PRIMARY-BACKUP REPLICATION

George Porter
Nov 14, 2018
UC San Diego

ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Tanenbaum and Van Steen, Dist. Systems: Principles and Paradigms
  • Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
ANNOUNCEMENTS

Outline

1. Primary-backup replication
2. Safety and liveness
3. Two-phase commit
4. Two-phase commit failure scenarios
LIMITED FAULT TOLERANCE IN TOTALLY-ORDERED MULTICAST

• Stateful server replication for fault tolerance...
• But no story for server replacement upon a server failure → no replication

Today: Make stateful servers fault-tolerant?

PRIMARY-BACKUP: GOALS

• Mechanism: Replicate and separate servers

• Goal #1: Provide a highly reliable service
  • Despite some server and network failures
  • Continue operation after failure

• Goal #2: Servers should behave just like a single, more reliable server
STATE MACHINE REPLICATION

- Any server is essentially a state machine
- Set of (key, value) pairs is state
- Operations transition between states

- Need an op to be executed on all replicas, or none at all
- *i.e.*, we need distributed all-or-nothing atomicity
- If op is deterministic, replicas will end in same state

- Key assumption: Operations are deterministic

PRIMARY-BACKUP (P-B) APPROACH

- Nominate one server the primary, call the other the backup
- Clients send all operations (get, put) to current primary
- The primary orders clients’ operations
- Should be only one primary at a time

Need to keep clients, primary, and backup in sync: who is primary and who is backup
CHALLENGES

- Network and server failures

- Network partitions
  - Within each network partition, near-perfect communication between servers
  - Between network partitions, no communication between servers

PRIMARY-BACKUP (P-B) APPROACH

1. Primary logs the operation locally
2. Primary sends operation to backup and waits for ack
   - Backup performs or just adds it to its log
3. Primary performs op and acks to the client
   - After backup has applied the operation and ack’ed
**VIEW SERVER**

- A **view server** decides who is primary, who is backup
  - Clients and servers depend on view server
    - Don’t decide on their own (might not agree)

- Challenge in designing the view service:
  - Only want one primary at a time
  - Careful protocol design needed

- For now, **assume** view server **never fails**

**MONITORING SERVER LIVENESS**

- Each replica periodically **pings** the view server
  - View server declares replica **dead** if it missed N pings in a row
  - Considers the replica **alive** after a single ping

- **Can a replica be alive but declared “dead” by view server?**
  - Yes, in the case of network failure or partition
THE VIEW SERVER DECIDES THE CURRENT VIEW

- **View** = (view #, primary server, backup server)

  ![Diagram showing seven interactions between clients and servers]

  **Challenge:** All parties make their own local decision of the current view number

AGREEING ON THE CURRENT VIEW

- In general, any number of servers can ping view server

- Okay to have a view with a primary and **no backup**

- Want everyone to **agree** on the **view number**
  - Include the view # in RPCs between all parties
TRANSITIONING BETWEEN VIEWS

- **How to ensure new primary has up-to-date state?**
  - Only promote a previous backup
    - *i.e.*, don’t make a previously-idle server primary
  - Set liveness detection timeout > state transfer time
- **How does view server know whether backup is up to date?**
  - View server sends `view-change` message to all
  - Primary **must ack new view** once backup is up-to-date
  - View server stays with current view until ack
    - Even if primary has or appears to have failed

SPLIT BRAIN

[Diagram showing a split brain scenario with labeled nodes and edges representing different states and interactions between View Server, Client, S1, and S2.]
SERVER $S_2$ IN THE OLD VIEW

SERVER $S_2$ IN THE NEW VIEW
STATE TRANSFER VIA OPERATION LOG

• How does a new backup get the current state?
  • If $S_2$ is backup in view $i$ but was not in view $i-1$
  • $S_2$ asks primary to transfer the state
  • One alternative: transfer the entire operation log

Simple, but inefficient (operation log is long)

STATE TRANSFER VIA SNAPSHOT

• Every op must be either before or after state transfer
  • If op before transfer, transfer must reflect op
  • If op after transfer, primary forwards the op to the backup after the state transfer finishes

• If each client has only one RPC outstanding at a time, state = map + result of the last RPC from each client
  • (Had to save this anyway for “at most once” RPC)
SUMMARY OF RULES

1. View i's primary must have been primary/backup in view i−1

2. A non-backup must reject forwarded requests
   - Backup accepts forwarded requests only if they are in its idea of the current view

