order

- Who should receive an update?
- What update ordering to use?
- How expensive is the ordering property?

Questions to ask about order

- Delivery order for concurrent updates
  - Issue is more subtle than it looks!
  - We can fix a system-wide order, but...
    - Sometimes nobody notices out of order delivery
    - System-wide ordering is expensive
    - If we care about speed we may need to look closely at cost of ordering

Ordering example

- System replicates variables x, y
  - Process p sends “x = x/2”
  - Process q sends “x = 83”
  - Process r sends “y = 17”
  - Process s sends “z = x/y”
- To what degree is ordering needed?

Ordering example

- x=x/2   x=83
- These clearly “conflict”
  - If we execute x=x/2 first, then x=83, x will have value 83.
  - In opposite order, x is left equal to 41.5

Ordering example

- x=x/2   y=17
- These don’t seem to conflict
  - After the fact, nobody can tell what order they were performed in

Ordering example

- z=x/y
- This conflicts with updates to x, updates to y and with other updates to z

Commutativity

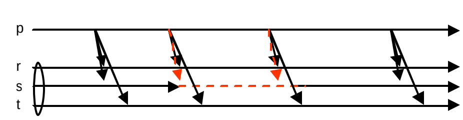
- We say that operations “commute” if the final effect on some system is the same even if the order of those operations is swapped
- In general, a system worried about ordering concurrent events need not worry if the events commute

Single updater

- In many systems, there is only one process that can update a given type of data
  - For example, the variable might be “sensor values” for a temperature sensor
  - Only the process monitoring the sensor does updates, although perhaps many processes want to read the data and we replicate it to exploit parallelism
  - Here the only “ordering” that matters is the FIFO ordering of the updates emitted by that process

Single updater

- If p is the only update source, the need is a bit like the TCP “fifo” ordering



Mutual exclusion

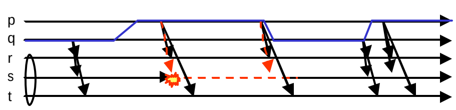
- Another important case we'll study closely
- Arises in systems that use locks to control access to shared data
  - This is very common, for example in "transactional" systems (we'll discuss them next week)
  - Very often without locks, a system rapidly becomes corrupted

Mutual exclusion

- Suppose that before performing conflicting operations, processes must lock the variables
- This means that there will never be any true concurrency
- And it simplifies our ordering requirement

Mutual exclusion

- Dark blue when holding the lock



- How is this case similar to "FIFO" with one sender? How does it differ?

Mutual exclusion

- Are these updates in "FIFO" order?
  - No, the sender isn't always the same
  - But yes in the sense that there is a unique path through the system (corresponding to the lock) and the updates are ordered along that path
- Here updates are ordered by Lamport's happened before relation: $\rightarrow$

Types of ordering we've seen

- *cheapest* — Deliver updates in an order matching the FIFO order in which they were sent
- *Still cheap* — Deliver updates in an order matching the $\rightarrow$ order in which they were sent
- *More costly* — For conflicting concurrent updates, pick an order and use that order at all replicas
- *Most costly* — Deliver an update to all members of a group according to "membership view" determined by ordering updates wrt view changes

Types of ordering we've seen

- ***fbcast*** — Deliver updates in an order matching the FIFO order in which they were sent
- ***cbcast*** — Deliver updates in an order matching the $\rightarrow$ order in which they were sent
- ***abcast*** — For conflicting concurrent updates, pick an order and use that order at all replicas
- ***gbcast*** — Deliver an update to all members of a group according to "membership view" determined by ordering updates wrt view changes

Recommended readings

- In the textbook, we're at the beginning of Part III (Chapter 14)
- We'll build up the "virtual synchrony" replication model in the next lecture and see how it can be built with 2PC, 3PC, consistent cuts and ordering