CSE 552
Paxos

Steve Gribble
Department of Computer Science & Engineering
University of Washington
Remember from last time...

VS
Replication for availability

Execute multiple replicas of a server

- keep the replicas “in sync” somehow

- if a replica fails, there is enough redundancy in the system for clients to receive service from the surviving replicas
Strawman #1

Simple idea:
- each client sends all writes to all available replicas

Issues?
Issue: ordering

Have to make sure each replica processes requests in the same order as other replicas

- somehow, the replicas need to agree on the order of inputs
Issue: determinism

Replicas need to be deterministic, otherwise they can diverge

- determinism: the next state is a function only of the input and the current state
- non-determinism
  - many potential sources: checking the system time, multithreaded timing, etc.
Issue: recovery

Failed replicas will miss requests from clients

- need to have some mechanism for them to play catch-up after they recover
Strawman #2

Primary-backup

- clients communicate with a single replica (the primary)
  ‣ primary chooses order of requests
  ‣ primary updates the backups
  ‣ primary “resolves” non-determinism

- backups detect failure of the primary using timeout
  ‣ clients failover to a backup in case of primary failure

Issues?
Issue: failure detection

The usual problem: how long of a timeout to set for our failure detector?

- too short and primary will be falsely accused of failure
- too long and availability/performance is affected
Issue: lag

When is it safe for the primary to ACK a client’s request?

- option #1: wait until all backups are updated
  ‣ slow but safe
- option #2: once primary is updated, but before backups
  ‣ fast but unsafe
Issue: primary election

If primary fails, which backup should become the new primary?

- issue becomes complicated in the case of multiple simultaneous perceived failures -- laggy networks or partitions, for example
- need to come to agreement on who the primary is, otherwise have dueling primaries
Issue: recovery

Same problem as with initial strawman
- once primary has failed, need to bring it back online and play catch-up with the new primary
Paxos

A set of protocols for dealing with replicated state machines

- attempts to solve several problems
  ‣ agreeing on the order and value of inputs
  ‣ dealing with asynchrony (both of network and of processors)
  • hence, dealing with failure (both of network and of processors)
The “synod” protocol

The basic building block of Paxos

- goal: get the system to agree on a single value
- three roles: proposers, acceptors, learners
  - proposers issue a series of rounds of proposals, suitably constrained
  - a value is “chosen” when a majority of acceptors accept it
  - sometime later, learners learn that the value is chosen
Outline

Synod
- how it works
- why it works

Replicated state machine protocol
- how it works
- optimizations
Outline

Synod
- how it works
- why it works

Replicated state machine protocol
- how it works
- optimizations
Synod

Two phase protocol

- **phase 1:** proposer decides it wants to propose a value, and to do that, has to learn constraints on what can be proposed

- **phase 2:** proposer proposes a value, acceptors accept or reject it
Phase 1, step a

Proposer:

- selects an unused proposal number N
  ‣ each proposer owns some subset of proposal number space
  ‣ e.g., proposal number = localcount.proposerID
- sends “prepare” request with N to a majority of acceptors

Implications

- proposer must stably store previously used local proposer #s and proposer ID
Phase 1, step b

Acceptor:

- if acceptor receives a “prepare” request with number N, and N is greater than any prepare request to which it has responded:
  
  ‣ it makes a promise not to accept any more proposals \(< N
  
  ‣ it responds with the highest numbered proposal, if any, it has already accepted

- otherwise, do nothing

Implications: must stably store...

- highest numbered prepare request to which it has responded
- highest numbered proposal/value that it has ever accepted
Phase 2, step a

Proposer:

- if receives a response to step 1b from a majority of acceptors:
  ‣ send to a majority of acceptors an “accept” request for proposal numbered N with value V
  ‣ V is the value of the highest-numbered proposal among responses, or any value if responses reported no proposals
- else, do nothing
Phase 2, step b

Acceptor:

- if receive an “accept” request for proposal N value V:
  
  ‣ accept the proposal, unless it has already responded to a prepare request having number greater than N

  ‣ optional: inform a distinguished learner of the outcome

- else do nothing
Common case is simple

\[ N = 0.0 \]

proposer + learner

acceptor
\[ (N, V)_{accept} = \{ \} \]
\[ N_{prepare} = \{ \} \]

acceptor
\[ (N, V)_{accept} = \{ \} \]
\[ N_{prepare} = \{ \} \]

acceptor
\[ (N, V)_{accept} = \{ \} \]
\[ N_{prepare} = \{ \} \]
Common case is simple

N = 1.0

proposer + learner

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = \{ \}

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = \{ \}

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = \{ \}
Common case is simple.

N = 1.0

proposer + learner

prepare (N=1.0)

acceptor

(N,V)accept = { }
Nprepare = { }

acceptor

(N,V)accept = { }
Nprepare = { }

acceptor

(N,V)accept = { }
Nprepare = { }

CSE552 paxos // 04-29-13 // gribble
Common case is simple

\[ N = 1.0 \]

- proposer + learner

\[ (N, V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.0 \]

- acceptor

\[ (N, V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.0 \]
Common case is simple

\[
N = 1.0
\]

(proposer + learner) \rightarrow

prepared \((N, V = \{ \})\)

\rightarrow

prepared \((N, V = \{ \})\)

\rightarrow

prepared \((N, V = \{ \})\)

\rightarrow

\((N, V)_{accept} = \{ \}\)

\(N_{prepare} = 1.0\)
Common case is simple

N = 1.0

proposer + learner

accept (N,V=1.0,X) → acceptor

accept (N,V=1.0,X)

accept (N,V=1.0,X) → acceptor

acceptable = { }  
N_{prepare} = 1.0

acceptor

accept (N,V=1.0,X) → acceptor

acceptable = { }  
N_{prepare} = 1.0

acceptor

acceptor

acceptable = { }  
N_{prepare} = 1.0

acceptor
Common case is simple

- N = 1.0

**proposer + learner**

- (N,V)accept = 1.0,X
- Nprepare = 1.0

**acceptor**

- (N,V)accept = 1.0,X
- Nprepare = 1.0
Common case is simple

\[
\begin{align*}
N &= 1.0 \\
\text{proposer + learner} &\quad\text{accepted \,(N,V = 1.0,X)} \\
\text{acceptor} &\quad\text{accepted \,(N,V = 1.0,X)} \\
\text{acceptor} &\quad\text{accepted \,(N,V = 1.0,X)} \\
\text{acceptor} &\quad\text{accepted \,(N,V = 1.0,X)}
\end{align*}
\]
Worst case is unlikely

N = 0.0

proposer + learner

(N,V)_{accept} = \{\}
N_{prepare} = \{\}

N = 0.1

proposer + learner

(N,V)_{accept} = \{\}
N_{prepare} = \{\}

CSE552 paxos // 04-29-13 // gribble
Worst case is unlikely

\[ N = 1.0 \]

\[ (N, V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = \{ \} \]

\[ N = 0.1 \]

\[ (N, V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = \{ \} \]
Worst case is unlikely

N = 1.0

proposer + learner

prepare (N=1.0)

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = \{ \}

N = 0.1

proposer + learner

prepare (N=1.0)

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = \{ \}

prepare (N=1.0)

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = \{ \}
Worst case is unlikely

\[
\begin{align*}
N = 1.0 & \quad \text{proposer + learner} \\
N = 0.1 & \quad \text{proposer + learner}
\end{align*}
\]
Worst case is unlikely

\[ N = 1.0 \]

\[ \text{prepared} \ (N, V = \{ \}) \]

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.0 \]

\[ N = 0.1 \]

\[ \text{prepared} \ (N, V = \{ \}) \]

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.0 \]
Worst case is unlikely

\[
\begin{align*}
\text{proposer + learner} & \quad \text{acceptor} \\
N = 1.0 & \quad (N,V)_{\text{accept}} = \{ \} \\
\text{prepare (N=1.1)} & \quad N_{\text{prepare}} = 1.0 \\
\text{proposer + learner} & \quad \text{acceptor} \\
N = 1.1 & \quad (N,V)_{\text{accept}} = \{ \} \\
\text{prepare (N=1.1)} & \quad N_{\text{prepare}} = 1.0 \\
\text{proposer + learner} & \quad \text{acceptor} \\
& \quad (N,V)_{\text{accept}} = \{ \} \\
& \quad N_{\text{prepare}} = 1.0
\end{align*}
\]
Worst case is unlikely

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.1 \]

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.1 \]

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.1 \]
Worst case is unlikely

N = 1.0
proposer + learner

N = 1.1
proposer + learner

acceptor
(N,V)_{accept} = \{ \}
N_{prepare} = 1.1

acceptor
(N,V)_{accept} = \{ \}
N_{prepare} = 1.1

acceptor
(N,V)_{accept} = \{ \}
N_{prepare} = 1.1

prepared (N,V = \{ \})

prepared (N,V = \{ \})

prepared (N,V = \{ \})
Worst case is unlikely

\[ N = 1.0 \]

\[ \text{proposer + learner} \]

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.1 \]

\[ N = 1.1 \]

\[ \text{proposer + learner} \]

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.1 \]

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.1 \]

\[ (N,V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 1.1 \]
Worst case is unlikely

\[ N = 1.0 \]

\[ N = 1.1 \]
Worst case is unlikely

N = 2.0

N = 1.1

proposer + learner

acceptor

acceptor

acceptor

\[(N, V)_{accept} = \{ \} \]

\[N_{prepare} = 1.1\]
Worst case is unlikely

- Proposer + learner
  - $N = 2.0$
  - Prepare (N=2.0)
  - (N,V)_{accept} = \{ \}
  - $N_{prepare} = 1.1$

- Proposer + learner
  - $N = 1.1$
  - Prepare (N=2.0)
  - (N,V)_{accept} = \{ \}
  - $N_{prepare} = 1.1$
Worst case is unlikely

\[
\begin{align*}
N &= 2.0 \\
\text{proposer + learner} & \quad \text{acceptor} \\
(N,V)_{\text{accept}} &= \{ \} \\
N_{\text{prepare}} &= 2.0 \\
N &= 1.1 \\
\text{proposer + learner} & \quad \text{acceptor} \\
(N,V)_{\text{accept}} &= \{ \} \\
N_{\text{prepare}} &= 2.0 \\
\end{align*}
\]
Worst case is unlikely

\(N = 2.0\)

proposer + learner

\((N,V)_{accept} = \{\}\)
\(N_{prepare} = 2.0\)

acceptor

\(N = 1.1\)

proposer + learner

\((N,V)_{accept} = \{\}\)
\(N_{prepare} = 2.0\)

acceptor

\(N = 1.1\)

proposer + learner

\((N,V)_{accept} = \{\}\)
\(N_{prepare} = 2.0\)

acceptor
Worst case is unlikely

N = 2.0
proposer + learner

N = 1.1
proposer + learner

accept (N,V=1.1,Y)

accept (N,V=1.1,Y)

accept (N,V=1.1,Y)

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = 2.0

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = 2.0

acceptor

(N,V)_{accept} = \{ \}
N_{prepare} = 2.0
Worst case is unlikely

N = 2.0

N = 1.1

(Case 1)

(Case 2)
Worst case is unlikely

N = 2.0

(proposer + learner)

N = 2.1

(proposer + learner)

prepare (N=2.1)

acceptor

\((N,V)_{\text{accept}} = \{\} \)

\(N_{\text{prepare}} = 2.0\)

acceptor

\((N,V)_{\text{accept}} = \{\} \)

\(N_{\text{prepare}} = 2.0\)

acceptor

\((N,V)_{\text{accept}} = \{\} \)

\(N_{\text{prepare}} = 2.0\)
Worst case is unlikely

\[ N = 2.0 \]

- Proposer + Learner

\[ N = 2.1 \]

- Proposer + Learner

- Acceptor
  
  - Acceptor
    
    \[ (N,V)_{\text{accept}} = \{ \} \]
    \[ N_{\text{prepare}} = 2.1 \]
Worst case is unlikely

\[ N = 2.0 \]

- proposer + learner

\[ N = 2.1 \]

- proposer + learner

\[ (N, V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 2.1 \]

\[ (N, V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 2.1 \]

\[ (N, V)_{\text{accept}} = \{ \} \]
\[ N_{\text{prepare}} = 2.1 \]
Worst case is unlikely

\[ N = 2.0 \]

Proposer + Learner

\[ (N,V)_{accept} = \{ \} \]
\[ N_{prepare} = 2.1 \]

\[ accept \ (N,V=2.0,X) \]

\[ accept \ (N,V=2.0,X) \]

\[ accept \ (N,V=2.0,X) \]

\[ (N,V)_{accept} = \{ \} \]
\[ N_{prepare} = 2.1 \]

\[ (N,V)_{accept} = \{ \} \]
\[ N_{prepare} = 2.1 \]

\[ (N,V)_{accept} = \{ \} \]
\[ N_{prepare} = 2.1 \]
Worst case is unlikely

- **N = 2.0**
  - Proposer + Learner
  - Reject \( (N_{\text{prepare}}=2.1) \)

- **N = 2.1**
  - Proposer + Learner
  - Reject \( (N_{\text{prepare}}=2.1) \)

**Acceptor**
- \( (N,V)_{\text{accept}} = \{ \} \)
- \( N_{\text{prepare}} = 2.1 \)
Worst case is unlikely

\[ N = 2.0 \]
proposer + learner

\[ N = 2.1 \]
proposer + learner

acceptor
\( (N,V)_{\text{accept}} = \{ \} \)
\( N_{\text{prepare}} = 2.1 \)

acceptor
\( (N,V)_{\text{accept}} = \{ \} \)
\( N_{\text{prepare}} = 2.1 \)

acceptor
\( (N,V)_{\text{accept}} = \{ \} \)
\( N_{\text{prepare}} = 2.1 \)

AND SO ON...
Fun paxos games

Assume there are 5 agents, and three leaders L1/L2/L3

- leaders don’t know if a value is yet chosen

Leader L1 issues prepare in round 3. L1, and gets back:

- (2.L2, X), -, -, (2.L2, X), (2.L2, X)
  - what is the correct next step?
Fun paxos games

Leader L1 issues prepare in round 3. L1, and gets back:

- (2.L2, X), -, -, -, (2.L2, X)

› what is the correct next step?
Fun paxos games

Leader L1 issues prepare in round 3. L1, and gets back:

- (2.L2, X), - , (2.L3, Y), - , (2.L2, X)

› what is the correct next step?
Fun paxos games

Leader L1 issues prepare in round 3. L1, and gets back:
- {}, {}, (2.L3, Y), {}, {}  
  ▸ what is the correct next step?
Outline

Synod
- how it works
- why it works

Replicated state machine protocol
- how it works
- optimizations
Deriving paxos

Context

- assume an asynchronous system
  - messages can be dropped, reordered, delayed, but not corrupted
  - agents can take arbitrarily long to respond to messages
- assume fail-stop failures only
  - agents function correctly, or not at all
What Paxos promises

Paxos will:

- guarantee “safety” under all circumstances
  ‣ including many simultaneous leaders, high rate of failure/recovery
- terminate under some circumstances
  ‣ if a single leader runs by itself in a round for a long enough time period that it can talk to a majority of agents twice
Safety

“Safety” = consistency + validity

- only a single value is chosen
  - an agent never learns that a value is chosen unless it has been
- only a value that has been proposed may be chosen
Let’s start deriving

Imagine a single leader exists, does phase 1a, sends out accepts in 2a, then dies.

- if a majority of agents hear 1a and 2a, the proposal must be chosen according to our termination criteria

- HENCE, an agent must accept the first proposal it hears
Only a single value is chosen

Assume in round M that value V is chosen

- then, every higher-numbered proposal that is chosen must have value V
  - but, a proposer can’t predict if its proposal will be chosen

- Hence, if a proposal M with value V has been chosen, every higher-numbered proposal must have value V
Implications

During phase 1, a proposer must find out what proposals might have been chosen already

- and if it is conceivable that a proposal has been chosen, it must select the same value for its future proposals

During phase 1, a proposer must prevent “temporally concurrent” proposals from previous rounds from being chosen

- otherwise it might not learn about them
So...

During phase 1

- acceptors must:
  ‣ (a) report back on values they have accepted
  ‣ (b) promise not to accept values from lower-numbered proposals

- proposer must:
  ‣ assume the highest-numbered, accepted proposal might have been chosen, and adopt it for its next proposal in phase 2

Together, can use induction to prove safety
Why majority?

If a proposal is chosen, a majority of agents accepted the value

- any two majority sets share at least one agent
- during interrogation in phase 1, if you hear back from a majority of agents, at least one will be in that “chosen” set
Outline

Synod
- how it works
- why it works

Replicated state machine protocol
- how it works
- optimizations
Model

client

D W Y X

D W Y X

D W Y X

proposer + learner + acceptor

proposer + learner + acceptor

proposer + learner + acceptor
Implementation

One paxos agent is elected to be the “leader”
- we’ll talk about how in a bit

All clients funnel their requests through the leader
- the leader orders the requests
- for each request, the leader runs a paxos synod instance
  ‣ the Jth instance of paxos synod determines the Jth command in the sequence passed to the replicas
Implementation

client

request
response

leader

paxos group

client

proposer + learner + acceptor

proposer + learner + acceptor

proposer + learner + acceptor

client

synod
Newly elected leader

Since L is a learner in all instances of consensus, it should already know most of the commands that have been chosen. For example, it might know commands 1-10, 13, and 15.

- it executes phase 1 of 11, 12, and 14, and of all instances 16 and larger
  
  ‣ maybe it learns 14 and 16 are constrained, and 11, 12, and all commands after 16 are unconstrained
  
  ‣ L executes phase 2 of 14 and 16, choosing commands for them
Stop-gap

Now, all replicas can execute 1-10, but not 13-16, because 11 and 12 haven’t yet been chosen

- L can either:
  - (a) take the next two commands issued to be 11 and 12
  - (b) immediately propose a special “no-op” command for 11 and 12
- L then runs phase 2 of consensus for 11 and 12
  - once consensus achieved, all replicas execute all commands through 16
Multipaxos

Now, the leader has executed phase 1 for all open slots

- it can just proceed to phase 2 for those slots

- “short-circuit” the two-phase protocol in the common case
On leader failure...

Any of the surviving agents can self-promote to leader

- do so by running phase 1
- paxos synod takes care of multiple concurrent self-promotions
- if you get your value chosen, you’re the new leader, otherwise, step back
Membership changes

We need to distinguish between:
- temporary failure plus eventual recovery
- permanent failure leading to membership change

Membership change requires consensus
- use Paxos to agree on membership change proposals