Agreement Generally

• Most distributed systems problems can be reduced to this one:
  • Despite being separate nodes (with potentially different views of their data and the world)...
  • All nodes that store the same object O must apply all updates to that object in the same order (consistency)
  • All nodes involved in a transaction must either commit or abort their part of the transaction (atomicity)
• Easy?
  • … but nodes can restart, die or be arbitrarily slow
  • … and networks can be slow or unreliable too
Properties of Agreement

- 2 kinds of properties, just like for mutual exclusion:
- Safety (correctness)
  - All nodes agree on the same value (which was proposed by some node)
- Liveness (fault tolerance, availability)
  - If less than N nodes crash, the rest should still be OK
2-Phase Commit

• Separate the commit into two steps:
  • 1: Voting
    • Each participant prepares to commit and votes of whether or not it can commit
  • 2: Committing
    • Once voting succeeds, every participant commits or aborts
• Assume that participants and coordinator communicate over RPC
2PC Event Sequence

Coordinator
Transaction state:

- prepared
- committed

Can you commit?

Participant
Local state:

- prepared
- uncertain
- committed
Fault Recovery Example

Coordinator (client or 3rd party)  
transaction .commit()  

Participant Goliath National

Participant Duke & Duke

"Yes"  
response_{GNB}  
prepare  

"Yes"  
response_{D&D}  
prepare  

outcome  

Crash :(

Example: Participant crashes after voting “yes” to commit

Solution: Participants must keep track of transaction status on persistent storage for recovery on reboot
Fault Recovery Example

Example: Coordinator crashes after receiving votes

Solution: Coordinator must keep track of transaction status on persistent storage for recovery on reboot
2PC Timeouts

• We can solve a lot (but not all of the cases) by having the participants talk to each other
• But, if coordinator fails, there are cases where everyone stalls until it recovers
• Can the coordinator fail?… yes
• We’ll come back to this “discuss amongst yourselves” kind of transactions today!
Today

• More discussion of fault tolerance, in the context of transactions
• Agreement and transactions in distributed systems - 3PC
• Reminders:
  • HW3 due next week!
Digging Deeper into 2PC Failures

• Fundamental problem:
  • Once coordinator says commit we can not go back
  • That’s the property of transactions though!
  • In what situations can we reach consensus if the coordinator fails?
  • Let’s go through some examples again, this time using Socrative to poll your answers

Go to socrative.com and select “Student Login” Room: CS475; ID is your G-Number
Digging Deeper into 2PC Failures

If they can talk to each other, we know we can commit (good)

Question 1

Participant A: Voted yes
Heard back “commit”

Participant B: Voted yes
Did not hear result

Participant C: Voted yes
Did not hear result

Participant D: Voted yes
Did not hear result
Digging Deeper into 2PC Failures

Question 2

If they can talk to each other, we know that we can all abort (good)

- Coordinator
- Participant A: Voted no
- Participant B: Voted yes
- Participant C: Voted yes
- Participant D: Voted yes

Did not hear result
Digging Deeper into 2PC Failures

If they can talk to each other, we do not know if we can commit/abort (who knows what the coordinator will do?)

Question 3

Coordinator

Participant A: Voted yes, Did not hear result
Participant B: Voted yes, Did not hear result
Participant C: Voted yes, Did not hear result
Participant D: Voted yes, Did not hear result
Question 4

Digging Deeper into 2PC Failures

If they can talk to each other, we do not know if we can commit/abort (who knows that there was a vote no?)

Participant A    Voted no
Did not hear result

Participant B    Voted yes
Did not hear result

Participant C    Voted yes
Did not hear result

Participant D    Voted yes
Did not hear result

Coordinator

X
Digging Deeper into 2PC Failures

Question 5

If they can talk to each other, we do not know if we can commit/abort (do not know what the coordinator heard/said)

<table>
<thead>
<tr>
<th>Participant</th>
<th>Voted</th>
<th>Heard result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participant A</td>
<td>yes</td>
<td>Heard back “commit”</td>
</tr>
<tr>
<td>Participant B</td>
<td>yes</td>
<td>Did not hear result</td>
</tr>
<tr>
<td>Participant C</td>
<td>yes</td>
<td>Did not hear result</td>
</tr>
<tr>
<td>Participant D</td>
<td>yes</td>
<td>Did not hear result</td>
</tr>
</tbody>
</table>
3 Phase Commit

- Goal: Eliminate this class of failure from blocking liveness

- Coordinator
- Participant A: Voted yes, Heard back “commit”
- Participant B: Voted yes
- Participant C: Voted yes
- Participant D: Voted yes
- Did not hear result
- Did not hear result
- Did not hear result
3 Phase Commit

• Goal: Avoid blocking on node failure
• How?
  • Think about how 2PC is better than 1PC
    • 1PC means you can never change your mind or have a failure after committing
    • 2PC still means that you can’t have a failure after committing (committing is irreversible)
3 Phase Commit

- 3PC idea:
  - Split commit/abort into 2 sub-phases
    - 1: Tell everyone the outcome
    - 2: Agree on outcome
  - Now: EVERY participant knows what the result will be before they irrevocably commit!
3PC Example

Coordinator       Participants (A,B,C,D)

Soliciting votes

Timeout causes abort

Commit authorized (if all yes)

Timeout causes abort

Done

prepare

response

pre-commit

OK

commit

OK

Status: Uncertain

Timeout causes abort

Status: Prepared to commit

Timeout causes commit

Status: Committed
3PC Exercise

Scenario:
1 Coordinator, 4 participants
No failures, all commit
3PC Crash Handling

- Can B/C/D reach a safe decision...
  - If any one of them has received preCommit?
    - YES! Assume A is dead. When A comes back online, it will recover, and talk to B/C/D to catch up.
    - Consider equivalent to in 2PC where B/C/D received the “commit” message and all voted yes
3PC Crash Handling

• Can B/C/D reach a safe decision…
  • If NONE of them has received preCommit?
    • YES! It is safe to abort, because A can not have committed (because it couldn’t commit until B/C/D receive and acknowledge the pre-commit)
  • This is the big strength of the extra phase over 2PC
• Summary: Any node can crash at any time, and we can always safely abort or commit.
3PC Exercise

Coordinator          Participants (A,B,C,D)

Soliciting votes

Timeout causes abort

Commit authorized (if all yes)

Timeout causes abort

Done

Status: Uncertain
Timeout causes abort

Status: Prepared to commit
Timeout causes commit

Status: Committed

Scenario:
1 Coordinator, 4 participants
After pre-commit sent, coordinator and A fail
Properties of Agreement

- **Safety** (correctness)
  - All nodes agree on the same value (which was proposed by some node)
- **Liveness** (fault tolerance, availability)
  - If less than N nodes crash, the rest should still be OK
Does 3PC guarantee agreement?

• Reminder, that means:
  • Liveness (availability)
    • Yes! Always terminates based on timeouts
  • Safety (correctness)
    • Yes!

*Assuming that the only way things fail is by crashing*
Safety in Crashes

Timeout behavior: abort!

Prepared to commit

Coordinator

Soliciting Authorized

Participant A: Yes
Participant B: X
Participant C: X
Participant D: X

Crashed: do not commit or abort. When recovers, asks coordinator what to do
Partitions

Implication: if networks can delay arbitrarily, 3PC does not guarantee safety!!!!

Timeout behavior: abort

Coordinator

Soliciting Authorized

Network Partition!!!

Prepared to commit

Participant A

Yes

Participant B

Yes

Participant C

Yes

Participant D

Yes

Committed

Aborted

Aborted

Aborted

Implication: if networks can delay arbitrarily, 3PC does not guarantee safety!!!!
Modeling our Systems

To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions.

Generally: Stronger assumptions -> worse performance
Weaker assumptions -> more complicated

- Synchronous
- Asynchronous
- Byzantine (we’ll come back to this, but blockchains are here)
- Partitions
- Crash-fail

Strength
System model
Failure Model
Synchronous vs Asynchronous Messages

- Synchronous: There is a bound on how long a message takes to arrive
- Asynchronous: There is no bound on how long a message takes to arrive
- Key implication: what does a timeout mean?
  - Synchronous: Something must have crashed
  - Asynchronous: Network might just be slow
- Note: real networks are asynchronous
Failure Models: Crash-Fail vs Partition Tolerant

- Crash-fail: Our system will be correct if the only failures we can ever see are a node crashing
- Partition tolerant: Our system will be correct for crashing failures and for arbitrary network delays
- NB: If the network is synchronous, we are partition-tolerant by default (no partitions possible)
2PC vs 3PC

• 2PC
  • Safety (always, for crash and partition failures)
  • Liveness (if 1 node fails, we may block)

• 3PC
  • Safety (assuming the only failure mode is crash, never partition)
  • Liveness (can always proceed if 1 node fails)
  • Can we have some hybrid/best of both worlds?
Can we fix it?

