CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

RECAP: Agreement Protocols

- These are used when a set of processes needs to make a decision of some sort
- The problem arises often and can take many forms
  - An *agreement protocol* solves a simple (single-bit) instance of the general problem
  - Gives us a "template" for building fancier protocols that solve other problems

---

When is agreement needed?

- Recall Sam and Jill from lecture 5
  - Sam was hoping he and Jill could eat outside but they couldn't get their act together and ended up eating inside
  - It illustrated a type of impossibility result:
    - Impossible to learn new "common knowledge" facts in an asynchronous distributed system
    - Defn: "I know that you know that I know…" without any finite limit on the chain

---

FLP was about agreement

- There we focused on agreement on the value of a single bit
- We concluded that
  - One can build decision protocols
    - And prove that they decide correctly
    - And can even include failure handling
  - But they can't guarantee progress
    - If if we have *many* processes and know that *at most one* of them might crash

---

We don't always need the FLP "version" of agreement

- Sam and Jill needed an impossible-to-achieve form of agreement!
  - Had they sought a weaker guarantee they might have been able to eat outside without risk!
    - For example: suppose Sam sends "Let's eat outside" and Jill replies "Sounds good," and Sam replies "See yah!"
    - 3-way handshake has risk built in (if the last message doesn't get through, what to do?) but the risk isn't large.
    - If they can live with that risk… it solves the problem
- FLP is about impossible "progress" properties

When is agreement needed?

- Situations where agreement arises
  - Ordering updates to a replicated data item
    - Might allow concurrent updates from sources that don't coordinate their actions
    - Or could have updates that are ordered by, say, use of locking but that might then get disordered in transmission through the network
  - Decision on which updates "occurred"
    - An issue in systems that experience faults

More needs for agreement

- Agreement on the membership
- Agreement on the leader, or some other process with a special role
- Agreement on a ranking
- Agreement on loads or other inputs to a load balancing service
- Agreement on the mapping of a name to a target IP address, or on routing

One protocol isn't enough!

- We'll need different solutions for these different agreement problems
  - But if we abstract away the detail can learn basic things about how such protocols should be structured
  - Also can learn to prove correctness
  - Then can build specific specialized ones, optimized for a particular use and *engineered to perform well*

Agreement: Paradigms

- We've already seen two examples
  - FLP involved consensus on a single bit
    - Processes have a bit values 0 or 1
    - Protocol executes
    - Outcome: all agree on a value (and it was a legitimate input, and they tolerate faults, and we are faithful to the decisions of the dead)
  - Byzantine Agreement: same idea; different model
- But paradigms are about clean theory. *Engineering* implies a focus on speed!

Things we know

- From FLP we know that this statement of the problem…
  - … can be solved in asynchronous settings
  - … but solution can't guarantee liveness
    - There is at least one input scenario and "event sequence" that prevents progress
- From BA, we know that in a system with synchronized rounds, solutions *can* be found, but they are costly
  - Anyhow, that synchronous model is impractical

What about real systems?

- Real world is neither synchronous nor asynchronous
  - We've got a good idea of messages latency
  - … and pretty good clocks
  - Like Sam and Jill, we may be able to tolerate undesired but unlikely outcomes
  - Anyhow, no real system can achieve perfect correctness (at best we depend on the compiler, the operating system)

Real world goals?

- Practical solutions that:
  - Work if most of the system is working
  - Tolerate crashes and perhaps even some mild forms of "Byzantine" behavior, like accidental data corruption
  - "Strive to be live" (to make progress) but accept that some crash/partitioning scenarios could prevent this, like it or not
- We still want to be rigorous

Performance goals

- Want solutions that are cheap, but what should this mean?
    - Traditionally: low total number of messages sent (today, only rarely an important metric)
    - Have low costs in per-process messages sent, received (often important)
    - Have low delay from when update was generated to when it was applied (always VERY important)

Other goals

- Now we'll begin to work our way up to really good solutions. These:
    - Are efficient in senses just outlined
    - Are packaged so that they can be used to solve real problems
    - Are well structured, so that we can understand the code and (hopefully) debug/maintain it easily

