15-440/15-640 Distributed Systems

Distributed Replication

Lecture 12 – Thursday, Oct 7th, 2021

Carnegie Mellon University
School of Computer Science

FIRST,

Announcements

- **Midterm-1 in class, Tuesday Oct 12th, 10:10am – 11:30am**
  - Please try and arrive early. Between 10am and 10:05am
  - 1 page (double sided) cheat sheet allowed. First time for the course!
    - **<u>However</u>**, you must add your name, Andrew ID, *<u>and submit with exam.</u>*

- **P1 Part A (Due 10/08), Part B (10/14)**

- **HW2 Due 10/10 – we will release solutions promptly after deadline**

CONTEXT

Fault Tolerance Techniques So Far?

Redundancy: information / time / physical redundancy

- *E.g., used in airplanes*

Recovery: checkpointing and logging (ARIES)

- *E.g., used in commercial databases*

Previous (concurrency) protocols rely on recovery techniques

- *E.g., Two Phase Commit is not fault tolerant by itself*

Why not always use these techniques?

- → *Long wait in the case of failure*

OUR GOAL TODAY:

Stay Up During Failures

- **Provide a service**
- **Replicate the machines that serve clients**
- **Survive the failure of up to f replicas**
- **Provide identical service to a non-replicated version**
    - *(except more reliable, and perhaps different performance)*

LOOKING AHEAD

Outline for Today

- Consistency when content is replicated
- Primary-backup replication model
- Consensus replication model

LOOKING AHEAD

Outline for Today

- Consistency when content is replicated
- Primary-backup replication model
- Consensus replication model

CONTEXT

Simple Examples of Replication

- **Replicated web sites**
- **e.g., Google or Amazon:**
  - DNS-based load balancing (DNS returns multiple IP addresses for each name)
  - Hardware load balancers put multiple machines behind each IP address

- **When is replication easy? When hard?**
  - *Workload assumptions*

CONTEXT

Read-Only Content

Easy to replicate - just make multiple copies of it.

- **Performance boost 1:** Get to use multiple servers to handle the load;
- **Performance boost 2**: Locality. We'll see this later when we discuss CDNs, can often direct client to a replica near it
- **Availability boost:** Can fail-over (done at both DNS level -- slower, because clients cache DNS answers -- and at front-end hardware level)

CONTEXT

But *Read-Write* Data

Requires write replication, and some degree of consistency

- **Strict Consistency**
  - Read always returns value from latest write
- **Sequential Consistency**
  - All nodes see operations in some sequential order
  - Operations of each process appear in-order in this sequence

BUT FIRST,

A Question...

P1:	$W(x)a$		
P2:		$R(x)NIL$	$R(x)a$

Is this example demonstrating **strict consistency**?

A note on notation: $W_i(x)a$ denotes that process **Pi** writes value to data item x. Similarly, $R_i(x)a$ represents **Pi** reading x and is returned value b.

CONSISTENCY

Sequential Consistency (1)

P1:	*W(x)a*		
P2:		*R(x)NIL*	*R(x)a*

Behavior of two processes operating on the same data item. The horizontal axis is time.

- **P1:** Writes *W* value a to variable *x*
- **P2:** Reads *NIL* from *x* first and then *a*

CONSISTENCY

Sequential Consistency (2)

P1:	*W(x)a*				
P2:		*W(x)b*			
P3:			*R(x)b*		*R(x)a*
P4:				*R(x)b*	*R(x)a*

(a) A sequentially consistent data store.

P1:	*W(x)a*				
P2:		*W(x)b*			
P3:			*R(x)b*		*R(x)a*
P4:				*R(x)a*	*R(x)b*

(b) A data store that is not sequentially consistent

CONTEXT

But *Read-Write* Data

Requires write replication, and some degree of consistency

- **Strict Consistency**
  - Read always returns value from latest write
- **Sequential Consistency**
  - All nodes see operations in some sequential order
  - Operations of each process appear in-order in this sequence
- **Causal Consistency**
  - All nodes see potentially causally related writes in same order
  - But concurrent writes may be seen in different order on different machines

Causal Consistency (1)

P1:	*W(x)a*			*W(x)c*		
P2:		*R(x)a*	*W(x)b*			
P3:		*R(x)a*			*R(x)c*	*R(x)b*
P4:		*R(x)a*			*R(x)b*	*R(x)c*

This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

Causal Consistency (2)

P1:	*W(x)a*				
P2:		*R(x)a*	*W(x)b*		
P3:				*R(x)b*	*R(x)a*
P4:				*R(x)a*	*R(x)b*

A violation of a causally-consistent store.

(W(x)a causally related to R(x)a, W(x)b.)

EXAMPLES

Consistency Guarantees in the Real World

In practice we often have a choice

Google Mail

- Sending mail is replicated to ~2 physically separated datacenters (users hate it when they think they sent mail and it got lost); mail will pause while doing this replication.
  - Q: How long would this take with 2-phase commit? in the wide area?
- Marking mail read is only replicated in the background - you can mark it read, the replication can fail, and you'll have no clue (re-reading a read email once in a while is no big deal)

Weaker consistency is cheaper if you can get away with it.

Replication Strategies

What to replicate: State versus Operations

- Propagate only a **notification of an update**
  - *Sort of an "invalidation" protocol*
- **Transfer data** from one copy to another
  - *Read-to-Write ratio high, can propagate logs (save bandwidth)*
- Propagate the **update operation** to other copies
  - *Don't transfer data modifications, only operations – "Active replication"*

When to replicate: Push vs Pull

- Pull Based
  - *Replicas/Clients poll for updates (caches)*
- Push Based
  - *Server pushes updates (stateful)*

LOOKING AHEAD

Outline for Today

- Consistency when content is replicated
- Primary-backup replication model
- Consensus replication model

BUT FIRST,

Assumptions Today

Group membership manager

- Allow replica nodes to join/leave

Fail-stop (not Byzantine) failure model

- Servers might crash, might come up again
- Delayed/lost messages

Failure detector

- E.g., process-pair monitoring, etc.

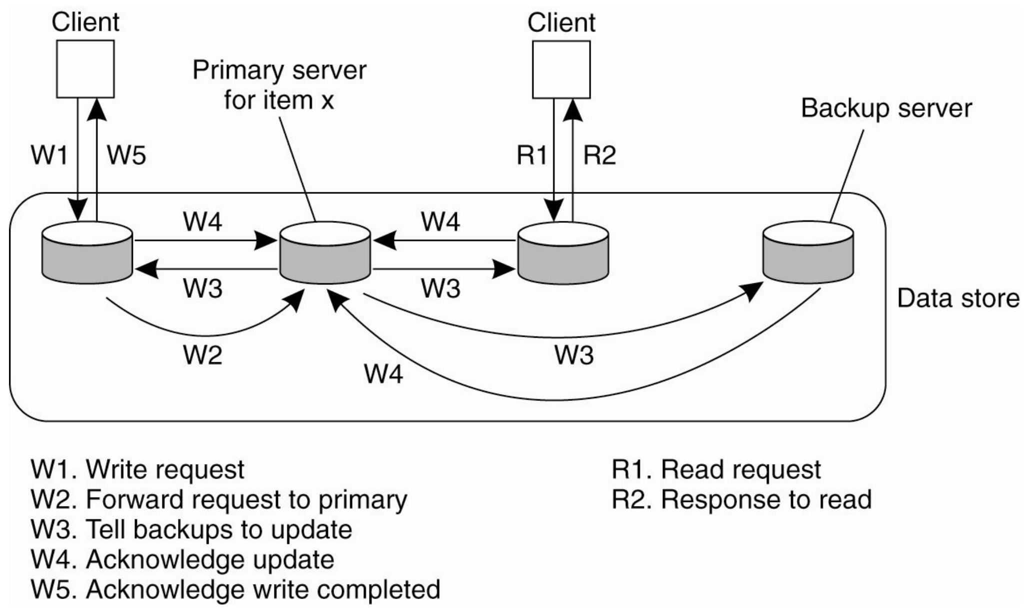
PRIMARY BACKUP

Remote Write Protocol

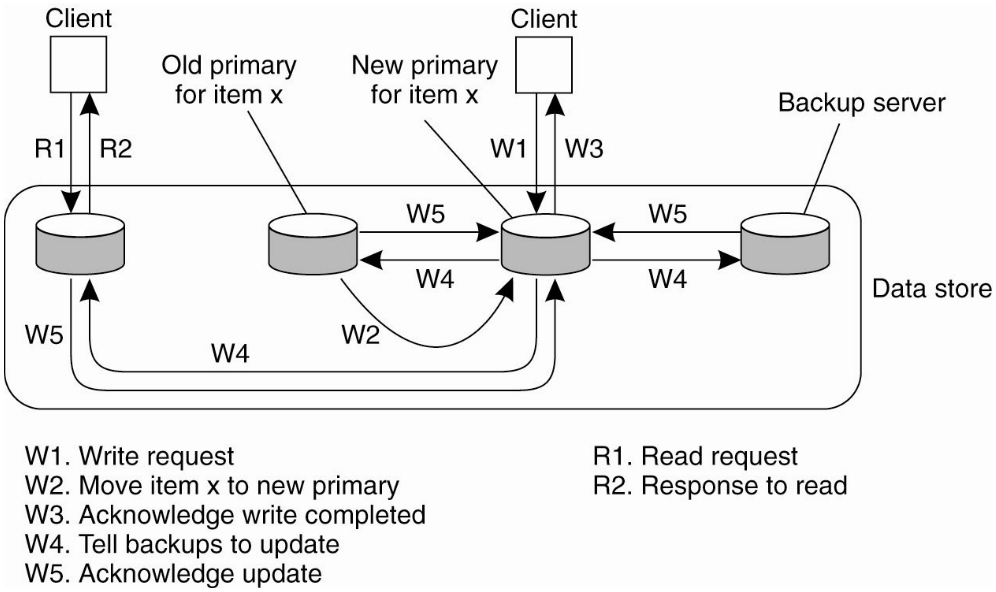
- Writes always go to primary, read from any backup

- Implementation
  - Stream the log

- Common in practice
  - Simple
- Are updates blocking?



Local-Write P-B Protocol



- Primary migrates to the process wanting to process update
- For performance, use non-blocking op.
- What does this scheme remind you of?

Properties of Primary Backup

This looks cool. How many failures can we deal with? What are some problems?

- What do we do if a replica has failed?
- We wait… how long? Until it's marked dead.

- Advantage: With N servers, can tolerate loss of N-1 copies
- Not a great solution if you want very tight response time even when something has failed: Must wait for failure detector

Note: If you don't care about strong consistency (e.g., the "mail read" flag), you can reply to client before reaching agreement with backups (sometimes called "asynchronous replication").

LOOKING AHEAD

Outline for Today

- **Consistency when content is replicated**
- **Primary-backup replication model**
- **Consensus replication model**

CONSENSUS

Quorum-Based Consensus

- Designed to have fast response time even under failures
- Operate as long as majority of machines is still alive

- No master, per se
- To handle f failures, must have $2f + 1$ replicas
- Also, for replicated-write => write to all replicas not just one

- Usually boils down to Paxos [Lamport]

The Paxos Approach

Decompose the problem:

Basic Paxos ("single decree"):

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

Multi-Paxos:

- Combine several instances of Basic Paxos to agree on a series of values forming the log

Requirements for Basic Paxos

Correctness (safety):

- Only a single value may be chosen
- A machine never learns that a value has been chosen unless it really has been
- The agreed value X has been proposed by some node

Liveness (termination) :

- Some proposed value is eventually chosen
- If a value is chosen, servers eventually learn about it

Fault-tolerance:

- If less than $N/2$ nodes fail, the rest should reach agreement eventually w.h.p
- **Liveness is not guaranteed**

[FLP'85] Impossibility Result

- **Synchronous** DS: bounded amount of time node can take to process and respond to a request
- **Asynchronous** DS: timeout is not perfect

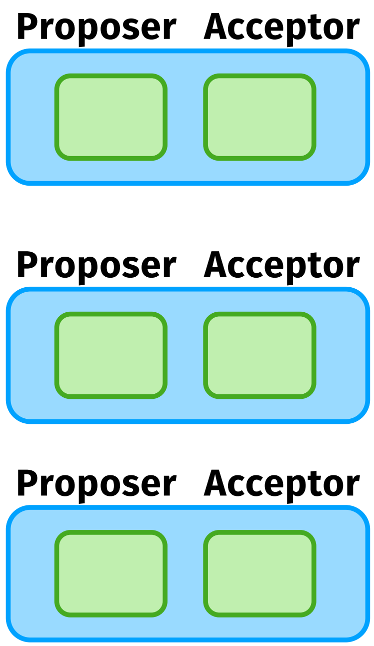
Fischer-Lynch-Paterson Result

It is impossible for a set of processors in an asynchronous system to agree on a binary value, even if only a single processor is subject to an unannounced failure.

Paxos Components

- **Proposers:**
  - Active: put forth particular values to be chosen
  - Handle client requests
- **Acceptors:**
  - Passive: respond to messages from proposers
  - Responses represent votes that form consensus
  - Store chosen value, state of the decision process

- **For this presentation:**
  - Each Paxos server contains both components
  - Ignore third role, aka Learner
- **"Round": (proposal, messages/voting, decision)**
  - We may need several rounds



Basic Two-Phase *(Strawman)*

- **Coordinator tells replicas: "Value *V*"**
- **Replicas ACK**
- **Coordinator broadcasts "Commit!"**

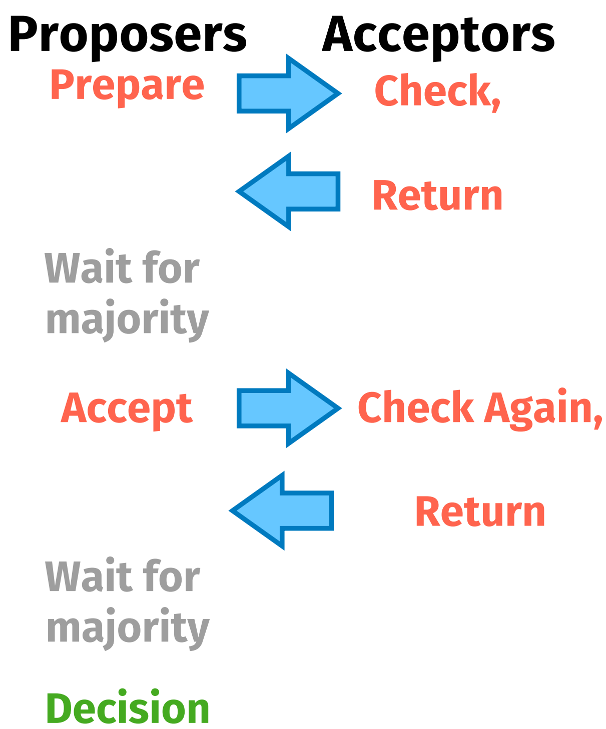
- **This isn't enough**
  - What if some of the nodes or the coordinator fails during the communication?
  - What if there is a network partition?
  - What if there's more than 1 coordinator at the same time?
  - What if new coordinator chooses a different value?

Let’s Discuss Some Problems & Solutions

- **Problem: can’t trust a single node**
  - Solution: everyone can potentially propose
- **Problem: several concurrent proposers**
  - Solution: Quorum (require majority of acceptors)
- **Problem: split votes, no proposer reaches majority**
  - Solution: acceptors need to allow updating of their value
- **Problem: conflicting choices (due to updating)**
  - Solution a): prioritize proposal with highest unique time stamp (Lamport clocks)
  - Solution b): once majority has agreed on value, future proposals forced to propose/choose same value

Single Decree Paxos: Informal Description

- **Phase 1: Prepare message**
  - Find out about any chosen values
  - Block older proposals that have not yet completed
- **Phase 2: Accept message**
  - Ask acceptors to accept a specific value
- **(Phase 3): Proposer decides**
  - If majority again: chosen value, commit.
  - If no majority: delay and restart Paxos



Single Decree Paxos: Protocol

Proposers

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

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

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

Acceptors

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

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

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

PAXOS

Single Decree Paxos: Protocol

Proposers

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

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

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

Acceptors

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

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

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

PAXOS

Single Decree Paxos: Protocol

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)

Proposers

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

Acceptors

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

Proposers

7) 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)*

Single Decree Paxos: Protocol

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)

Proposers

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

Acceptors

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

Proposers

7) 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)*

Single Decree Paxos: Protocol

Proposers

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

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

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

Acceptors

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

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

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

PAXOS

Single Decree Paxos: Protocol

Proposers

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

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

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

Acceptors

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

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

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

Single Decree Paxos: Protocol

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)

Proposers

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

Acceptors

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

Proposers

7) 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)*

Single Decree Paxos: Protocol

Proposers

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

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

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

Acceptors

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

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

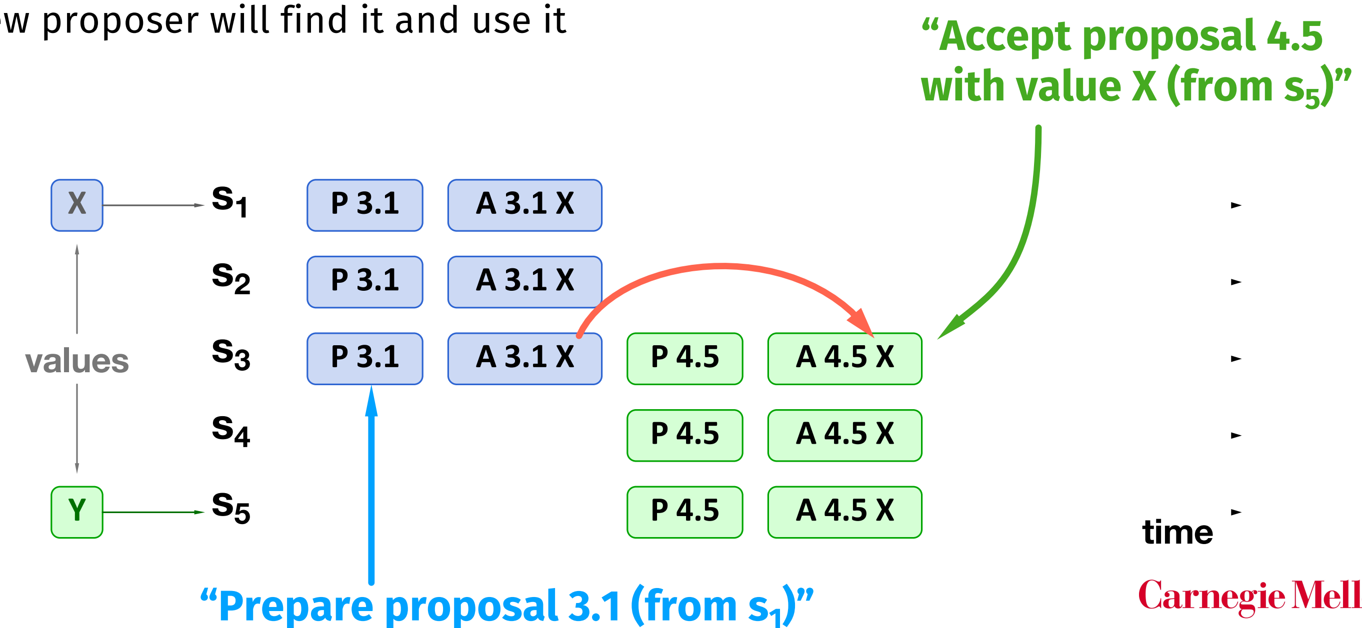
Acceptors must record ***minProposal***, ***acceptedProposal***, *and* ***acceptedValue*** *on stable storage (disk)*

Basic Paxos Examples

Three possibilities when later proposal prepares:

1. Previous value already chosen:

- New proposer will find it and use it

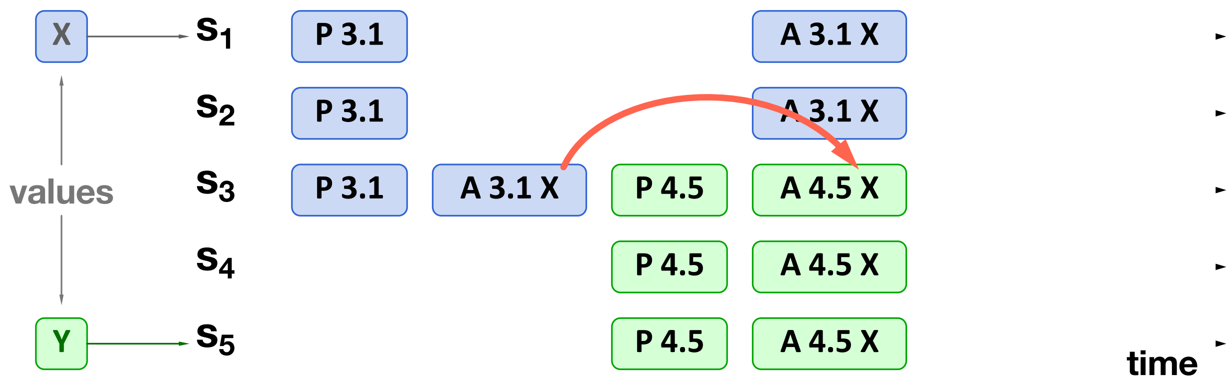


Basic Paxos Examples, cont'd

Three possibilities when later proposal prepares:

2. Previous value not chosen, but new proposer sees it:

- New proposer will use existing value
- Both proposers can succeed



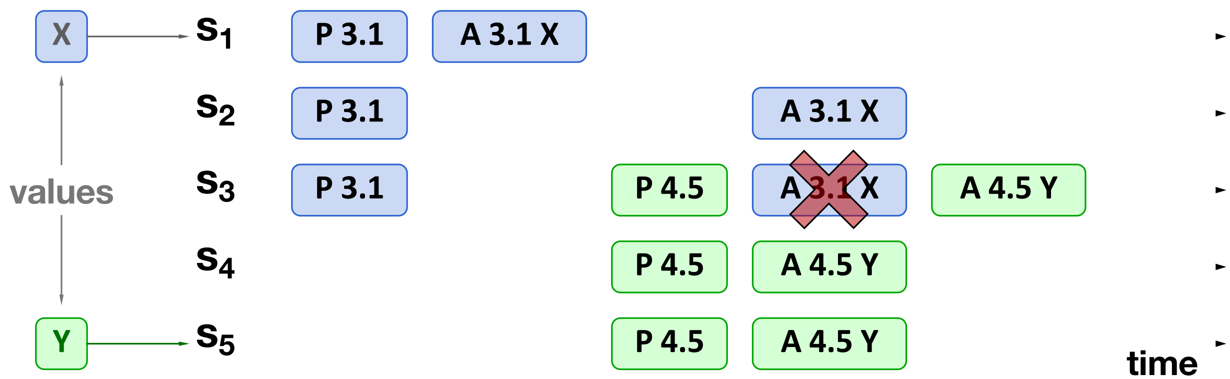
PAXOS

Basic Paxos Examples, cont'd

Three possibilities when later proposal prepares:

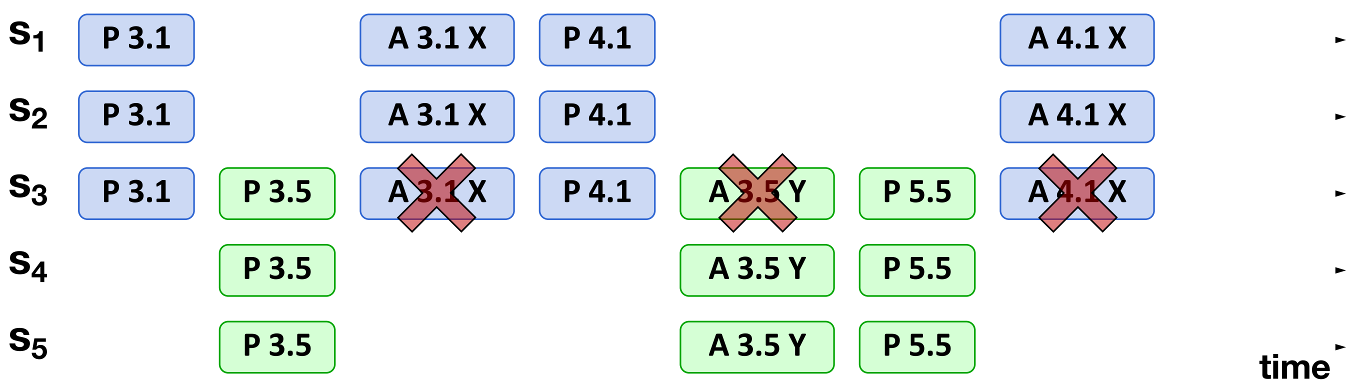
3. Previous value not chosen, new proposer doesn't see it:

- New proposer chooses its own value
- Older proposal blocked



Liveness

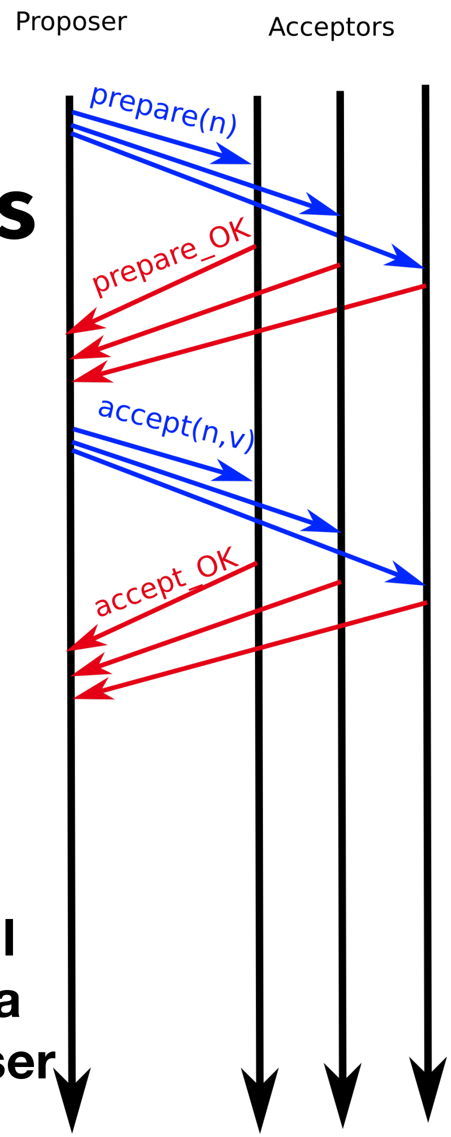
- Competing proposers can livelock:



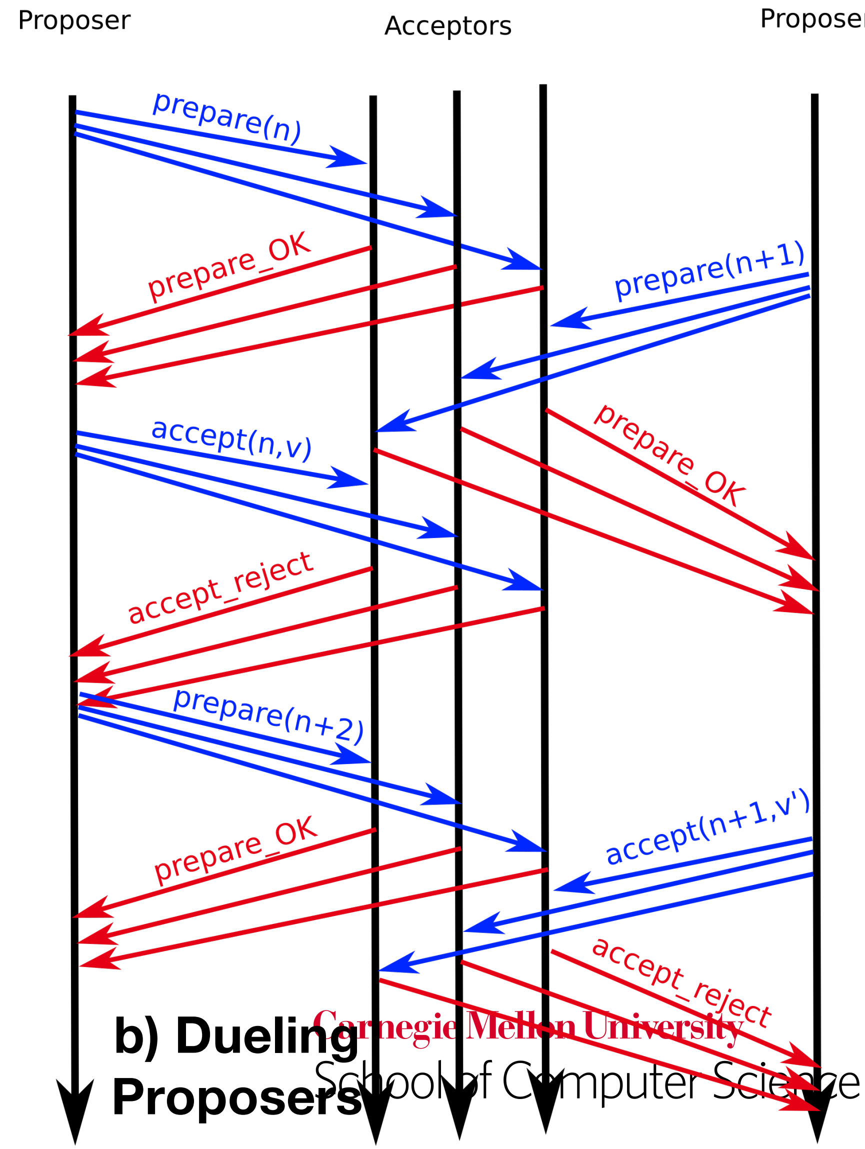
- One solution: randomized delay before restarting
  - Give other proposers a chance to finish choosing
- Multi-Paxos will use leader election instead

PAXOS

Paxos Examples



a) Successful Round with a Single Proposer



b) Dueling Proposers

Single Decree Paxos: Protocol

Proposers

1) Choose new proposal number *n*
2) Broadcast `Prepare(n)` to all servers
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
7) When responses received from majority:
   - Any rejections (result > n)? goto (1)
   - Otherwise, **value is chosen**

Acceptors

3) Respond to `Prepare(n)`:
   - If n > minProposal then minProposal = n
   - Prepare-OK(acceptedProposal, acceptedValue)
   - else
   - Prepare-REJECT()
6) Respond to `Accept(n,value)`:
   - If n ≥ minProposal then
     acceptedProposal = minProposal = n
     acceptedValue = value
   - Accept-OK()
   - else
   - Accept-REJECT()

PAXOS

Some Remarks

- **Only proposer knows chosen value (majority accepted)**
- **Only a single value is chosen -> MultiPaxos**

- **No guarantee that proposer's original value *v* is chosen by itself**

- **Number *n* is basically a Lamport clock -> always unique *n***
- **Key invariant:**
  - If a proposal with value *v* is chosen, all higher proposals must have value *v*

- **Dueling proposer**
  - Resolved using number *n* in prepare

- **There are challenging corner cases**

Paxos is Widespread!

- **Industry and academia**
  - Google: Chubby (distributed lock service)
  - Yahoo: Zookeeper (distributed lock service)
  - MSR: Frangipani (distributed lock service)
  - OpenSource implementations
    - Libpaxos (paxos based atomic broadcast)
    - Zookeeper is open source, integrated w/Hadoop

Paxos History

It took 25 years to come up with safe protocol

- 2PC proposed in 1979 (Gray)
- In 1981, Stonebraker proposed a basic, unsafe 3PC
- 1988, Brian Oki and Barbara Liskov created Viewstamped Replication, which has the core protocol.
- In 1998, Lamport rediscovered it and explained the protocol formally, naming it Paxos
- 2001 ”Paxos made simple”.
- In ~2007 RAFT appears, presenting the Viewstamped Replication approach to Paxos as a cleanly isolated protocol.

More Remarks

- Paxos is painful to get right, particularly the corner cases. Start from a good implementation if you can. See Yahoo's "Zookeeper" as a starting point.
- There are lots of optimizations to make the common / no or few failures cases go faster; if you find yourself implementing, research these.
- Paxos is expensive. Usually, used for critical, smaller bits of data and to coordinate cheaper replication techniques such as primary-backup for big bulk data.

Beyond Paxos

- **Many follow ups and variants**
- **RAFT consensus algorithm**
  - https://raft.github.io/
  - **Great visualization of how it works**
    - http://thesecretlivesofdata.com/raft/

WRAPPING UP

Summary

- **Primary-backup**
  - Writes handled by primary, stream log to backup(s)
  - Replicas are “passive”, follow primary
  - Good: Simple protocol. *N* machines, can handle *N-1* failures
  - Bad: Slow response times in case of failures.

- **Quorum consensus**
  - Designed to have fast response time even under failures
  - Replicas are “active” - participate in protocol; there is no master, per se.
  - Good: Clients don’t even see the failures
  - Bad: More complex (corner cases). To handle *f* failures, must have *2f + 1* replicas.