Cloud-Based Data Processing

Consensus

Jana Giceva

Fault-tolerant total order broadcast

- Total order broadcast is very useful for state machine replication.
- Can implement total order broadcast by sending all messages via a single **leader.**

- **Problem: what if the leader crashes / becomes unavailable?**

- **Manual failover:**
  a human operator chooses a new leader, and reconfigures each node to use a new leader.

  Used in many databases. Fine for planned maintenance.
  Unplanned outage? Humans are slow, may take a long time until the system recovers.

- **Can we automatically choose a new leader?**

Consensus and total order broadcast

- Traditional formulation of consensus:
  several nodes want to come to an agreement about a single value.

- In context of total order broadcast – this value is the next message to be delivered.
- Once one node **decides** on a certain message order, all nodes will decide the same order.

- A consensus algorithm must satisfy the following properties:
  - **Uniform agreement** – no two nodes decide differently
  - **Integrity** – no node decides twice
  - **Validity** – if a node decides value v, then v was proposed by some node.
  - **Termination** – every node that does not crash, eventually decides some value.

- **Common consensus algorithms:**
  - **Paxos:** single-value consensus
  - **Multi-Paxos:** generalization to total order broadcast
  - **Raft, Viewstamped Replication, Zab:** FIFO-total order broadcast by default

Consensus system models

- Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

- Why not asynchronous?
  - **FLP result (**Fischer, Lynch, Paterson):
    There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model.
  - **Paxos, Raft, etc.** use clocks only used for timeouts/failure detector to ensure progress. Safety (correctness) does not depend on timing.

- There are also consensus algorithms for a partially synchronous **Byzantine** system model (used in Blockchain).

Core of consensus: Leader

- Leader election

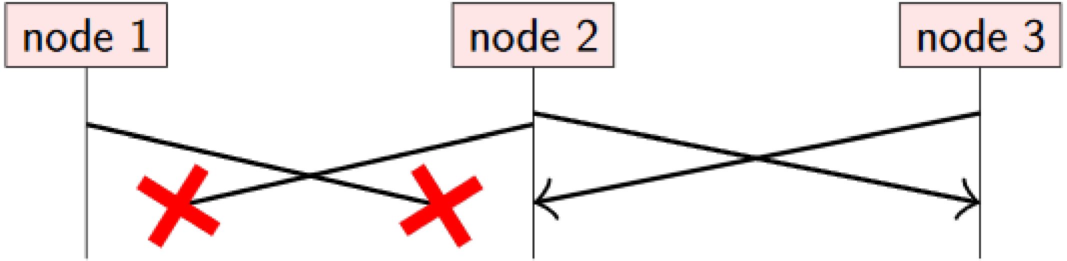
- Multi-Paxos, Raft, etc. use a leader to sequence messages.
  - Use a **failure detector** (timeout) to determine suspected crash or unavailability of a leader.
  - On suspected leader crash, **elect a new one.**
  - Prevent **two leaders at the same time** (“split brain” problem).

- Ensure <= 1 leader per **term:**
  - Term is incremented every time a leader election is started
  - A node can only **vote once per term**
  - Require a **quorum** of nodes to elect a leader in a term

Can we guarantee there is only one leader?

- Can guarantee unique leader **per term.**
- **Cannot** prevent having multiple leaders from different terms.

Example: node 1 is leader in term t, but due to network partitioning, it can no longer communicate with nodes 2 and 3.

