CS 582: Distributed Systems

Multi-Paxos

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgments: Slides use some material from John Ousterhout (Stanford University)

Recap: Basic Paxos

- One or more replicas propose values
- All replicas must agree on a single value as chosen
- Only one value is ever chosen

Recap: Two-Phase Approach in Basic Paxos

- **Phase 1:** Broadcast Prepare RPC
  - Find out about any chosen values
  - Block older proposals that have not yet been completed
- **Phase 2:** Broadcast Accept RPC
  - Ask acceptors to accept a specific value

Recap: Basic Paxos

Proposers

1) Choose new proposal number n
2) Broadcast Prepare(n) to all servers

Acceptors

3) Respond to Prepare(n):
   - If n > minProposal then minProposal = n
   - Return(acceptedProposal, acceptedValue)

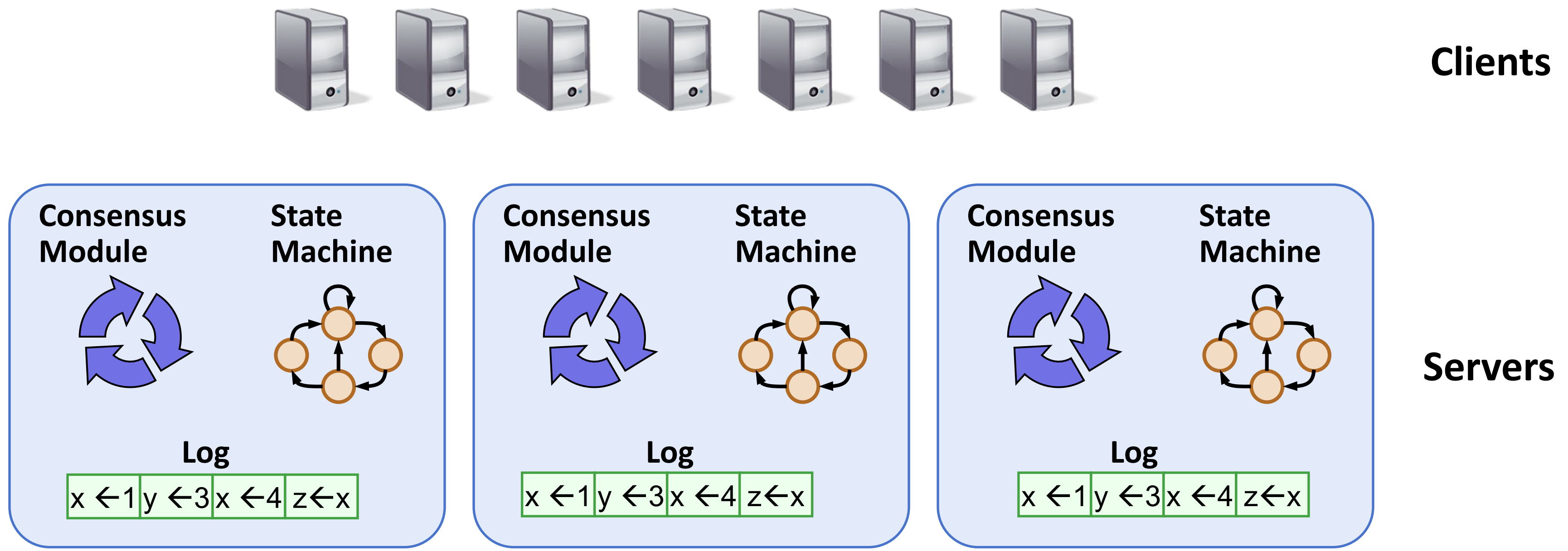
4) When responses received from majority:
   - If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal
5) Broadcast Accept(n, value) to all servers

6) Respond to Accept(n, value):
   - If n ≥ minProposal then
     acceptedProposal = minProposal = n
     acceptedValue = value
   - Return(minProposal)

6) When responses received from majority:
   - Any rejections (result > n)? goto (1)
   - Otherwise, **value is chosen**

Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)

