Snapshot Terminology

• A process’s state -> memory region
  – E.g., Variables used and defined

• Channel -> connection between two processes
  – Each channel is unidirectional
  – One channel from P1 to P2 and a different one from P2 to P1
  – Channel deliver messages in FIFO order
  – Channel does not lose messages
  – Channel does not fail

• Snapshot are done one at a time
  – Initiated by a global and external entity
Snapshot Rules

• Marker receiving rule for process $p_i$
  On $p_i$’s receipt of a marker message over channel $c$:
  \[ \text{if } (p_i \text{ has not yet recorded its state}) \]
  it records its state
  it records the state of $c$ as the empty sequence
  it turns on recording of messages arriving over other channels

  \[ \text{else} \]
  $p_i$ records the state of $c$ as the set of messages it has received over $c$ since it saved its state and before it received the marker over $c$

• Marker sending rule for process $p_i$
  After $p_i$ has recorded its state, for each outgoing channel $c$:
  $p_i$ sends one marker message over $c$ (before it sends any other messages over $c$)
Termination

• Process P has completed its part of the algorithm when it has processed markers on all input channels

• It sends its saved local state and channel histories to the initiator
  – the intent is that collection of local states form consistent cut
    - channel histories are the messages in transit at time of cut
Analysis

• Does it find a consistent cut?
  – if so, then for any $P_a$ and $P_b$, if $m$ is a message sent from $P_a$ to $P_b$, then if $\text{recv}(m)$ is in the cut, so is $\text{send}(m)$
    - i.e., if $\text{recv}(m)$ occurred before $P_b$ recorded its state, then $\text{send}(m)$ occurred before $P_a$ recorded its state
  – stronger statement: if for any $P_a$ and $P_b$, if $e_a$ and $e_b$ are events in $P_a$ and $P_b$, such that $e_a$ happens before $e_b$ ($e_a \rightarrow e_b$), then if $e_b$ is in the cut, so is $e_a$
    - i.e., if $e_b$ occurred before $P_b$ recorded its state, then $e_a$ occurred before $P_a$ recorded its state
Proof

• Assume no: $P_a$ recorded its state before $e_a$ occurred ($e_b$ is in the cut, but $e_a$ is not)
  – since $e_a \rightarrow e_b$, there was some sequence of messages $m_1, m_2, \ldots, m_h$ that brought on $e_a \rightarrow e_b$
  – since $P_a$ recorded its state before $e_a$ occurred, it sent marker messages out on all its outgoing channels before transmitting $m_1$
  – since the channels are FIFO, a marker reached $P_b$ before $m_h$
  – but then $P_b$ would have recorded its state before $e_a$
  – but then $e_b$ would not have been in the cut
    - contradiction
Failures, Elections, and Raft
Distributed Banking

SFO

add interest based on current balance

PVD

deposit $1000
What is Consensus?
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
Use of Distributed Consensus

• Processes may need to agree on:
  – Leader of the group
  – Set of processes in the group
  – Msg committed to a distributed queue
  – Who holds the lock?
  – What is the value for a key?

Case Study 1: “The split-brain problem”

• Service uses files replicated on two racks
• Service should only write to one rack
  – Heartbeats are used to detect failure
• If initial Node A is primary file but heartbeats fail
  – Service switches to Node B
• But …
  – What if heartbeats are slow??
Oceanstore: Replicated Primary Replica

Inner ring of servers
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]
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)
<table>
<thead>
<tr>
<th>Project</th>
<th>Consensus System</th>
<th>SR</th>
<th>LR</th>
<th>SS</th>
<th>BO</th>
<th>SD</th>
<th>GM</th>
<th>LE</th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFS</td>
<td>Chubby</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Borg</td>
<td>Chubby/Paxos</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kubernetes</td>
<td>etcd</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Megastore</td>
<td>Paxos</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Spanner</td>
<td>Paxos</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Bigtable</td>
<td>Chubby</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Hadoop/HDFS</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Hive</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Configurator</td>
<td>Zeus</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Accumulo</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>BookKeeper</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Hedwig</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Kafka</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Solr</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Giraph</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Hama</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Mesos</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CoreOS</td>
<td>etcd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>OpenStack</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Neo4j</td>
<td>ZooKeeper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

- server replication (SR),
- log replication (LR),
- synchronization service (SS),
- barrier orchestration (BO),
- service discovery (SD),
- group membership (GM),
- leader election (LE),
- metadata management (MM)

https://www.cse.buffalo.edu/tech-reports/2016-02.pdf
Desired Properties of a Consensus protocol

• Liveness: a process must eventually become leader

• Safety: No two processes may be the leader

• Fairness:
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
Synchronous vs. Asynchronous

• Execution speed
  – synchronous: bounded
  – asynchronous: unbounded

• Message transmission delays
  – synchronous: bounded
  – asynchronous: unbounded

• Local clock drift rate:
  – synchronous: bounded
  – asynchronous: unbounded
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
Detecting Crashes

- **Synchronous systems**
  - Timeouts

- **Asynchronous systems**
  - ?

- **Fail-stop**
  - an oracle lets us know
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
Raft

• Proposed by Ongaro and Ousterhout in 2014

• Five components
  – Leader election
  – Log replication
  – Safety
  – Client protocol
  – Membership changes

• Assumes crash failures (so no byzantine failures)
• No dependency on time for safety
  – But depends on time for availability
• Tolerates (N-1)/2 failures
# Raft Implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language</th>
<th>Developer(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>kanaka/raft.js</td>
<td>JS</td>
<td>Joel Martin</td>
</tr>
<tr>
<td>go-raft</td>
<td>Go</td>
<td>Ben Johnson (Sky) and Xiang Li (CoreOS)</td>
</tr>
<tr>
<td>hashicorp/raft</td>
<td>Go</td>
<td>Armon Dadgar (HashiCorp)</td>
</tr>
<tr>
<td>LogCabin</td>
<td>C++</td>
<td>Diego Ongaro (Stanford)</td>
</tr>
<tr>
<td>ckite</td>
<td>Scala</td>
<td>Pablo Medina</td>
</tr>
<tr>
<td>peterbourgon/raft</td>
<td>Go</td>
<td>Peter Bourgon</td>
</tr>
<tr>
<td>rafter</td>
<td>Erlang</td>
<td>Andrew Stone (Basho)</td>
</tr>
<tr>
<td>barge</td>
<td>Java</td>
<td>Dave Rusek</td>
</tr>
<tr>
<td>py-raft</td>
<td>Python</td>
<td>Toby Burress</td>
</tr>
<tr>
<td>ocaml-raft</td>
<td>OCaml</td>
<td>Heidi Howard (Cambridge)</td>
</tr>
</tbody>
</table>

...
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
  - Term incremented when new voting is initiated
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 → follower becomes candidate
  - Timeouts typically 100-500ms
Randomized Timeouts

- If we choose election timeouts randomly,

- One server usually times out and wins election before others wake up
Election Basics

• When election timeout, start a new election:
  – 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 then send 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 the 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
Raft

- Proposed by Ongaro and Ousterhout in 2014

- Five components
  - Leader election
  - Log replication
  - Safety
  - Client protocol
  - Membership changes

- Assumes crash failures (so no byzantine failures)
- No dependency on time for safety
  - But depends on time for availability
- Tolerates \((N-1)/2\) failures