

Advanced Course

Distributed Systems

Omni-Paxos



COURSE TOPICS

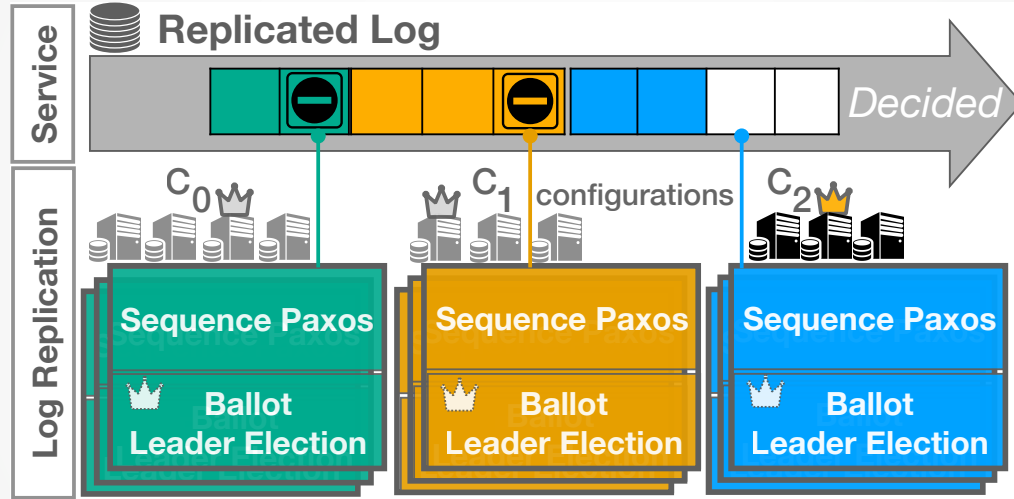


- ▶ Intro to Distributed Systems
- ▶ Basic Abstractions and Failure Detectors
- ▶ Reliable and Causal Order Broadcast
- ▶ Distributed Shared Memory, CRDTs
- ▶ Consensus, RSMs (Omni-Paxos, Raft, etc.)
- ▶ Dynamic Reconfiguration
- ▶ Time Abstractions and Interval Clocks (Spanner etc.)
- ▶ Consistent Snapshotting (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?
- The final assembly — 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 only send 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 .

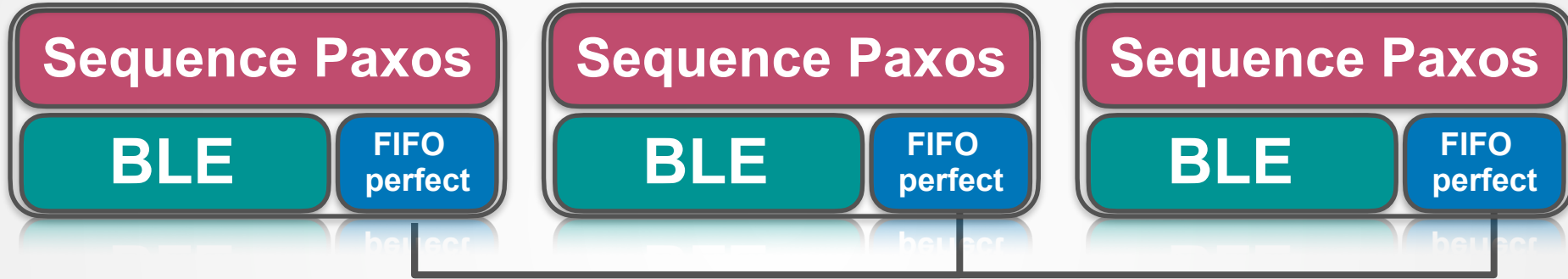
Properties:

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)

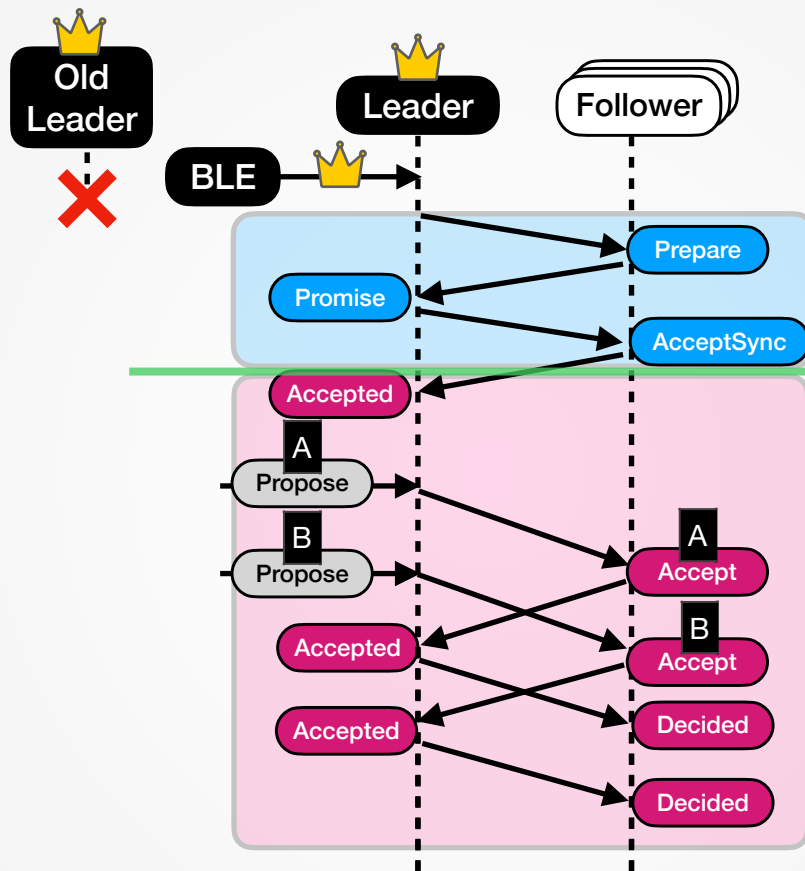
BLE Ensures termination (liveness)
(Leader ~ Proposer)

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

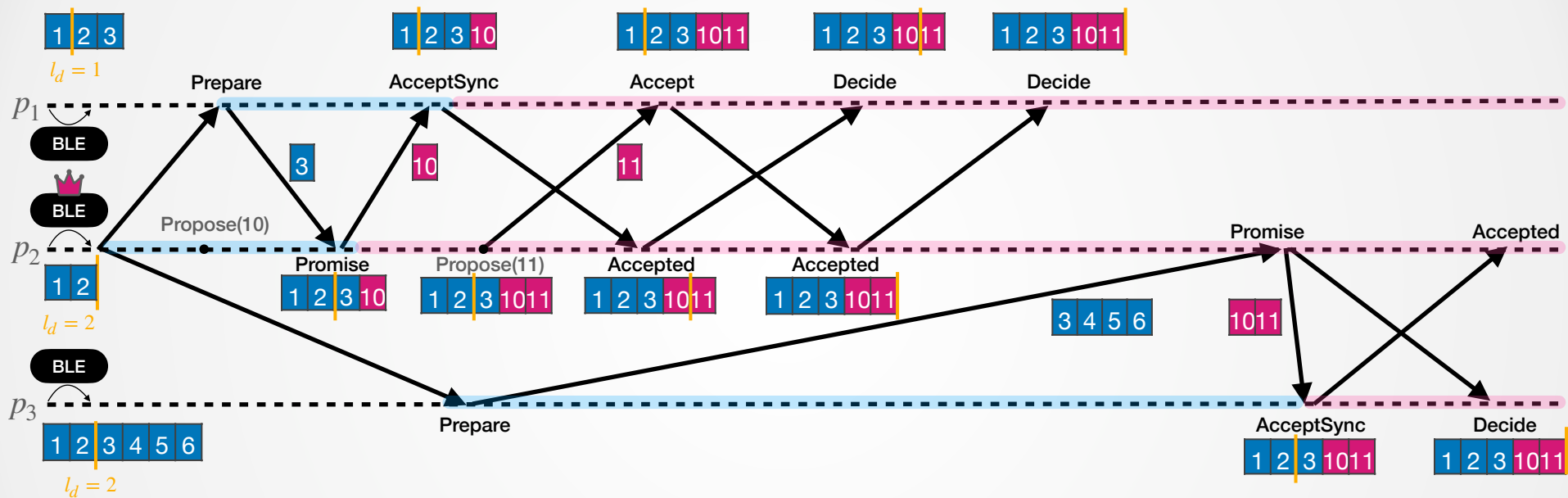
ACCEPTSYNC

- 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



FULL PSEUDO CODE - STATE AND BLE

① State and Functions

Persistent state on all servers:

log[]	log with entries (0-indexed)
promisedRnd	the round a server has promised to not accept entries from any lower round
acceptedRnd	the latest round a server has accepted entries in
decidedIdx	the log index that a server has decided up to

Volatile state on all servers:

state	the role and phase a server is in. Initially (FOLLOWER, PREPARE)
--------------	--

Volatile state of leader:

currentRound	the round that this server is leading in
promises[]	set of received promises
maxProm	the highest promise received during the prepare phase
accepted[]	the accepted index per server. Initialized to 0 for all servers
chosenIdx	the highest index accepted by a majority. Initially set to 0
buffer[]	buffer for client requests received during the prepare phase

Functions:

stopped()	true if last entry in the log is SS else false (§5)
prefix(idx)	the entries in the log with index $\{0..idx-1\}$
suffix(idx)	if $idx > log $ then \square else the entries in the log with index $\{idx.. log -1\}$

② **Leader** from BLE

Fields:

s	the elected server
n	the round s got elected in

Leader implementation (if $s = \text{self}$ AND $n > \text{promisedRnd}$):

1. reset all volatile state of leader
2. $\text{state} \leftarrow (\text{LEADER}, \text{PREPARE})$,
 $\text{currentRnd} \leftarrow n$, $\text{promisedRnd} \leftarrow n$,
3. insert own promise to promises:
(acceptedRnd, $\lfloor \log \rfloor$, self, decidedIdx, suffix(decidedIdx))
4. send $\text{Prepare}(\text{currentRnd}, \text{acceptedRnd}, \lfloor \log \rfloor, \text{decidedIdx})$
to all peers

Follower implementation: (if $s \neq \text{self}$):

- ```
1. state.role ← FOLLOWER
```

# PREPARE PHASE

④ **⟨Promise⟩ from follower  $f$**

**Fields:**

- $n$  promised round
- $accRnd$  the acceptedRnd of  $f$
- $logIdx$  the log length of  $f$
- $decIdx$  the decidedIdx of  $f$
- $sfx$  suffix of entries the leader might be missing

**Receiver implementation:**

- return if  $n \neq \text{currentRnd}$
- 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  $\text{maxProm}.accRnd \neq \text{acceptedRnd}$  then  $\text{log} \leftarrow \text{prefix}(\text{decidedIdx})$

P4. append  $\text{maxProm}.sfx$  to the log

P5. if  $\text{stopped}()$  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  $\text{syncIdx} \leftarrow$  if  $p.accRnd = \text{maxProm}.accRnd$  then  $p.logIdx$  else  $p.decIdx$ ,

P7. send (AcceptSync,  $\text{currentRnd}$ ,  $\text{suffix}(\text{syncIdx})$ ,  $\text{syncIdx}$ ) to  $p.f$

**If state = (LEADER, ACCEPT) then:**

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

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

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



③ **⟨Prepare⟩ from leader  $l$**

**Fields:**

- $n$  round of leader  $l$
- $accRnd$  the acceptedRnd of  $l$
- $logIdx$  the length of the leader's log
- $decIdx$  the decidedIdx of  $l$

**Receiver implementation:**

- return if  $\text{promisedRnd} > n$
- state  $\leftarrow$  (FOLLOWER, PREPARE)
- $\text{promisedRnd} \leftarrow n$
- 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 []
- send (Promise,  $n$ ,  $\text{acceptedRnd}$ ,  $|\text{log}|$ ,  $\text{decidedIdx}$ ,  $sfx$ ) to  $l$

⑤ **⟨AcceptSync⟩ from leader  $l$**

**Fields:**

- $n$  round of leader  $l$
- $sfx$  entries to be appended to the log
- $\text{syncIdx}$  the position in the log where  $sfx$  should be appended at

**Receiver implementation:**

- if  $\text{promisedRnd} = n$  AND state = (FOLLOWER, PREPARE)
- $\text{acceptedRnd} \leftarrow n$ , state  $\leftarrow$  (FOLLOWER, ACCEPT)
- $\text{log} \leftarrow \text{prefix}(\text{syncIdx})$ , append  $sfx$  to the log
- send (Accepted,  $n$ ,  $|\text{log}|$ ) 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 (Accept, currentRnd, C) to all promised followers

⑧ (Accept) 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 state  $\neq$  (LEADER, ACCEPT)
2.  $\text{accepted}[f] \leftarrow \text{logIdx}$   
if  $\text{logIdx} > \text{chosenIdx}$  AND a majority has accepted *logIdx*  
then  $\text{chosenIdx} \leftarrow \text{logIdx}$ ,  $\text{decidedIdx} \leftarrow \text{logIdx}$ ,
3. send (Decide, currentRnd, chosenIdx) to all promised followers

⑦ (Accept) from leader **l**

**Fields:**

*n* round of leader *l*

*C* client request

**Receiver implementation:**

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

⑨ (Decide) 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 state = (FOLLOWER, 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 problem
  - 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.

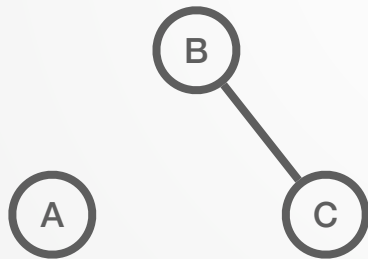
## For Sequence Paxos:

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

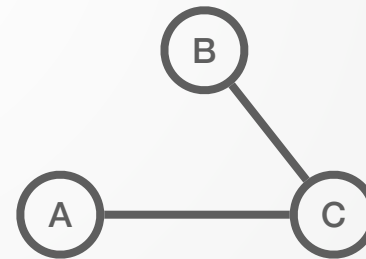
# Partial Connectivity

# THE PROBLEM OF PARTIAL CONNECTIVITY

- Thus far, we have assumed network failures to
  - In practice, network partitions can be more unpredictable.
- Partial connectivity
  - Failures at the link level.
  - Caused 6+ hours outage at Cloudflare in 2020



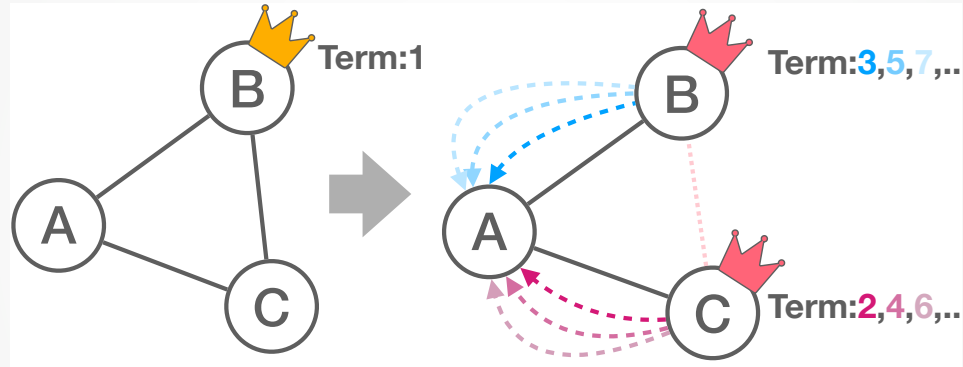
Full partition



Partial partition

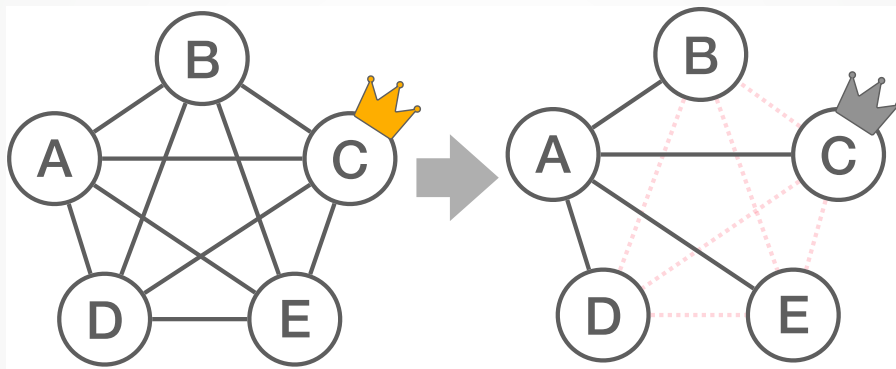
# CHAINED SCENARIO

---

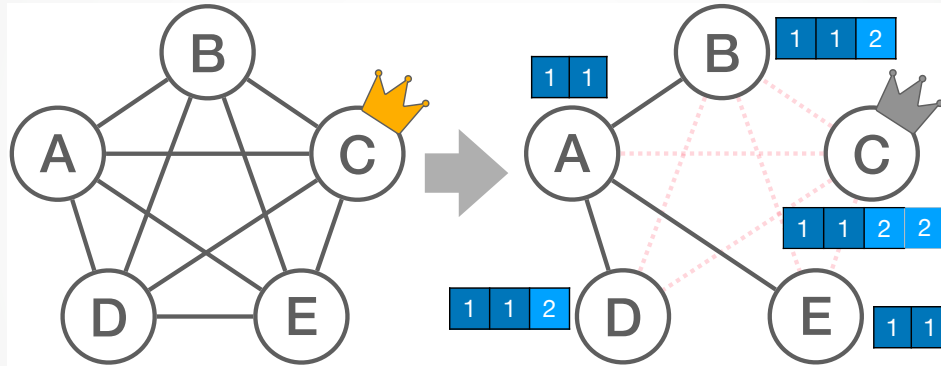


# QUORUM-LOSS SCENARIO

---



# CONSTRAINED ELECTION SCENARIO





# TEASER: EXISTING ALGORITHMS CANNOT HANDLE THIS!

|                      | Protocol Properties<br>(QC= quorum-connected, EQC=elected by quorum-connected) |                        |                       |                      |                                               | Partial-Connectivity Scenarios<br>(✓:stable progress, ✗:unavailable) |                               |                  |
|----------------------|--------------------------------------------------------------------------------|------------------------|-----------------------|----------------------|-----------------------------------------------|----------------------------------------------------------------------|-------------------------------|------------------|
|                      | Log Sync Phase                                                                 | Candidate Requirements | Leader Vote Gossiping | QC Status Heartbeats | Guaranteed Progress Requirement (#QC servers) | Quorum-Loss Scenario                                                 | Constrained Election Scenario | Chained Scenario |
| Multi-Paxos [28, 37] | ✓                                                                              | QC                     | ✓                     |                      | $\geq \lceil N/2 \rceil$                      | ✗                                                                    | ✓                             | ✗                |
| Raft [34]            |                                                                                | QC + max log           | ✓                     |                      | $\geq \lceil N/2 \rceil$                      | ✗*                                                                   | ✗                             | ✗*               |
| VR [31]              | ✓                                                                              | QC + EQC               | ✓                     |                      | $\geq \lceil N/2 \rceil$                      | ✗                                                                    | ✗                             | ✗                |
| Zab [25, 33]         |                                                                                | QC + EQC + max log     | ✓                     |                      | $\geq \lceil N/2 \rceil$                      | ✗                                                                    | ✗                             | ✗                |
| Omni-Paxos           | ✓                                                                              | QC                     |                       | ✓                    | $\geq 1$                                      | ✓                                                                    | ✓                             | ✓                |

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

# CORRECTNESS

---

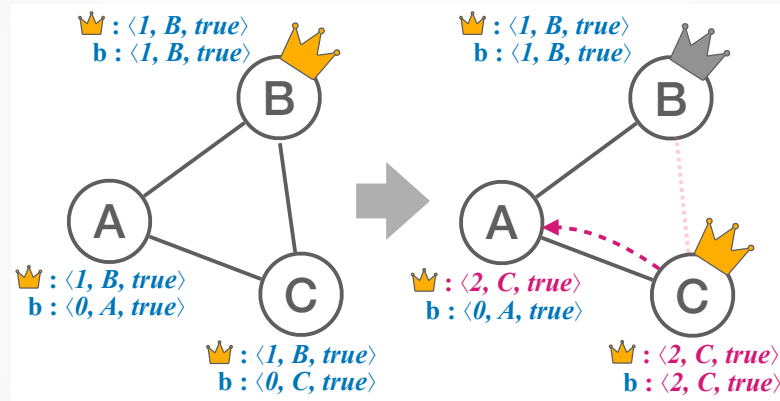
- Assuming we learn a time out s.t. no late  $\langle \text{HBReply} \rangle$  is received.
  - A late heartbeat is ignored and does not affect correctness.
- *QLE1. Quorum-connected Completeness*
  - A server can only elect if it got a majority of  $\langle \text{HBReply} \rangle$  i.e. is quorum-connected. 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

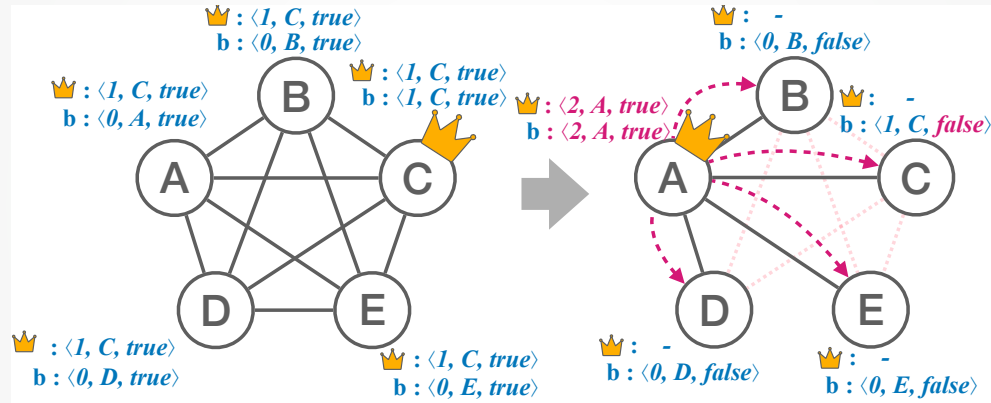
---

- *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 QC server in the cluster.
  2. Multiple QC 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. As they are QC but disconnected, each of them is connected to a majority of servers. And 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)

# OMNI-PAXOS: CHAINED SCENARIO

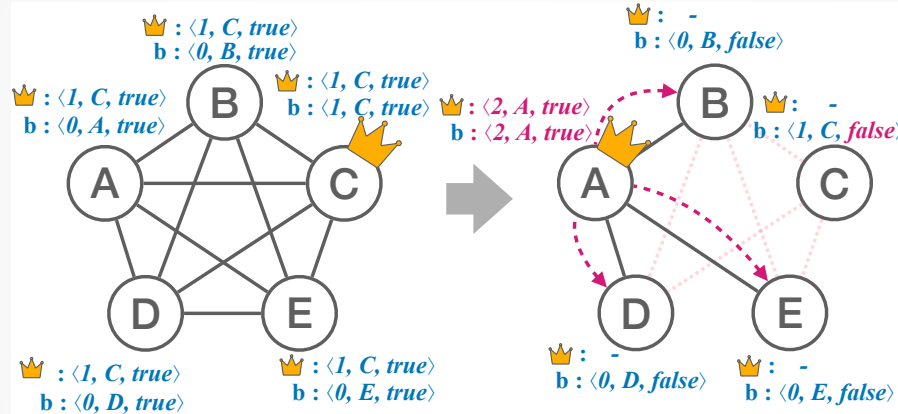


# OMNI-PAXOS: QUORUM-LOSS SCENARIO





# CONSTRAINED ELECTION SCENARIO



# 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 anyway.
  - 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.

# Failure 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.

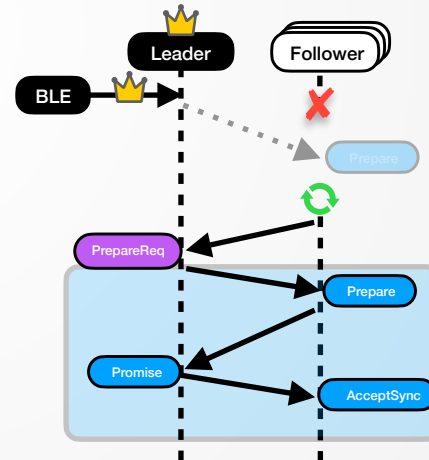
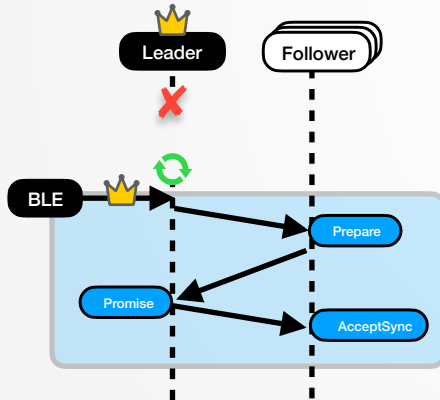
# RECOVERY

---

- 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  $s$

- ID2203
-  KTH  
VETENSKAP  
OCH KONST
- KTH-2023

# UP NEXT

---

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