How to Build a Highly Available System Using Consensus

Workshop on Distributed Algorithms (WDAG 1996)

Butler W. Lampson, Microsoft

Presentation by: Amaiya Singhal, Aryan Dhaka, Aman Singh Dalawat

26/08/2025

What is High Availability?

- A system is highly available if it provides services promptly on demand.
- In the ideal world, a single system should be always available and we do not need to study this paper.
- But in reality no single server, disk or process is perfectly reliable.
- Your system’s availability is capped at the available of the weakest link.

How to achieve High Availability?

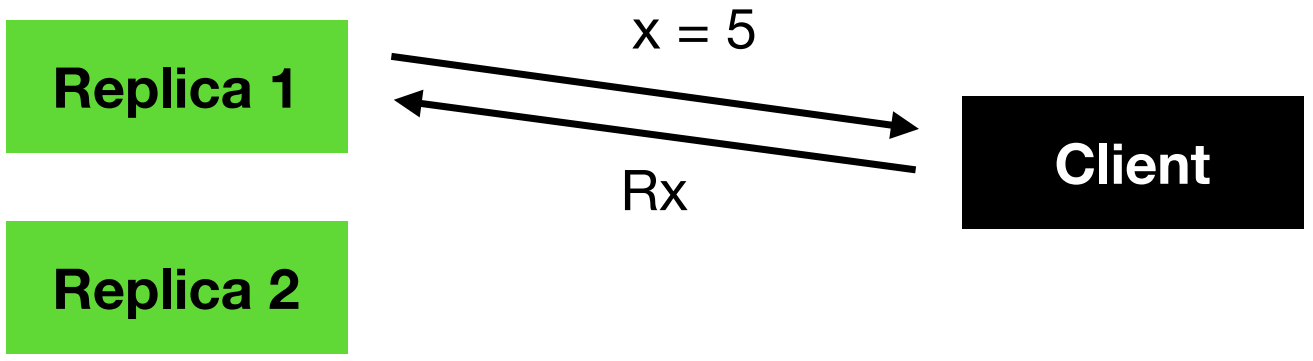
with less available components

- The way to make a highly available system out of less available components is to use redundancy.
- **Replication** is a simple way to achieve redundancy
- Replication: Make several copies or ‘replicas’ of each part

How to achieve High Availability?

with less available components

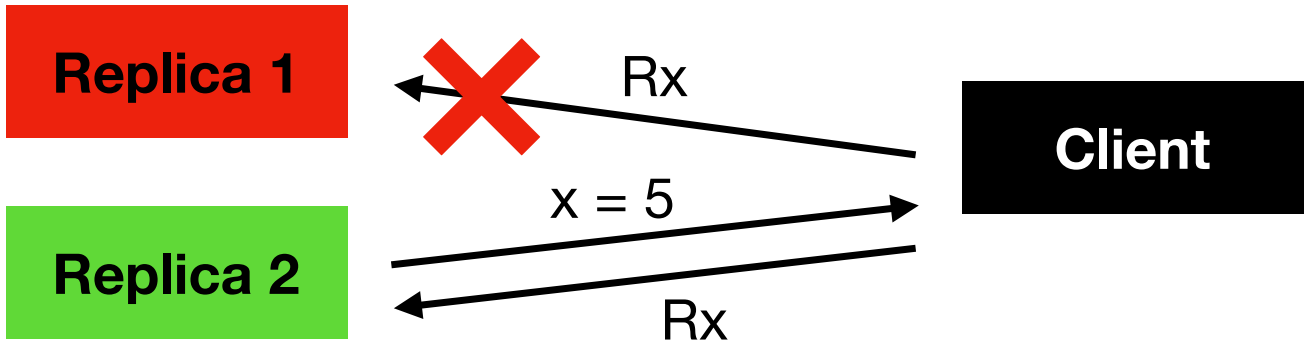
- The way to make a highly available system out of less available components is to use redundancy.
- **Replication** is a simple way to achieve redundancy
- Replication: Make several copies or ‘replicas’ of each part



How to achieve High Availability?

