Replicated State Machines, Sequence Consensus

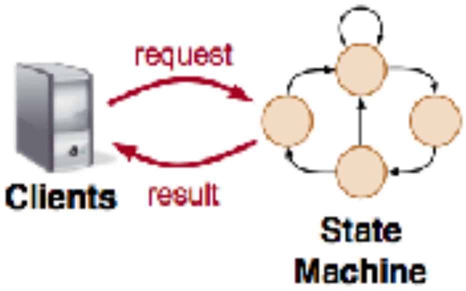
Seif Haridi

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

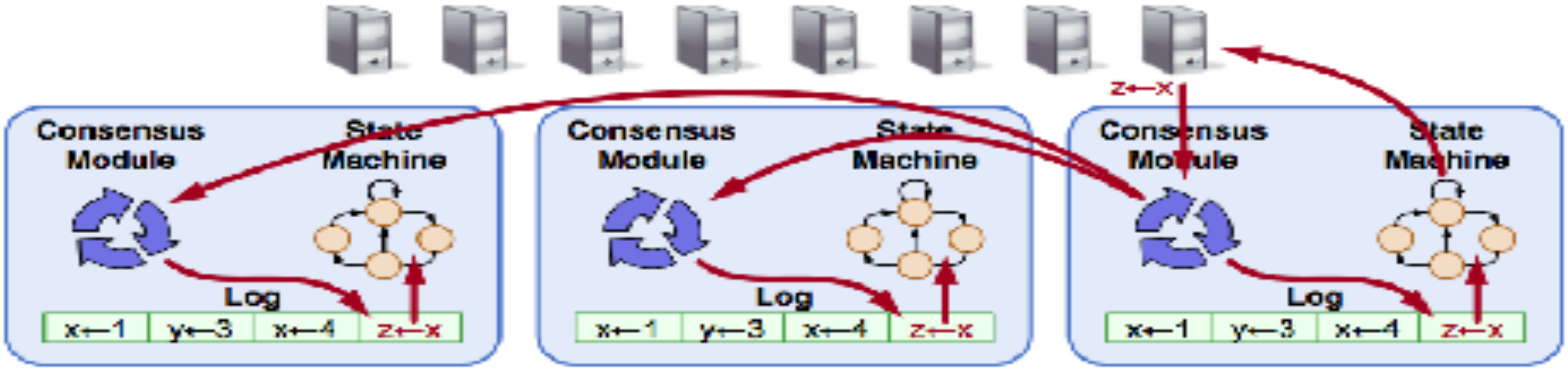
What is a state machine?

- A state machine
  - Executes a sequence of commands
  - Transform its state and may produce some output



- Commands are deterministic
  - Outputs of the state machine are solely determined by the initial state and by the sequence of commands that it has executed

Replicated State Machine



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

Our Trial (1)

- Consensus is an agreement on a single value/command
- Let us use multiple instances of Paxos

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

Single Value 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

Our Trial (Informal)

- Consensus is agreement on a single value
- Let us use multiple instances of Paxos
- Organize the algorithm in rounds
- 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*, it reliably rb-broadcast ⟨*C*, Pid_q⟩ to all servers
- **upon** delivery ⟨*C*, Pid_q⟩ at p_j , the command pair is added to *ProCmds* unless it is already in *Log*

Our Trial

- At round i, each server p_j:
  - Start new instance i of Paxos (single-value)
- **If** *ProCmds* $\neq \varnothing \wedge$ 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

Problems with our Trial !

