CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Quorum replication

- We developed a whole architecture based on our four-step recipe
- But there is a second major approach that also yields a complete group communication framework and solutions
  - Based on “quorum” read and write operations
  - Omits notion of process group views

Today’s topic

- Quorum methods from a mile high
  - Don’t have time to be equally detailed
- We’ll explore
  - How the basic read/update protocol works
  - Failure considerations
  - State machine replication (a form of lock-step replication for deterministic objects)
  - Performance issues

A peek at the conclusion

- These methods are
  - Widely known and closely tied to consensus
  - Perhaps, easier to implement
- But they have serious drawbacks:
  - Need deterministic components
  - Are *drastically* slower (10s-100s of events/second)
- Big win?
  - Recent systems combine quorums with Byzantine Agreement for ultra-sensitive databases

---

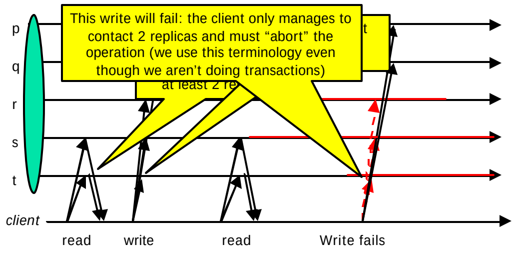
Static membership

- Subsets of a *known* set of processes
  - E.g. a cluster of five machines, each running replica of a database server
  - Machines can crash or recover but don’t depart permanently and new ones don’t join “out of the blue”
- In practice the dynamic membership systems can easily be made to work this way… but usually aren’t

---

Static membership example

$Q_{read} = 2, Q_{write} = 4$



This write will fail: the client only manages to contact 2 replicas and must “abort” the operation (we use this terminology even though we aren’t doing transactions)

Quorums

- Must satisfy two basic rules
  1. A quorum *read* should “intersect” any prior quorum *write* at >= 1 processes
  2. A quorum *write* should also intersect any other quorum write
- So, in a group of size N:
  1. $Q_r + Q_w > N$, and
  2. $Q_w + Q_w > N$

Versions of replicated data

- Replicated data items have “versions”, and these are numbered
  - I.e. can’t just say “$X_p=3$”. Instead say that X_p has timestamp [7,q] and value 3
  - Timestamp must increase monotonically and includes a process id to break ties
  - This is NOT the pid of the update source… we’ll see where it comes from

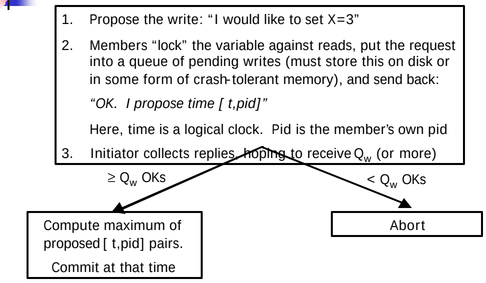
Doing a read is easy

- Send RPCs until Q_r processes reply
- Then use the value with the largest timestamp
  - Break ties by looking at the pid
  - For example
    - [6,x] < [9,a] (first look at the “time”)
    - [7,p] < [7,q] (but use pid as a tie-breaker)
- *Even if a process owns a replica,* it can’t just trust it’s own data. Every “read access” must collect Q_r values first…

Doing a write is trickier

- First, we can’t support incremental updates (x=x+1), since no process can “trust” its own replica.
  - Such updates require a read followed by a write.
- When we initiate the write, we don’t know if we’ll succeed in updating a quorum of processes
  - wE can’t update just some subset; that could confuse a reader
  - Hence need to use a commit protocol
- Moreover, must implement a mechanism to determine the version number as part of the protocol. We’ll use a form of voting

The sequence of events



1. Propose the write: “I would like to set X=3”
2. Members “lock” the variable against reads, put the request into a queue of pending writes (must store this on disk or in some form of crash-tolerant memory), and send back:

   “OK. I propose time [t,pid]”

   Here, time is a logical clock. Pid is the member’s own pid
3. Initiator collects replies, hoping to receive Q_w (or more)

$\geq Q_w$ OKs

Compute maximum of proposed [t,pid] pairs.

Commit at that time

$< Q_w$ OKs

Abort

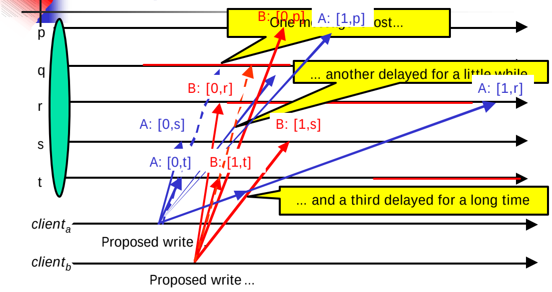
Which votes got counted?

- It turns out that we also need to know which votes were “counted”
  - E.g. suppose there are five group members, A…E and they vote:
    - {[17,A] [19,B] [20,C] [200,D] [21,E]}
  - But somehow the vote from D didn’t get through and the maximum is picked as [21,E]
  - We’ll need to also remember that the votes used to make this decision were from {A,B,C,E}

What's with the [t,pid] stuff?

- Lamport's suggestion: use logical clocks
  - Each process receives an update message
  - Places it in an ordered queue
  - And responds with a proposed time: [t,pid] using *its own process id* for the time
- The update source takes the maximum
  - Commit message says "commit at [t,pid]"
  - Group members who's votes were considered deliver committed updates in timestamp order
  - Group members who votes were not considered discard the update and don't do it, at all.

Example



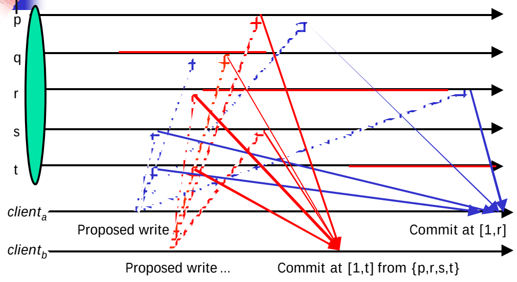
Timestamps seen by the clients?

- A sees (in order):
  - [0,p], [0,s], [1,p], [1,r]
  - This is a write quorum (Q_w=4)
  - A picks [1,r] from {p,r,s} as the largest "time"
- B sees
  - [0,r], [0,p], [1,t], [1,s]
  - B picks [1,t] from {p,r,s,t} as the largest time.
  - Note that [1,r] < [1,t], so A goes first

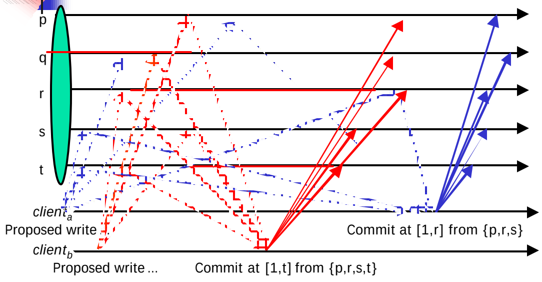
Where are the updates?

- Each member has a queue of pending, uncommitted updates
  - Even if a member crashes and restarts, it remembers this pending queue
- Example: at process p the queue has
  - {B: [0,p]}; {A: [1,p]}
  - Neither can be delivered (acted upon) since neither time is committed yet
  - Right now, process p can only respond to reads using the *old* value of the variable!

Example



Example



When are updates performed?

- In this example, A is supposed to go before B but processes learn commit time for B *first*
- Look at two cases
  - Pending queue at process P was
    - {B: [0,p]}; {A: [1,p]}
  - Pending queue at process T was
    - {A: [0,t]}; {B: [1,t]}
- Now they learn commit time for B: [1,t]
  - A reorders its queue: {A: [1,p]}, ***{B: [1,t]}***
  - B just notes the time: {A: [0,t]}; ***{B: [1,t]}***

When are updates performed?

- After they learn commit time for B: [1,t]
  - A reorders its queue: {A: [1,p]}, ***{B: [1,t]}***
  - B just notes the time: {A: [0,t]}; ***{B: [1,t]}***
- Now they learn commit time for A: [1,r]
  - A notes the time: ***{A: [1,r]},{B: [1,t]}***
  - B just notes the time: ***{A: [1,r]};{B: [1,t]}***
- ... So both deliver committed messages from the front of their respective queues, and use the same update ordering

What if "my vote wasn't used?"

- A process that had a pending update but discovers it wasn't used when computing the maximum discards the pending update request *even though it committed.*
  - Issue is that perhaps this vote should have been the largest one…
  - Discarding the request won't hurt: this replica will lag the others, but a quorum read would always "see" one of the updated copies!

Recovery from a crash

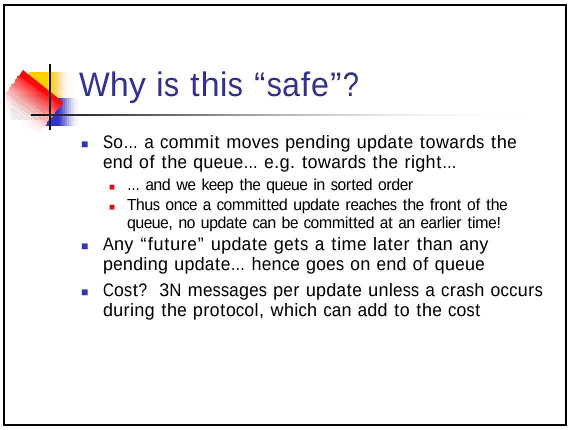
- So… to recover from a crash, a replica
  - First recovers its queue of pending updates
  - Next must learn the outcome of the operation
    - May need to contact Q_r other replicas
  - Checks to see if the operation committed and if its own vote counted
    - If so, applies the pending update
    - If not, discards the pending update