3. A non-primary must reject direct client requests

4. Every operation must be before or after state transfer

PRIMARY-BACKUP: SUMMARY

- First step in our goal of making stateful replicas fault-tolerant

- Allows replicas to provide continuous service despite persistent net and machine failure

- Finds repeated application in practical systems
Outline

1. Primary-backup replication
2. Safety and liveness
3. Two-phase commit
4. Two-phase commit failure scenarios

REASONING ABOUT FAULT TOLERANCE

• This is hard!
  • How do we design fault-tolerant systems?
  • How do we know if we’re successful?
• Often use “properties” that hold true for every possible execution
• We focus on safety and liveness properties
SAFETY

• “Bad things” don’t happen
  • No stopped or deadlocked states
  • No error states

• Examples
  • Mutual exclusion: two processes can’t be in a critical section at the same time
  • Bounded overtaking: if process 1 wants to enter a critical section, process 2 can enter at most once before process 1

LIVENESS

• “Good things” happen
  • ...eventually

• Examples
  • Starvation freedom: process 1 can eventually enter a critical section as long as process 2 terminates
  • Eventual consistency: if a value in an application doesn’t change, two servers will eventually agree on its value
OFTEN A TRADEOFF

• “Good” and “bad” are application-specific
• Safety is very important in banking transactions
  • May take some time to confirm a transaction
• Liveness is very important in social networking sites
  • See updates right away (what about the “breakup problem”?)

Outline

1. Primary-backup replication
2. Safety and liveness
3. Two-phase commit
4. Two-phase commit failure scenarios
MOTIVATION: SENDING MONEY

\[
\text{send\_money}(A, B, \text{amount}) \{ \\
\quad \text{Begin\_Transaction();} \\
\quad \text{if (A.balance - amount} \geq 0 \} \{ \\
\quad \quad \text{A.balance = A.balance - amount;} \\
\quad \quad \text{B.balance = B.balance + amount;} \\
\quad \quad \text{Commit\_Transaction();} \\
\quad \} \text{ else } \{ \\
\quad \quad \text{Abort\_Transaction();} \\
\quad \} \\
\}
\]

SINGLE-SERVER: ACID

- **Atomicity**: all parts of the transaction execute or none (A’s decreases and B’s balance increases)
- **Consistency**: the transaction only commits if it preserves invariants (A’s balance never goes below 0)
- **Isolation**: the transaction executes as if it executed by itself (even if C is accessing A’s account, that will not interfere with this transaction)
- **Durability**: the transaction’s effects are not lost after it executes (updates to the balances will remain forever)
DISTRIBUTED TRANSACTIONS?

- Partition databases across multiple machines for scalability (A and B might not share a server)
- A transaction might touch more than one partition
- How do we guarantee that all of the partitions commit the transactions or none commit the transactions?

TWO-PHASE COMMIT (2PC)

- **Goal**: General purpose, distributed agreement on some action, with failures
  - Different entities play different roles in the action
- **Running example**: Transfer money from A to B
  - Debit at A, credit at B, tell the client “okay”
  - Require **both** banks to do it, or **neither**
  - Require that **one bank never act alone**
STRAW MAN PROTOCOL

1. **C → TC: “go!”**

   - **Client C**
   - **Transaction Coordinator TC**
   - **Bank**
   - **A**
   - **B**

2. **TC → A: “debit $20!”**
   - **TC → B: “credit $20!”**
   - **TC → C: “okay”**

   - A, B perform actions on receipt of messages
REASONING ABOUT THE STRAW MAN PROTOCOL

What could possibly go wrong?

1. Not enough money in A's bank account?
2. B's bank account no longer exists?
3. A or B crashes before receiving message?
4. The best-effort network to B fails?
5. TC crashes after it sends debit to A but before sending to B?

SAFETY VERSUS LIVENESS

• Note that TC, A, and B each have a notion of committing
• We want two properties:

1. Safety
   • If one commits, no one aborts
   • If one aborts, no one commits

2. Liveness
   • If no failures and A and B can commit, action commits
   • If failures, reach a conclusion ASAP
A CORRECT ATOMIC COMMIT PROTOCOL

1. **C ↔ TC:** “go!”

![Diagram](image1)

2. **TC → A, B:** “prepare!”

