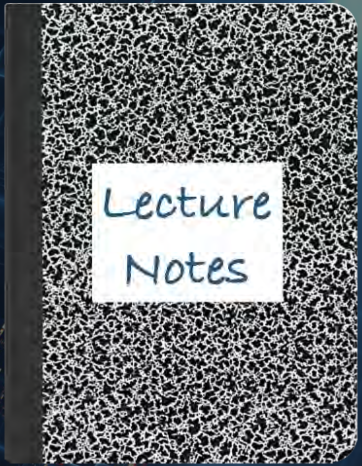


CS 417 – DISTRIBUTED SYSTEMS

Week 5: Part 3

Quorum-Based Consensus: Raft

Paul Krzyzanowski

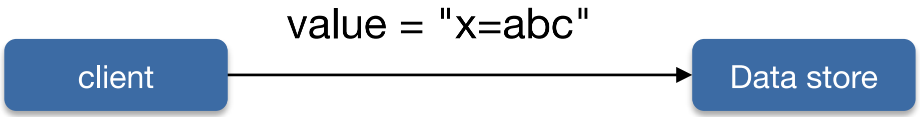
Consensus Goal

- Consensus problem statement:
  - *How do we get unanimous agreement on a given value?*
    value = sequence number of a message, key=value, operation, whatever…
- Enable a group of processes to agree on a result
- All processes must agree on the same value
- The value must be one that was submitted by at least one process (the consensus algorithm cannot just make up a value)

We saw versions of this

- Mutual exclusion – *choose which process can access a resource from all who want it*
  - Agree on who gets a resource or who becomes a coordinator
- Election algorithms – *choose one process from the set of willing processes*
- Other uses of consensus
  - Synchronize state to create replicas:
    Have every group member agree on the sequence # of the following operation
  - Manage group membership: *have everyone agree on the set of group members*
  - Agree on distributed transaction commit: *agree everyone is done with a set of operations*

Achieving consensus seems easy!



value = "x=abc"

client

Data store

- One request at a time
- Trivial … but we must hope the server never dies

FLP Impossibility Result

Impossibility of distributed consensus with one faulty process
by Fischer, Lynch and Patterson

- Consensus protocols with asynchronous communication & faulty processes

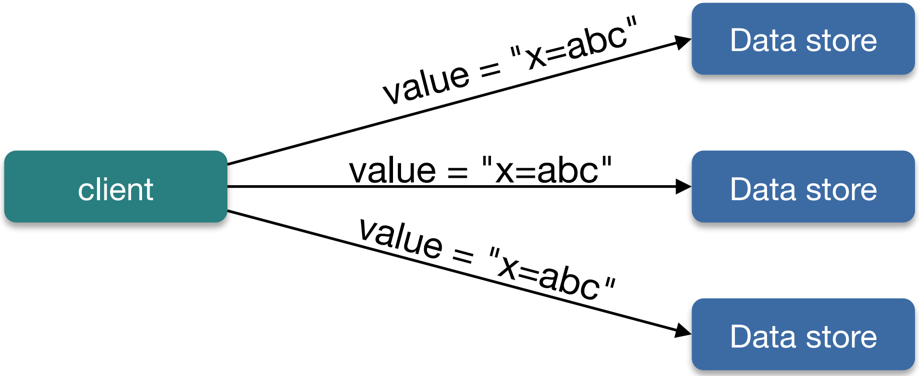
  "Every protocol for this problem has the possibility of nontermination, even with only one faulty process"

It really means *we cannot achieve consensus in bounded time*

- But we can with *partially synchronous* networks
  - Partially synchronous = network with a bounded time for message delivery but we don't know ahead of time what that bound is
- We can either wait long enough for messaging traffic so the protocol can complete or else terminate

