Distributed Transactions: 3 Phase Commit and Beyond

CS 475, Spring 2019
Concurrent & Distributed Systems
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**

**Participant**

Local state:
- **prepared**
- **uncertain**
- **committed**

---

**Coordinating Event**

- Can you commit?
  - Yes
  - OK, commit

---

**Participant Event**

- prepared
- uncertain
- committed
Fault Recovery Example

Coordinator (client or 3rd party) | Participant Goliath National | Participant Duke & Duke

transaction .commit()

prepare

“Yes” response

prepare

“Yes” response

outcome

outcome

Example: Participant crashes after voting “yes” to commit

Crash :(]

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

Coordinator (client or 3rd party) | Participant Goliath National | Participant Duke & Duke

transaction .commit()

“Yes”

prepare

response_GNB

“Yes”

prepare

response_D&D

Crash :(

Example: Coordinator crashes after receiving votes

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

**Example: Participant times out while waiting to hear the outcome**

**Problem: Can the participant unilaterally determine the outcome?**

<table>
<thead>
<tr>
<th>Coordinator (client or 3rd party)</th>
<th>Participant Goliath National</th>
<th>Participant Duke &amp; Duke</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepare</td>
<td>response</td>
<td>response</td>
</tr>
<tr>
<td>&quot;Yes&quot;</td>
<td>prepare</td>
<td>response</td>
</tr>
<tr>
<td>&quot;Yes&quot;</td>
<td>response</td>
<td>outcome</td>
</tr>
<tr>
<td></td>
<td></td>
<td>outcome</td>
</tr>
</tbody>
</table>

Solution: As long as we vote "no" outcome is always abort! If we voted "yes"... no idea!
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 Thursday!
  • Study opportunity - help improve software engineering, get $40 - https://cs.gmu.edu/~tlatoza/studies/AuthoringDesignRules.pdf
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
If they can talk to each other, we know we can commit (good)

Coordinator

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

Coordinator

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
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?)

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
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?)

- 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 (do not know what the coordinator heard/said)

- Coordinator: Heard back “commit”
- Participant A: Voted yes
- Participant B: Voted yes
- Participant C: Voted yes
- Participant D: Voted yes

All participants voted yes, but the coordinator did not hear the result.
3 Phase Commit

- Goal: Eliminate this class of failure from blocking liveness

Diagram:

- Coordinator
- 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
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

Soliciting votes
Timeout causes abort
Commit authorized (if all yes)
Timeout causes abort
Done

Participants (A, B, C, D)

prepare
response
pre-commit
OK
commit
OK

Status: Uncertain
Timeout causes abort

Status: Prepared to commit
Timeout causes commit

Status: Committed
3PC Exercise

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

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

prepare

response

pre-commit

OK

commit

OK

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
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

Exercise round 2:
1 Coordinator, 4 participants
Coordinator sends pre-commit message then fails
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!

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!!!
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
Coordinator sends pre-commit message ONLY to A, then Coordinator fails, A partitioned
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

Strength

System model
- Synchronous
- Asynchronous

Failure Model
- Byzantine (we’ll come back to this, but blockchains are here)
- Partitions
- Crash-fail
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
FLP - Intuition

- 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)

Coordinator

Participant A
Uncertain
Committed

Participant B
Yes
Prepared to commit

Participant C
Yes
Prepared to commit

Participant D
Yes
Prepared to commit

Soliciting Authorizing

Network Partition!!!

Timeout behavior: Commit!

Timeout behavior: abort
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 Algorithm

Abstract

Raft is a consensus algorithm for many replicated log. It produces a result equivalent to that of Paxos; it is as efficient as Paxos, but in a way that improves practical systems. In order to enhance Paxos, Raft separates the key elements of leader election, log replication, and safety into a stronger degree of coherency to define states that must be considered. Raft demonstrates that Raft is easier for small Paxos. Raft also includes a mechanism for (a failure of Raft), the cluster membership, which uses timeouts to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of processes to work as a coherent group that can use some of its members. Because Paxos is so strong that it is key in building reliable large-scale distributed systems, the last decade: most of consensus are based on Paxos or it is Paxos has become the primary vehicle for discussion about consensus.

Unfortunately, Paxos is quite difficult to build and few understand how to use some of its members. Because Paxos is so strong, designers still struggle with Paxos. After struggling with Paxos ourselves, our conclusion is that our network needs and that we can no longer trust the Paxos implementation.

ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Grid
{patrick.hunt,mahadev}@yahoo.com

Flavio P. Junqueira and Benjamin Reed
Yahoo! Research
{mpj,breed}@yahoo.com

Abstract

In this paper, we describe ZooKeeper, a service that addresses the needs of distributed systems. Since ZooKeeper is part of critical infrastructure, ZooKeeper aims to provide a simple and high-performance service for building more complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, central service. The interface exposed by ZooKeeper has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems to provide a simple, yet powerful coordination service.

The ZooKeeper interface enables a high-performance service implementation. In addition to the wait-free property, ZooKeeper provides a per-client guarantee of FIFO execution of requests and linearizability for all requests that change the ZooKeeper state. These design decisions enable the implementation of a high-performance processing pipeline with real requests being satisfied by local servers. We show for the local workloads, 2.1 to 100.1 read to write ratio, that ZooKeeper can handle up to hundreds of thousands of transactions per second. This performance allows ZooKeeper to be used extensively by client applications.

One approach to coordination is to develop services for each of the different coordination needs. For example, Apache Simple Queue Service (SQS) focuses specifically on queueing. Other services have been developed specifically for state management (Cassandra) and configuration (Consul). Services that implement powerful primitives can be used to implement more powerful ones. For example, Chubby is a lock with strong synchronization guarantees. Chubby can then be used to implement leader election, group membership, etc.

When designing our coordination service, we moved away from implementing specific primitives on the server side, and instead we opted for exposing an API that enables application developers to implement their own primitives. Such a choice led to the implementation of a coordination kernel, providing a richer primitives without requiring changes to the server core. This approach enables multiple forms of coordination adapted to the requirements of applications, instead of constraining developers to a fixed set of primitives.
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.