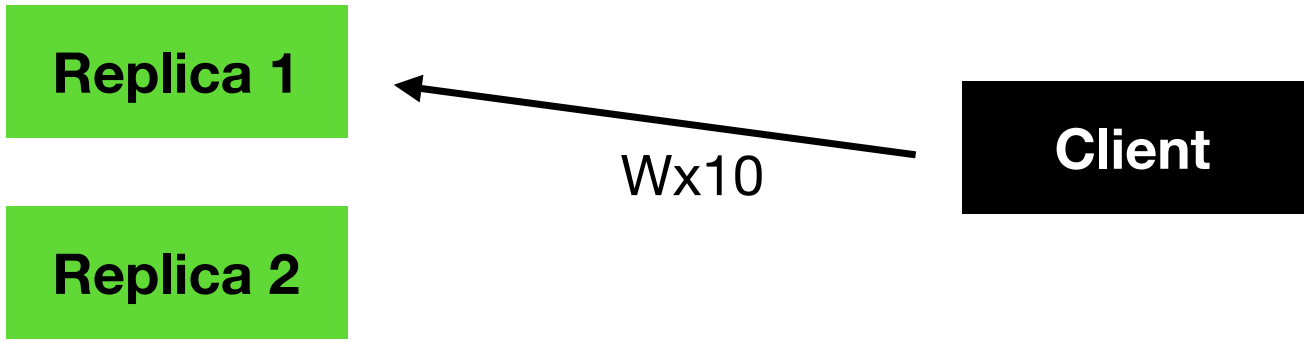
with less available components

- The way to make a highly available system out of less available components is to use redundancy.
- **Replication** is a simple way to achieve redundancy
- Replication: Make several copies or ‘replicas’ of each part



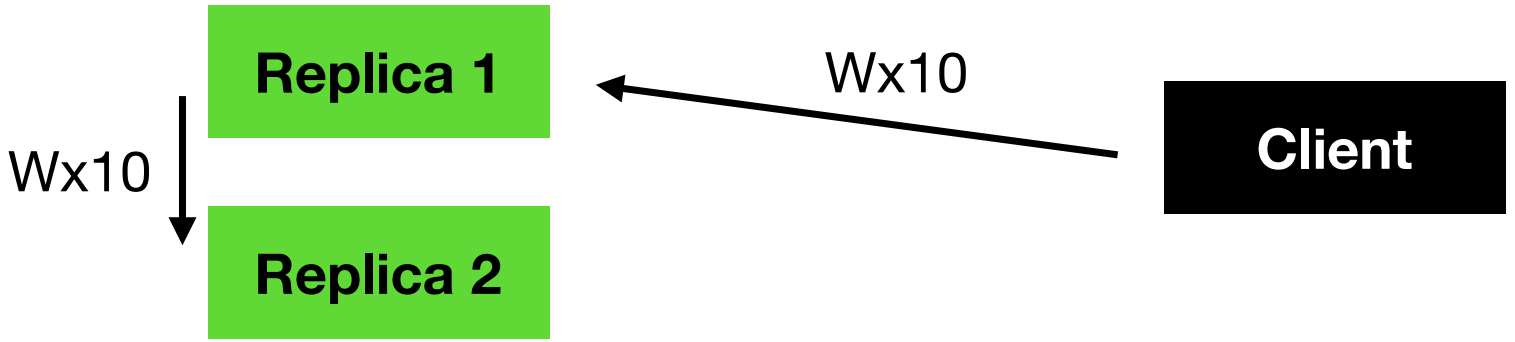
Is Redundancy enough?

- To be useful, the replicas must be coordinated



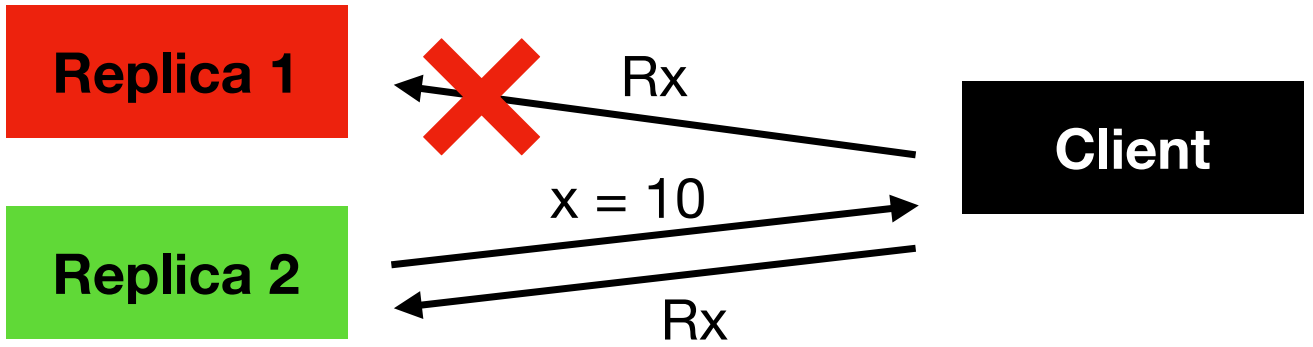
Is Redundancy enough?

- To be useful, the replicas must be coordinated
- The simplest way to achieve this is to make each non-faulty replica do the same thing
- Then any non-faulty replica can provide the outputs



Is Redundancy enough?

