Advanced Course Distributed Systems

Replicated Logs and State Machines

Paris Carbone

COURSE TOPICS

- Intro to Distributed Systems
- Basic Abstractions and Failure Detectors
- Reliable and Causal Order Broadcast
- Distributed Shared Memory
- Consensus (Paxos, Raft, etc.)
- Replicated State Machines + Virtual Logs
- Time Abstractions and Interval Clocks (Spanner etc.)
- Consistent Snapshotting (Stream Data Management)
- Distributed ACID Transactions (Cloud DBs)

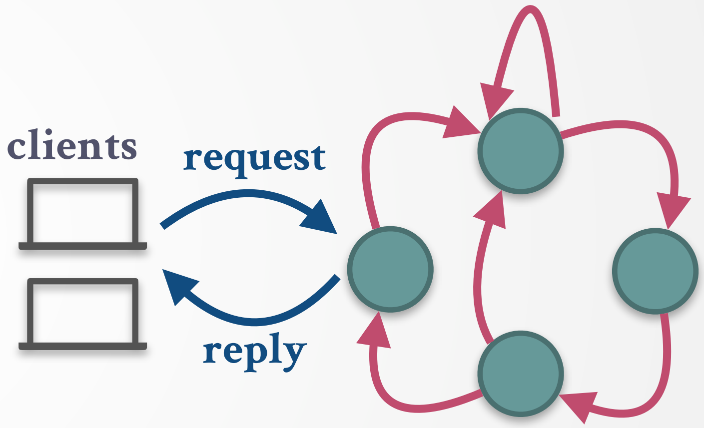
Motivation

- We wish to implement a Replicated State Machine (RSM).
- Processes need to agree on the sequence of commands (or messages) to execute.
- The standard approach is to use multiple instances of Paxos for single-value consensus (MultiPaxos).