Recap: Goal → Replicated Log

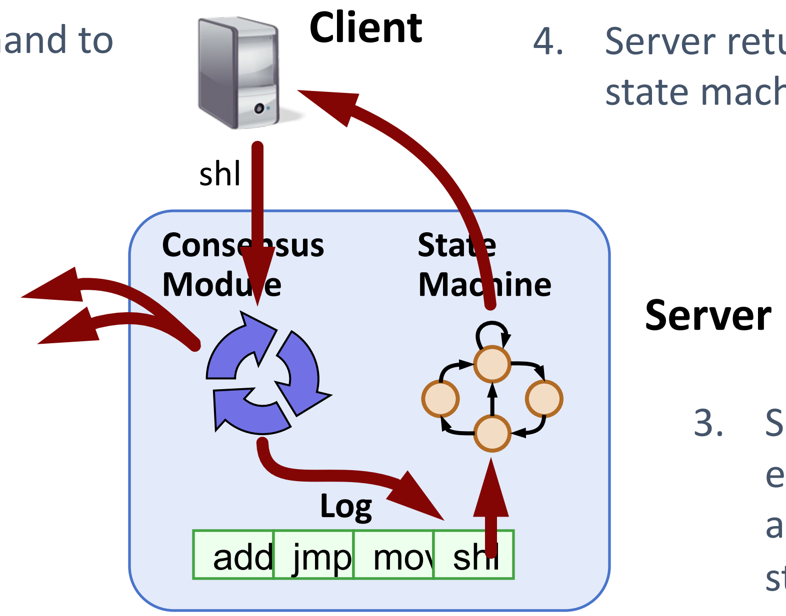


Basic Paxos solves the problem for only a single log entry.
How do we extend it to solve the log replication problem?

Multi-Paxos

- Separate instance of Basic Paxos for each entry in the log:
  - Add index argument to Prepare and Accept (selects entry in log)

1. Client sends command to server
2. Server uses Paxos to choose command as value for a log entry
3. Server waits for previous log entries to be applied, then applies new command to state machine
4. Server returns result from state machine to client



Multi-Paxos Issues

- Which log entry to use for a given client request?
- Performance optimizations:
  - Use leader to reduce proposer conflicts
  - Eliminate most Prepare requests
- Ensuring full replication
- Client protocol
- Configuration changes

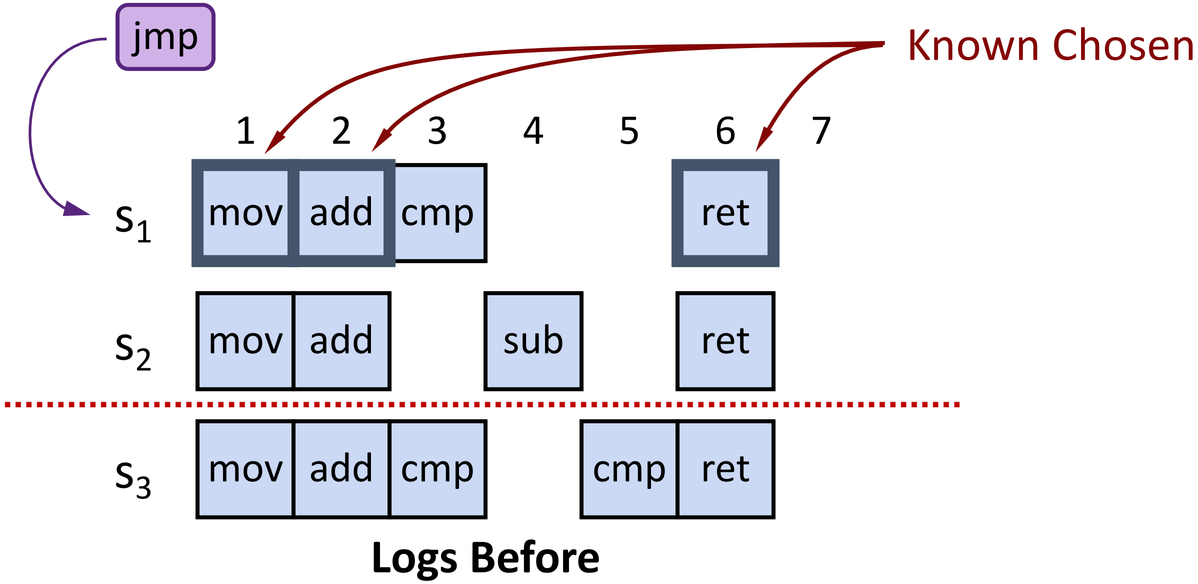
Note: Multi-Paxos not specified precisely in literature