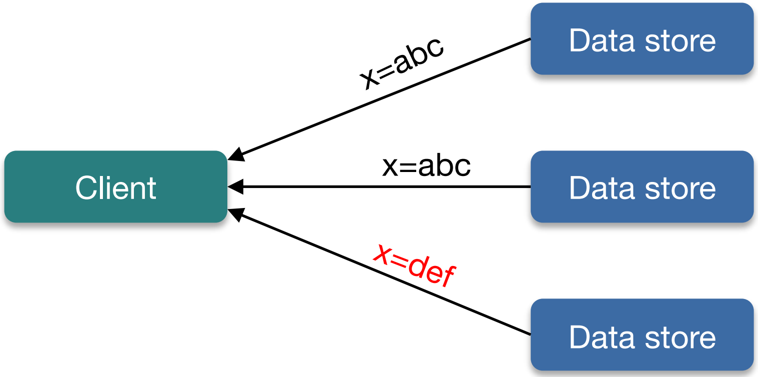
References:
the-paper-trail: https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/
original paper: https://dl.acm.org/doi/10.1145/3149.214121

Servers might die – let's add replicas



Easy if only one client sends request at a time

Reading from replicas is easy

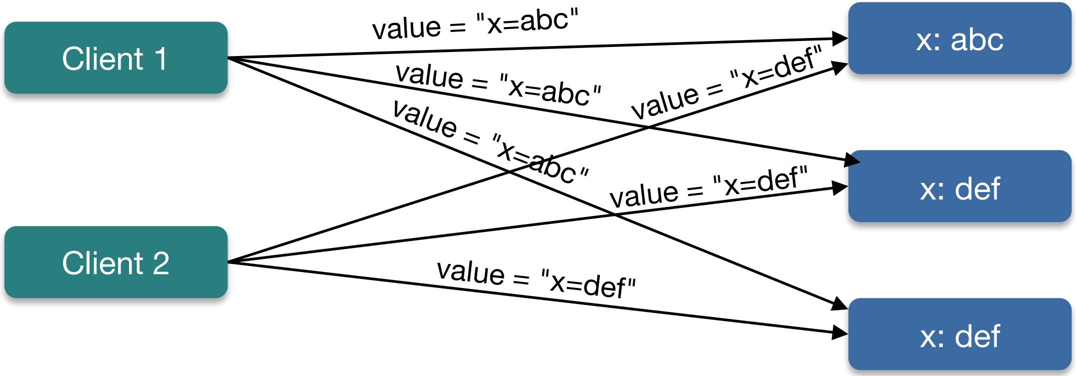


We rely on a **quorum** (majority) for reads & writes

If we have to write to a majority of servers for the *write* to succeed *and* we have to read from a majority of servers for the *read* to succeed then we can be certain that at least one server has the latest version of data.

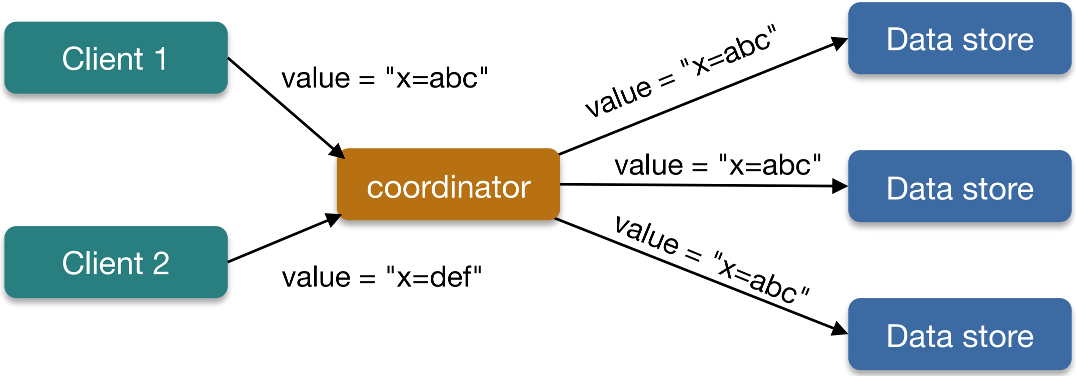
No quorum = failed read!

What about **concurrent updates?**



We risk inconsistent updates

Send all updates through a coordinator?



- Coordinator (or sequence # generator) processes requests one at a time
- But now we have a **single point of failure**!
- We need something safer

Consensus algorithm goal

Goal: agree on one result among a group of participants

Create a fault-tolerant consensus algorithm that does not block if a *majority of processes* are working

- Processors may fail (some may need stable storage)
- Messages may be lost, out of order, or duplicated
- If delivered, messages are not corrupted

Quorum: majority (>50%) agreement is the key part:

It avoids split-brain: you cannot have two majorities doing their own thing

It ensures continuity: if members die and others come up, **there will be one member in common** with the old group that still holds the information.

