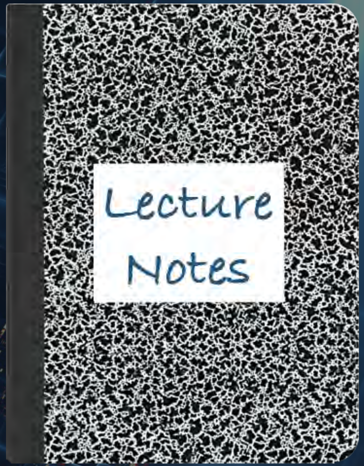


CS 417 – DISTRIBUTED SYSTEMS

Week 10: Distributed Transactions

Part 2: Three-Phase Commit and the CAP Theorem

Paul Krzyzanowski

Three-Phase Commit Protocol

What's wrong with the 2PC protocol?

Biggest problem: it's a blocking protocol with failure modes that require all systems to recover eventually

- If the coordinator crashes, participants have no idea whether to commit or abort
  - A recovery coordinator helps
- If a coordinator AND a participant crashes
  - The system has no way of knowing the result of the transaction
  - It might have committed for the crashed participant – hence all others must block

The protocol cannot pessimistically abort because some participants may have already committed

When a participant gets a commit/abort message, it does not know if every other participant was informed of the result

Three-Phase Commit Protocol

- Same setup as the two-phase commit protocol:
  - Coordinator & Participants
- Add timeouts to each phase that result in an abort
- Propagate the result of the commit/abort vote to each participant *before* telling them to act on it
  - This will allow us to recover the state if any participant dies

Three-Phase Commit Protocol

Split the second phase of 2PC into two parts:

2a. "*Precommit*" (*prepare to commit*) phase

- Send *Prepare* message to all participants when it received a *yes* from all participants in phase 1
- Participants can prepare to commit but cannot do anything that cannot be undone
- Participants reply with an acknowledgement
- Purpose: *let every participant know the state of the result of the vote so that state can be recovered if anyone dies*

2b. "*Commit*" phase (same as in 2PC)

- If coordinator gets ACKs for all *prepare* messages
  - It will send a *commit* message to all participants
- Else it will abort – send an *abort* message to all participants

Three-Phase Commit Protocol: Phase 1

Phase 1: *Voting phase*

- Coordinator sends *CanCommit?* queries to participants & gets responses
- Purpose: Find out if everyone agrees to commit

- [!] If the coordinator gets a *timeout* from any participant, or *any* NO replies are received
  - Send an *abort* to all participants

- [!] If a participant times out waiting for a request from the coordinator
  - It aborts itself (assume coordinator crashed)

- Else continue to phase 2

We can abort if the participant and/or coordinator dies

Three-Phase Commit Protocol

Phase 2: *Prepare to commit phase*

- Send a ***prepare*** message to all participants
- Get *OK* messages from **all** participants
  - We need to hear from all before proceeding so we can be sure the state of the protocol can be properly recovered if the coordinator dies
- Purpose: let all participants know the decision to commit
- [!] If a participant times out: assume it crashed; send *abort* to all participants

Phase 3: *Finalize phase*

- Send ***commit*** messages to participants and get responses from all
- [!] If participant times out: *contact any other participant* and move to that state (*commit* or *abort*)
- [!] If coordinator times out: that's ok – we know what to do

3PC Recovery

If the coordinator crashes

A recovery node can query the state from any available participant

Possible states that the participant may report:

Already committed

- That means that *every* other participant has received a *Prepare to Commit*
- Some participants may have committed

⇒ Send *Commit* message to all participants (just in case they didn't get it)

Not committed but received a *Prepare* message

- That means that all participants agreed to commit; some may have committed
- Send *Prepare to Commit* message to all participants (just in case they didn't get it)
- Wait for everyone to acknowledge; then commit

Not yet received a *Prepare* message

- This means no participant has committed; some may have agreed
- Transaction can be aborted or the commit protocol can be restarted

3PC Weaknesses

- May have problems when the network gets partitioned
  - Partition A: nodes that received ***Prepare*** message
    - Recovery coordinator for A: **allows commit**
  - Partition B: nodes that did not receive ***Prepare*** message
    - Recovery coordinator for B: **aborts**
  - Either of these actions are legitimate as a whole
    - But when the network merges back, the system will be inconsistent