Selecting Log Entries

- When request arrives from client:

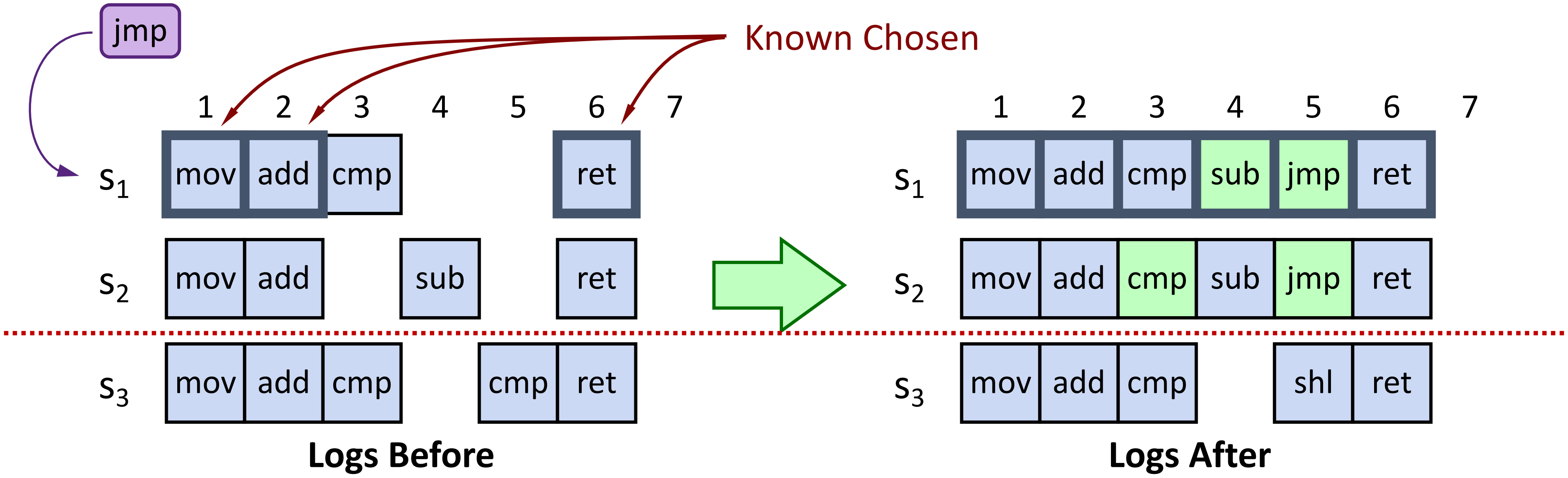
Selecting Log Entries

- When request arrives from client:
  - Find first log entry not known to be chosen



Selecting Log Entries

- When request arrives from client:
  - Find first log entry not known to be chosen
  - Run Basic Paxos to propose client's command for this index
  - Prepare returns acceptedValue?
    - Yes: finish choosing acceptedValue, start again
    - No: choose client's command



Selecting Log Entries (Cont'd)

- Servers can handle multiple client requests concurrently:
  - Select different log entries for each
- Must apply commands to state machine in log order

Improving Efficiency

Improving Efficiency

- Using Basic Paxos is inefficient:
  - With multiple concurrent proposers, conflicts and restarts are likely (higher load → more conflicts)
  - 2 rounds of RPCs for each value chosen (Prepare, Accept)

Solution:

1. Pick a leader
   - At any given time, only one server acts as Proposer
2. Eliminate most Prepare RPCs
   - Prepare once for the entire log (not once per entry)
   - Most log entries can be chosen in a single round of RPCs

Leader Election

One simple approach from Lamport:

- Let the server with highest ID act as leader
- Each server sends a heartbeat message to every other server every T ms
- If a server hasn't received heartbeat from server with higher ID in last 2T ms, it acts as leader:
    - Accepts requests from clients
    - Acts as proposer and acceptor
- If server not leader:
    - Rejects client requests (redirect to leader)
    - Acts only as acceptor

Eliminating Prepares

- Why is Prepare needed?
  - Block old proposals
    - Make proposal numbers refer to the entire log, not just one entry
  - Find out about (possibly) chosen values
    - Return highest proposal accepted for current entry
    - Also return noMoreAccepted: no proposals accepted for any log entry beyond current one
- If acceptor responds to Prepare with noMoreAccepted, skip future Prepares with that acceptor (until Accept rejected)
- Once leader receives noMoreAccepted from majority of acceptors, no need for Prepare RPCs
  - Only 1 round of RPCs needed per log entry (Accepts)

Full Disclosure

- So far, information flow is incomplete:
  - Log entries not fully replicated (majority only)
    Goal: full replication
  - Only proposer knows when entry is chosen
    Goal: all servers know about chosen entries
- Solution part 1/4: keep retrying Accept RPCs until all acceptors respond (in background)
  - Fully replicates most entries
- Solution part 2/4: track chosen entries
  - **Mark entries** that are known to be chosen:
    acceptedProposal[i] = ∞
  - Each server maintains **firstUnchosenIndex**: index of earliest log entry not marked as chosen

Full Disclosure (Cont'd)

- Solution part 3/4: proposer tells acceptors about chosen entries
  - Proposer includes its firstUnchosenIndex in Accept RPCs.
  - Acceptor marks all entries i chosen if:
    - i < request.firstUnchosenIndex
    - acceptedProposal[i] == request.proposal
  - Result: acceptors know about *most* chosen entries

log index	1	2	3	4	5	6	7	8	9	
acceptedProposal	∞	∞	∞	2.5	∞	3.4				*before Accept*

… Accept(proposal = 3.4, index=8, value = v, firstUnchosenIndex = 7) …

	1	2	3	4	5	6	7	8	9	
acceptedProposal	∞	∞	∞	2.5	∞	∞		3.4		*after Accept*

Still don't have complete information

Full Disclosure (Cont'd)

- Solution part 4/4: entries from old leaders
  - Acceptor returns its firstUnchosenIndex in Accept replies
  - If proposer's firstUnchosenIndex > firstUnchosenIndex from response, then proposer sends Success RPC (in background)
- Success(index, v): notifies acceptor of chosen entry:
  - acceptedValue[index] = v
  - acceptedProposal[index] = ∞
  - return its new firstUnchosenIndex
  - Proposer sends additional Success RPCs, if needed

Client Protocol (Similar to Raft)

- 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 chosen for log entry 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 (Cont'd)

