

Advanced Course

Distributed Systems

Omni-Paxos



- Check for comments on your project proposal.
 - Submit project proposal ASAP if you haven't yet.

COURSE TOPICS

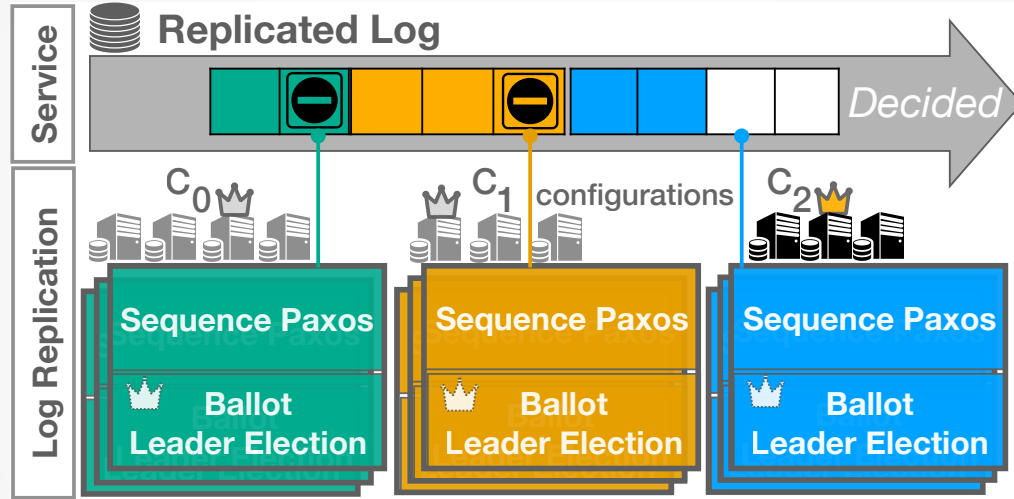


- ▶ Intro to Distributed Systems
- ▶ Basic Abstractions and Failure Detectors
- ▶ Reliable and Causal Order Broadcast
- ▶ Distributed Shared Memory
- ▶ Consensus, RSMs (Omni-Paxos, Raft, etc.)
- ▶ Dynamic Reconfiguration
- ▶ Time Abstractions and Interval Clocks (Spanner etc.)
- ▶ Consistent Snapshotting (Stream Data Management)
- ▶ Distributed ACID Transactions (Cloud DBs)

RECAP

- From Paxos to naïve Sequence Paxos
 - no pipelining
 - too much IO
 - redundancy of local state
- Liveness
 - what makes a server a “good” candidate?
- This week: Putting everything together - Omni-Paxos:
 - Sequence Paxos: log replication
 - Ballot Leader Election: liveness
 - Reconfiguration: parallel log migration

OMNI-PAXOS OVERVIEW



Sequence Paxos

The final version

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

DESIGN CONSIDERATIONS

- We want to replicate a growing log.
 - Proposers should send only the new entries, rather than the whole log every time
- Assume there is a **single** proposer running for a longer period of time as a **leader**.
 - Will not be aborted for a while.
 - If aborted, safety must still be guaranteed.

ASSUMPTIONS

- FIFO perfect link
- Ballot Leader Election abstraction:

Events:

Indication (out): $\langle \text{Leader} \mid n, p_i \rangle$

Notify that p_i is elected as leader with ballot n .

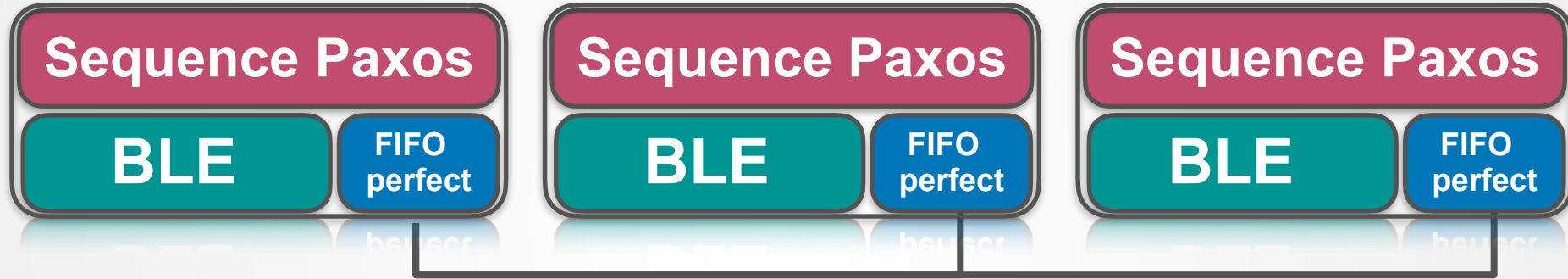
Properties:

BLE1. Completeness: Eventually, every correct process elects some correct process, if a majority of processes is correct.

BLE2. Eventual Accuracy: Eventually, no two correct processes elect different correct processes.

BLE3. Monotonically Increasing Unique Ballots: If a process p_i with ballot n is elected as leader by a process p_j , then all previously elected leaders by p_j have ballot numbers $m < n$, and the pair (n, p_i) is unique.

ABSTRACTIONS



Sequence Paxos Ensures correctness (safety)

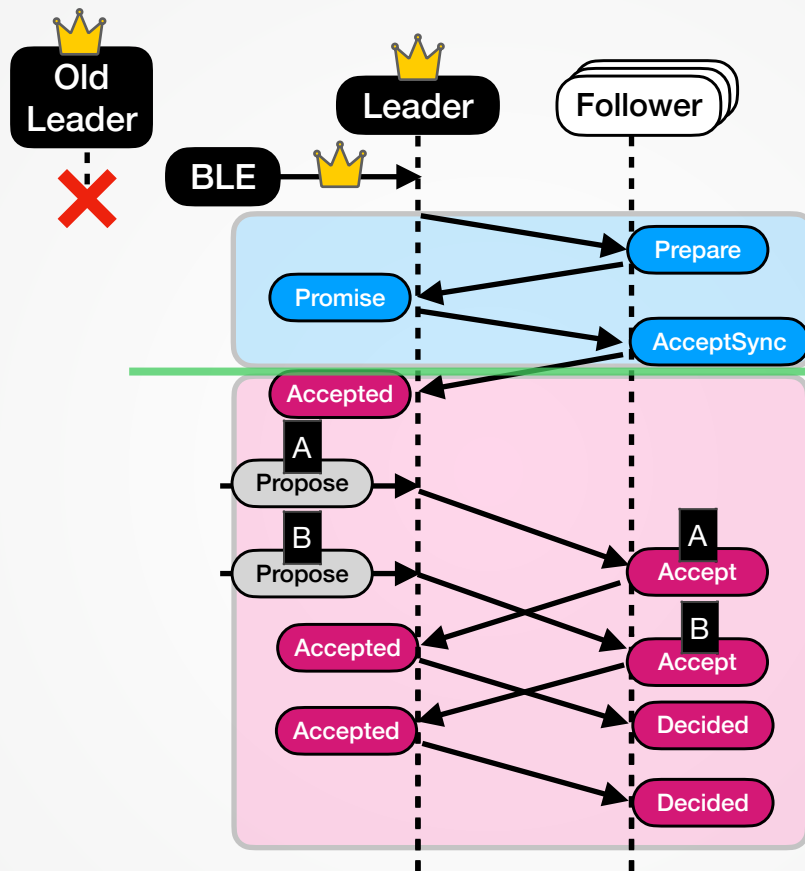
BLE Ensures termination (liveness)
(Leader ~ Proposer)

SEQUENCE PAXOS

- Each process acts in all roles as proposer, acceptor and learner
 - Every process maintains a single log: v_a
 - Use decided index l_d s.t. the decided sequence is $prefix(v_a, l_d)$
- A process acts as the *leader* or a *follower* in a round n
 - The leader acts as the sole proposer for round n
 - Until aborted by another leader $n' > n$
- A round has a *Prepare* and an *Accept* phase
 - Log synchronization in the Prepare phase
 - Replicate new entries in the Accept phase

