Transactions

CS 475, Spring 2018
Concurrent & Distributed Systems
boolean transferMoney(Person from, Person to, float amount){
  if(from.balance >= amount) {
    from.balance = from.balance - amount;
    to.balance = to.balance + amount;
    return true;
  }
  return false;
}

Assume running on a single machine: What can go wrong here?
Review: Properties of Transactions

• Traditional properties: ACID
  • **Atomicity**: transactions are “all or nothing”
  • **Consistency**: Guarantee some basic properties of data; each transaction leaves the database in a valid state
  • **Isolation**: Each transaction runs as if it is the only one; there is some valid serial ordering that represents what happens when transactions run concurrently
  • **Durability**: Once committed, updates cannot be lost despite failures
Review: 2PC

transaction
.commit()

Coordinator
(client or 3rd party)

Participant
Goliath National

Participant
Duke & Duke

If we can commit, then lock
our customer, vote “yes”

If everyone can commit, then
outcome == commit, else
abort

outcome

outcome

prepare

response_{GNB}

response_{D&D}

prepare

outcome
Review: Recovery on Reboot

• If coordinator finds no “commit” message on disk, abort
• If coordinator finds “commit” message, commit
• If participant finds no “yes, ok” message, abort
• If participant finds “yes, ok” message, then replay that message and continue protocol
Announcements

• HW4 is out!
• Today:
  • Agreement & transactions in distributed systems (continued)
  • Reminder: lecture from last week is posted on YouTube
• Additional readings:
  • http://the-paper-trail.org/blog/consensus-protocols-two-phase-commit/
  • http://the-paper-trail.org/blog/consensus-protocols-three-phase-commit/
  • Tannenbaum Note 8.13 (“Advanced”!)
Timeouts in 2PC

• Example:
  • Coordinator times out waiting for Goliath National Bank’s response
  • Bank times out waiting for coordinator’s outcome message

• Causes?
  • Network
  • Overloaded hosts
  • Both are very realistic…
Coordinator Timeouts

• If coordinator times out waiting to hear from a bank
  • Coordinator hasn’t sent any commit messages yet
  • Can safely abort - send abort message
  • Preserves correctness, sacrifices performance (maybe didn’t need to abort!)
  • If either bank decided to commit, it’s fine - they will eventually abort
Handling Bank Timeouts

• What if the bank doesn’t hear back from coordinator?
• If bank voted “no”, it’s OK to abort
• If bank voted “yes”
  • It can’t decide to abort (maybe both banks voted “yes” and coordinator heard this)
  • It can’t decide to commit (maybe other bank voted yes)
• Does bank just wait for ever?
Handling Bank Timeouts

- Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit
- Bank asks other bank for status (if it heard from coordinator)
- If other bank heard “commit” or “abort” then do that
- If other bank didn’t hear
  - but other voted “no”: both banks abort
  - but other voted “yes”: no decision possible!
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
• Hence, 2PC does not guarantee **liveness**: a single node failing can cause the entire set to fail
2PC Exercise

Exercise round 1:
1 Coordinator, 4 participants
No failures, all commit

<table>
<thead>
<tr>
<th>Coord</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>
2PC Exercise

Exercise round 2:
1 Coordinator, 4 participants
Coordinator fails before providing outcome

Coordinator (client or 3rd party)

Participant

prepare
response
outcome

 Participants

<table>
<thead>
<tr>
<th>Coord</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>
2PC Exercise

Exercise round 3:
1 Coordinator, 4 participants
Coordinator provides outcome to
1 participant, then coordinator
and that participant fail

Coordinator (client or 3rd party)  Participant

prepare

response

outcome

<table>
<thead>
<tr>
<th>Coord</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>
3 Phase Commit

- Goal: Eliminate this specific failure from blocking liveness
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)
• 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

Coordinator

Participants (A,B,C,D)

Scenario:
1 Coordinator, 4 participants
No failures, all commit

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

Status: Uncertain
Timeout causes abort

Status: Prepared to commit
Timeout causes commit

Status: Committed

<table>
<thead>
<tr>
<th>Status</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>B</td>
</tr>
<tr>
<td>OK</td>
<td>C</td>
</tr>
<tr>
<td>OK</td>
<td>D</td>
</tr>
</tbody>
</table>