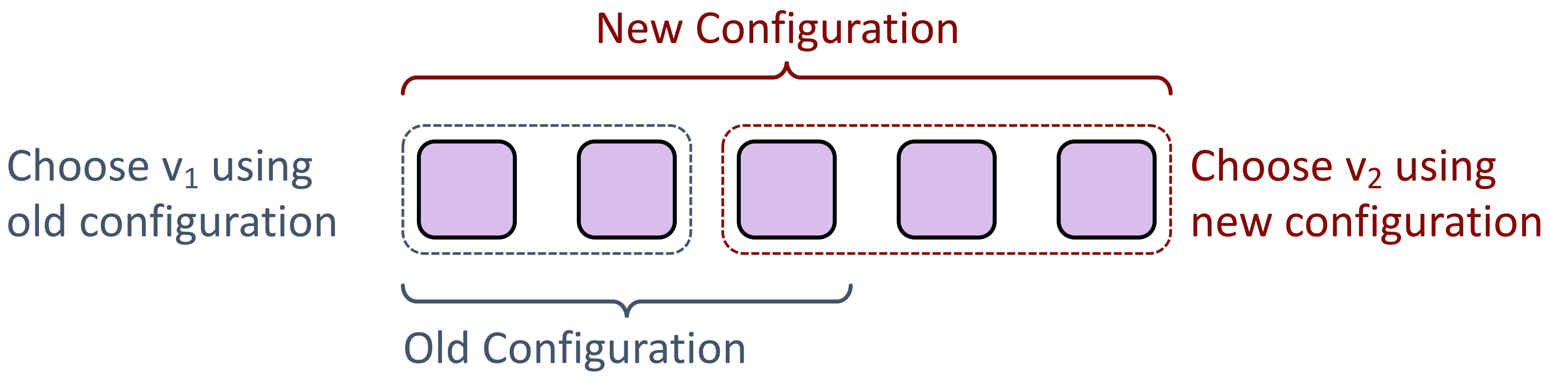
- What if the leader crashes after executing the command but before responding?
  - Must not execute command twice
- Client embeds a unique id in each command (just like Raft)
  - Server includes id in log entry
  - State machine records most recent command executed for each client
  - Before executing command, state machine checks to see if the command already executed, if so:
    - Ignore command
    - Return response from old command

Configuration Changes

- Consensus mechanism must support changes in the configuration:
  - Replace failed machine
  - Change degree of replication

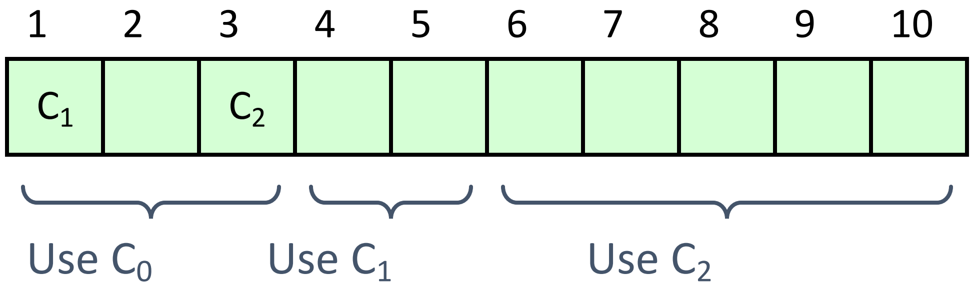
Configuration Changes (Cont'd)

- Safety requirement:
  - During configuration changes, it must not be possible for different majorities to choose different values for the same log entry:



Configuration Changes (Cont'd)

- Paxos solution: use the log to manage configuration changes:
  - Configuration is stored as a log entry
  - Replicated just like any other log entry
  - Configuration for choosing entry i determined by entry i-**a**. Suppose **a** = 3:



- Notes:
  - **a** limits concurrency: can't choose entry i+**a** until entry i chosen
  - Issue no-op commands if needed to complete change quickly

Paxos Summary

- Basic Paxos:
  - Prepare phase
  - Accept phase
- Multi-Paxos:
  - Choosing log entries
  - Leader election
  - Eliminating most Prepare requests
  - Full information propagation
- Client protocol
- Configuration changes

So far, we have assumed fail-stop failures when dealing with consensus

Consensus with Fail-stop failures

- Failure-Stop failures: Nodes fail by crashing
  - A machine is either working correctly or it is doing nothing
- Paxos/Raft can ensure safety and solve the state machine replication problem as long as at least a majority of nodes are up
  - With $N = 2f + 1$ replicas can tolerate upto f simulatenous fail-stop failures
  - Ensure replicas execute operations in the same order
- Remember FLP proof still applies: can’t sure both safety and liveness in an asynchronous system
  - Raft/Paxos provide liveness when at least a majority of nodes can communicate with reasonable timeliness

Byzantine Faults

- Nodes fail arbitrarily, and may
  - Perform incorrect computation
  - Send different and wrong messages
  - Not send message at all
  - Lie about the input value
  - Collude with other failed nodes
  - And more …

Why care about Byzantine faults?