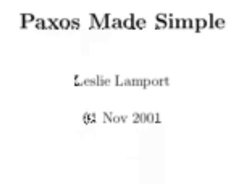
Consensus requirements

- Validity
  - Only proposed values may be selected – you can't make stuff up
- Uniform agreement
  - No two nodes may select different values – you agree with everyone else
- Integrity
  - A node can select only a single value – you cannot change your mind
- Termination (Progress)
  - Every node will eventually decide on a value – you come to a decision

Distributed Consensus Protocols: Paxos

The Part-Time Parliament

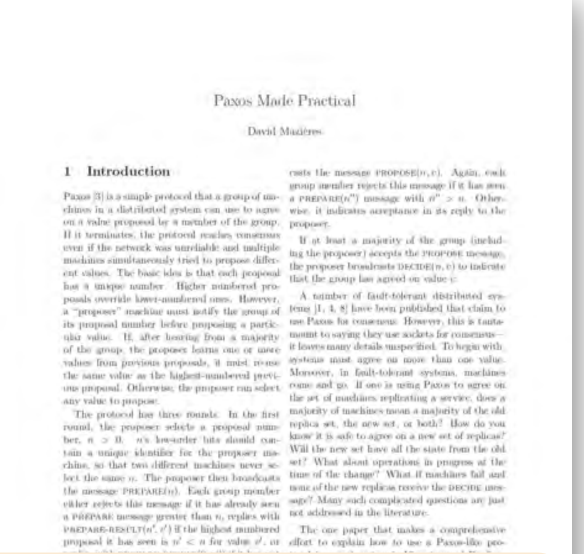
LESLIE LAMPORT
Digital Equipment Corporation



Paxos Made Simple

Leslie Lamport

01 Nov 2001



Paxos Made Practical

David Mazières

1 Introduction

Leslie Lamport's Paxos algorithm is the best-known distributed consensus algorithm. It won't hurt to read about it (or watch this video). It's not complex but how it's designed to handle failures is not obvious. Moreover, it does not incorporate leader election and requires running multiple instances for state machine replication (log replication).

Raft Distributed Consensus

Instead, we cover Raft. It was designed to be an alternative to Paxos: it's cleaner, easier to understand, incorporates elections, supports log replication, and supports bringing recovered systems up to date.

Goal: fault-tolerant replicated state machines

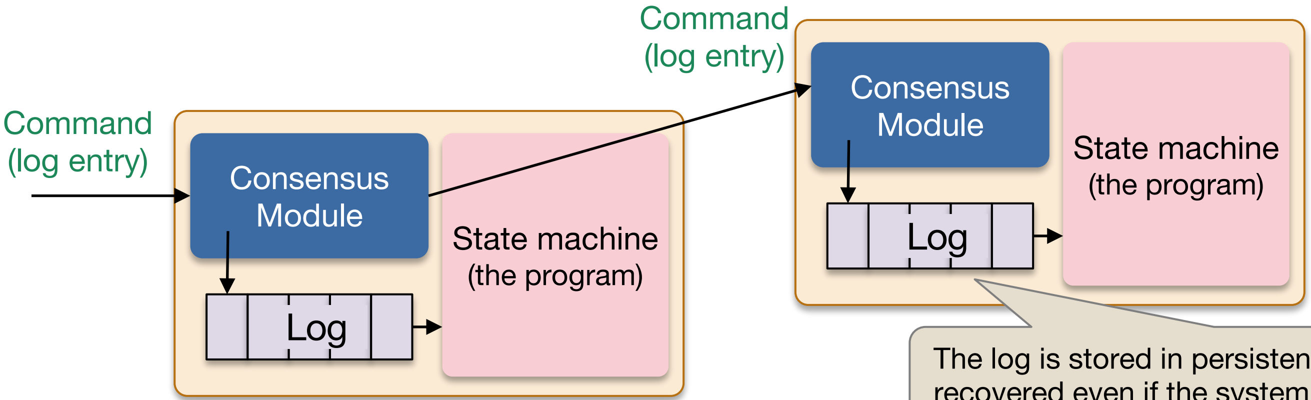
Allow a collection of systems to stay in sync and withstand the failure of some members