- Not good when a crashed coordinator recovers
  - It needs to find out that someone took over and stay quiet
  - Otherwise, it will mess up the protocol, leading to an inconsistent state

3PC coordinator recovery problem

Suppose a coordinator sent a *Prepare* message to all participants
- All participants acknowledged the message
- BUT the coordinator died before it got all acknowledgements

- A recovery coordinator queries a participant
  - It continues with the commit: Sends *Prepare, gets ACKs, sends Commit*
- Around the same time…*the original coordinator recovers*
  - Realizes it is still missing some replies from the *Prepare*
  - Gets timeouts from some and decides to send an *Abort* to all participants
- Some processes may commit while others abort!
- 3PC works well when servers crash (fail-stop model)
- **3PC is not resilient against fail-recover environments**
- **3PC is not resilient against network partitions**

Consensus-based Commit

What about Raft?

- Consensus-based protocols (Raft, Paxos) are designed to be resilient against network partitions
- What does Raft consensus offer?
  - Total ordering of proposals (replicated log)
  - Fault tolerance: proposal is accepted if a **majority** of nodes accept it
    - There is always enough data available to recover the state of proposals
  - Is provably resilient in asynchronous networks
- For a two-phase commit protocol using a consensus algorithm:
  - Use replicated nodes to avoid blocking if the coordinator fails
  - Run a consensus algorithm on the commit/abort decision of **EACH** participant

What do we want to do with a consensus protocol?

- Each participant must get its chosen value – *can_commit* or *must_abort* – accepted by the majority of replicated nodes

- Transaction Leader
  - Chosen via election algorithm
  - Coordinates the commit algorithm
  - Not a single point of failure – we can elect a new one; Raft nodes store state

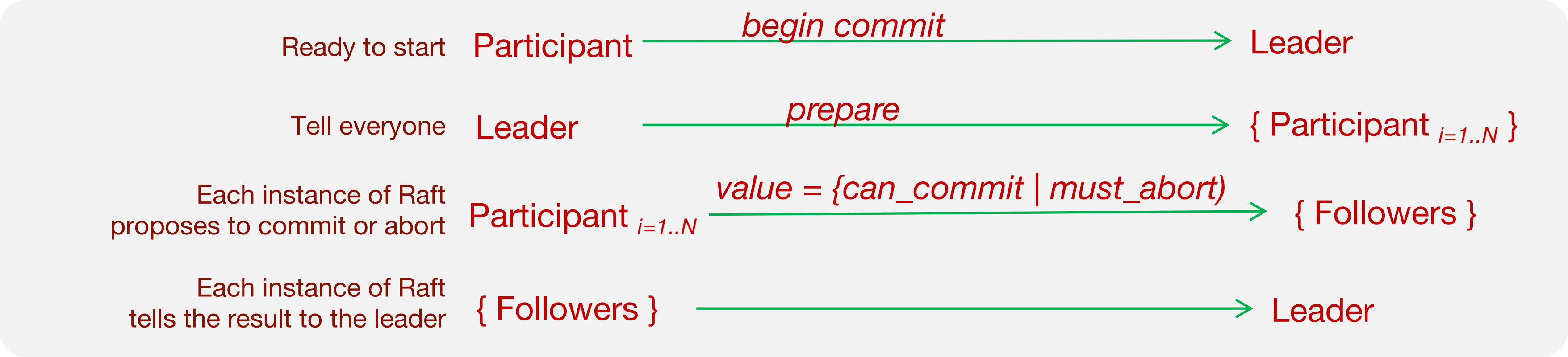
How do we do it?

- Some participant decides to begin to *commit*
  - Sends a message to the Transaction Leader
- Transaction Leader: Sends a *prepare* message to each participant
- Each participant now sends a *can_commit* or *must_abort* message to its instance of the consensus protocol (Raft)
  - All participants share the elected Transaction Leader
  - "*Can_commit*" or "*Must_abort*" is sent to majority of followers
  - Result is sent to the leader
- Transaction Leader tracks all instances of the commit protocol
  - Commit *iff* every participant's instance of the consensus protocol chooses "*can_commit*"
  - Tell each participant to **commit** or **abort**

Consensus-based fault-tolerant coordinator

The cast:

- One instance of Raft per participant (N participants)
- Set of 2F+1 nodes and a leader play the role of the coordinator
  - We can withstand the failure of F nodes
  - Leader = node elected to be in charge & run the protocol



- A leader will get at least *F+1* messages for each instance
- Commit *iff* every participant's instance of Raft chooses *can commit*
- Raft commit acts like 2PC if only one node

Virtual Synchrony vs. Transactions vs. Raft

- **Virtual Synchrony**
  - Fast & scalable
  - State machine replication: multicast messages to the entire group
  - Focuses on group membership management & reliable multicasts
  - Does not handle partitions!