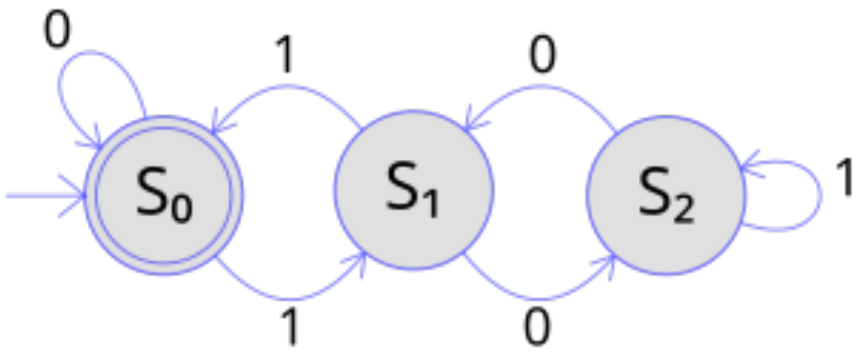
- To be useful, the replicas must be coordinated
- The simplest way to achieve this is to make each non-faulty replica do the same thing
- Then any non-faulty replica can provide the outputs



Coordinating the replicas

How to ensure each replica does the same thing?

- Build each replica as a **Deterministic State Machine**
- **What is a Deterministic State Machine?**
  - A set of states Q
  - A finite set of input symbols Σ
  - A transition function $\delta : Q \times \Sigma \rightarrow Q$
  - An initial state $q_0 \in Q$



Coordinating the replicas

Why a Deterministic State Machine?

- Call each replica a “process”.
- Processes that start in the same state and see the same sequence of inputs will end up in the same state and produce the same outputs.
- Remember, the transition relation is a **function.**
- So in this model, all we need is to ensure all non-faulty processes see the same inputs. This is **Consensus!**

What is consensus?

and why do we care about it

- Several processes must agree on a single value, even if some fail.
- 2 key properties:
  - **Safety:** No two non-faulty processes decide on different values.
  - **Liveness:** A decision is eventually reached (if the system keeps running).
- Consensus is hard:
  - Failures, crashes, message loss, delays
  - Asynchrony (cannot tell if a process is slow or dead)

Coordinating the replicas

How does consensus help?

- Informally, several processes achieve “consensus” if they all agree on some value.
- If several processes are implementing the same DSM and achieve consensus on the values and order of inputs, they will do the same thing.
- This allows replication of an arbitrary computation, making it highly available

Applications of Consensus

Distributed Transactions

- All the processes need to agree on whether a transaction commits or aborts.
- No split-brain: some can’t commit while others abort.
- Despite failures, all processes must agree on one value.
- Each transaction needs a separate consensus on its outcome.

Applications of Consensus

Membership

- A group of processes cooperating need to agree on which processes are currently functioning as members of the group.
- Every time a process fails or starts again there must be a new consensus.
- If different processes have a different view of membership, it can violate correctness as some replicas may not see some writes.
- Some protocols rely on majority vote, different view of membership can lead to different majorities for different replicas

Consensus

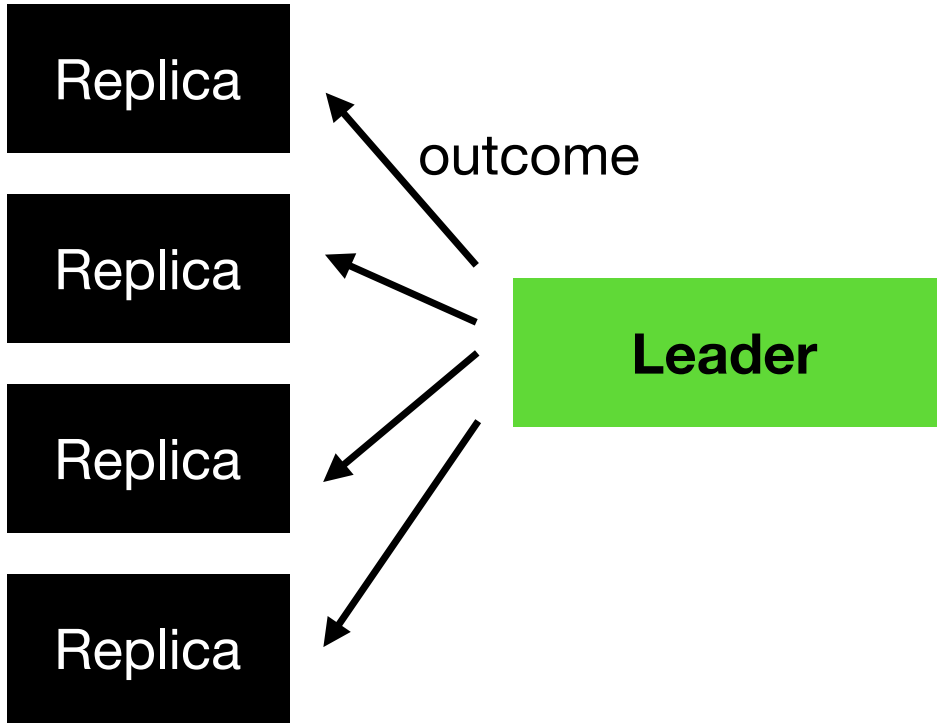
Some jargon

- The value agreed on is called the “outcome”
- The value must be “allowed” (if any value was allowed, every process would agree on 0).
- The interface to consensus has 2 actions:
  - Allow a value
  - Read the outcome
- A consensus algorithm terminates when all the non-faulty processes know the outcome.

