Failures, Elections, and Raft
We saw the example of a bank with two branches last class, and on the importance of their applying the same operations in the same order to their databases.

This generalizes to a problem known as consensus, which is tricky to achieve when we allow processes to fail.
Failures

- Omission failures
  - something doesn’t happen
    - process crashes
    - data lost in transmission
    - etc.
- Byzantine (arbitrary) failures
  - something bad happens
    - message is modified
    - message received twice
    - any kind of behavior, including malicious
- Timing failures
  - something takes too long

This material is from Section 2.4.2 of the text.
These definitions are abstracted from the textbook, page 64.
Detecting Crashes

- Synchronous systems
  - timeouts
- Asynchronous systems
  - ?
- Fail-stop
  - an oracle lets us know
We saw the example of a bank with two branches last class, and on the importance of their applying the same operations in the same order to their databases.
This generalizes to a problem known as consensus, which is tricky to achieve when we allow processes to fail.
Consensus

• Setting: a group of processes, some correct, some faulty
• Two primitives, \(propose(v)\) and \(decide(v)\)
• If each correct process proposes a value:
  – Termination: every correct process eventually decides on a value
  – Agreement: if a correct process decides \(v\), then all processes eventually decide \(v\)
  – Integrity: if a correct process decides \(v\), then \(v\) was previously proposed by some process

Informally: think of this as voting for a value
Consensus

• Variations on the problem, depending on assumptions
  – Synchronous vs asynchronous
  – Fail-stop vs crash/omission failures vs Byzantine failures

• Equivalent to reliable, totally- and causally-ordered broadcast (Atomic broadcast)
Byzantine Agreement Problem

- Will cover in a future lecture
Impossibility of Consensus

- There is no deterministic protocol that solves consensus in a message-passing asynchronous system in which at most one process may fail by crashing.
  - Solve means to satisfy safety and liveness
  - Famous result from Fisher, Lynch, Paterson (FLP), 1985
- There is hope, though
  - Introducing (some) synchrony – timeouts
  - Introducing randomization
  - Introducing failure detectors

The primary replica’s function is too important to be trusted to a single server. Thus all its actions must be agreed upon by a set of servers known as the *inner ring.*
State-Machine Replication

- An approach to dealing with failures by replication
- A data-structure with deterministic operations replicated among servers
- State consistent if every server sees the same sequence of operations
  - Multiple rounds of consensus
- Common algorithms (non-Byzantine):
  - Paxos [Lamport], Viewstamped Replication [Oki and Liskov], Zab [Junqueira et al.], Raft [Ongaro and Ousterhout]
Raft
Raft

- Proposed by Ongaro and Ousterhout in 2014
- Four components
  - Leader election
  - Log replication
  - Safety
  - Membership changes
- Assumes crash failures
- No dependency on time for safety
  - But depends on time for availability
Server States

- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - Follower: completely passive (issues no RPCs, responds to incoming RPCs)
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers
Terms

• Time divided into terms:
  – Election
  – Normal operation under a single leader
• At most 1 leader per term
• Some terms have no leader (failed election)
• Each server maintains current term value
• Key role of terms: identify obsolete information
Heartbeats and Timeouts

- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority
- If electionTimeout elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms
Election Basics

- Increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs to all other servers, retry until either:
  1. Receive votes from majority of servers:
     - Become leader
     - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election
Elections, cont’d

- Safety: allow at most one winner per term
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can’t accumulate majorities in same term

- Liveness: some candidate must eventually win
  - Choose election timeouts **randomly** in \([T, 2T]\)
  - One server usually times out and wins election before others wake up
  - Works well if \(T >>\) broadcast time

B can’t also get majority

.servers

Voted for candidate A
Entries committed when leader hears back from majority of followers.
Life as a Leader

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once new entry committed:
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- Crashed/slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers
Log Consistency Invariants

- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>div</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
</tr>
</tbody>
</table>

- If a given entry is committed, all preceding entries are also committed
AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request
- Implements an induction step, ensures coherency

---

**leader**

1. add
2. cmp
3. ret
4. mov
5. jmp

**follower**

1. add
2. cmp
3. ret
4. mov

---

**AppendEntries succeeds:** matching entry

**leader**

1. add
2. cmp
3. ret
4. mov
5. jmp

**follower**

1. add
2. cmp
3. ret
4. shi

---

**AppendEntries fails:** mismatch
Leader Changes

- At beginning of new leader’s term:
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - Leader’s log is “the truth”
  - Will eventually make follower’s logs identical to leader’s
  - Multiple crashes can leave many extraneous log entries:

<table>
<thead>
<tr>
<th>log index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S₂</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>S₃</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S₄</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S₅</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Safety Requirement

Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry

- Raft safety property:
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders

- This guarantees the safety requirement
  - Leaders never overwrite entries in their logs
  - Only entries in the leader’s log can be committed
  - Entries must be committed before applying to state machine

Committed → Present in future leaders’ logs

Restrictions on commitment → Restrictions on leader election
Picking the Best Leader

- Followers can’t tell which entries are committed!

