Logistics

• MP1 is due today.

• HW2 is due on Wednesday.

• MP2 has been released.

• HW3 has been released.
Agenda for today

• Consensus
  • Consensus in synchronous systems
    • Chapter 15.4
  • Impossibility of consensus in asynchronous systems
    • We will not cover the proof in details
  • Good enough consensus algorithm for asynchronous systems:
    • Paxos made simple, Leslie Lamport, 2001
• Other forms of consensus algorithm
  • Raft (log-based consensus)
  • Block-chains (distributed consensus)
Raft: A Consensus Algorithm for Replicated Logs

Slides from Diego Ongaro and John Ousterhout, Stanford University
Goal: Replicated Log

- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up
- Failure model: fail-stop (not Byzantine), delayed/lost messages
Raft Overview

1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader

2. Neutralizing old leaders

3. Normal operation (basic log replication)

4. Safety and consistency after leader changes
Raft Overview

1. Leader election:
   • Select one of the servers to act as leader
   • Detect crashes, choose new leader

2. Neutralizing old leaders

3. Normal operation (basic log replication)

4. Safety and consistency after leader changes
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 (requests), responds to incoming RPCs
  - **Candidate**: used to elect a new leader

- Normal operation: 1 leader, N-1 followers
Quick Detour: RPCs

- Raft servers communicate via RPCs.
- What are RPCs?
  - Remote Procedure Calls: procedure call between functions on different processes
  - Convenient programming abstraction.

```
P2.call("foo", args, reply)
```

```
P1

1. "foo", args

2. foo(args) {
   ....
   ....
   return reply
}

3. reply

P2
```
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 promotes itself to candidate and starts new election
  • Timeouts typically in range 100-500ms
    • Randomly chosen in some range to reduce probability of split election.
Election Basics

• On timeout:
  • Increment current term
  • Change to Candidate state
  • Vote for self
  • Send RequestVote RPCs to all other servers:
    1. Receive votes from majority of servers:
       • Become leader
       • Send AppendEntries heartbeats (RPCs) periodically to all other servers
    2. Receive RPC from valid leader (with same or higher term):
       • Return to follower state
    3. No-one wins election (election timeout elapses):
       • Increment term, start new election
State Diagram Revisit

Follower
- start
- timeout, start election
- discover current server or higher term
- "step down"

Candidate
- timeout, new election
- receive votes from majority of servers

Leader
- discover server with higher term
Election Basics: handling RequestVote RPCs

• Suppose a server in term `currentTerm` has voted for process with id `votedFor` in that term.
• When it receives RequestVote RPC from process `candidateId` with term `voteRequestTerm`:
  
  If `voteRequestTerm < currentTerm`
  
  reply false
  
  return.

  If `voteRequestTerm > currentTerm`
  
  `currentTerm = voteRequestTerm, votedFor = null`

If `(votedFor is null or candidateId)`*

  //should not have voted for anyone else in that term

Grant vote, `votedFor = candidateId`

*we will extend on this condition later.
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 \([T, kT]\)
  - One server usually times out and wins election before others wake up
  - Works well if \(T \gg \) broadcast time

*Safety is guaranteed. Liveness is not guaranteed.*
Implication of terms

• Each term has at most one leader (safety condition).

• Terms always increase with time.

• If the latest term has an elected leader, majority of processes must have updated themselves to the latest term.

• Only the leader of the latest term can commit log entries (we will discuss this next).
Raft Overview

1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader

2. Neutralizing old leaders

3. Normal operation (basic log replication)

4. Safety and consistency after leader changes
Neutralizing Old Leaders

- Deposed 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
  - Deposed server cannot commit new log entries
Raft Overview

1. Leader election:
   • Select one of the servers to act as leader
   • Detect crashes, choose new leader

2. Neutralizing old leaders

3. Normal operation (basic log replication)

4. Safety and consistency after leader changes
Log entry = index, term, command

Log stored on stable storage (disk); survives crashes

Entry is **committed** by the leader when certain conditions are met*.  
  - Durable, will eventually be executed by state machines  
  - * we will get back to this.

### Log Structure

<table>
<thead>
<tr>
<th></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>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>div</td>
<td>shl</td>
<td>sub</td>
</tr>
<tr>
<td>S2</td>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>div</td>
<td>shl</td>
<td>sub</td>
</tr>
<tr>
<td>S3</td>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>div</td>
<td>shl</td>
<td>sub</td>
</tr>
<tr>
<td>S4</td>
<td>add</td>
<td>cmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>div</td>
<td>shl</td>
<td></td>
</tr>
</tbody>
</table>

---

* log index
  - leader

---

* followers
  - committed entries

---

**Term**

---

**Command**

---
Normal Operation

• Client sends command to leader
• Leader appends command to its log (*not yet committed*)
• Leader sends AppendEntries RPCs to followers
• Once new entry committed* (we will discuss when and how):
  • 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

High level of coherency between logs:

Raft guarantees that:

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

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

• Each AppendEntries RPC contains index and term of entry preceding new ones

• Follower must contain matching entry; otherwise it rejects request

• Implements an induction step, ensures coherency

### AppendEntries succeeds: matching entry

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

### AppendEntries fails: mismatch

<table>
<thead>
<tr>
<th>leader</th>
<th>follower</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 add</td>
<td>1 add</td>
</tr>
<tr>
<td>1 cmp</td>
<td>1 cmp</td>
</tr>
<tr>
<td>1 ret</td>
<td>1 ret</td>
</tr>
<tr>
<td>2 mov</td>
<td>1 shl</td>
</tr>
<tr>
<td>3 jmp</td>
<td></td>
</tr>
</tbody>
</table>
**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
    - *Unless a new leader gets elected during the process.*
  - 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><strong>s₁</strong></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><strong>s₂</strong></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><strong>s₃</strong></td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s₄</strong></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s₅</strong></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
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>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

(a)  
(b)  
(c)  
(d)  
(e)  
(f)  

possible followers

Missing Entries

Extraneous Entries
Repairing Follower Logs

• New leader must make follower logs consistent with its own
  • Delete extraneous entries
  • Fill in missing entries

• Next class:
  • How leader repairs logs?
  • How does Raft guarantee safety of consensus and when can a log entry be committed?
MP2: Raft Leader Election and Log Consensus

- Lead TA: Jiangran Wang

- Objective:
  - Implement a leader-based consensus protocol for replicated state machine, that maintains log consensus even when nodes crash or get temporarily disconnected.

- Task:
  - Beef up a skeleton code provided to you to implement Raft leader election and log consensus.
  - We provide an emulation framework and a test suite.
  - Strive to pass all the test cases provided in our test suite.
MP2: Logistics

• Due on April 5th.
  • Late policy: Can use part of your 168 hours of grace period accounted per student over the entire semester.

• Must be implemented in Go.
  • The framework we provide is in Go.

• Read the specification and the comments in the provided code carefully.

• Start early!!
  • MP2 is harder than MP1.