Achieving Consensus

Assuming no faults

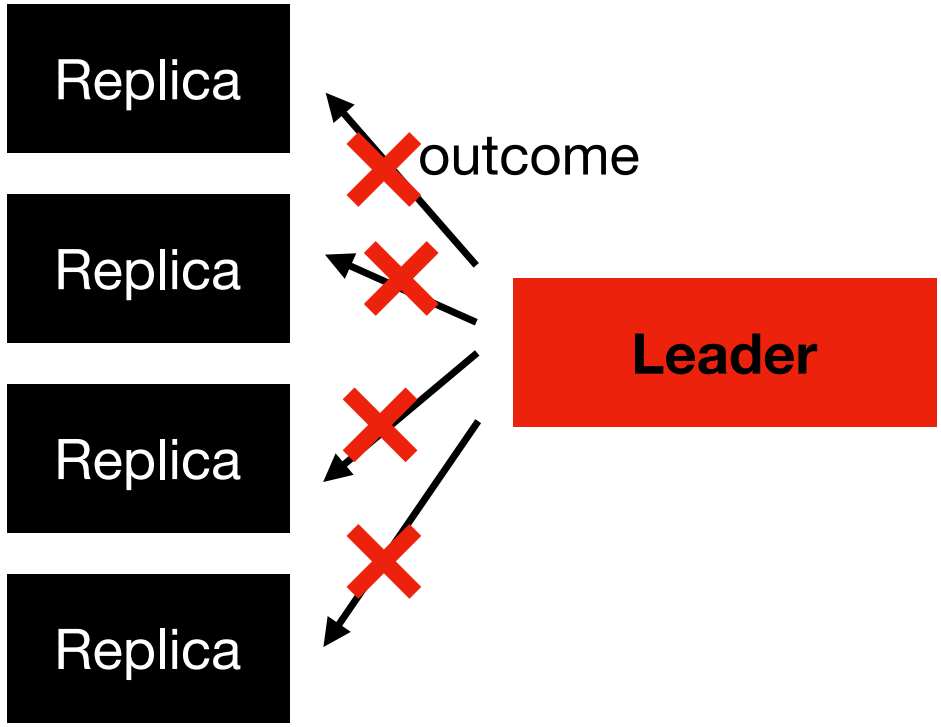
- Have a fixed leader process.
- The leader gets all the **Allow** actions, chooses the outcome and tells everyone.
- If the leader fails, the outcome may be unknown



Achieving Consensus

Assuming no faults

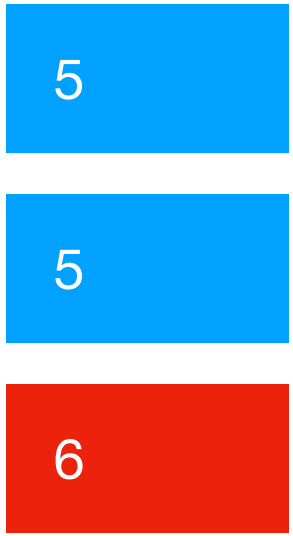
- Have a fixed leader process.
- The leader gets all the **Allow** actions, chooses the outcome and tells everyone.
- If the leader fails, the outcome may be unknown



Achieving Consensus

Another implementation (Assuming no faults)

- Have a set of processes, each choosing a value.
- If majority choose the same value, that is the outcome.
- Possible majorities are subsets such that any 2 majorities have a non-empty intersection.
- If there is no majority or if some members of the majority fail, there is no outcome.

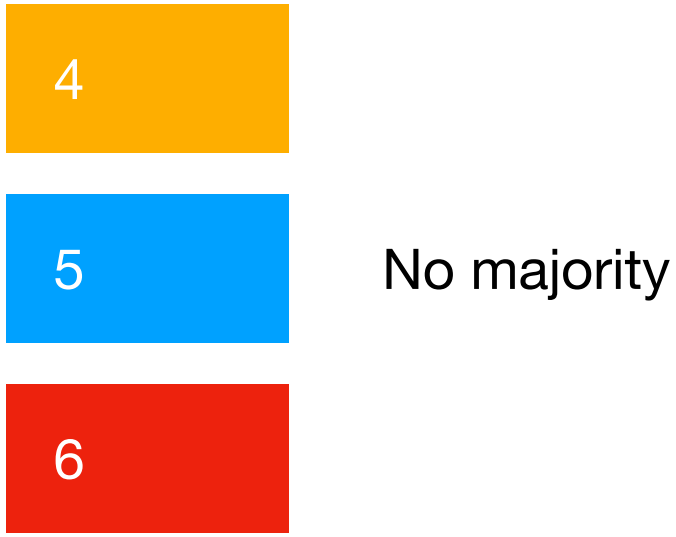


5 forms the majority

Achieving Consensus

Another implementation (Assuming no faults)

- Have a set of processes, each choosing a value.
- If majority choose the same value, that is the outcome.
- Possible majorities are subsets such that any 2 majorities have a non-empty intersection.
- If there is no majority or if some members of the majority fail, there is no outcome.