Coord
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.
Scenario:
1 Coordinator, 4 participants
After pre-commit sent, coordinator and A fail

Coordinator  Participants (A,B,C,D)

Soliciting votes

Timeout causes abort

Commit authorized (if all yes)

Timeout causes abort

Done

pre-commit

Timeout causes abort

pre-commit

OK

Timeout causes commit

OK

Status: Uncertain

Status: Prepared to commit

Status: Committed
3PC Timeout Handling

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

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

- Status: Uncertain
  
  Timeout causes abort

- Status: Prepared to commit

  Timeout causes abort

- Status: Committed

  Timeout causes commit
Agreement

• In distributed systems, we have multiple nodes that need to all agree that some object has some state

• Examples:
  • Who owns a lock
  • Whether or not to commit a transaction
  • The value of a file
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

- **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)
    • Hmm…
Partitions

Network Partition!!!

Coordinator

Participant A
Yes
Committed

Participant B
Yes
Prepared to commit

Participant C
Yes
Prepared to commit

Participant D
Yes
Prepared to commit

Timeout behavior: Commit!

Timeout behavior: abort

Commit Authorized

Uncertain

Uncertain

Uncertain

Uncertain

Commit!
3PC Exercise

Scenario:
1 Coordinator, 4 participants
Coordinator sends pre-commit message ONLY to A, then Coordinator fails, A partitioned

Timeout causes abort
Timeout causes abort
Timeout causes commit

Coord | A | B | C | D
--- | --- | --- | --- | ---
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 \textit{eventually} the partition will heal, and the network will deliver the delayed packages

• But the messages might be delayed \textit{forever}

• Hence, your protocol would not come to a result, until \textit{forever} (it would not have the \textit{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)

<table>
<thead>
<tr>
<th>Participant</th>
<th>Prepared to commit</th>
<th>Commit Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Yes</td>
<td>Commit!</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
<td>Timeout behavior: abort</td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>Timeout behavior: abort</td>
</tr>
<tr>
<td>D</td>
<td>Yes</td>
<td>Timeout behavior: abort</td>
</tr>
</tbody>
</table>

Timeout behavior: **Commit!**
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 Ultimate Consensus Algorithm

Abstract

Raft is a consensus algorithm for managing logs. It produces a result equivalent to Paxos, but it is as efficient as Paxos, but its stability rules are different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for various applications. In order to enhance scalability, Raft separates the key elements of cooperative protocols, such as leader election, log replication, and safety. Raft also includes a simple and intuitive algorithm for building a distributed system with a stronger degree of cohesion to recover from states that are not consistent. Results show that Raft is easier for developers to understand and use compared to Paxos.

1 Introduction

Consensus algorithms allow a collection of processes to work as a coherent group that can achieve agreement on a common value. Because of its key role in building reliable large-scale distributed systems, Paxos [9, 10] has dominated the consensus algorithms over the last decade. Most of the consensus algorithms are based on Paxos or its variants. In recent years, Paxos has become the primary vehicle on the Internet for achieving consensus.

Unfortunately, Paxos is quite difficult to understand and implement. Despite of numerous attempts to make it more understandable, Paxos requires a significant amount of effort to support practical systems. As a result, builders and users struggle with Paxos.

After struggling with Paxos ourselves, we found a new consensus algorithm that has the best of both worlds: it is as efficient as Paxos, but its stability rules are different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for various applications. In order to enhance scalability, Raft separates the key elements of cooperative protocols, such as leader election, log replication, and safety. Raft also includes a simple and intuitive algorithm for building a distributed system with a stronger degree of cohesion to recover from states that are not consistent. Results show that Raft is easier for developers to understand and use compared to Paxos.
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:
  - Maintaining group membership, distributed data structures, distributed locks, distributed protocol state, etc
ZooKeeper - Guarantees

- **Liveness guarantees**: if a majority of ZooKeeper servers are active and communicating the service will be available

- **Durability guarantees**: if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover