Consensus and related problems

To do ...

- Failure
- Consensus and related problems
- Raft
Consensus and failures

- How to make process agree on a value after one or more have proposed what the value should be?

  ... reach **consensus** even if some processes may fail
  - Failure – System cannot meet its promises
  - Error – Part of system’s state that can lead to a failure

- Many ways to fail ...
Failure models

- **Crash failures**
  - Simply halts, but behaves correctly before halting
  - Fail-stop, crash failures and others can tell

- **Omission failures**
  - ... fails to respond to incoming requests, data lost ...

- **Timing failures**
  - Takes too long, > a specified real-time interval

- **Response failures**
  - Output is incorrect

- **Byzantine (arbitrary) failures**
  - Anything, arbitrary output, arbitrary timing failures
Failure ➔ Redundancy

- Redundancy – basic approach to masking faults
- Information
  - Add extra bits to a message for reconstruction
- Time
  - Do something more than once if needed
- Physical
  - Add copies of software/hardware
Consensus – Definition

- A group of $N$ processes, to reach consensus
  - Every process $p_i$ begins *undecided* and *proposes* a value $v_i$
  - Processes exchange values
  - Each sets the value of a *decision variable* $d_i$, entering the *decided* state

- Requirements
  - Termination – All correct processes eventually decide
  - Agreement – If a correct process decides $v$, all correct processes decide the same
  - Integrity – If the correct processes all proposed $v$, then any correct process in the decided state has chosen $v$
Dreamland solution – Just as illustration

- A system where processes cannot fail

Each of $N$ process $p_i$ R-multicasts its proposed value to $g$
Every process $p_j$
  Collects all $N$ values (including its own)
  Evaluates $a = \text{majority}(v_i \text{ for } i: 1..N)$, returns $a$ or $NIL$ if no majority exists

- Could be something different than majority, e.g., min or max

- Requirements
  - Termination guaranteed by the reliability of multicast
  - Agreement and integrity by the definition of $\text{majority}$ and the integrity property of reliable multicast
  - (every process receives the same set of values and runs the same function, so they must agree on the same value)
Consensus

- Variations on the problem, depending on assumptions
  - Synchronous or asynchronous system
  - Fail-stop or Byzantine failures

- We have seen protocols tailored for individual types of consensus/agreements
  - Which process can enter the critical section
  - Who is the leader
  - What’s the order of messages
Consensus and related problems

- Besides consensus …
- Byzantine general problem
  - 3+ generals need to agree to attack or retreat
  - Commander issues an order, lieutenants decide what to do
  - One of the generals may be “treacherous”
  - Slightly different integrity: if commander is correct, all decide on the value proposed by commander
Consensus and related problems

- Interactive consistency
  - All processes agree on a *vector* of values, one per process
  - Similar requirements, e.g. for integrity: if $p_i$ is correct, all correct processes decide on $v_i$ as the *i*-th entry of its vector

- Consensus, Byzantine generals, Interactive consistency
  - All can be define in the presence of crash or arbitrary failures and for synchronous or asynchronous systems
  - It is possible to derive a solution to one problem using a solution for another
Byzantine general problem

- If the commander is a traitor
  - Can propose attack to one lieutenant and retreat to another
- If the lieutenant is a traitor,
  - Can tell one lieutenant that the commander said attack and tell another that he said retreat

- A variant of consensus with a distinguished process proposing a value
  - Integrity – if the commander is correct, all correct processes agree with what the commander proposed
Byzantine general problem

- Assumptions
  - Synchronous system: Correct processes can detect absence of a message (timeout), but can’t conclude the sender has crashed
  - Arbitrary/Byzantine failures: A faulty process can send any message with any value at any time (or omit to send)
  - The communication channel between processes are private

Impossibility with three processes

If $L_1$ has to decide to accept $v$ on the first example, has to choose $w$ in the second one

- Lamport et al. solution solves BGP for $3m+1$ or more generals in the presence of at most $m$ traitors
Algorithm $BG(\theta)$

1. The commander (C) sends value to every lieutenant.
2. Lieutenants use the value received from C or RETREAT if they received no value.

Algorithm $BG(m), m>0$

1. C sends value to every lieutenant.
2. For each $i$, let $v_i$ be the value Lieutenant $i$ receives from C or RETREAT. Lieutenant $i$ acts as C in algorithm $BG(m-1)$ to send the value $v_i$ to each of the $n-2$ other lieutenants.
3. For each $i$, and each $j \neq i$, let $v_j$ be the value Lieutenant $i$ received from Lieutenant $j$ in step (2) (using algorithm $BG(m-1)$), or else RETREAT. Lieutenant $i$ uses the value $\text{majority}(v_1, \ldots, v_{n-1})$. 
A run with $N \geq 4$ and $f = 1$

- **A lieutenant is the traitor**
  - $L1$: $\text{majority}(v,v,x) = v$
  - $L2$: $\text{majority}(v,v,y) = v$

- **The commander is the traitor**
  - $L1$: $\text{majority}(v,w,z) = \text{NIL}$
  - $L2$: $\text{majority}(v,w,z) = \text{NIL}$
  - $L3$: $\text{majority}(v,w,z) = \text{NIL}$
Consensus and related problems

- It is possible to derive a solution to one problem using a solution to another
  - Consensus based on Interactive consistency, IC based on Byzantine general, ...
- E.g. suppose there’s a solution to Byzantine generals
  - $BG_i(j, v)$ returns decision value of $p_i$ in a run of the solution to Byzantine generals where commander $p_j$, proposed value $v$
  - $IC_i(v_1, v_2, ..., v_N)[j]$ returns the $j$th value in decision vector of $p_i$ in a run of the solution to interactive consistency
- IC from BG – by running BG $N$ times, one per process
  - $IC_i(v_1, v_2, ..., v_N)[j] = BG_i(j, v_j)$ ($i, j = 1 \ldots N$)
- ...

...
Impossibility of consensus

- But what if the system is asynchronous?
- No algorithm can guarantee to reach consensus with even one faulty (by crashing)
  - Famous result from Fischer, Lynch, Paterson (FLP), 1985
  - Basic idea: there’s always a continuation of a processes’ execution that avoids reaching consensus (*why? Can’t tell if a process is running slow or is dead*)
  - Guarantee – it is possible, just not guaranteed, …

- How do we work around this?
  - Masking faults: process keep data in persistent storage so that they can restart after crashing (so it just seems slow)
  - Using perfect *by design* failure detectors – processes agree to a maximum response time (otherwise, the process has failed)
  - …
Replicated state-machines

- Consensus typically appears in the context of replicated state machines
  - State machines (SM) on a collection of servers
  - A data-structure with deterministic operations replicated among servers
  - State consistent if every server sees the same sequence of operations

- Common algorithms (non-Byzantine)
  - Paxos [Lamport], Viewstamped replication [Oki, Liskov], Raft [Ongaro, Ousterhout]
• An algorithm for managing a replicated log
  – Designed for understandability – making Paxos easier

• Two general approaches to consensus
  – Symmetric, leader-less
    • All servers are equal
    • Client contact any server
  – Asymmetric, leader-based - Raft
    • At any given point in time, one in charge
    • Clients communicate with the leader
    • our key components

• Assumes crash failures

• No dependency on times for safety
  – Yes for availability

*Partially based on the authors’ slides
Raft overview

- Relies on a distinguished leader
  - With full responsibility for managing the replicated log
  - Leader accepts log entries from clients
  - Replicates them on other servers
  - Tell servers when it is safe to apply log entries

- Raft decomposes the consensus problem
  - Leader election – Choose a new one when the existing one fails
  - Log replication – Accept log entries from clients and replicate
  - Safety – key safety property, SM safety
    - If any server has applied a particular log entry to its SM, no other server may apply a different command for the same log index
Back in 5’
1. Leader election
2. Normal operation
3. Safety and consistency
4. Neutralize old leaders
5. Client protocol
6. Configuration changes
A Raft cluster contains several servers