Specifying Consensus

consensus.tla

Achieving Consensus

When nodes can fail

- **Goal:** All non-faulty nodes must agree on the same value even if some nodes crash or messages are delayed.
- **Solution:** Use quoroms (majority) - only a majority of nodes is required to be alive and responsive.
- Even if some nodes fail, as long as a majority can be contacted, consensus can be achieved.

Quoroms?

Achieving Consensus when nodes can fail

- A quorom is any possible subset of nodes that satisfies the following:
- **Key Property:** Any 2 quoroms must overlap
  - This overlap prevents 2 different values from being chosen.
  - If one quorom accepts a value, any future quorom will remember it through the overlapping node(s).
- A majority is the most simple form of a quorom

Quoroms

Example. Majority

Set of quoroms:

{1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {1, 3, 4},

{1, 3, 5}, {2, 3, 5}, {1, 2, 4}, {1, 2, 5},

{2, 4, 5}, {1, 4, 5}, {1, 2, 3, 4},

{1, 2, 3, 5}, {1, 3, 4, 5}, {1, 2, 4, 5},

{2, 3, 4, 5}, {1, 2, 3, 4, 5}

The Paxos Algorithm

Achieving consensus when nodes can fail

- One node acts as the *proposer* and is responsible for initiating the protocol.
- The other nodes conspire to make a decision about the value being proposed and are called *acceptors*
- There can only be one proposer at a time, but if two or more choose to then the protocol will fail to terminate until only one node is a proposer
- This sacrifices termination for correctness.

FLP Impossibility Theorem

(Fischer, Lynch, Paterson, 1985)

- Asynchronous Networks:
  - No global clock, messages can be delayed arbitrarily
  - Processes cannot tell if a node is slow or failed
- **Theorem:** In a purely asynchronous system, no deterministic consensus algorithm can guarantee both simultaneously:
  - **Safety:** all non-faulty nodes agree on the same value.
  - **Liveness:** a decision is eventually reached.