Read requests received when updates are pending wait…

- Suppose someone does a read while there are pending, uncommitted updates
  - These must wait until those commit, abort, or are discarded
  - Otherwise a process could do an update, then a read, and yet might not see its own updated value

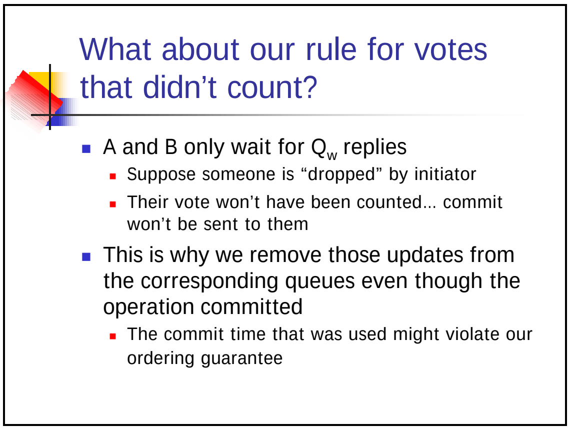
Why is this "safe"?

- Notice that a commit can only move a pending update to a *later* time!
  - This is why we discard a pending update if the vote wasn't counted when computing the commit time
  - Otherwise that "ignored" vote might have been the maximum value and could have determined the event ordering… by discarding it we end up with an inconsistent replica, but that doesn't matter, since to do a read, we always look at Q_r replicas, and hence can tolerate an inconsistent copy
  - This is also why we can't support incremental operations ("add six to x")



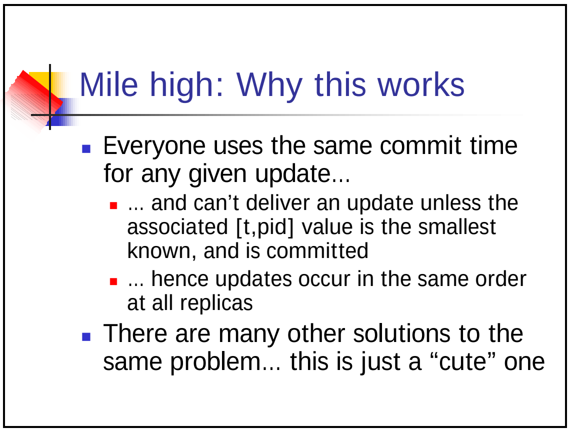
Why is this “safe”?

- So… a commit moves pending update towards the end of the queue… e.g. towards the right…
  - … and we keep the queue in sorted order
  - Thus once a committed update reaches the front of the queue, no update can be committed at an earlier time!
- Any “future” update gets a time later than any pending update… hence goes on end of queue
- Cost? 3N messages per update unless a crash occurs during the protocol, which can add to the cost



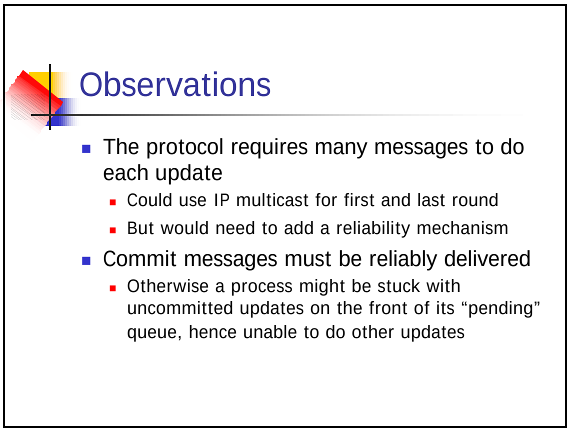
What about our rule for votes that didn’t count?