At any given time, each server is either

- Leader – handles client interaction, log replication, <1
- Follower – completely passive, only responds to incoming RPCs, doesn’t issue RPCs itself
- Candidate – leader wannabe

Normal state – 1 leader, N-1 followers
- Time split in terms (acting as logical clocks)
  - Election + normal operation under a single leader
- At most one leader per term
- Some terms have no leader (failed election)
- Each server maintains the current term value
- Key roles of terms: identify obsolete information
RPCs, heartbeats and timeouts

- Servers start as followers
- Followers receive RPCs from leaders or candidates
  - Two RPCs overall, `AppendEntries` and `RequestVote`
  - Both idempotent
- Leaders must send heartbeats (empty `AppendEntries` RPCs) to maintain authority
- If `electionTimeout` passes with no RPCs
  - Follower assumes leader has crashed
  - … starts a new election, putting itself up as candidate
Leader election

- Increment current term
- Become a candidate and vote for self
- Send RequestVote RPC to other servers, retrying until either
  1. Receive vote from majority
     - Becoming the leader
     - Send AppendEntries heartbeats to all others
  2. Receive RPC from valid leader
     - Return to being a follower
  3. No-one wins election (timeout)
     - Increment term, start new election
Election – Safety and liveness

- Safety – allow at most one winner per term
  - Each server gives out only one vote per term
    - Vote is made persistent on disk
  - Thus, two different candidates can’t get majority in same term

- Liveness – some candidate must eventually win
  - In principle, you could see repeated split votes …
  - Serves choose election timeouts randomly within [T, 2T]
  - One server usually times out and wins election before other noticed the absence
  - Works well election timeout, $T \gg$ broadcast time
Normal operation

- Clients sends command to the leader
- Leader appends commands to its log
- … sends \textit{AppendEntries} RPCs to followers
- Once new entry committed
  - Leader passes command to its SM, return result to client
  - … notifies followers of committed entries in subsequent \textit{AppendEntries} RPCs
  - Followers pass committed commands to their state machine
- Crashed/slow followers?
  - Leader retries RPC
- Performance is optimal in common case
  - One successful RPC to any majority of servers
- Log structure: every log entry \{index, term, command\}
- Log stored on stable storage (disk) to survive crashes
- Entry committed if stored on majority of servers
  - Durable, will eventually execute by state machines
Log consistency

- If two entries in different logs have the same index and term, they store the same command.
- If two entries in different logs have the same index and terms, the logs are identical in all preceding entries.
AppendEntries consistency check

- Each *AppendEntries* RPC contains index, term of entry preceding new one
- Follower must contain matching entry, else reject request
- Implements an induction step that ensure coherency

![Diagram showing AppendEntries process]

**Leader**
- 1 add
- 1 add
- 1 add
- 2 move
- 3 jmp

**Followers**
- 1 add
- 1 add
- 1 add
- 2 move

AppendEntries succeeds

AppendEntries fails
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 (a bit narrower)
  - If a leader has decided that a log entry is committed, that entry will be present in the log of all future leaders.

- This guarantees the safety requirement
  - Leader don’t overwrite or delete entries
  - Only entries in the leader’s log can be committed
  - Entry must be committed before applying to state machine
### Picking the best leader

- **Can’t tell which entries are committed**
  - Old leader knows but its log is unavailable during transition!

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>move</td>
<td>move</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>move</td>
<td>move</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>move</td>
<td>move</td>
</tr>
</tbody>
</table>

**Committed?**

- **During election, choose candidate with log most likely to contain all committed entries**
  - Candidates include log info in `RequestVote` RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is more complete
    
    $$(\text{lastTerm}_v > \text{lastTerm}_c) \lor (\text{lastTerm}_v = \text{lastTerm}_c) \land (\text{lastIndex}_v > \text{lastIndex}_c)$$
  - Leader will have most complete log among electing majority

**Unavailable during transition**
Committing entry from current term

- **Case #1/2:** Leader decides entry in current term is committed

  ![Diagram]

  - Leader in term 2
  - AppendEntry just succeeded
  - So entry 4 is committed
  - Cant’ be elected as leader for term 3

- **Safe:** leader for term 3 must contain entry 4
Committing entry from earlier term

- Case #2/2: Leader is trying to finish committing entry from an earlier term

- Entry 3 not safely committed:
  - $s_5$ can be elected as leader for term 5
  - If elected, it will overwrite entry 3 on $s_1$, $s_2$, and $s_3$!