Roadmap

- To do this
    - First look at 2-phase and 3-phase commit
        - This pattern of communication arises in many protocols and will be a basic building block
    - Next look at "agreeing on membership"
        - Protocols that track membership give fastest update rates, often by orders of magnitude!
    - Then, implement an ordered update (or multicast) over these mechanisms
    - Finally, think about software architecture issues

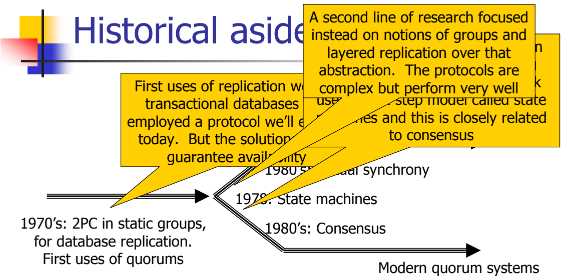
Roadmap

- This will give us
    - A notion of a "process group"
        - Has a name… and a set of members… and the system can dynamically track membership
        - Membership *ranking* is useful in applications
    - Ways to do ordered, reliable, multicast
    - Things built over these primitives: leader election, replication, fault-tolerant request execution, etc

Historical aside

- We're following the evolution of the area now called "distributed systems"
- But we're focused on the path that gave the highest performance solutions
    - Also known as virtual synchrony systems
- Historically, many researchers focused on quorum systems, a second path
    - Much slower, although also has some benefits
    - Closely related to "State Machine" replication

Historical aside



A second line of research focused instead on notions of groups and layered replication over that abstraction. The protocols are complex but perform very well

n
k
use… step model called state …es and this is closely related to consensus

First uses of replication w… transactional databases employed a protocol we'll e… today. But the solution… guarantee avail…ility

1980's …al synchrony

1978: State machines

1980's: Consensus

1970's: 2PC in static groups, for database replication. First uses of quorums

Modern quorum systems

Historical Aside

- Two major classes of real systems
  - Virtual synchrony
    - Weaker properties – not quite “FLP consensus”
    - Much higher performance (orders of magnitude)
    - Requires that majority of system remain connected. Partitioning failures force protocols to wait for repair
  - Quorum-based state machine protocols are
    - Closer to FLP definition of consensus
    - Slower (by orders of magnitude)
    - Sometimes can make progress in partitioning situations where virtual synchrony can’t

Names of some famous systems

- Isis was first practical virtual synchrony system
  - Later followed by Transis, Totem, Horus
  - Today: Best options are Jgroups, Spread, Ensemble
  - Technology is now used in IBM Websphere and Microsoft Windows Clusters products!
- Paxos was first major state machine system
  - BASE and other Byzantine Quorum systems now getting attention from the security community
- (End of Historical aside)

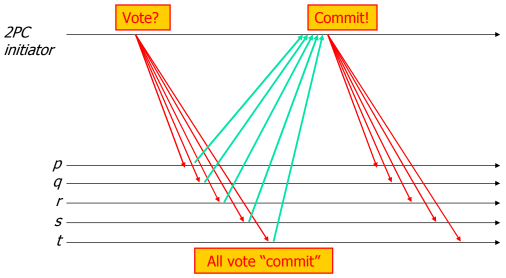
We’re already on track “A”

- We’re actually focused more on the virtual synchrony “track”
  - Not enough time to do justice to both
  - And systems engineers tend to prefer very high performance
  - But for systems doing secure replication, the Byzantine Quorums approach is probably better despite costs

The commit problem

- An initiating process communicates with a group of actors, who vote
  - Initiator is often a group member, too
  - Ideally, if all vote to commit we perform the action
  - If any votes to abort, none does so
- Asynchronous model
  - Network is reliable, but no notion of time
  - Fail-stop processes
- In practice we introduce timeouts;
  - If timeout occurs the leader can presume that a member wants to abort. Called the *presumed abort* assumption.