- Systems are deterministic – if they receive the same input then they produce the same results
- Required for any system that uses a single coordinator & makes it fault tolerant
  - Examples: Centralized mutual exclusion algorithms,
    - Lock/configuration managers: Google Chubby, Apache Zookeeper
    - Data stores: Google File System, Hadoop Distributed File System, Google Bigtable, HBase
    - Big data processing frameworks: Bulk Synchronous Parallel, Google Pregel, Apache Giraph, Apache Spark, …
- Implement as a **replicated log**
  - Log = ordered list of commands (updates) processed by each server

Raft Consensus Goal

Keep the replicated log consistent across all systems

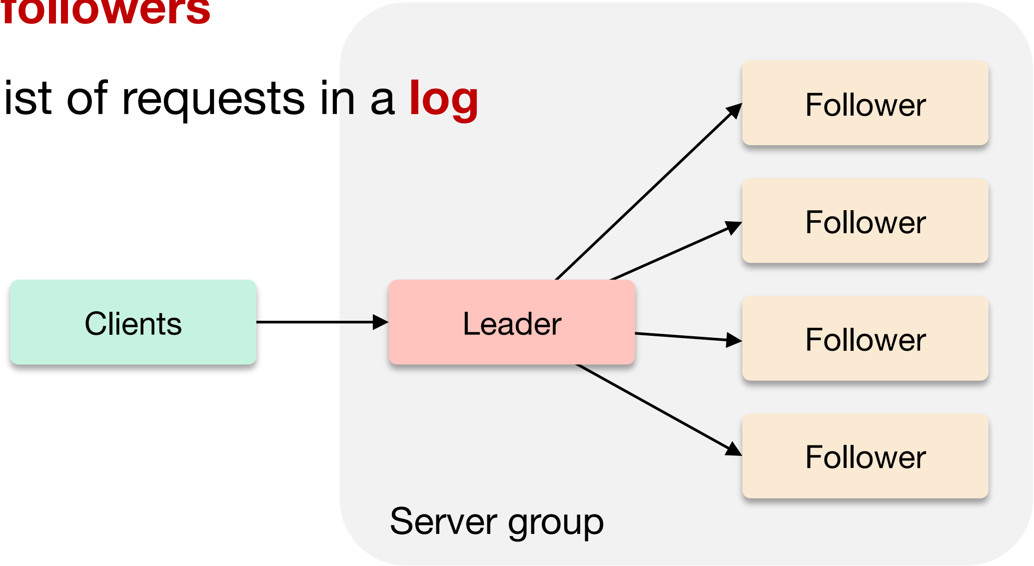
- A consensus module on a server runs Raft and receives commands from clients
- It propagates the commands to consensus modules on other systems to get everyone to agree on the next log entry
- The entry is added to the log (queue) and a state machine on each server can then process the log data



The log is stored in persistent storage – it can be recovered even if the system reboots

Raft environment

- **Server group** = set of replicas (replicated state machine)
  - Typically a small odd number (5, 7) of systems
- Clients send data to an elected **leader**
- The leader forwards the data to **followers**
- Each leader & follower stores a list of requests in a **log**
- Raft has two phases
  1. Leader election
  2. Log propagation



Participant states

- **Leader**: handles all client requests
  - There is only one leader at a time
- **Candidate**: used during leader election
  - One leader will be selected from one or more candidates
- **Follower**: doesn't talk to clients
  - Responds to requests from leaders and candidates

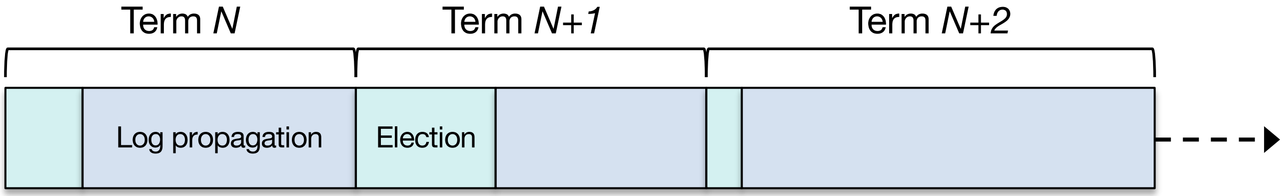
Raft RPCs

The Raft protocol uses two RPCs

- **RequestVotes**
  - Used during elections
- **AppendEntries**
  - Used by leaders to
    - Propagate log entries to replicas (followers)
    - Send commit messages (inform that a majority of followers received the entry)
    - Send heartbeat messages – a message with no log entry