- Need a new rule for commitment
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 the leader’s term must also be stored on majority of servers*

- Once entry 4 committed:
  - $s_5$ cannot be elected leader for term 5
  - Entries 3 and 4 both safe

- Combination of election rules and commitment rules makes Raft safe
Leader changes

- Leader crashes can leave the log inconsistent
  - Inconsistencies can be compound over a series of leader and follower crashes
  - Missing and extraneous entries in a log may span multiple terms

- Raft handling of inconsistencies
  - No special steps for new leader
  - Leader’s log is the truth
  - Will eventually makes follower’s log identical to its own
Leader changes can result in log inconsistencies
**Repairing follower logs**

- New leader makes 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 \((1 + \text{leader's last index})\)
- **When `AppendEntries`** consistency check fails, decrement `nextIndex` and try again
- **When follower overwrites inconsistent entry, it deletes all subsequent entries**

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
\text{Leader} & 1 \text{ add} & 1 \text{ add} & 1 \text{ add} & 4 \text{ sub} & 4 \text{ sub} & 5 \text{ add} & 5 \text{ add} & 6 \text{ jmp} & 6 \text{ jmp} & 6 \text{ jmp} \\
\text{Follower log before} & 1 \text{ add} & 1 \text{ add} & 1 \text{ add} & 2 \text{ move} & 2 \text{ move} & 2 \text{ move} & 3 \text{ jmp} & 3 \text{ jmp} & 3 \text{ jmp} & 3 \text{ jmp} & 3 \text{ jmp} \\
\text{And after} & 1 \text{ add} & 1 \text{ add} & 1 \text{ add} & 4 \text{ sub} \\
\end{array}
\]
Neutralizing old leaders

- Deposed leader may not be dead
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader reconnects and 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 (as a good follower)

- Election process updates terms of majority of servers
  - Candidate includes its own term in its request, everybody updates
  - So after election, deposed server cannot commit new log entries
Client protocol

- Send commands to leader
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader

- Leader does not respond until command has been logged, committed, and executed by leader’s state machine

- If request times out (e.g., leader crash):
  - Client reissues command to some other server
  - Eventually redirected to new leader
  - Retry request with new leader
Client protocol

- 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 an entry with that id
  - If id is in log, ignore new command, return response from old command

- Result: exactly-once semantics as long as client doesn’t crash
Finally, configuration changes

- Until now, system configuration was considered fixed
  - Determines what constitutes a majority
- Consensus mech must support configuration changes
  - Replace failed machine
  - Change degree of replication
- Cannot switch directly from one configuration to another: conflicting majorities could arise

![Diagram showing two disjoint majorities](image-url)
Two-phase change – Joint consensus

- Intermediate phase uses joint consensus
  - Need majority of old and new config for elections, commitment
- Config change is just a log entry; applied immediately on receipt (committed or not)
- Once joint consensus is committed, begin replicating log entry for final configuration
Joint consensus, additional details

- Any server from either configuration can serve as leader.
- If current leader is not in $C_{\text{new}}$, must step down once $C_{\text{new}}$ is committed.

![Diagram showing the transition between configurations and the timeline of events.](image-url)
Summary

- How to make process agree on a value after one or more have proposed what the value should be?
  - Consensus
- In synchronous and asynchronous systems
- Not just theory – Chubby, Zookeeper, …
- Several Raft implementations out there (and in Go)
- A good demo of Raft

http://thesecretlivesofdata.com/raft/