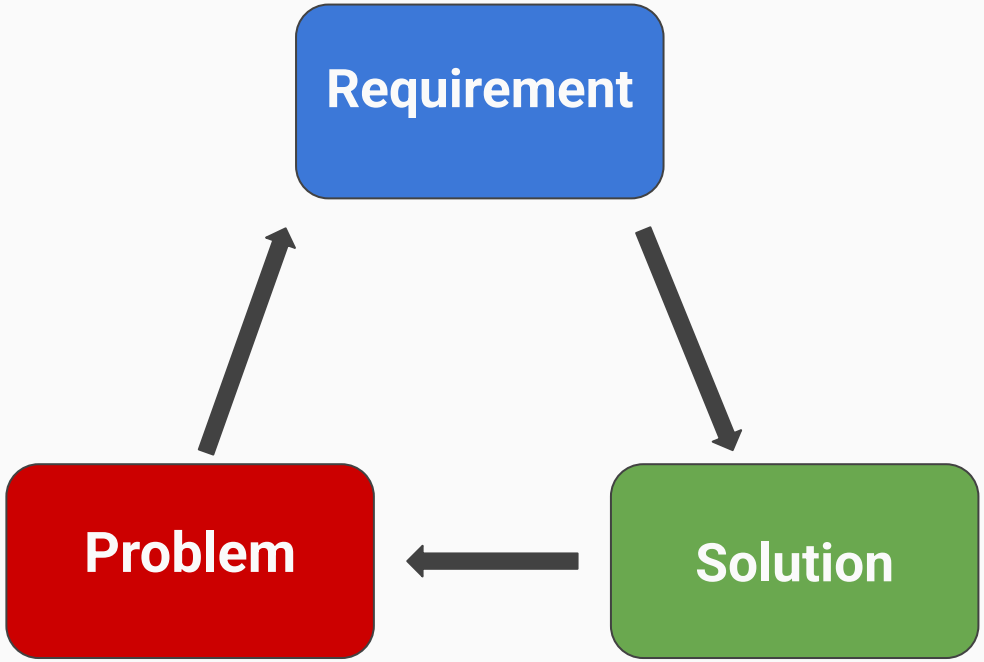
In Search of an Understandable Consensus Algorithm

Authors : **Diego Ongaro and John Ousterhout**

Presented by:
Ankit Kumar Meena
Sathya Pramod Batni
Vipul Nayyar

Stepping into the shoes of a System Designer

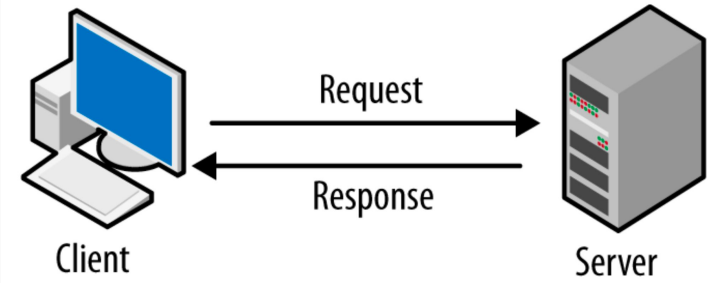


Requirement

- Write data about an entity
- Read latest stored data

Client/Server Model

- Running a state machine on single server
- Stores latest version of data
- Provides Read and Write operations
- Persists data to disk



Problem

- Server failure causes unavailability of data
- May lead to permanent data loss (disk failure)