State Machines

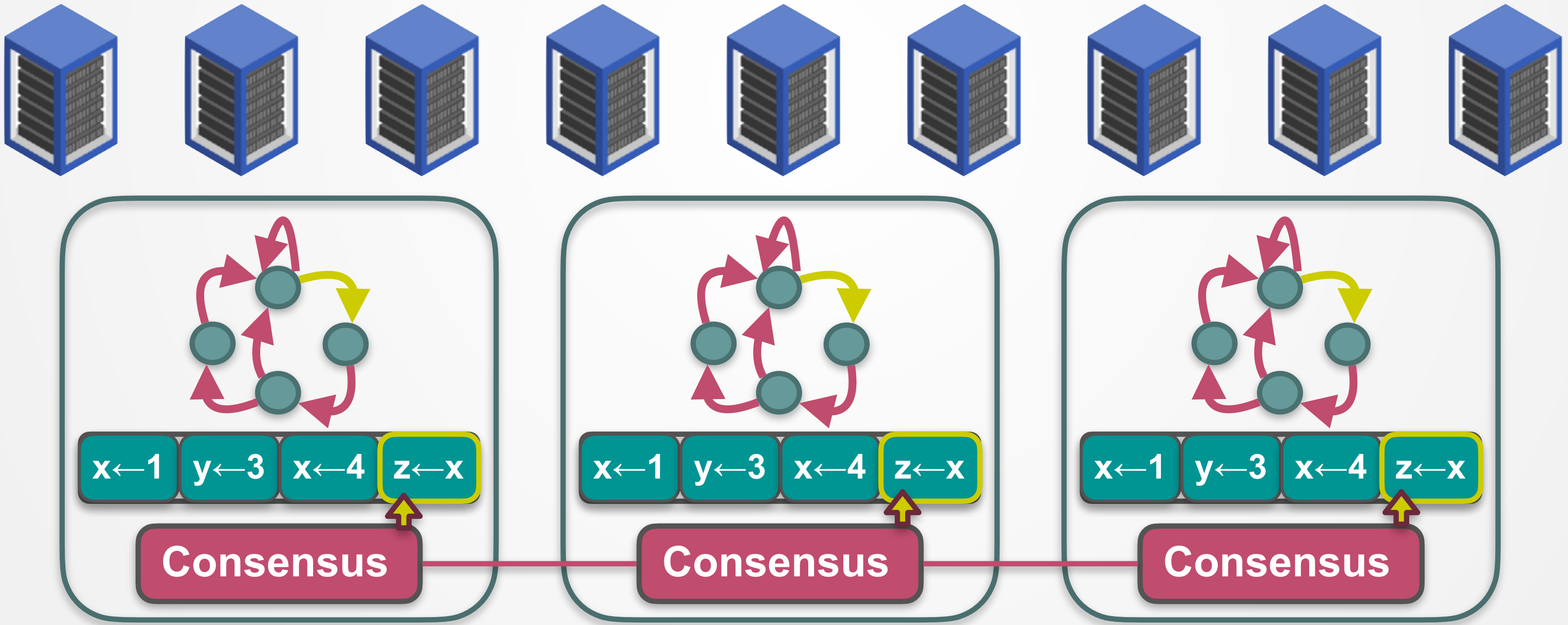
A State Machine

- **Executes** a **sequence** of commands
- **Transforms** its **state** and may produce some **output**
- Commands are **deterministic**
  - i.e., **Outputs** of the state machine are solely **determined** by the initial state and by the **sequence** of commands that it has executed



REPLICATED STATE MACHINES

- A **Replicated Log** ensures state machines execute same commands in same order.
- **Consensus** guarantees agreement on command sequence in the replicated log.
- System makes progress as long as any majority of servers are up.



MultiPaxos Approach

- Consensus is an agreement on a **single** value/command
- Let us use **multiple Paxos instances**. (MultiPaxos)

- Single-value consensus has two events
  - Request: Propose(C)
  - Indication/Response: Decide(C')

MultiPaxos Approach

- Consensus is agreement on a single value
- Let us use multiple instances of Paxos
- Organise the algorithm in rounds

MultiPaxos Approach

Initially all processes p_j (servers) are at round 1

- **ProCmds** := ∅; **Log** := ⟨⟩; s_0 (initial state); **proposed** := **false**
- A client q that wants to execute a command C, triggers **rb-broadcast** $\langle C, Pid_q \rangle$
- **upon** delivery $\langle C, Pid_q \rangle$ at p_j , the command pair is added to *ProCmds* unless it is already in *Log*.

MultiPaxos Approach

- At round *i*, each server p_j:
  - Start new instance *i* of Paxos (single-value)
- **If** *ProCmds* $\neq \emptyset \wedge$ not *proposed***:**
  - Choose a command $\langle C, Pid \rangle$ in *ProCmds*
  - **Propose** $\langle C, Pid, i \rangle$ in instance i; *proposed* := **true**
- **upon Decide**($\langle C_d, Pid', i \rangle$):
  - remove $\langle C_d, Pid' \rangle$ from *ProCmds;* Append (C_d, *Pid', i*) to *Log*
  - Execute C_d on s_{i-1} to get (s_i , res_i) and return res_i to *Pid'*
  - *Proposed* := **false**;
  - Move to the next round i+1

MULTIPAXOS … CAN BE A MESS

- The algorithms works
- This algorithm is sequential!
  - In order to select a command at round i any process (learner) have to agree on the sequence of commands $C_1 \dots C_{i-1}$
  - Using Paxos every round takes 4 communication steps, 2 for the prepare phase, and 2 for the accept phase
- Not easy to pipeline proposals
  - Same proposal C might end decided in different slots
  - Holes in the *Log* might arise

Sequence Consensus

WHAT IS THE PROBLEM?

- We need to agree on each command
  - Handled well by Paxos
- We also need to agree on the sequence of commands
  - A mismatch with the consensus specification
- We would like to agree on a **growing sequence of commands**

Consensus Mismatch

- **Integrity** property says that a process can decide at most one value
  - ”Cannot change one’s mind”
- But, we don’t want to change what’s been decided before
  - Just extend it with more information
- This is allowed by Sequence Consensus
  - Can decide again if old decided sequence is a prefix of the new one

CONSENSUS PROPERTIES

- **Validity**
  - Only proposed values may be decided
- **Uniform Agreement**
  - No two processes decide different values
- **Integrity**
  - Each process can decide at most one value
- **Termination**
  - Every correct process eventually decides a value

Sequence Consensus Properties

- Validity
  - If process p decides v then v is a sequence of proposed commands (without duplicates)
- Uniform Agreement
  - If process p decides u and process q decides v then one is a prefix of the other
- Integrity
  - If process p decides u and later decides v then u is a strict prefix of v

- Termination (liveness)
  - If command C is proposed by a correct process then eventually every correct process decides a sequence containing C

Sequence Consensus

- Event Interface
  - **propose(C)**
    - request event to append single command C to the sequence of decided command
  - **decide(CS)**
    - Indication event where CS is a decided command sequence
- Abortable Sequence Consensus adds
  - **abort**
    - Indication event