- 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 $C_1 \ldots C_{i-1}$
  - 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|= ⌈(N+1)/2⌉:
  - (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 = ⌈(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_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
  - $\perp$ is the empty sequence ($\perp = \langle\rangle$)
- 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 $\langle$Decide, v$\rangle$ 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 $\oplus$:
- Non-duplicate $\oplus$:
  - $\langle C_1, \ldots, C_m \rangle \oplus C \stackrel{\text{def}}{=} \begin{cases} \langle C_1, \ldots, C_m \rangle \; \mathit{if} \text{ C is equal some } C_i \\ \langle C_1, \ldots, C_m, C \rangle, \text{otherwise} \end{cases}$
- Duplication allowed $\oplus$
  - $\langle C_1, \ldots, C_m \rangle \oplus C \stackrel{\text{def}}{=} \langle C_1, \ldots, C_m, C \rangle$

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
  - $n_{prom} := 0$ Promise not to accept in lower rounds
  - $n_a := 0$ Round number in which a value is accepted
  - $v_a := \langle\rangle$ Accepted value (empty sequence)
- Learner
  - $v_d := \langle\rangle$ 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

- **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

Accept phase

- **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

- **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

Accept phase

- **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 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 q 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

Multi-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

On ⟨Append, 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

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

Accept phase

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 ≤ 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

Leader- Based Sequence Paxos

Problems with current algorithm

- The previous algorithm as presented satisfies all the safety properties but may not make progress
  - A proposer can run only one proposal until decide before taking the next proposal. No pipelining of proposals
  - Multiple proposers may lead to live-locks (liveness violation)
  - Two round-trips for each sequence chosen
  - Entire sequences are sent back and forth
  - v_p, v_a and v_d are mostly redundant

Assumptions

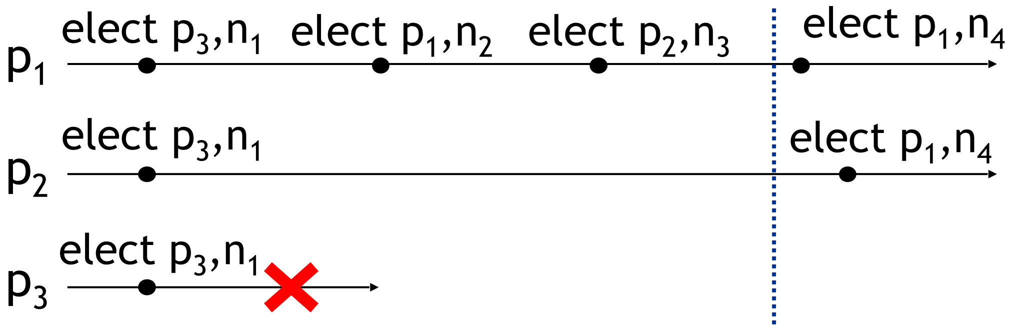
- Assume eventual leader election abstraction with a ballot number BLE ⟨**Leader**, L, n⟩
  - BLE satisfies completeness and eventually accuracy
  - And also monotonically unique ballots
- The Leader-based Sequence Paxos is optimized for the case when a single proposer runs for a longer period of time as a leader
  - Thus, will not be aborted for a while
  - But must guarantee safety if aborted

Interface of Leader Election

- **Module**:
  - Name: BallotLeaderElection (Ble)
- **Events**:
  - **Indication**: ⟨ble, Leader | p_i, n⟩
    - Indicate that leader is node p_i with ballot number n
- **Properties:**
  - ***BLE1 (completeness)***. Eventually every correct process elects some correct process if a majority are correct
  - ***BLE2 (eventual agreement)***. Eventually no two correct processes elect different correct processes
  - ***BLE3 (monotonic unique ballots)***. If a process L with ballot n is elected as leader by p_i, all previously elected leaders by p_i have ballot numbers less than n, and (L,n) is a unique number