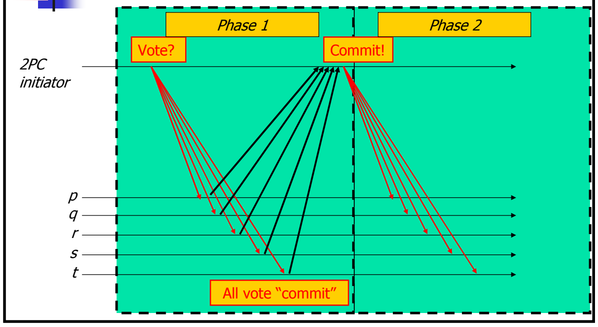
As a time-line picture



Observations?

- Any member can abort any time it likes, even before the protocol runs
  - E.g. if we are talking “about” some pending action that the group has known for a while
- We call it “2 phase” even though it actually has 3 rounds of messages

As a time-line picture



In fact we're missing stuff

- Eventually will need to do some form of garbage collection
  - Issue is that participants need memory of the protocol, at least for a while
  - But can delay garbage collection and run it later on behalf of many protocol instances
- Part of any real implementation but not thought of as part of the protocol

Fault tolerance

- We can separate this into three cases
  - Group member fails; initiator remains healthy
  - Initiator fails; group members remain healthy
  - Both initiator and group member fail
- Further separation
  - Handling recovery of a failed member
  - Recovery after "total" failure of the whole group

Fault tolerance

- Some cases are pretty easy
  - E.g. if a member fails before voting we just treat it as an abort
  - If a member fails after voting commit, we assume that when it recovers it will finish up the commit and perform whatever action we requested
- Hard cases involve crash of initiator

Initiator fails, members healthy

- Must ask "when did it fail"?
  - Could fail before starting the 2PC protocol
    - In this case if the members were expecting the protocol to run, e.g. to terminate a pending transaction on a database, they do "unilateral abort"
  - Could fail after some are prepared to commit
    - Those members need to learn the outcome before they can "finish" the protocol
  - Could fail after some have learned the outcome
    - Others may still be in a prepared state

Ideas?

- Members could do an all-to-all broadcast
  - But this won't actually work… problem is that if a process seems to have failed, perhaps some of us will have seen its messages and some not
- Could elect a leader to solve the problem
  - Forces us to inject leader election into our system
- Could use some sort of highly available log server that remembers states of protocols
  - This is how Web Services does it

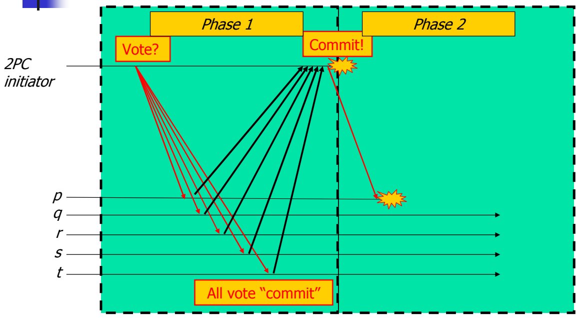
Leads to two ideas

- Initiator should record the decision in a logging server for use after crashes
  - We saw this in the Web Services transactional systems slide set last week
- Also, members can help one-another terminate the protocol
  - E.g., a leader can take over if the initiator fails
  - This is needed if a failure happens before the initiator has a chance to log its decision

Problems?

- 2PC has a "bad state"
  - Suppose that the initiator and a member both fail and we aren't using a "log"
    - As 2PC is normally posed, we don't have a log server in the problem statement
    - (In practice, log server can eliminate this issue)
  - There is a case in which we can't terminate the protocol!

As a time-line picture



Why do we get stuck?

- If process p voted "commit", the coordinate may have committed the protocol
  - And p may have learned the outcome
  - Perhaps it transferred $10M from a bank account…
  - So we want to be consistent with that
- If p voted "abort", the protocol must abort
  - And in this case we can't risk committing

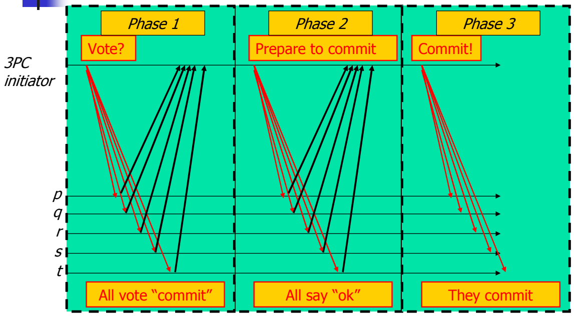
Why not always have a log?

- In some sense, a log service is just another member
  - In effect, Web Services is willing to wait if its log server crashes and must reboot
  - And guarantees that if this doesn't happen you never need to wait
  - But in many systems we just want to use 2PC. Using a separate server is a pain
- Can we solve the problem without it?

3 phase commit

- Protocol was introduced by Skeen and Stonebraker
- And it assumes *detectable* failures
  - We happen to know that real systems can't detect failures, unless they can unplug the power for a faulty node
  - But Skeen and Stonebraker set that to the side
- Idea is to add an extra "prepared to commit" stage

3 phase commit



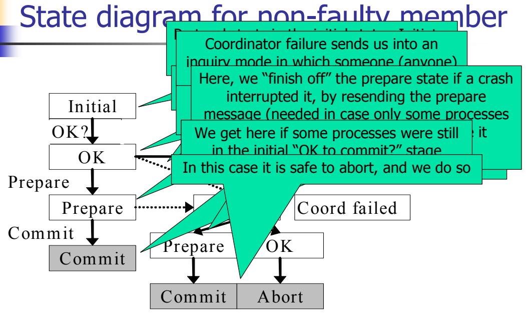
Why 3 phase commit?

- A “new leader” in the group can deduce the outcomes when this protocol is used
- Main insight?
  - Nobody can enter the commit state unless all are first in the prepared state
  - Makes it possible to determine the state, then push the protocol forward (or back)
- But does require accurate failure detections
  - If it didn’t, would violate the FLP result!

Value of 3PC?

- Even with inaccurate failure detections, it greatly reduces the window of vulnerability
  - The bad case for 2PC is not so uncommon
    - Especially if a group member is the initiator
    - In that case one badly timed failure freezes the whole group
  - With 3PC in real systems, the troublesome case becomes very unlikely
- But the risk of a freeze-up remains

State diagram for non-faulty member



Coordinator failure sends us into an
inquiry mode in which someone (anyone)

Here, we “finish off” the prepare state if a crash
interrupted it, by resending the prepare
message (needed in case only some processes

We get here if some processes were still
in the initial “OK to commit?” stage

In this case it is safe to abort, and we do so

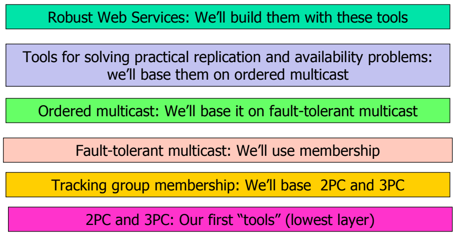
Some additional details

- Like 2PC, 3PC needs some extra work
  - Issue is that members need to save some information about the protocol until it terminates
  - In practice this requires an extra round for garbage collection
  - Often we do this just now and then, on behalf of *many* terminated protocols, so costs are amortized and very low

What next?

- We’ll use a protocol based on 2PC and 3PC (both are used) to build a group membership service
  - This is a system service that tracks membership of process groups
  - The service itself tries to be highly available (but can’t always do so)
  - Other processes use it in place of a failure detection system

Layering



Robust Web Services: We'll build them with these tools

Tools for solving practical replication and availability problems: we'll base them on ordered multicast

Ordered multicast: We'll base it on fault-tolerant multicast

Fault-tolerant multicast: We'll use membership

Tracking group membership: We'll base 2PC and 3PC

2PC and 3PC: Our first "tools" (lowest layer)

But first…

- We've seen several new mechanisms
- Let's pause and ask if we can already apply them in some practical real-world settings
- Then resume and work our way up the protocol stack!

What should you be reading?

- We're working our way through Chapter 14 of the textbook now
- Read the introduction to Part III and Chapters 13, 14 and 15