- **Two-Phase & Three-Phase Commit**
  - Most expensive – requires extensive use of stable storage
  - 2PC efficient in terms of # of messages
  - Designed for transactional activities
  - Not suitable for high-speed messaging

- **Raft (or Paxos) Consensus**
  - General purpose fault-tolerant consensus algorithm
  - Performance limited by its two-phase protocol
  - Useful for fault-tolerant log replication & elections
  - Using consensus-based commit overcomes dead coordinator and network partition problems of 2PC and 3PC
  - Need mechanisms to restore state on *abort.*

Scaling & Consistency

Reliance on multiple systems affects availability

- One database with 99.9% availability
  - 8 hours, 45 minutes, 35 seconds downtime per year
- If a transaction uses 2PC protocol and requires two databases, each with a 99.9% availability:
  - Total availability = (0.999*0.999) = 99.8%
  - 17 hours, 31 minutes, 12 seconds downtime per year
- If a transaction requires 5 databases:
  - Total availability = 99.5%
  - 1 day, 19 hours, 48 minutes, 0 seconds downtime per year

Scaling Transactions

- Transactions require locking part of the database so that everyone sees consistent data at all times
  - Good on a small scale.
    - Low transaction volumes: getting multiple databases consistent is easy
  - Difficult to do efficiently on a huge scale
- Add replication – processes can read any replica
  - But all replicas must be locked during updates to ensure consistency
- **Risks of not locking:**
  - Users run the risk of seeing stale data
  - The "I" of ACID may be violated
    - E.g., two users might try to buy the last book on Amazon

Delays hurt

The delays to achieve consistency can hurt business

- Amazon: 0.1 second increase in response time costs 1% of sales
- Google: 0.5 second increase in latency causes traffic to drop by 20%
- Latency is due to lots of factors
  - OS & software architecture, computing hardware, tight vs loose coupling, network links, geographic distribution, …
  - We're only looking at the problems caused by the tight software coupling due to achieving the ACID model

http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it

http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

Eric Brewer's CAP Theorem

Three core requirements in a shared data system:

1. **Atomic, Isolated Consistency**
   – Operations must appear totally ordered and each is isolated
2. **Availability**
   – Every request received by a non-failed node must result in a response
3. **Partition Tolerance**: tolerance to network partitioning
   Messages between nodes may be lost

No set of failures less than total failure is allowed to cause the system to respond incorrectly*

CAP Theorem: when there is a network partition, you cannot guarantee both availability & consistency

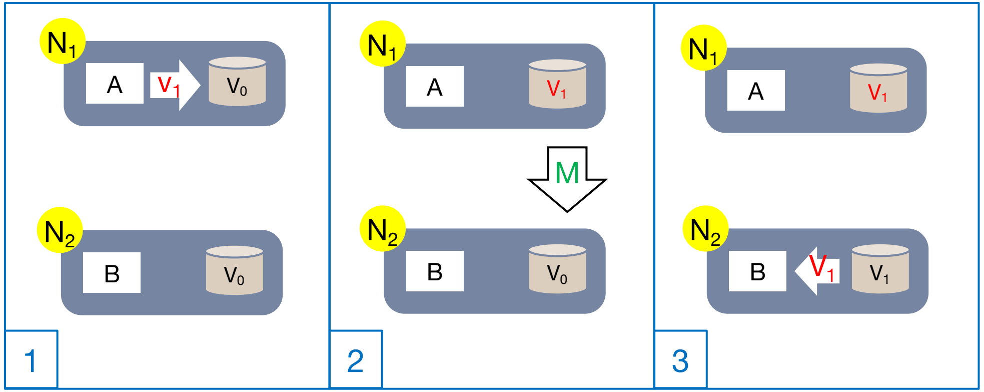
*goo.gl/7nsj1R

Commonly (not totally accurately) stated as *you can have at most two of the three: C, A, or P*