BLE desirable properties

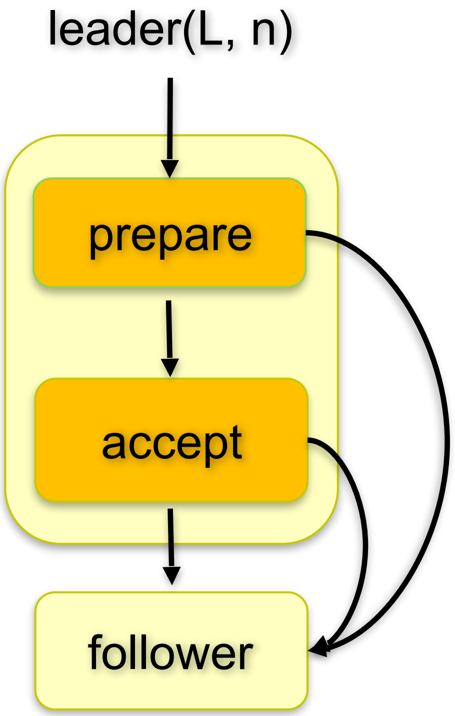
- Ballot leader election elects a leader L with higher ballot number n than all previous leaders L'
- If a process p elects a leader $\langle$**Leader**, L, n$\rangle_p$ then for previously elected leader at p $\langle$**Leader**, L', n'$\rangle_p$, n' > n and all pairs (L', n') are unique



$n_1<n_2<n_3<n_4$

The state of proposers

- We still have a set of proposers
- Any proposer will be either a leader or a follower
- A leader may be in either:
  - Prepare state, or
  - Accept state
- Until overrun by a higher leader, and moves to a follower state



Ballot Leader Election BLE

BLE desirable properties

- We will allow a process p to "inaccurately" leave a correct leader as long as the new leader has a higher ballot number
- We will also require that a process is elected as a leader only if a majority of processes are correct and alive. This fits Sequence Paxos (see later)
  - **BLE1:** Eventually every correct process trusts some correct correct process if **a majority are correct**
  - **BLE 2: Eventually** no two correct correct processes trust different correct processes

Assumptions

- We assume initially a Fail-Noisy model
  - Processes fail by crashing
  - Initial arbitrary network delays but eventually stabilizes (partially synchronous system)
  - Perfect point-to-point links
- However the algorithms works for a weaker model where the network may drop messages and processes crash and recover

Basic idea

- Ballots are unique
  - Each process p has its own ballot (n, pid_p). This pair is always unique since pid_p is unique can comes from an totally ordered set
  - A ballot is the **rank of a process**
- Max ballot is available at each correct process
  - Each correct process periodically gossips its ballot to all processes
- Processes are ranked
  - Eventually each correct process will elect the process with the highest rank (max ballot) given good network conditions ***(eventual agreement)***

Basic idea

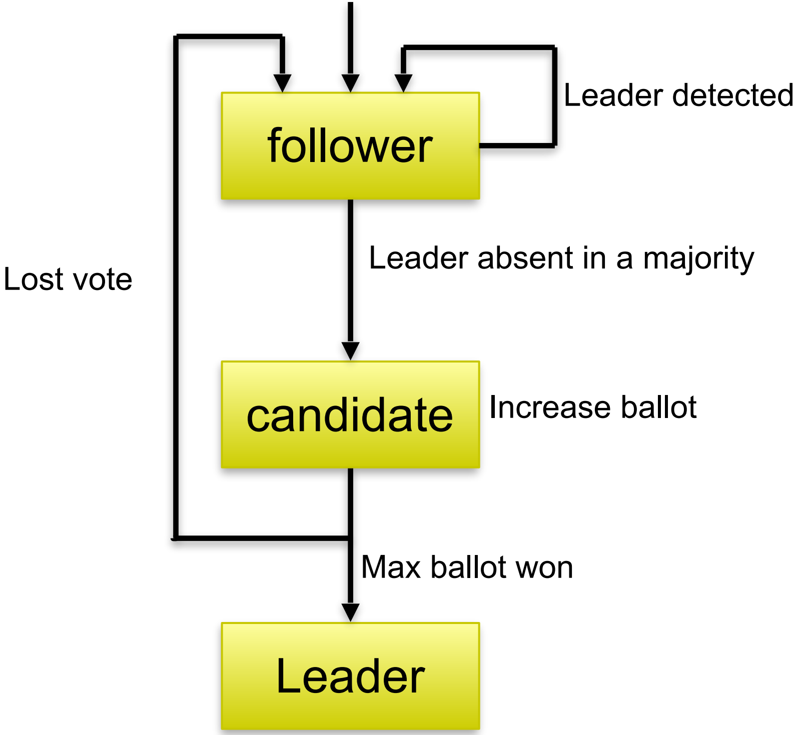
- Majority requirement
  - Each correct process will trust a leader only if the leader's max ballot is among the collected ballots from a majority of processes
- Monotonically increasing ballots
  - Every process p that do not receive the leader's ballot (n, pid_L) among collected ballots consider the leader has crashes
  - p increases his own ballot (n+1, pid_p)
- **BLE3 *(monotonic unique ballots)*** is satisfied and also **BLE1 *(completeness)*** assuming eventual synchrony