![Diagram](image2)
**A CORRECT ATOMIC COMMIT PROTOCOL**

1. \( C \to TC: \text{“go!”} \)

2. \( TC \to A, B: \text{“prepare!”} \)

3. \( A, B \to P: \text{“yes” or “no”} \)

4. \( TC \to A, B: \text{“commit!” or “abort!”} \)
   - TC sends commit if both say yes
   - TC sends abort if either say no
A CORRECT ATOMIC COMMIT PROTOCOL

1. C → TC: “go!”
2. TC → A, B: “prepare!”
3. A, B → P: “yes” or “no”
4. TC → A, B: “commit!” or “abort!”
   - TC sends commit if both say yes
   - TC sends abort if either say no
5. TC → C: “okay” or “failed”
   - A, B commit on receipt of commit message

REASONING ABOUT ATOMIC COMMIT

• Why is this correct?
  • Neither can commit unless both agreed to commit
• What about performance?
  1. Timeout: I’m up, but didn’t receive a message I expected
     • Maybe other node crashed, maybe network broken
  2. Reboot: Node crashed, is rebooting, must clean up
TIMEOUTS IN ATOMIC COMMIT

Where do hosts wait for messages?

1. **TC** waits for “yes” or “no” from A and B
   - **TC** hasn’t yet sent any commit messages, so can safely abort after a timeout
   - But this is **conservative**: might be network problem
     - We’ve preserved correctness, sacrificed performance

2. A and B wait for “commit” or “abort” from **TC**
   - If it sent a no, it can safely abort *(why?)*
   - If it sent a yes, can it unilaterally abort?
   - Can it unilaterally commit?
   - A, B could wait forever, but there is an alternative...

SERVER TERMINATION PROTOCOL

- Consider Server B (Server A case is symmetric) waiting for commit or abort from **TC**
  - Assume B voted yes (else, unilateral abort possible)
- **B → A**: “status?” A then replies back to B. Four cases:
  - (No reply from A): no decision, B waits for **TC**
  - Server A received commit or abort from **TC**: Agree with the **TC**’s decision
  - Server A hasn’t voted yet or voted no: both abort
    - **TC** can’t have decided to commit
  - Server A voted yes: both must wait for the **TC**
    - **TC** decided to commit if both replies received
    - **TC** decided to abort if it timed out
REASONING ABOUT THE SERVER TERMINATION PROTOCOL

- **What are the liveness and safety properties?**
  - **Safety**: if servers don’t crash, all processes will reach the same decision
  - **Liveness**: if failures are eventually repaired, then every participant will eventually reach a decision
- Can resolve some timeout situations with guaranteed correctness
- Sometimes however A and B must block
  - Due to failure of the TC or network to the TC
- But what will happen if TC, A, or B crash and reboot?

HOW TO HANDLE CRASH AND REBOOT?

- Can’t back out of commit if already decided
  - **TC** crashes just after sending “commit!”
  - **A** or **B** crash just after sending “yes”
- If all nodes knew their state before crash, we could use the termination protocol...
  - Use **write-ahead log** to record “commit!” and “yes” to disk
RECOVERY PROTOCOL WITH NON-VOLATILE STATE

- If everyone rebooted and is reachable, TC can just check for commit record on disk and resend action
- TC: If no commit record on disk, abort
  - You didn’t send any “commit!” messages
- A, B: If no yes record on disk, abort
  - You didn’t vote “yes” so TC couldn’t have committed
- A, B: If yes record on disk, execute termination protocol
  - This might block

TWO-PHASE COMMIT

- This recovery protocol with non-volatile logging is called Two-Phase Commit (2PC)
- Safety: All hosts that decide reach the same decision
  - No commit unless everyone says “yes”
- Liveness: If no failures and all say “yes” then commit
  - But if failures then 2PC might block
  - TC must be up to decide
- Doesn’t tolerate faults well: must wait for repair
Outline

1. Primary-backup replication
2. Safety and liveness
3. Two-phase commit
4. Two-phase commit failure scenarios

WHAT IF PARTICIPANT FAILS BEFORE SENDING RESPONSE?
WHAT IF PARTICIPANT FAILS AFTER SENDING VOTE

WHAT IF PARTICIPANT LOST A VOTE?
WHAT IF COORDINATOR FAILS BEFORE_SENDING PREPARE?

WHAT IF COORDINATOR FAILS AFTER_SENDING PREPARE?
WHAT IF COORDINATOR FAILS AFTER RECEIVING VOTES

WHAT IF COORDINATOR FAILS AFTER SENDING DECISION?
DO WE NEED THE COORDINATOR?

UC San Diego