- Short answer: No.
- Fischer, Lynch & Paterson (FLP) Impossibility Result:
  - Assume that nodes can only fail by crashing, network is reliable but can be delayed arbitrarily
  - Then, there can not be a deterministic algorithm for the consensus problem subject to these failures
Why can’t we make a protocol for consensus/agreement that can tolerate both partitions and node failures?

To tolerate a partition, you need to assume that eventually the partition will heal, and the network will deliver the delayed packages.

But the messages might be delayed forever.

Hence, your protocol would not come to a result, until forever (it would not have the liveness property).
Partitions

Insight: There is a “majority” partition here (B, C, D)
The “minority” know that they are not in the majority (A can only talk to Coordinator, knows B, C, D might exist)

Timeout behavior: Commit!
Timeout behavior: abort

Participant A: Committed
Participant B: Committed
Participant C: Aborted
Participant D: Aborted

Coordinator

Soliciting Authoriz.

Network Partition!!!

Prepared to commit
Prepared to commit
Prepared to commit
Prepared to commit

Timeout behavior: abort

Insight: There is a “majority” partition here (B, C, D)
The “minority” know that they are not in the majority (A can only talk to Coordinator, knows B, C, D might exist)
Partition Tolerance

• Key idea: if you always have an odd number of nodes…
• There will always be a minority partition and a majority partition
• Give up processing in the minority until partition heals and network resumes
• Majority can continue processing
Partition Tolerant Consensus Algorithms

- Decisions made by **majority**
- Typically a fixed coordinator (**leader**) during a time period (**epoch**)
- How does the leader change?
  - Assume it starts out as an arbitrary node
  - The leader sends a heartbeat
  - If you haven’t heard from the leader, then you **challenge** it by advancing to the next epoch and try to elect a new one
  - If you don’t get a **majority** of votes, you don’t get to be leader
  - …hence no leader in a minority partition
Partition Tolerant Consensus Algorithms

In Search of an Efficient and Practical Raft Implementation

Abstract
Raft is a consensus algorithm for managing a distributed log. It produces a result equivalent to Google's Paxos, but in an order of magnitude more efficient. Raft is designed to be simple, yet practical for real-world systems. It allows for easy implementation while still providing strong consistency guarantees across a wide variety of crash scenarios.

1 Introduction
Raft guarantees safety, as long as the majority of members is operational. Raft is easy to implement, thanks to a simple state machine abstraction and a convenient API for interacting with the consensus engine.

ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Research
{phunt,mahadev}@yahoo-inc.com

Abstract
In this paper, we describe ZooKeeper, a service for coordinating processes of distributed applications. Since it is stateless, ZooKeeper is part of critical infrastructure. It provides a simple and high-performance kernel for building more complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services.

2 Introduction
ZooKeeper is a distributed configuration service for distributed applications. It guarantees that all processes agree on the configuration data, which is stored in a distributed file system. A ZooKeeper instance consists of a set of nodes, or clients, that communicate with one another to store and manage configuration data.
Paxos: High Level

• One (or more) nodes decide to be leader (proposer)
• Leader proposes a value, solicits acceptance from the rest of the nodes
• Leader announces chosen value, or tries again if it failed to get all nodes to agree on that value
• Lots of tricky corners (failure handling)
• In sum: requires only a majority of the (non-leader) nodes to accept a proposal for it to succeed
Paxos: Implementation Details

Just kidding!
ZooKeeper

• Distributed coordination service from Yahoo! originally, now maintained as Apache project, used widely (key component of Hadoop etc)
• Highly available, fault tolerant, performant
• Designed so that YOU don’t have to implement Paxos for:
  • Distributed transactions/agreement/consensus
• We’ll come back to ZooKeeper in a few weeks
This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/

- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.

- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.