- Can be caused by
  - Malicious attack → increasing
  - Software errors → commonplace
- Example applications
  - Aircraft flight control systems (e.g., Boeing)
  - Blockchain platforms (e.g., Zilliqa, Hyperledger)
  - Byzantine-tolerant Distributed File System

Mini-case study: Boeing 737 Max

- Launched at the end of 2017
  - Had more efficient engines than 737 aircrafts
  - Introduced a new software system for automatic maneuvering of the plane (called MCAS)

- Multiple crashes
  - Lion Air Flight 610 – Oct 2018
    - Killing all 189 people on board
  - Ethiopian Airlines Flight 302 – March 2019
    - Killing all 157 people on board

Mini-case study: Boeing 737 Max (cont'd)

"Shortly after Flight 610 left Jakarta on Oct. 29, the sensor began sending bad information to the jet's computer, triggering the automated feature and driving down the plane's nose, according to the investigators' report …"

Washington Post

washingtonpost.com

Sections

The Washington Post

Democracy Dies in Darkness

Subscribe

Sign in

0:28 / 1:38

Indonesian investigators said Oct. 29 that Boeing 737 Max's design, regulatory lapses and pilot training were at fault for last year's fatal Lion Air crash. (Reuters)

By Ainur Rohmah, Ian Duncan, Michael Laris and Shibani Mahtani

413

October 25, 2019

JAKARTA, Indonesia — Design flaws in an automated feature on Boeing's 737 Max, attached to a sensor that had likely been poorly repaired, plus regulatory lapses and false assumptions about pilots' responses to new systems, combined to cause last year's fatal Lion Air crash, Indonesian investigators said Friday.

Shortly after Flight 610 left Jakarta on Oct. 29, the sensor began sending bad information to the jet's computer, triggering the automated feature and driving down the plane's nose, according to the investigators' report. The pilots — who knew nothing about the feature, which was new to the Max — struggled to respond, grappling with controls that felt as though they weighed 100 pounds.

Byzantine Fault Tolerance

⬇

Design services that continue to function <u>correctly</u> despite Byzantine faults

⬇

Solve the replicated state machine problem machine with Byzantine faults

⬇

Solving <u>consensus</u> with Byzantine faults

Can we reach consensus in the presence of Byzantine Faults with Paxos/Raft?

Can’t use Paxos/Raft with Byzantine Faults

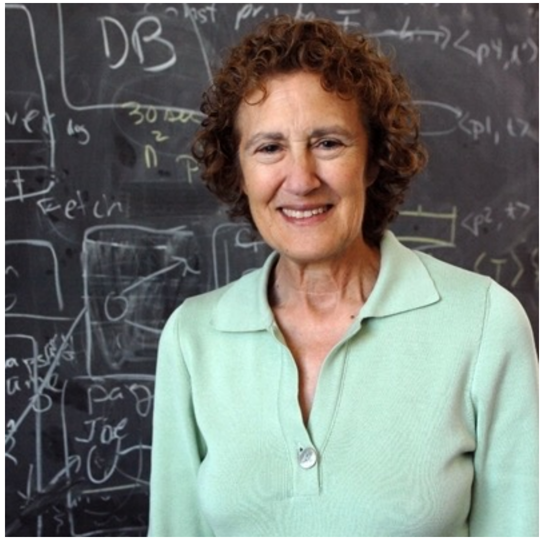
- The intersection of two majority (f + 1 node) quorums may be a byzantine node
- Byzantine node tells different quorums different things!
  - Leading to nodes committing different values
- Raft: Can’t rely on the leader to assign log index
  - Could assign same log index to different requests

Bare majority quorums may not be enough in the presence of byzantine faults

A leader might be a byzantine node so we can’t trust it

Byzantine Fault Tolerance

- Practical Byzantine Fault Tolerance (PBFT) Algorithm
  - [Liskov and Castro, 1999]



Barbara Liskov