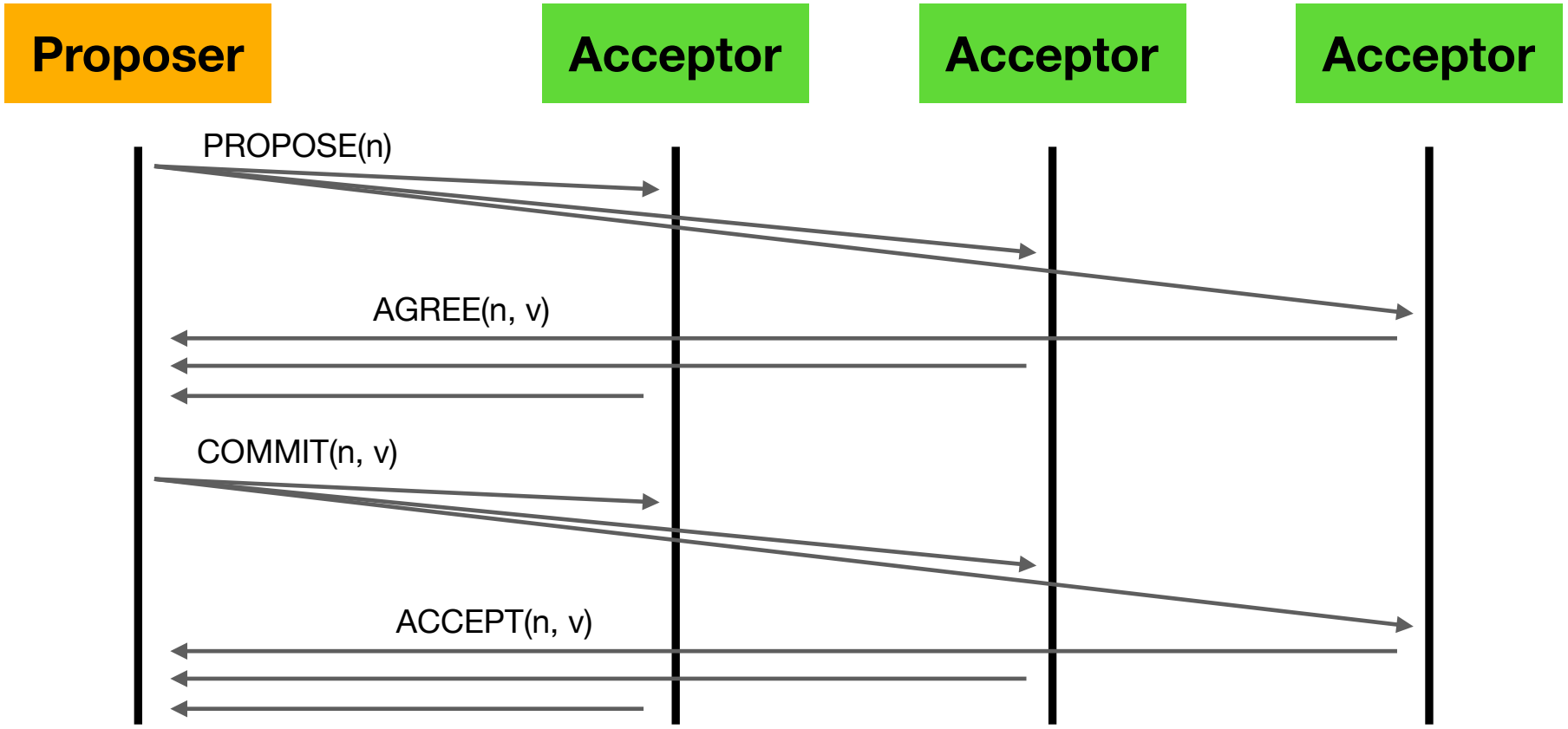
The Paxos Algorithm

Skeleton Protocol

- The proposer sends a ‘prepare’ request to the acceptors.
- When the acceptors have indicated their agreement to accept the proposal, the proposer sends a commit request to the acceptors.
- Finally, the acceptors reply to the proposer indicating the success or failure of the commit request.
- Once enough acceptors have committed the value and informed the proposer, the protocol terminates.

The Paxos Algorithm

Skeleton Protocol



The Paxos Algorithm

Is this fault tolerant?

- Not yet. The first change we need is: **Ordering the proposals.**
- Every proposal is tagged with a unique sequence number which are used to totally order the proposals.
- This is needed to determine which amongst multiple proposals should be accepted.
- The acceptor makes a promise that it will not accept any proposals ordered before the highest proposal it has received.

The Paxos Algorithm

Is this fault tolerant now?

- Still no. The next change we need is:
- A proposal is accepted when a **majority** of acceptors have indicated that they have decided upon it.
- This is different from 2 Phase Commit where proposals were accepted only if every acceptor agreed to do so.
- With 2f + 1 nodes, Paxos can tolerate upto f failures.
- As long as a quorom is alive and responding, the algorithm continues to work correctly.

The Paxos Algorithm

Revised Algorithm (Proposers)

- Submit a proposal numbered **n** to a majority of acceptors.
- If the majority reply with ‘agree’, they will also send back the values of any proposals already accepted.
- Pick one of these values and send a ‘commit’ message with the proposal number and value.
- If no values have been already accepted, choose a new one.
- If the majority reply ‘reject’, or fail to reply, abandon and start again
- If majority reply to commit with ‘accepted’, the protocol is terminated. Otherwise, abandon and start again

The Paxos Algorithm

Revised Algorithm (Acceptors)

- Once a proposal is received, compare its numbers to the highest numbered proposal you have already agreed to.
- If the new proposal is higher, reply ‘agree’ with the value of any proposals you have already accepted.
- If lower, reply ‘reject’ with the sequence number of the highest proposal.
- When a ‘commit’ message is received, accept if
  - The value is same as any previously accepted proposal
  - Sequence number is the highest proposal number you have agreed to
- Otherwise reject

The Paxos Algorithm

Some small details

- We need to ensure all proposals are uniquely numbered.
- Just having a sequence number is not enough since 2 leaders can choose the same sequence number.
- One practical way to ensure each proposer draws from a disjoint set of sequence numbers is to construct a pair (seqnumber, address)
- Address is the proposer’s unique network address
- These pairs can be totally ordered and at the same time all proposers can outbid all others with a sufficiently high sequence number

The Paxos Algorithm

An example

- **{A, B, C} reply:** Leader knows no majority in previous rounds. Can choose any allowed value i.e any of 7, 8 or 9
- **{A, B} or {A, C} reply:** Leader knows round 3 failed but doesn't know if rounds 1 or 2 failed. Hence chooses latest value of 8
- **{B, C} reply:** Leader doesn't know if round 3 failed. Chooses 9
- **Less replies:** Abandon and start again

Sequence Number	A	B	C
1	7	X	X
2	8	X	X
3	X	X	9
4	**What should the leader choose?**		

The Paxos Algorithm

Another example

- **{A, B, C} or {A, C} reply:** Leader knows round 3 failed but round 2 succeeded. Chooses 9
- **{A, B} reply:** Leader knows round 3 failed but doesn't know if rounds 1 or 2 failed. Hence chooses latest value of 9
- **{B, C} reply:** Leader doesn't know if round 3 or 2 failed. Chooses 9
- **Less replies:** Abandon and start again