Terms

- Each **term** begins with an election
- Any requests from smaller term numbers are rejected
- If a participant discovers its term is smaller than another's
  - This is an indication of a recovery after failure
  - It updates its term number
  - If the participant was a **leader** or **candidate** then it reverts to a **follower** state



Leader Election

Everyone starts off as a *follower* and waits for messages from the *leader*

Leaders periodically send *AppendEntries* messages

- A *leader* must send a message to all followers at least every ***heartbeat*** interval
- These might contain no entries but act as a heartbeat

If a *follower* times out waiting for a heartbeat from a *leader*, it starts an election

- Follower changes its state to ***candidate***
- Increments its term number
- Sets a random election timeout
- Votes for itself
- Sends **RequestVote** RPC messages to all other members
  - Any receiving process will vote for this candidate if it has not voted yet in this term

Leader Election: Outcomes

Possible outcomes

1. **Candidate receives votes from a majority of servers**
   - It becomes a leader and starts to send *AppendEntries* messages to others
2. **Candidate receives an *AppendEntries* RPC**
   - That means someone else thinks they're the leader – check the *term* # in the message
   - If term # in message > candidate's term #
     It accepts the server as the leader and becomes a follower
   - If term # in message < candidate's term #
     It rejects the RPC and remains a candidate
3. **Election timeout is reached with no majority response**
   - Split vote: if more than one server becomes a candidate at the same time, there is a chance the vote may be split with no majority

Leader Election: Randomized timeouts

If more than one server becomes a candidate at the same time, there is a chance the vote may be split with no majority

- Raft uses **randomized timeouts** to ensure concurrent elections and split votes are rare
- Each participant chooses a random election timeout (e.g., 150-300 ms)
  - Timeout must expire before the candidate can start another election
- If multiple servers hold concurrent elections and we have a split vote
  - They simply restart their elections: it's highly unlikely that both will choose the same random *election timeout*

Log replication: *leader* to *followers*

- Commands from clients are sent *only* to the current leader
  - Leader appends the request to its own log
    - Log entry has a term # and an index # associated with it
  - Sends an ***AppendEntries*** RPC to all the followers
    - Retry until all followers acknowledge it
- Each ***AppendEntries*** RPC request contains:
  - Command to be run by each server
  - Index to identify the position of the entry in the log (first is 1)
  - Term number - identifies when the entry was added to the leader's log
  - Index and term # of previous log entry

Log replication: *followers*

A follower receives an ***AppendEntries*** message only from the leader

- If leader's term < follower's term
  - Reject the message
- If the log does not contain an entry at the previous (index, term)
  - Reject the message
- If the log contains a conflicting entry (same index, different term)
  - Delete that entry and all following entries from the log
- If none of those conditions apply
  - Add the data in the message to the log

Log replication: execution

- When a log entry is accepted by the ***majority*** of servers, it is considered **committed**
- The leader can then execute the log entry & send a result to the client
- Each ***AppendEntries*** RPC request contains a *commit index*
  - Index of the highest committed log entry
  - When followers are told the entry is committed, they apply the log entry to their state machine
    - It tells them that a majority of systems in the server group acknowledged the entry and wrote it into their log

Forcing consistency

- Leaders & followers may crash
  - Causes logs (& knowledge of current term) to become inconsistent
- Leader tries to find the last index where its log matches that of the follower
  - Leader tracks `nextIndex` for ***each*** follower
    (index of next log entry that will be sent to that follower)
  - If ***AppendEntries*** returns a rejection
    - Leader decrements `nextIndex` for that follower
    - Sends an ***AppendEntries*** RPC with the previous entry
  - Eventually, the leader will find an index entry that matches the follower’s

This technique means no special actions need to be taken to restore logs when a system restarts

The End