  
  ![Committed and Unavailable Entries Diagram]

- During elections, choose candidate with log most likely to contain all committed entries
  
  - Candidates C include log info in RequestVote RPCs (index & term of last log entry)

  - Voting server V **doesn’t vote** if its log is “more complete”:
    
    \[
    \text{(lastTerm}_v \geq \text{lastTerm}_C) \lor \text{(lastTerm}_v = \text{lastTerm}_C) \land (\text{lastIndex}_v > \text{lastIndex}_C)
    \]

  - Leader will have “most complete” log among electing majority
Safety criterion: can only commit entries which are guaranteed to be in the log of ANY possible future leader. In this case, because the leader is trying to commit an entry that is in the current term, it means that if the entry gets replicated to a majority of nodes, then no other leader can be elected for a new term that won't have this entry. In the example, S5 wouldn't be elected because no one would vote for it (its term is still 1), and S4 would also not be elected (its log is shorter).
S1 was leader, crashed before replicating entry 3 to s3. s5 was elected with votes from s3, s4, s5, then crashed.
S1 came back (was elected by 1,2,3) and accepted 4. Now it tries to replicate 2. But this is not safe to commit, because s5 could be elected and overwrite entry 3.

You don’t have the same guarantee as in the previous slide because S5’s term is higher than that of a majority of nodes. Since the leader is trying to commit an entry from an earlier term, it can’t guarantee that no electable leader would not have the entry in question.
New Commitment Rules

- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - At least one new entry from leader’s term must also be stored on majority of servers

- Once entry 4 committed:
  - s₅ cannot be elected leader for term 5
  - Entries 3 and 4 both safe

Combination of election rules and commitment rules makes Raft safe
Log Inconsistencies

Leader changes can result in log inconsistencies:

<table>
<thead>
<tr>
<th>log index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader for term 8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

(a) 1 1 1 4 4 5 5 6 6 6
(b) 1 1 1 4 6
(c) 1 1 1 4 4 5 5 6 6 6 6
(d) 1 1 1 4 4 5 5 6 6 6 7 7
(e) 1 1 1 4 4 4 4
(f) 1 1 1 2 2 2 3 3 3 3 3

- Missing Entries
- Extraneous Entries
Repairing Follower Logs

- New leader must make follower logs consistent with its own
  - Delete extraneous entries
  - Fill in missing entries
- Leader keeps nextIndex for each follower:
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader’s last index)
- When AppendEntries consistency check fails, decrement nextIndex and try again:

```
<table>
<thead>
<tr>
<th>log index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader for term 7</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>followers (a)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>followers (b)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Repairing Logs, cont’d

- When follower overwrites inconsistent entry, it deletes all subsequent entries:

<table>
<thead>
<tr>
<th>log index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader for term 7</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>follower (before)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>follower (after)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Neutralizing Old Leaders

- Deposited leader may not be dead:
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries

- **Terms** used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - If sender's term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term, then processes RPC normally

- Election updates terms of majority of servers
  - Deposited server cannot commit new log entries
Some details of the client protocol are only in Diego Ongaro's PhD thesis, not in the Raft tech report.
Client Protocol, cont’d

- What if leader crashes after executing command, but before responding?
  - Must not execute command twice
- Solution: client embeds a unique id in each command
  - Server includes id in log entry
  - Before accepting command, leader checks its log for entry with that id
  - If id found in log, ignore new command, return response from old command
- Result: exactly-once semantics as long as client doesn’t crash
- Enforces linearizability (will see in upcoming lecture)
Partitioned Leader

- A client talking to a partitioned leader could be delayed forever.
  - Solution: leader will step down after a number of rounds of heartbeats with no response from majority
What if clients can crash?

- Servers maintain a session for each client
  - Keep track of latest sequence number processed for client, and response
- Generalizes for multiple outstanding requests
  - Server keeps set of <seq, resp> for client
  - Client includes with request lowest seq with no response
  - Server can discard smaller sequence numbers
- Must expire sessions
  - All replicas must agree on when to do this
  - Raft uses leader timestamp, committed to log
Alive clients with expired sessions

- How to distinguish between client which exited from client which just took too long?

- Require clients to register with the leader when starting a session
  - RegisterClient RPC
  - Leader returns unique ID to the client
  - Client uses this ID in subsequent request

- If server receives request for non-existing session...
  - Return an error. Current implementation crashes the client, forcing restart
Configuration Changes

Cannot switch directly from one configuration to another: conflicting majorities could arise

See the paper for details

- Majority of $C_{\text{old}}$
- Majority of $C_{\text{new}}$