- A and B only wait for Q_w replies
  - Suppose someone is “dropped” by initiator
  - Their vote won’t have been counted… commit won’t be sent to them
- This is why we remove those updates from the corresponding queues even though the operation committed
  - The commit time that was used might violate our ordering guarantee



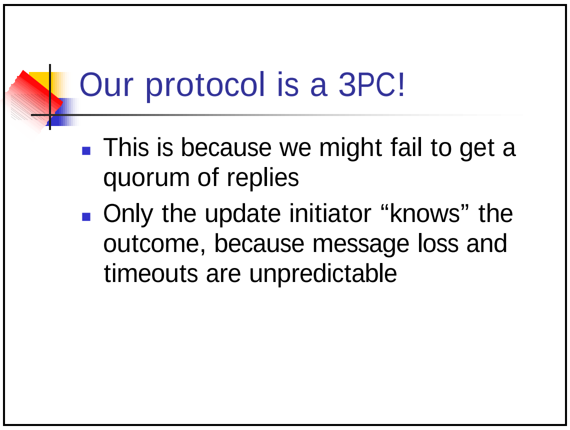
Mile high: Why this works

- Everyone uses the same commit time for any given update…
  - … and can’t deliver an update unless the associated [t,pid] value is the smallest known, and is committed
  - … hence updates occur in the same order at all replicas
- There are many other solutions to the same problem… this is just a “cute” one



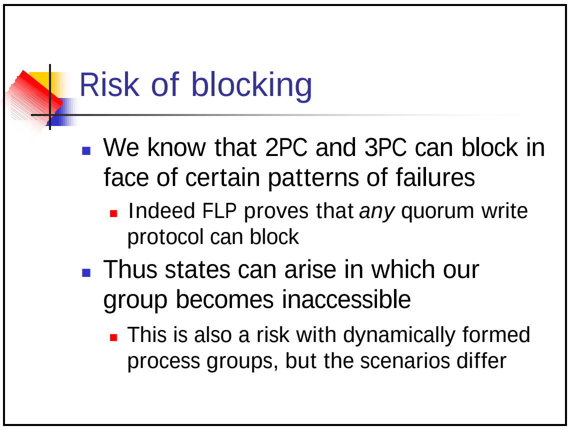
Observations

- The protocol requires many messages to do each update
  - Could use IP multicast for first and last round
  - But would need to add a reliability mechanism
- Commit messages must be reliably delivered
  - Otherwise a process might be stuck with uncommitted updates on the front of its “pending” queue, hence unable to do other updates



Our protocol is a 3PC!

- This is because we might fail to get a quorum of replies
- Only the update initiator “knows” the outcome, because message loss and timeouts are unpredictable



Risk of blocking

- We know that 2PC and 3PC can block in face of certain patterns of failures
  - Indeed FLP proves that *any* quorum write protocol can block
- Thus states can arise in which our group becomes inaccessible
  - This is also a risk with dynamically formed process groups, but the scenarios differ

Performance implications?

- This is a *much* slower protocol than the virtual synchrony solutions
  - With virtual synchrony we can read any group member's data…
    - But lacks dynamic uniformity (safety) unless we ask for it
  - Must read Q_r copies (at least 2)
    - A member can't even "trust" its own replica!
    - But has the dynamic uniformity property
  - And a write is a 3PC touching Q_w copies
    - An incremental update needs 4 phases…

Performance implications?

- In experiments
  - Virtual synchrony, using small asynchronous messages in small groups and packing them, reached 100,000's of multicasts per second
  - Quorum updates run at 10s-100s in same setup: 3 orders of magnitude slower

So why even consider them?

- Lamport uses this method in his Paxos system, which implements lock-step replication of components
  - Called the "State Machine" approach
  - Can be shown to achieve consensus as defined in FLP, including safety property
- Castro and Liskov use Byzantine Agreement for even greater robustness

Byzantine Quorums

- This is an extreme form of replication
  - Robust against failures
  - Tolerates Byzantine behavior by members
- Increasingly seen as a good choice when compromises are simply unacceptable

Typical approach?

- These use a quorum size of v N
  - Think of the group as if it was arranged as a square
  - Any "column" is a read quorum
  - Any "row" is a write quorum
- Then use Byzantine Agreement (not 3PC) to perform the updates or to do the read

Costs? Benefits?

- The costs are *very high*
  - Byzantine protocol is expensive
  - And now we're accessing vN members
- But the benefits are high too
  - Robust against malicious group members
  - Attacks who might change data on wire
  - Accidental data corruption due to bugs
  - Slow, but fast enough for many uses, like replicating a database of security keys

Virtual synchrony

- Best option if performance is a key goal
  - Can do a flush before acting on an incoming multicast if the action will be externally visible (if it “really matters”)
  - But not robust against Byzantine failures
- Has been more successful in real-world settings, because real-world puts such high value on performance

State Machines

- Paxos system implements them, using a quorum method
  - In fact has many optimizations to squeeze more performance out of the solution
  - Still rather slow compared to virtual sync.
- But achieves “safe abcast” and for that, is cheaper than abcast followed by flush
  - Use it if dynamic uniformity is required in app.
  - E.g. when service runs some external device

Take away?

- We can build groups in two ways
  - With dynamic membership
  - With static membership
  - (the former can also emulate the latter)
  - (the latter can be extended with Byz. Agreement)
- Protocols support group data replication
- Tradeoff between speed and robustness
  - User must match choice to needs of the app.