Example: Partition

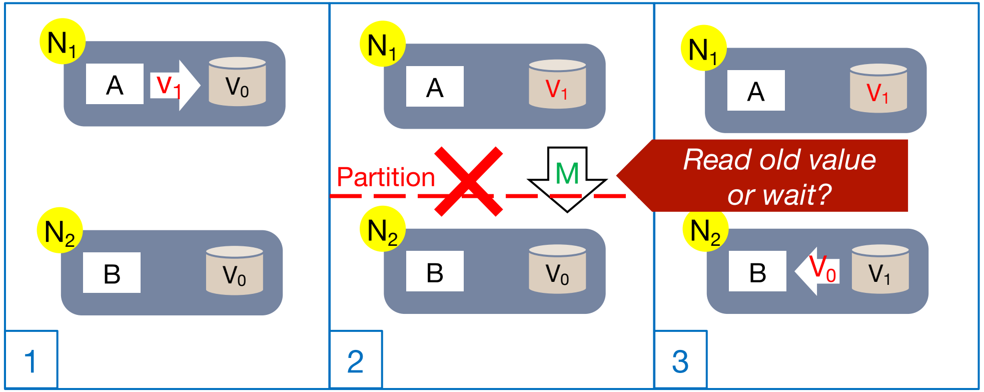
Life is good

A writes v_1 on N_1
v_1 propagates to N_2
B reads v_1 on N_2



Network partition occurs

A writes v_1 on N_1
*v_1 **cannot** propagate to N_2*
B reads v_0 on N_2



Do we want to give up consistency or availability?

From: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

Giving up one of {C, A, P}

- **Ensure partitions never occur**
  - Put everything on one machine or a cluster in one rack: high availability clustering
  - Use two-phase commit or three phase commit
  - **Scaling suffers**
- **Give up availability** [system is consistent & can handle partitioning]
  - Lock data: have services wait until data is consistent
  - Classic ACID distributed databases (also 2PC)
  - **Response time suffers**
- **Give up consistency** [system is available & can handle partitioning]
  - ***Eventually consistent*** data
  - Use expirations/leases, queued messages for updates
  - *Often not as bad as it sounds!*
  - Examples: DNS, web caching, Amazon Dynamo, Cassandra, CouchDB

We *really* want partition tolerance & high availability for a distributed system!

Partitions will occur

- With distributed systems, we expect partitions to occur
  - Maybe not a true partition but high latency can act like a partition
  - This is a property of the distributed environment
  - The CAP theorem says we have a tradeoff between availability & consistency
- But we want availability *and* consistency
  - We get availability via replication
  - We get consistency with atomic updates
    1. *Lock all copies before an update*
    2. *Propagate updates*
    3. *Unlock*
- We can choose high availability: allow reads before all nodes are updated (avoid locking)
- or choose consistency: enforce proper locking of nodes for updates

Eventual Consistency Model

- Traditional database systems want ACID
  - But scalability is a problem (lots of transactions in a distributed environment)

- Give up ***Consistent*** and ***Isolated***
  in exchange for *high availability* and *high performance*
  - Get rid of locking in exchange for multiple versions
  - Incremental replication

- **BASE** = Basically Available • Soft-state • Eventual Consistency

Consistency model:

If no updates are made to a data item, *eventually* all accesses to that item will return the last updated value

ACID vs. BASE

ACID

- Strong consistency
- Isolation
- Focus on *commit*
- Nested transactions
- Availability can suffer
- Pessimistic access to data (locking)

BASE

- Weak (eventual) consistency: stale data at times
- High availability
- Best effort approach
- Optimistic access to data
- Simpler model
(but harder for app developer)
- Faster

From Eric Brewer's PODC Keynote, July 2000
http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

A place for BASE

- ACID is neither dead nor useless
  - Many environments require it
  - It's safer – the framework handles ACID for you
- BASE has become common for large-scale web apps where replication & fault tolerance is crucial
  - eBay, Twitter, Amazon
  - Eventually consistent model not always surprising to users
    - Cellphone usage data
    - Banking transactions (e.g., fund transfer activity showing up on statement)
    - Posting of frequent flyer miles

But … the app developer has to worry about update conflicts and reading stale data … and programmers often write buggy code

The End