The algorithm I

- Each process p_i is ranked with a ballot: (n, pid_i) where n is an increasing epoch number and pid_i is a process identifier
- At any epoch n, ‘under stable network conditions’ the correct process with the highest pid is the leader and remains the leader if supported by a majority
- Periodically (delay Δ) each process collects the ballots of correct process in ballots (votes) and disseminates the known max ballot $ballot_{max}$

The algorithm II

- Each process pi starts as a follower
- Periodically each process pi collects ballots from a majority to check the leader
- If the leader's ballot is absent after collecting ballots from a majority at pi
  - pi moves to become a candidate
  - pi increases in own ballot to a value one higher than $ballot_{max}$
  - The one with highest rank wins and is elected
- If message from a suspected process is received the delay is increased by Δ



Implementing BLE

- BallotLeaderElection, **instance** ble
- **Uses:** PerfectPointToPointLinks, instance pp2p
- **upon event** ⟨ble, Init⟩ **do**
  - round := 0; ballots := ∅
  - ballot := (0; pid); leader := ⊥; $ballot_{max}$:= ballot
  - delay := Δ; **startTimer(delay)**
- **upon event** ⟨ Timeout ⟩ **do**
  - **if** ballots + 1 ≥ ⌈Π/2⌉ **then** checkLeader()
  - ballots := ∅, round := round + 1
  - **for all** p ∈ Π **do**
    - **if** p ≠ self **then**
    - **trigger** ⟨pp2p, Send | p, [HeartbeatRequest, round, $ballot_{max}$] ⟩
  - startTimer(delay)

Implementing BLE

- **upon event** ⟨pp2p, Deliver | p, [HeartbeatRequest, r, bmax] ⟩ **do**
  - **if** bmax > $ballot_{max}$ **then** $ballot_{max}$:= bmax
  - **trigger** ⟨pp2p, Send | p, [HeartbeatRelpy, r, ballot] ⟩

- **upon event** ⟨ ⟨pp2p, Deliver | p, [HeartbeatReply, r, b] ⟩ ⟩ **do**
  - **if** r = round **then** ballots := ballots ∪ { (p,b) }
  - **else**
    delay := delay + Δ

CheckLeader

- **Procedure** CheckLeader()
  - top := (topProcess, topBallot) := MaxByBallot(ballots ∪{(self , ballot) })
  - **if** topBallot < ballot_{max} **then**
  - leader := ⊥
  - **while** ballot ≤ ballot_{max} **do**
  - ballot := Increment(ballot)
  - **Else (**topBallot ≥ ballot_{max}
  - **if** top ≠ leader **then**
  - ballot_{max} := topBallot; leader := top
  - **trigger** ⟨ble, Leader | topProcess, topBallot ⟩

BLE conclusions

- The algorithm satisfies eventual agreement since the period Δ will increase so that heartbeats are delivered to each correct process by all correct process
- Once a leader L crashes or is disconnected from a majority, this majority with increase their ballot to a number higher than that of L
- In the next round one of processes will be elected based on the highest rank among them satisfying eventual completeness and monotonic ballots
- The algorithm works even if messages even if messages are lost or a process crashes and recovers