Sequence Number	A	B	C
1	8	X	X
2	9	X	9
3	X	X	9
4	**What should the leader choose?**		

The Paxos Algorithm

Another example

- The leader sends Commit(seqnumber = 1, value = 8) to a majority {A, B, C}

Seq Number	A	B	C	D	E
1	8	8	8	X	X

The Paxos Algorithm

Another example

- The leader sends Commit(seqnumber = 1, value = 8) to a majority {A, B, C}
- A and B fail and do not send a response to the leader
- The leader does not reach a majority and restarts for round 2

Seq Number	A	B	C	D	E
1	8	8	8	X	X

The Paxos Algorithm

Another example

- The leader sends Commit(seqnumber = 1, value = 8) to a majority {A, B, C}
- A and B fail and do not send a response to the leader
- The leader does not reach a majority and restarts for round 2
- Queries {C, D, E} and gets 8, X, X. Does not know if round 1 succeeded or failed so can only choose 8

Seq Number	A	B	C	D	E
1	8	8	8	X	X
2	Leader chooses 8				

The Paxos Algorithm

Another example

- The leader sends Commit(seqnumber = 1, value = 8) to a majority {A, B, C}
- A and B fail and do not send a response to the leader
- The leader does not reach a majority and restarts for round 2
- Queries {C, D, E} and gets 8, X, X. Does not know if round 1 succeeded or failed so can only choose 8
- If they all respond, consensus is achieved

Seq Number	A	B	C	D	E
1	8	8	8	X	X
2	X	X	8	8	8
	CONSENSUS!				

The Paxos Algorithm

Another example

- If the overlap property of quoroms did not hold, the leader could not have known 8 had to be chosen
- The overlap rule ensures information is not lost as atleast one node was in the previous quorom
- This ensures correctness