Sequence-Paxos

Roadmap: From Paxos to Sequence-Paxos

- Make the minimal modifications to Paxos to obtain correct **Sequence-Paxos** algorithm
- Then add optimizations to make the algorithm efficient
- In Paxos each process may assume any or all of the three roles: proposer, acceptor, and learner

Initial State for Paxos

- Proposer
  - $n_p := 0$ Proposer's current round number
  - $v_p := \bot$ Proposer's current value
- Acceptor
  - $n_{prom} := 0$ Promise not to accept in lower rounds
  - $n_a := 0$ Round number in which a value is accepted
  - $v_a := \bot$ Accepted value
- Learner
  - $v_d := \bot$ Decided value

Paxos Algorithm

Proposer

On ⟨Propose, C⟩ :

- n_p := unique higher proposal number
- S := ∅, acks := 0
- **send** ⟨Prepare, n_p⟩ **to** all acceptors

On ⟨Promise, n, n', v'⟩ **s.t.** $n = n_p$:

- add (n', v') to S (multiset union)
- **if** $|S| = \lceil (N+1)/2 \rceil$:
  - (k, v) := max(S) // **adopt v**
  - v_p := if v ≠ ⊥ then v else C
  - **send** ⟨Accept, n_p, v_p⟩ **to** all acceptors

On ⟨Accepted, n⟩ **s.t.** $n = n_p$:

- acks := acks + 1
- **if** $acks = \lceil (N+1)/2 \rceil$:
  - **send** ⟨Decide, v_p⟩ **to** all learners

On ⟨Nack, n⟩ **s.t.** $n = n_p$:

- trigger Abort()
- n_p := 0

Acceptor

- **On** ⟨Prepare, n⟩:
  - **if** $n_{prom} < n$:
  - n_{prom} := n
  - **send** ⟨Promise, n, n_a, v_a⟩ **to** Proposer
  - **else: send** ⟨Nack, n⟩ **to** Proposer

- **On** ⟨Accept, n, v⟩:
  - **if** $n_{prom} \leq n$:
  - n_{prom} := n
  - $(n_a, v_a) := (n, v)$
  - **send** ⟨Accepted, n⟩ **to** Proposer
  - **else: send** ⟨Nack, n⟩ **to** Proposer

Learner

- **On** ⟨Decide, v⟩:
  - If v_d = ⊥:
  - v_d := v
  - **trigger** Decide(v_d)

max(S) is any element (k, v) of S s.t. k is highest proposal number

From Paxos to Sequence-Paxos

- Values are sequences
  - $\bot$ is the empty sequence ($\bot = \langle\rangle$)
- We make two changes:
  - After adopting a value (seq) with highest proposal number, the proposer is allowed to extend the sequence with (nonduplicate) new command(s)
  - Learner that receives ⟨Decide, v⟩ will decide v if v is longer sequence than previously decided sequence

AGREEING ON (NON-DUPLICATE) COMMANDS

- As a client is allowed to issue the same (instance) command C multiple times we cannot avoid proposing the same command C multiple times
- We hide this issue in the sequence append operator ⊕:
- Non-duplicate ⊕ :
  - $$\langle C_1, \ldots, C_m \rangle \oplus C \stackrel{\text{def}}{=} \begin{cases} \langle C_1, \ldots, C_m \rangle \; \mathit{if} \text{ C is equal some } C_i \\ \langle C_1, \ldots, C_m, C \rangle, \text{otherwise} \end{cases}$$
- Duplication allowed ⊕
  - $\langle C_1, \ldots, C_m \rangle \oplus C \stackrel{\text{def}}{=} \langle C_1, \ldots, C_m, C \rangle$

INITIAL STATE FOR SEQUENCE PAXOS

- Proposer
  - $n_p := 0$ Proposer's current round number
  - $v_p := \langle\rangle$ Proposer's current value (empty sequence)
- Acceptor
  - $n_{\text{prom}} := 0$ Promise not to accept in lower rounds
  - $n_{\text{a}} := 0$ Round number in which a value is accepted
  - $v_{\text{a}} := \langle\rangle$ Accepted value (empty sequence)
- Learner
  - $v_{\text{d}} := \langle\rangle$ Decided value (empty sequence)

Sequence Paxos Algorithm

Proposer

On ⟨Propose, C⟩ :

n_p := unique higher proposal number
S := ∅, acks := 0
send ⟨Prepare, n_p⟩ **to** all acceptors

On ⟨Promise, n, n', v'⟩ **s.t.** n = n_p:

add (n', v') to S (multiset union)
if |S|= ⌈(N+1)/2⌉:
(k, v) := max(S) **// adopt v**
v_p := if v ≠ ⊥ then v else ⟨⟩
v_p := v ⊕ ⟨C⟩
send ⟨Accept, n_p, v_p⟩ **to** all acceptors

On ⟨Accepted, n⟩ **s.t.** n = n_p:

acks := acks + 1
if acks = ⌈(N+1)/2⌉:
send ⟨Decide, v_p⟩ **to** all learners

On ⟨Nack, n⟩ **s.t.** n = n_p:

trigger Abort()
n_p := 0

Acceptor

- **On** ⟨Prepare, n⟩:
  - **if** n_{prom} < n:
  - n_{prom} := n
  - **send** ⟨Promise, n, n_a, v_a⟩ **to** Proposer
  - **else**: **send** ⟨Nack, n⟩ **to** Proposer
- **On** ⟨Accept, n, v⟩:
  - **if** n_{prom} ≤ n:
  - n_{prom} := n
  - (n_a, v_a) := (n, v)
  - **send** ⟨Accepted, n⟩ **to** Proposer
  - **else**: **send** ⟨Nack, n⟩ **to** Proposer

Learner

- **On** ⟨Decide, v⟩:
  - **If |v_d| < |v|:**
  - v_d := v
  - **trigger** Decide(v_d)

Sequence Paxos Algorithm

Proposer

- **On** ⟨Propose, C⟩ :
  - n_p := unique higher proposal number
  - S := ∅, acks := 0
  - **send** ⟨Prepare, n_p⟩ **to** all acceptors
- **On** ⟨Promise, n, n', v'⟩ **s.t.** n = n_p:
  - add (n', v') to S (multiset union)
  - **if** |S|= ⌈(N+1)/2⌉:
  - (k, v) := max(S) // **adopt v**
  - **v_p := v ⊕ ⟨C⟩**
  - **send** ⟨Accept, n_p, v_p⟩ **to** all acceptors

Acceptor

- **On** ⟨Prepare, n⟩:
  - **if** n_{prom} < n:
  - n_{prom} := n
  - **send** ⟨Promise, n, n_a, v_a⟩ **to** Proposer
  - **else**: **send** ⟨Nack, n⟩ **to** Proposer

- S = {(n_1, v_1), …., (n_k,v_k)}
- **fun** max(S):
  - (n,v) =: (0,⟨⟩)
  - **for** (n',v') **in** S:
    - **if** n < n' **or** (n = n' **and** |v| < |v'|):
    - (n,v) := (n',v')
  - **return** (n,v)

WHERE TO GO FROM HERE?

- Correctness ?
  - Follow the steps of Lamport
  - Correctness in modeled after the single-value Paxos correctness proof

WHERE TO GO FROM HERE?

- Efficiency ?
  - Every proposal takes two round-trips
  - Proposals are not pipelined
  - Sequences are sent back and forth
  - Decide carries sequences

Prepare phase

Accept phase

Proposer

On ⟨Propose, C⟩ :
n_p := unique higher proposal number
S := ∅, acks := 0
send ⟨Prepare, n_p⟩ **to** all acceptors

On ⟨Promise, n, n', v'⟩ **s.t.** n = n_p:
add (n', v') to S (multiset union)
if |S|= ⌈(N+1)/2⌉:
(k, v) := max(S) // **adopt v**
v_p := if v ≠ ⊥ then v else C
v_p := v ⊕ ⟨C⟩
send ⟨Accept, n_p, v_p⟩ **to** all acceptors

On ⟨Accepted, n⟩ **s.t.** n = n_p:
acks := acks + 1
if acks = ⌈(N+1)/2⌉:
send ⟨Decide, v_p⟩ **to** all learners

On ⟨Nack, n⟩ **s.t.** n = n_p:
trigger Abort()
n_p := 0

Acceptor

- **On** ⟨**Prepare**, n⟩:
  - **if** n_{prom} < n:
  - n_{prom} := n
  - **send** ⟨Promise, n, n_a, v_a⟩ **to** Proposer
  - **else: send** ⟨Nack, n⟩ **to** Proposer
- **On** ⟨**Accept**, n, v⟩:
  - **if** n_{prom} ≤ n:
  - n_{prom} := n
  - (n_a, v_a) := (n, v)
  - **send** ⟨Accepted, n⟩ **to** Proposer
  - **else: send** ⟨Nack, n⟩ **to** Proposer

Learner

- **On** ⟨**Decide**, v⟩:
  - **If |v_d| < |v|:**
  - v_d := v
  - **trigger** Decide(v_d)

max(S) is any element (k, v) of S s.t. k is highest proposal number and v is a sequence

Correctness of Sequence Paxos

CORRECTNESS

- How do we know that algorithm is correct?
- Build on proof structure for Paxos

PREPARE PHASE

Accept phase

Proposer

On ⟨Propose, C⟩ :
- n_p := unique higher proposal number
- S := ∅, acks := 0
- **send** ⟨Prepare, n_p⟩ **to** all acceptors

On ⟨Promise, n, n', v'⟩ **s.t.** $n = n_p$:
- add (n', v') to S (multiset union)
- **if** |S|= ⌈(N+1)/2⌉:
- (k, v) := max(S) // **adopt v**
- v_p := if v ≠ ⊥ then v else C
- $\mathbf{v_p := v \oplus \langle C \rangle}$
- **send** ⟨Accept, n_p, v_p⟩ **to** all acceptors

On ⟨Accepted, n⟩ **s.t.** $n = n_p$:
- acks := acks + 1
- **if** acks = ⌈(N+1)/2⌉:
- **send** ⟨Decide, v_p⟩ **to** all learners

On ⟨Nack, n⟩ **s.t.** $n = n_p$:
- trigger Abort()
- n_p := 0

Acceptor

- **On** ⟨**Prepare**, n⟩:
  - **if** n_{prom} < n:
  - n_{prom} := n
  - **send** ⟨Promise, n, n_a, v_a⟩ **to** Proposer
  - **else**: **send** ⟨Nack, n⟩ **to** Proposer
- **On** ⟨**Accept**, n, v⟩:
  - **if** n_{prom} ≤ n:
  - n_{prom} := n
  - (n_a, v_a) := (n, v)
  - **send** ⟨Accepted, n⟩ **to** Proposer
  - **else**: **send** ⟨Nack, n⟩ **to** Proposer

Learner

- **On** ⟨**Decide**, v⟩:
  - **If** $|v_d| < |v|$:
  - v_d := v
  - **trigger** Decide(v_d)

max(S) is any element (k, v) of S s.t. k is highest proposal number and v is a sequence

Ballot (round) Array

Replicas p_1, p_2 and p_3

Round	Accepted by p_1	Accepted by p_2	Accepted by p_3
n = 5	$\langle C_2, C_3 \rangle$	$\langle C_2, C_3 \rangle$	
...			
n=2		$\langle C_2 \rangle$	$\langle C_2 \rangle$
n=1	$\langle C_1 \rangle$		
n=0	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$

We are looking at the state of acceptors at each p_i

Empty sequence accepted in round 0

Chosen Sequence V

Let $v_a[p,n]$ is the sequence accepted by acceptor p at round n

A sequence v is chosen at round n

if there exists an quorum Q of acceptors at round n such that v is prefix $v_a[p,n]$, for every acceptor p in Q

A sequence v is chosen

if v is chosen at n, for some round n

Round	Accepted by p_1	Accepted by p_2	Accepted by p_3
n = 5	$\langle C_2,C_3\rangle$	$\langle C_2,C_3\rangle$	
...			
n=2		$\langle C_2\rangle$	$\langle C_2\rangle$
n=1	$\langle C_1\rangle$		
n=0	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$

Chosen Sequences

When request arrives from proposer at round 5 the chosen sequences are

$<>$,

$<C_2>$,

$<C_2,C_3>$,

$<C_2,C_3,C_1>$

Round	Accepted by p_1	Accepted by p_2	Accepted by p_3
n = 5	$\langle C_2,C_3,C1_{,}\rangle$	$\langle C_2,C_3,C_1\rangle$	
...			
n = 2		$\langle C_2\rangle$	$\langle C_2\rangle$
n = 1	$\langle C_1\rangle$		
n = 0	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$

Paxos Invariants

- P2c. For any v and n, if a proposal with value v and number n is issued, then there is a Quorum S of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S

- ⇒ P2b. If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v
- ⇒ P2a. If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v
- ⇒ P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v

Sequence Paxos Invariants

P2c. if a proposal with seq v and number n is issued, then there is a quorum S of acceptors such that seq v is an extension of the sequence of the highest-numbered proposal less than n accepted by any acceptor in S

Round	Accepted by p_1	Accepted by p_2	Accepted by p_3
n=5	$\langle C_2,C_3,b,d \rangle$	$\langle C_2,C_3,b,d \rangle$	
n=4	$\langle C_2,C_3,a \rangle$		
n=3	$\langle C_2,C_3 \rangle$		$\langle C_2,C_3 \rangle$
n=2		$\langle C_2 \rangle$	$\langle C_2 \rangle$
n=1	$\langle C_1 \rangle$		
n=0	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$

Highest numbered proposal accepted before round 4 is <c2,c3>
It is ok to issue <c2,c3,a> at 4, or <c2,c3,b,d> at 5

PREPARE PHASE

Accept phase

Proposer

On ⟨Propose, C⟩ :

n_p := unique higher proposal number

S := ∅, acks := 0

send ⟨Prepare, n_p⟩ **to** all acceptors

On ⟨Promise, n, n', v'⟩ **s.t.** n = n_p:

add (n', v') to S (multiset union)

if |S|= ⌈(N+1)/2⌉:

(k, v) := max(S) // **adopt v**

v_p := if v ≠ ⊥ then v else C

v_p := v ⊕ ⟨C⟩

send ⟨Accept, n_p, v_p⟩ **to** all acceptors

On ⟨Accepted, n⟩ **s.t.** n = n_p:

acks := acks + 1

if acks = ⌈(N+1)/2⌉:

send ⟨Decide, v_p⟩ **to** all learners

On ⟨Nack, n⟩ **s.t.** n = n_p:

trigger Abort()

n_p := 0

Acceptor

- **On** ⟨**Prepare**, n⟩:
  - **if** n_{prom} < n:
  - n_{prom} := n
  - **send** ⟨Promise, n, n_a, v_a⟩ **to** Proposer
  - **else**: **send** ⟨Nack, n⟩ **to** Proposer
- **On** ⟨**Accept**, n, v⟩:
  - **if** n_{prom} ≤ n:
  - n_{prom} := n
  - (n_a, v_a) := (n, v)
  - **send** ⟨Accepted, n⟩ **to** Proposer
  - **else**: **send** ⟨Nack, n⟩ **to** Proposer

Learner

- **On** ⟨**Decide**, v⟩:
  - **If |v_d| < |v|:**
  - v_d := v
  - **trigger** Decide(v_d)

max(S) is any element (k, v) of S s.t. k is highest proposal number and v is a sequence

IF A SEQUENCE IS CHOSEN

Replicas p_1, p_2 and p_3

Round	Accepted by p_1	Accepted by p_2	Accepted by p_3
n = 5	$\langle C_2, C_3 \rangle$	$\langle C_2, C_3 \rangle$	
...			
n=2		$\langle C_2 \rangle$	$\langle C_2 \rangle$
n=1	$\langle C_1 \rangle$		
n=0	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$

If sequence v is issued in round n then v is an extension of all sequences chosen in rounds $\leq$ n

Paxos to Sequence-Paxos Invariants

P2b. If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

⇓

P2b. If a proposal with seq v is chosen, then every higher-numbered proposal issued by any proposer has v as a prefix

Paxos to Sequence-Paxos Invariants

P2a. If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v

⇓

P2a. If a proposal with seq v is chosen, then every higher-numbered proposal accepted by any acceptor has v as a prefix

Paxos to Sequence-Paxos Invariants

P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v

⇓

P2. If a proposal with seq v is chosen, then every higher-numbered proposal that is chosen has v as a prefix

Multi-Paxos Invariants

- Initially, the empty sequence is chosen in round n = 0
- P2c. If a proposal with seq v and number n is issued, then there is a set S consisting of a majority of acceptors such that seq v is an extension of the sequence of the highest-numbered proposal less than n accepted by the acceptors in S
- ⇒ P2b. If a proposal with seq v is chosen, then every higher-numbered proposal issued by any proposer has v as a prefix
- ⇒ P2a. If a proposal with seq v is chosen, then every higher-numbered proposal accepted by any acceptor has v as a prefix

- ⇒ P2. If a proposal with seq v is chosen, then every higher-numbered proposal that is chosen has v as a prefix

Discussion

Problems with Existing Algorithm?

We can do Better

- Safety properties are guaranteed but…

1. A proposer can run only one proposal until it decides before taking the next proposal (no pipelining).
2. Multiple Proposers? -> Livelock (flp ghost)
3. 2 round-trips for each sequence chosen
4. too much IO (whole sequences are sent back and forth)
5. the sequences kept in proposers, acceptors, deciders are mostly redudant.

Does the previous algorithm satisfy Liveness?

Name desirable properties of a leader election algorithm

When should a leader election algorithm take these transitions?