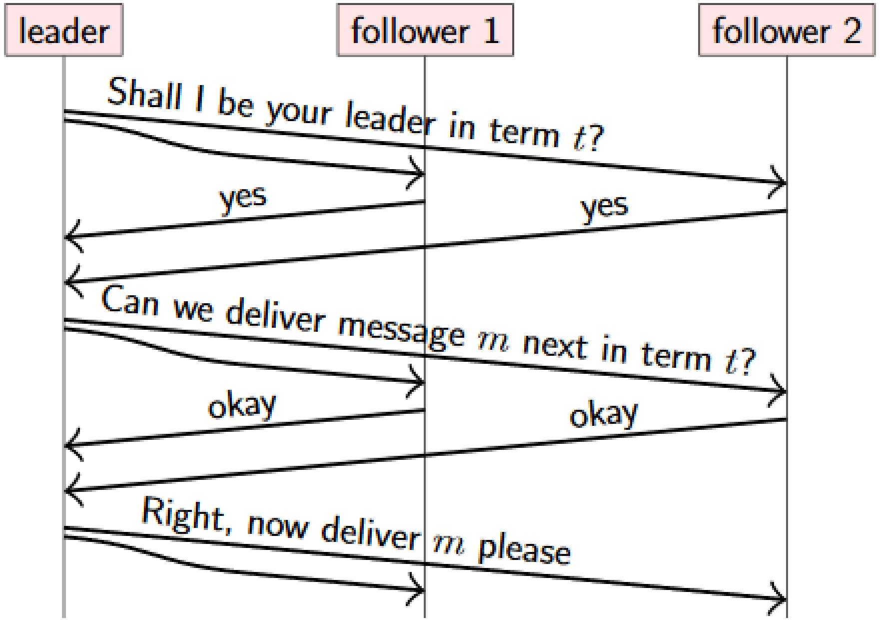


Nodes 2 and 3 may elect a new leader in term $t + 1$.

Node 1 may not even know that a new leader has been elected!

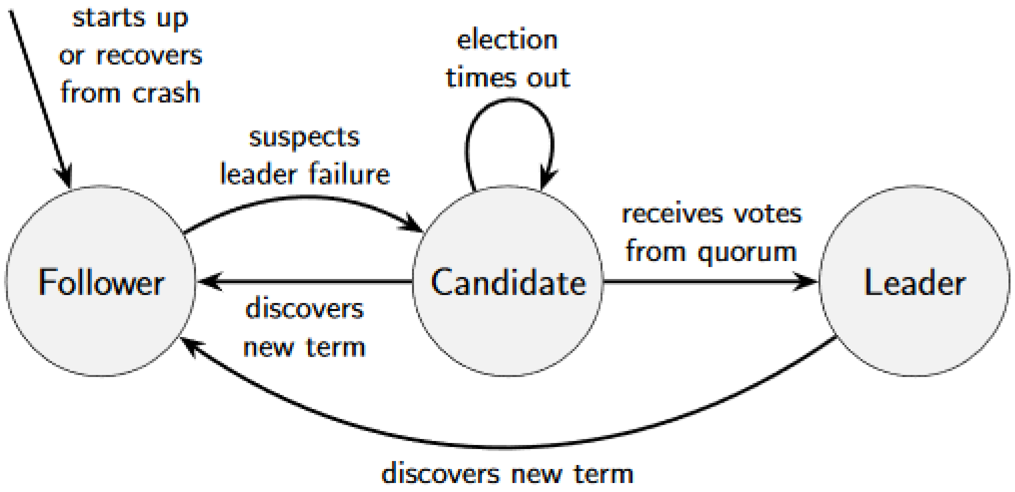
Checking if a leader has been voted out.

- For every decision (message to deliver), the leader must first get acknowledgement from a quorum.



The Raft consensus algorithm

Node state transitions in Raft



Graphical visualization of the Raft protocol

- http://thesecretlivesofdata.com/raft/

Reference for paper and pseudo-code

- https://raft.github.io/

Limitations of consensus

- Consensus brings a list of safety properties to systems where everything else is uncertain:
  - Support for **agreement, integrity** and **validity**, and **fault-tolerant**!

- But that all comes at a cost:
  - **Synchronous-based replication**
    - Much worse performance than asynchronous
  - **Strict quorum majority to operate**
    - Needs a minimum of 3 nodes to tolerate 1 failure, or minimum of 5 nodes to tolerate 2 failures
  - **Static membership algorithm**
    - Cannot simply add or remove nodes in the cluster
  - **Relies on timeouts to detect failed nodes**
    - Known to have issues for highly variable network delays

Case study: ZooKeeper

Membership and Coordination Services

References

The material covered in this class is mainly based on:

- The book *“Designing Data-Intensive Applications – The Big Ideas Behind Reliable, Scalable, and Maintainable Systems”* by Martin Kleppmann (Chapter 9) (link)
- Slides from “*Distributed Systems”* course from University of Cambridge (link)
- Raft (https://raft.github.io/)