PREPARE PHASE

- Initiated by the leader in a new round n
- Objective: prepare once, pipeline accepts
 - Leader sends $\langle \text{Prepare} \rangle$ to all followers.
 - Followers responds with $\langle \text{Promise} \rangle$ if not already promised $n' > n$.
 - Also includes the log suffix that the leader is missing.
 - Upon majority of promises: the leader adopts the most updated log and synchronizes it with the promised followers.
 - After the Prepare phase, any new entry extends the synchronized log
 - Allows multiple outstanding $\langle \text{Accept} \rangle$
 - Decision in a single round-trip



The leader and all promised followers have identical logs

LOG SYNCHRONIZATION

- For safety, the leader must adopt all chosen entries
 - Must be among at least one process in any majority
 - Adopt the log with highest n_a , or longest log if equal
- In $\langle \text{Prepare} \rangle$, the leader includes:
 - current round: n
 - accepted round: n_a
 - log length: $|v_a|$
 - decided index: l_d
- A follower responds with $\langle \text{Promise} \rangle$ only if its $n_{prom} < n$ and includes:
 - n and its own $n_a, |v_a|, l_d$
 - sfx : the log entries that the leader is missing
 - If greater n_a : $suffix(v_a, l_{d,leader})$
 - If same n_a and but longer log: $suffix(v_a, |v_a|_{leader})$
 - Else: $[]$

} more updated than leader

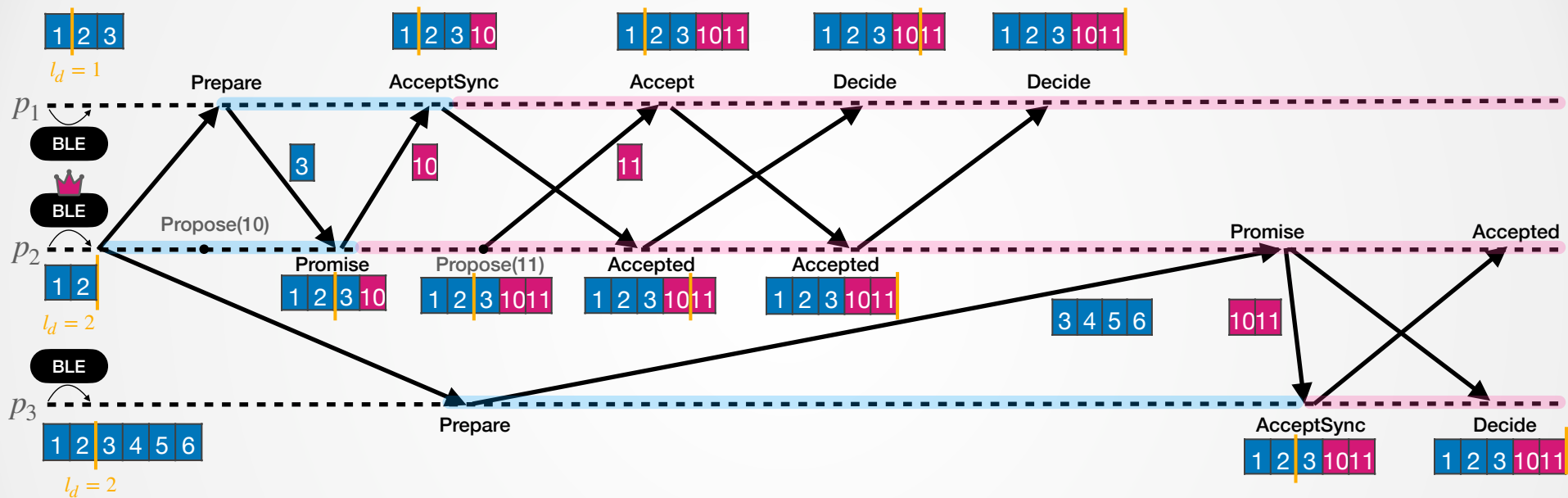
ACCEPTSYNC

- Upon majority of $\langle \text{Promise} \rangle$ adopt the sfx from the maximum promise:
 - If greater n_a : $v_a = \text{prefix}(v_a, l_d) \oplus sfx$
 - If same n_a : $v_a = v_a \oplus sfx$
- Synchronize updated log with all promised followers using $\langle \text{AcceptSync} \rangle$ including:
 - n
 - sfx : the log entries that the follower is missing
 - If greater n_a : $\text{suffix}(v_a, l_{d, \text{follower}})$
 - If same n_a and but longer log: $\text{suffix}(v_a, |v_a|_{\text{follower}})$
 - l_{sync} : the index to append sfx at in v_a

ACCEPT PHASE

- After the Prepare phase, the leader and all promised followers have the same common log prefix with all chosen entries.
- Leader replicates new command C with $\langle \text{Accept} \mid n, C \rangle$ to all promised followers.
 - Followers respond with accepted index $|v_a|$
 - When a majority has $\langle \text{Accepted} \mid n, idx \rangle$, send $\langle \text{Decide} \mid n, idx \rangle$
- Leader handles late $\langle \text{Promise} \rangle$ by synchronising that follower with its current log using $\langle \text{AcceptSync} \rangle$

EXAMPLE



①

②

KTH-2022

PREPARE PHASE

④ **⟨Promise⟩ from follower *f***

Fields:

<i>n</i>	promised round
<i>accRnd</i>	the acceptedRnd of <i>f</i>
<i>logIdx</i>	the log length of <i>f</i>
<i>decIdx</i>	the decidedIdx of <i>f</i>
<i>sfx</i>	suffix of entries the leader might be missing

Receiver implementation:

1. return if $n \neq \text{currentRnd}$
2. insert (*accRnd*, *logIdx*, *f*, *decIdx*, *sfx*) to promises

If state = (LEADER, PREPARE) then:

P1. return if |promises| < majority

P2. $\text{maxProm} \leftarrow$ the value with highest *accRnd* in promises (and highest *logIdx* if equal)

P3. if $\text{maxProm}.\text{accRnd} \neq \text{acceptedRnd}$ then
 $\text{log} \leftarrow \text{prefix}(\text{decidedIdx})$

P4. append *maxProm.sfx* to the log

P5. if *stopped()* then clear buffer else append buffer to the log

P6. $\text{acceptedRnd} \leftarrow \text{currentRnd}$,
 $\text{accepted}[\text{self}] \leftarrow |\text{log}|$, state \leftarrow (LEADER, ACCEPT)

foreach *p* in promises:
let *syncIdx* \leftarrow if $p.\text{accRnd} = \text{maxProm}.\text{accRnd}$ then
 $p.\text{logIdx}$ else $p.\text{decIdx}$,
send (AcceptSync, *currentRnd*, $\text{suffix}(\text{syncIdx})$, *syncIdx*)
to *p.f*

If state = (LEADER, ACCEPT) then:

A1. let *syncIdx* \leftarrow if $\text{accRnd} = \text{maxProm}.\text{accRnd}$ then
 $\text{maxProm}.\text{logIdx}$ else *decIdx*

A2. send (AcceptSync, *currentRnd*, $\text{suffix}(\text{syncIdx})$, *syncIdx*)
to *f*

A3. let *idx* $\leftarrow \max(\text{chosenIdx}, \text{decidedIdx})$,
if *idx* > *decIdx* then send (Decide, *currentRnd*, *idx*) to *f*

③ **⟨Prepare⟩ from leader *l***

Fields:

<i>n</i>	round of leader <i>l</i>
<i>accRnd</i>	the acceptedRnd of <i>l</i>
<i>logIdx</i>	the length of the leader's log
<i>decIdx</i>	the decidedIdx of <i>l</i>

Receiver implementation:

1. return if $\text{promisedRnd} > n$
2. state \leftarrow (FOLLOWER, PREPARE)
3. $\text{promisedRnd} \leftarrow n$
4. let *sfx* \leftarrow if $\text{acceptedRnd} > \text{accRnd}$ then
 $\text{suffix}(\text{decIdx})$ else if $\text{acceptedRnd} = \text{accRnd}$ then $\text{suffix}(\text{logIdx})$ else []
5. send (Promise, *n*, *acceptedRnd*, |log|, *decidedIdx*, *sfx*) to *l*

⑤ **⟨AcceptSync⟩ from leader *l***

Fields:

<i>n</i>	round of leader <i>l</i>
<i>sfx</i>	entries to be appended to the log
<i>syncIdx</i>	the position in the log where <i>sfx</i> should be appended at

Receiver implementation:

1. if $\text{promisedRnd} = n$ AND
state = (FOLLOWER, PREPARE)
2. $\text{acceptedRnd} \leftarrow n$,
state \leftarrow (FOLLOWER, ACCEPT)
3. $\text{log} \leftarrow \text{prefix}(\text{syncIdx})$,
append *sfx* to the log
4. send (Accepted, *n*, |log|) to *l*

ACCEPT PHASE

⑥ Proposal **C** from client

Receiver implementation:

1. return if stopped ()

If state = (LEADER, PREPARE) then:

P1. insert C into buffer

If state = (LEADER, ACCEPT) then:

A1. append C to the log, set $\text{accepted}[\text{self}] \leftarrow |\log|$

A2. send $\langle \text{Accept}, \text{currentRnd}, C \rangle$ to all promised followers



⑦ $\langle \text{Accept} \rangle$ from leader **l**

Fields:

n round of leader **l**

C client request

Receiver implementation:

1. return if $\text{promisedRnd} \neq n$ OR $\text{state} \neq (\text{FOLLOWER}, \text{ACCEPT})$
2. append C to the log, send $\langle \text{Accepted}, n, |\log| \rangle$ to **l**



⑧ $\langle \text{Accepted} \rangle$ from follower **f**

Fields:

n promised round

logIdx the position in the log **f** has accepted up to

Receiver implementation:

1. return if $\text{currentRnd} \neq n$ OR $\text{state} \neq (\text{LEADER}, \text{ACCEPT})$
2. $\text{accepted}[\text{f}] \leftarrow \text{logIdx}$
if $\text{logIdx} > \text{chosenIdx}$ AND a majority has accepted logIdx then $\text{chosenIdx} \leftarrow \text{logIdx}$, $\text{decidedIdx} \leftarrow \text{logIdx}$,
3. send $\langle \text{Decide}, \text{currentRnd}, \text{chosenIdx} \rangle$ to all promised followers



⑨ $\langle \text{Decide} \rangle$ from leader **l**

Fields:

n round of leader **l**

decIdx position in the log that has been decided





Receiver implementation:

1. if $\text{promisedRnd} = n$ AND $\text{state} = (\text{FOLLOWER}, \text{ACCEPT})$ then $\text{decidedIdx} \leftarrow \text{decIdx}$

CORRECTNESS

- We must guarantee that:
 - If a proposal (n, v) is chosen, then for every higher proposal (n', v') that is chosen, $v \leq v'$
- We have two cases:
 - $n = n'$: only successively longer sequences can be chosen within the same round since processes accept growing sequences.
 - $n < n'$: the prepare phase guarantees that all chosen sequences will be adopted in n' , and no new sequences can be chosen in round n after that.

SUMMARY

- Assume stable leader and FIFO perfect links.
 - Log synchronization in the Prepare phase
- Single round-trip to decide a command (most of the time) 
- Only new commands are being sent 
- Pipeline $\langle \text{Accept} \rangle$ without waiting for previous to be decided 
- Multiple Proposers and FLP ghost
 - Handled with BLE in the partially synchronous model (not solvable in async model)  SOON

Ballot Leader Election

REVISITING BLE

BLE1. Completeness: Eventually, every correct process elects some correct process, if a majority of processes is correct.

BLE2. Eventual Accuracy: Eventually, no two correct processes elect different correct processes.

BLE3. Monotonically Increasing Unique Ballots: If a process p_i with ballot n is elected as leader by a process p_j , then all previously elected leaders by p_j have ballot numbers $m < n$, and the pair (n, p_i) is unique.

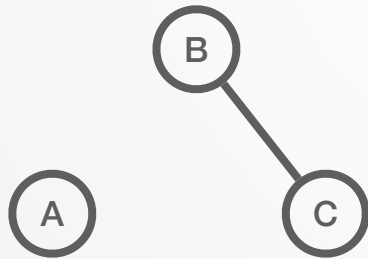
For Sequence Paxos:

Which processes really need to elect and agree with each other?

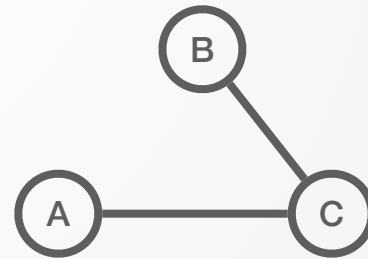
Partial Connectivity

THE PROBLEM OF PARTIAL CONNECTIVITY

- Thus far, we have assumed network failures to be full partitions.
 - In practice, network partitions can be more complex and unpredictable.
- Partial connectivity
 - Failures at the link level.
 - Servers A and B are disconnected but both can reach server C.

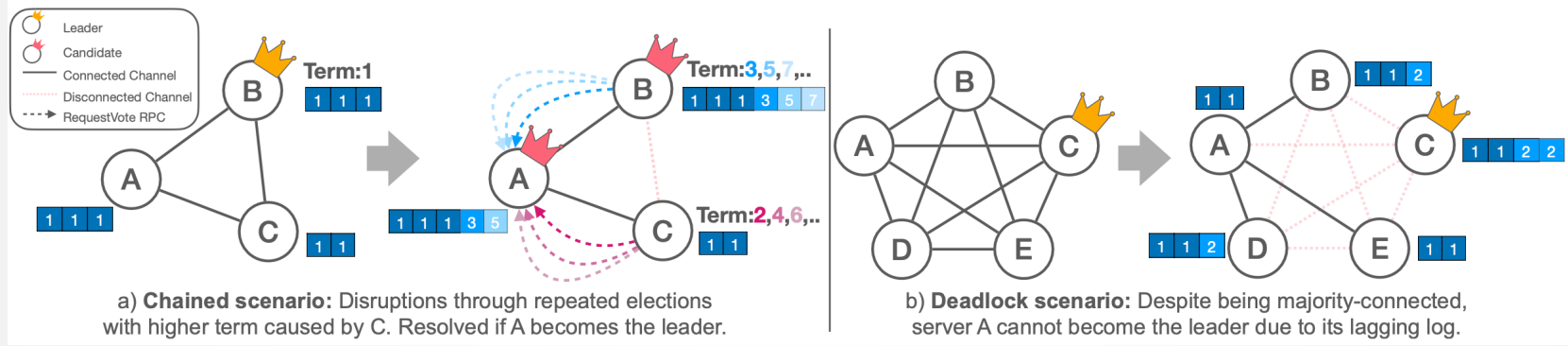


Full partition



Partial partition

TEASER: EXISTING ALGORITHMS CANNOT HANDLE THIS!



QUORUM-CONNECTED LEADER ELECTION

- **Observe:** In Sequence Paxos, only the leader must be connected to a majority for liveness.
 - Followers don't talk to each other!
- *Quorum-connected server*: a quorum-connected server is a server that is correct and has a direct link to at least a majority of correct servers (including itself).

QLE1. Quorum-Connected Completeness: Eventually, every quorum-connected server elects some quorum-connected server, if a quorum-connected server exists.

QLE2. Quorum-Connected Eventual Accuracy: Eventually, there is a majority of servers S where no two quorum-connected servers in S elect differently.

QLE3. Monotonically Increasing Unique Ballots: Unchanged.

BALLOT LEADER ELECTION

- A server has a ballot number b and a quorum-connected flag qc
- Periodically, all servers exchange heartbeats.
 - Broadcast $\langle \text{HBRequest} \mid r \rangle$
 - Reply $\langle \text{HBReply} \mid r, qc, b \rangle$
- Servers can determine two things with the heartbeats:
 1. Am I quorum-connected?
 2. Which of my peers are alive and quorum-connected?
- Upon timeout:
 - If received a majority of $\langle \text{HBReply} \rangle$:
 - Check if leader is still alive and quorum-connected. If not, increment b .
 - Elect the server with highest b and $qc = \text{true}$
 - Else: set $qc = \text{false}$

BLE PSEUDO CODE

① State and Functions	② Upon timeout of startTimer
<p>Persistent state on all servers:</p> <p>l ballot number of the current leader</p> <p>Volatile state on all servers:</p> <p>r current heartbeat round. Initially set to 0</p> <p>b ballot number. Initially set to (0, pid)</p> <p>qc quorum-connected flag. Initially set to true</p> <p>delay the duration a server waits for heartbeat replies within a single round</p> <p>ballots[] set of ballots received in the current round</p> <p>Functions:</p> <p>startTimer(d) schedules a timeout event in d timeunits. When starting: send $\langle \text{HBRequest}, r \rangle$ to all peers and $\text{startTimer}(\text{delay})$</p> <p>increment(b) increment the sequence number of ballot b</p> <p>max(ballots) pick the maximum ballot based on lexicographic order</p> <p>checkLeader()</p> <ol style="list-style-type: none"> let $\text{candidates} \leftarrow$ the values in ballots with $\text{quorumConnected} = \text{true}$ let $\text{max} \leftarrow \text{max}(\text{candidates})$ if $\text{max} < l$ then $\text{increment}(\text{b})$ s.t. $\text{b} > l$, $\text{quorumConnected} \leftarrow \text{true}$ else if $\text{max} > l$ then $l \leftarrow \text{max}$, trigger $\langle \text{Leader}, \text{max.pid}, \text{max} \rangle$ 	<p>Receiver implementation:</p> <ol style="list-style-type: none"> insert (b, qc) into ballots if $\text{ballots} \geq \text{majority}$ then $\text{checkLeader}()$ else $\text{qc} \leftarrow \text{false}$ clear ballots, $r \leftarrow r + 1$ send $\langle \text{HBRequest}, r \rangle$ to all peers, $\text{startTimer}(\text{delay})$ <p>③ $\langle \text{HBRequest} \rangle$ from server s</p> <p>Fields:</p> <p>rnd the round of this request</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> Send $\langle \text{HBReply}, \text{rnd}, \text{b}, \text{qc} \rangle$ to s <p>④ $\langle \text{HBReply} \rangle$ from server s</p> <p>Fields:</p> <p>rnd the round this reply was sent in</p> <p>ballot ballot number of s</p> <p>q qc of s</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> if $\text{rnd} = r$ then insert (ballot, q) into ballots <p>⑤ Upon Recovery</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> reload l from persistent storage $\text{startTimer}(\text{delay})$

CORRECTNESS

- Assuming we learn a time out s.t. no late $\langle \text{HBReply} \rangle$ is received.
 - A late heartbeat is ignored and does not affect correctness.
- *QLE1. Quorum-connected Completeness*
 - A server can only elect if it got a majority of $\langle \text{HBReply} \rangle$ i.e. is quorum-connected. The elected server must have $qc = \text{true}$.
- *QLE3. Monotonically Increasing Unique Ballots*
 - Each ballot (b, pid) is unique due to pid is unique. Servers only elect new leaders with higher ballot than previous leaders.

CORRECTNESS CONTINUED

- *QLE2. Quorum-Connected Eventual Accuracy*: Eventually, there is a majority of servers S where no two quorum-connected servers in S elect differently.
- Consider every possible case of connectivity between quorum-connected servers:
 1. Only one quorum-connected server in the cluster.
 2. Multiple quorum-connected servers:
 - A. That are connected to each other.
 - B. That are disconnected to each other.
- 1. That qc-server will be the only one receiving a majority of $\langle \text{HBReply} \rangle$ and its own ballot will be the only with $qc = \text{true}$.
- 2A. All qc-servers get each others $\langle \text{HBReply} \rangle$. They all elect the same leader with the highest ballot.
- 2B. Since they are quorum-connected but disconnected, they all are connected to a majority of servers. Any majority overlaps on at least one server.
 - That server is not qc: will not elect **in BLE**, but will follow (i.e. promise) the leader with the highest ballot **in Sequence Paxos**.
 - That server is qc: will elect the one with highest ballot (as in 2A)

OBSERVATIONS

- Omni-Paxos guarantees liveness as long as one quorum-connected server exists.
- BLE and its quorum-connected properties are **weaker** than a usual leader election. But it is **sufficient for Sequence Paxos**
 - Non qc-servers do not elect (QLE1)
 - But if they are connected to the leader, they will get the $\langle \text{Prepare} \rangle$ to participate in Sequence Paxos.
 - Different qc-servers might elect different leaders (QLE2)
 - One of them will have the highest ballot. That leader will also be the only one making progress in Sequence Paxos.

Fail Recovery

OUTLINE

- At this point, we have an efficient and resilient algorithm.
- Fail recovery model
 - Recover from crashes.
- FIFO perfect link assumption is impractical.
 - **Session-based** FIFO perfect links
 - Handle session drops

FAIL RECOVERY

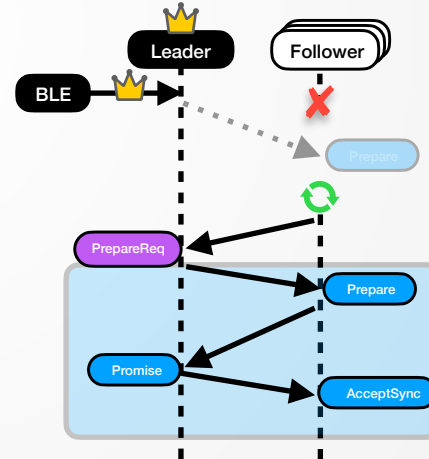
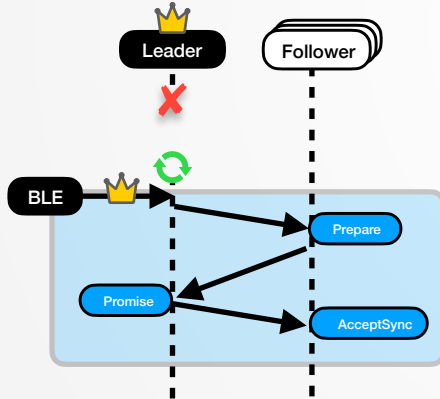
- A process is correct if it crashes and recovers a finite number of times.
 - By crashing and restarting, a process loses any arbitrary suffix of most recent messages in each FIFO perfect link.
- A recovered process must get its log synchronized to be up-to-date before doing anything further.

RECOVERY

- Each process must store the following variables in persistent storage
 - v_a : the log.
 - l_d : the decided index.
 - n_{prom} : the promised round.
 - n_a : the latest round entries were accepted in.
- Upon recovery, restore these variables.
 - Load n_{prom} into BLE
 - Set own state into (FOLLOWER, **RECOVER**)
 - Send **⟨ PrepareReq ⟩** to all peers
 - If a receiving process is the leader, it replies with **⟨ Prepare ⟩**

RECOVER STATE

- In (FOLLOWER, **RECOVER**), a process can only handle:
 - $\langle \text{Leader} \rangle$: got elected as the leader, will get synchronized by performing the prepare phase.
 - $\langle \text{Prepare} \rangle$: leader will help us get synchronized.



SESSION-BASED FIFO PERFECT LINKS

- Assume FIFO perfect links once a session has been established.
 - e.g. TCP sessions
 - Need to handle session-drops.
- If disconnected to a peer... do nothing
- When reconnecting to a peer p :
 - Send $\langle \text{PrepareReq} \rangle$ to p because p might have become the new leader during our down-time.
 - If p is the leader we last promised:
 - Go into recover mode to avoid handling anything before being synchronized.

PSEUDO CODE

⑩

Upon Recovery

Receiver implementation:

1. reload: log, promiseRnd, acceptedRnd and decidedIdx from persistent storage
2. state \leftarrow (FOLLOWER, RECOVER),
send $\langle \text{PrepareReq} \rangle$ to all peers



⑪

$\langle \text{PrepareReq} \rangle$ from follower f

Receiver implementation:

1. return if state \neq (LEADER, _)
2. send Prepare $\langle \text{currentRnd, acceptedRnd, } |\log|, \text{decidedIdx} \rangle$ to f

⑫

$\langle \text{Reconnected} \rangle$ to server s

Receiver implementation:

1. if s is the current leader then state \leftarrow (FOLLOWER, RECOVER)
2. send $\langle \text{PrepareReq} \rangle$ to s

SUMMARY

- From naïve Sequence Paxos to Sequence Paxos
 - Log Synchronization in Prepare phase
 - Pipeline $\langle \text{Accept} \rangle$ in Accept phase.
- Liveness with Ballot Leader Election
 - Quorum-connected leader election properties
 - Resilient: guaranteed progress with a single quorum-connected node.
- Handling Failures:
 - Session-based FIFO perfect link
 - Always get synchronized first when recovering.

- Reconfiguration: how to safely add/remove processes.
 - Parallel log migration
- Other replicated state machines
 - Raft and ZooKeeper (Zab)
 - Partial connectivity