Seq Number	A	B	C	D	E
1	8	8	X	X	X
2	X	X	100	100	100
	NO CONSENSUS :(				

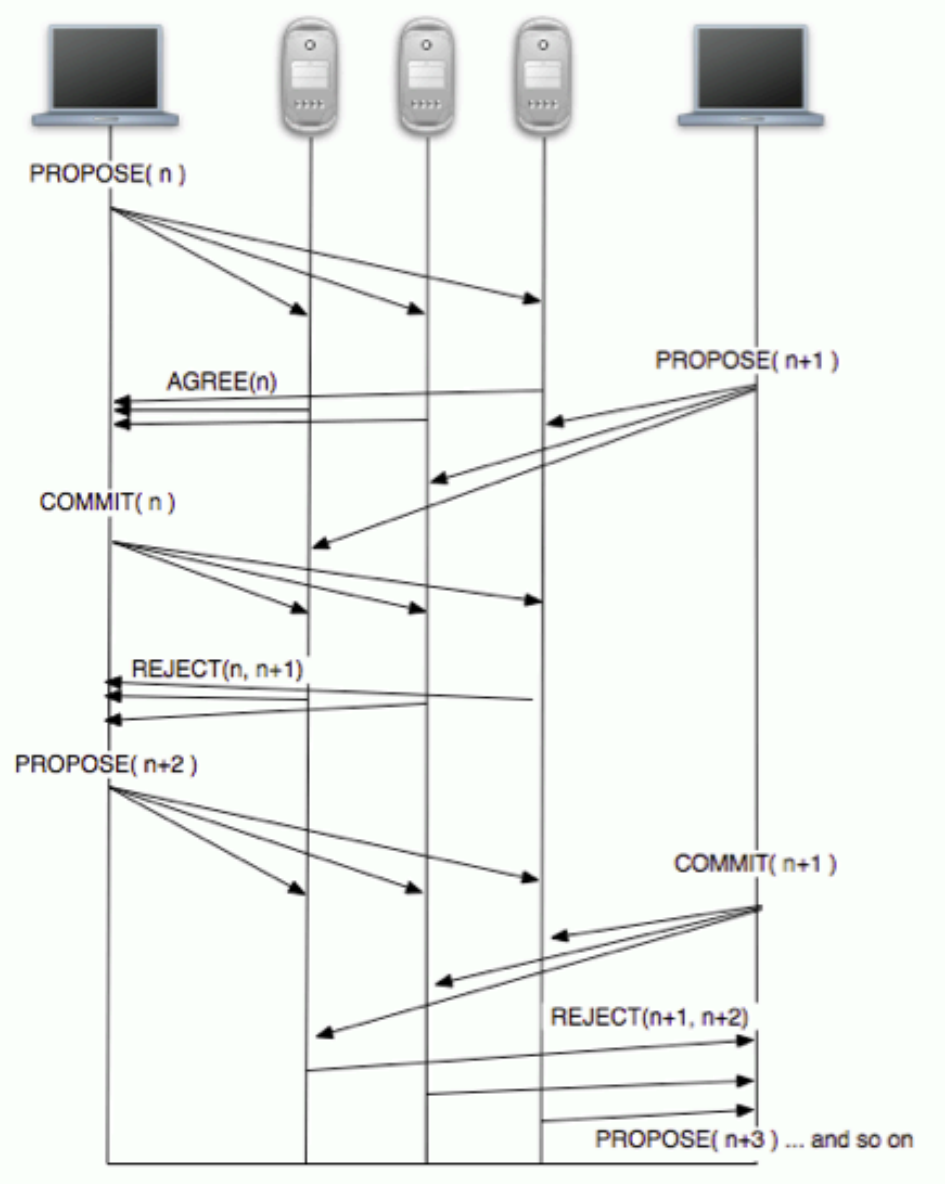
The Paxos Algorithm

Is this fault tolerant now?

- Yes*, atleast as good as we can get…
- When 2 proposers are active at the same time, they may duel for the highest proposal number by alternately issuing ‘one-up’ of the previous proposal.
- Until this is resolved and a single leader is agreed upon, Paxos may not terminate.
- This violates liveness, but we are okay with that.
- The likelihood is that eventually Paxos will return to a correct execution once network settles down and the 2 proposers see each other and one backs off

The Paxos Algorithm

Duelling proposers violate termination



Proving Paxos $\Longrightarrow$ Consensus

Showing Y implements X

- We want to show that a system Y implements a system X
- The authors define it as follows
  - Every trace of Y is a trace of X; that is, X's safety property implies Y's safety propery
  - Y's liveness property implies X's liveness property
- The authors use the notion of an abstraction function to map implementation states to specification states

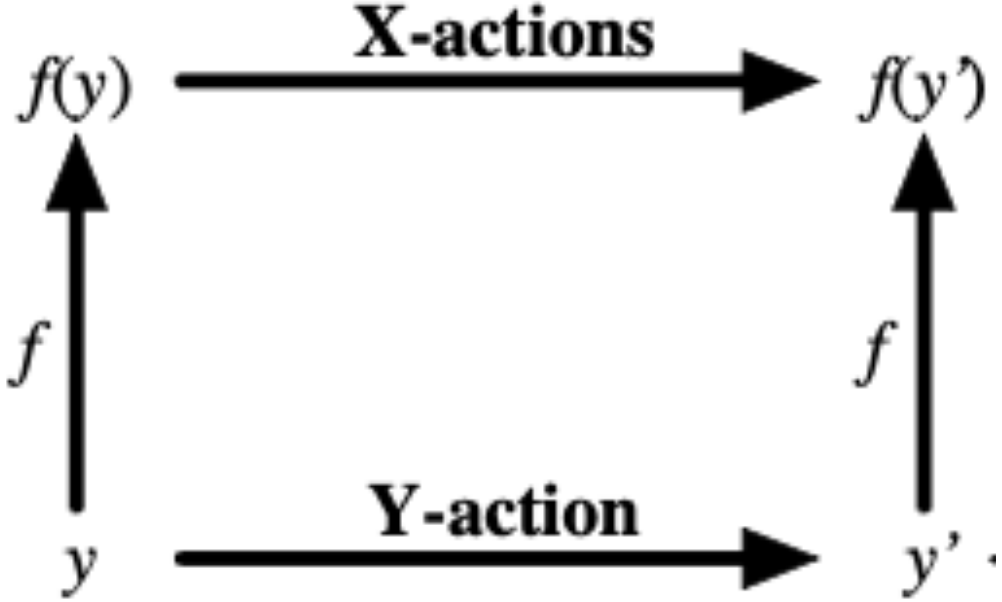
Proving Paxos $\Longrightarrow$ Consensus

The abstraction method

- **Define an abstraction function f:** Maps each state of implementation Y to a state of specification X
- Show that f(initial state of Y) = initial state of X
- For each Y-action and reachable state y: There exists a sequence of X-actions (possibly empty) that is externally the same.
- Define invariants that hold in all reachable states of Y and prove that every Y-action preserves them
- By induction on the sequence of Y-actions, construct an X-trace that matches externally

Proving Paxos $\Longrightarrow$ Consensus

The abstraction method



Proving Paxos $\Longrightarrow$ Consensus

Using .tla specs

- Using the abstraction method, we will now prove the following implications
  1. Voting.tla $\Longrightarrow$ Consensus.tla
  2. Paxos.tla $\Longrightarrow$ Voting.tla
- Combing this gives us Paxos.tla $\Longrightarrow$ Consensus.tla
- This shows Paxos implements Consensus!

So how to build a highly available system using consensus?

- Run replicated state machines
- Get consensus on each input to the state machines
- We discussed the most fault-tolerant algorithm for consensus without real-time guarantees
- This is Lamport's Paxos algorithm, based on repeating rounds until you get a majority, and ensuring every round after a majoriy has the same value