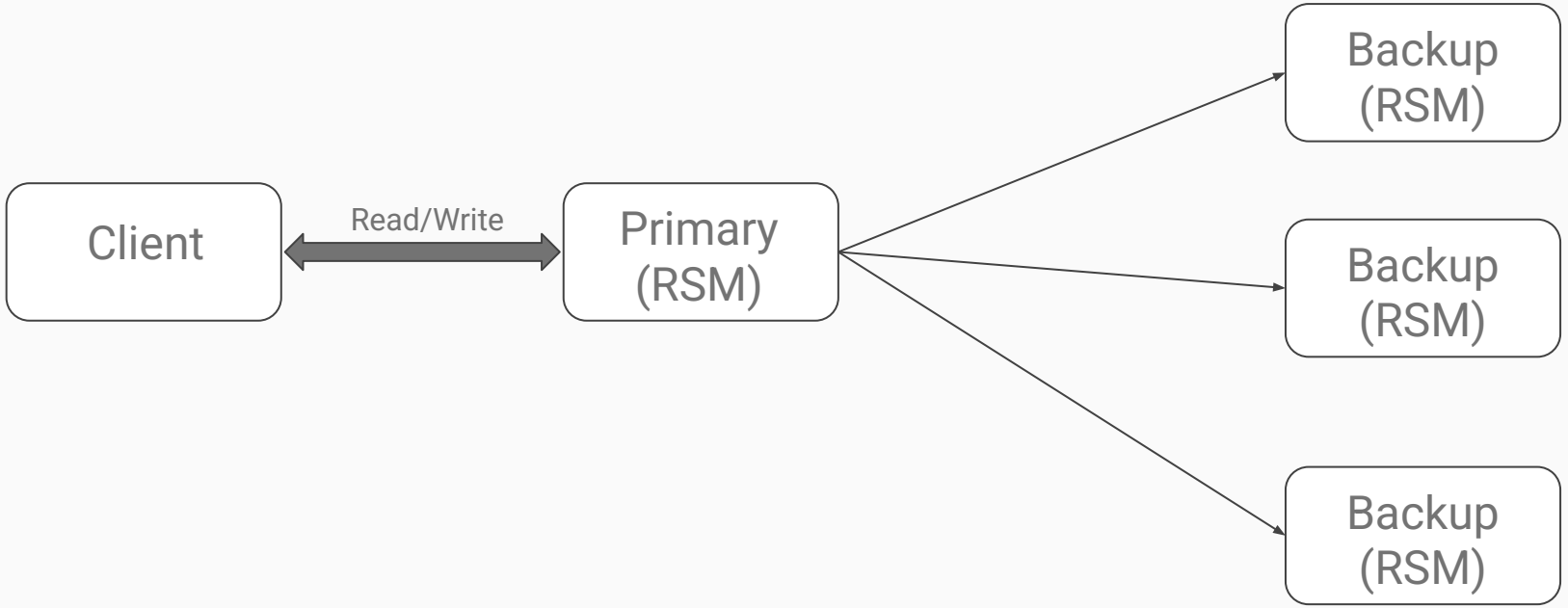
Requirement

- Ensure availability of data service across server failures
- Prevent permanent data loss

Data Redundancy with Replicated State Machines

- Store more than one copy of data across multiple servers
- State machine on each server stores identical copy of data. Implemented using a replicated log.
- Use one server as Primary that dispatches request to backup servers
- Primary serializes read and write requests

Simple P/B


Client
Read/Write
Primary
(RSM)
Backup
(RSM)
Backup
(RSM)
Backup
(RSM)

Problem : Simple P/B

- Single point of failure
- Require a coordination service, or internal mechanism to select new Primary after failure

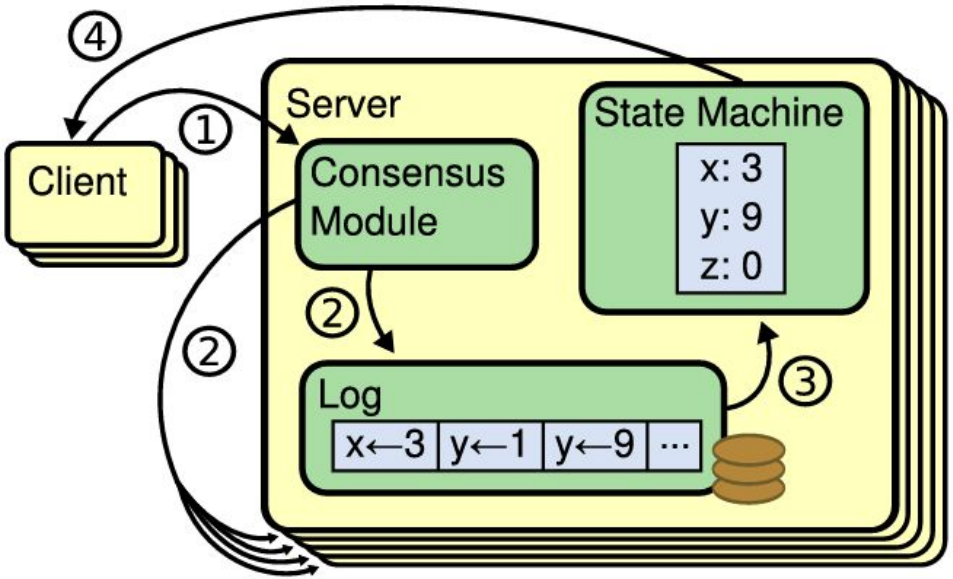
Requirement

- Replicated log used to apply change to RSM should be consistent across servers
- Do not need external coordinator

Consensus

- Allows a collection of machines to work as a coherent group that can survive the failures of some of its members.
- Servers appear to form a single, highly reliable state machine
- **Paxos :** Well known consensus algorithm

Consensus


④
①
Client
Server
Consensus
Module
State Machine
x: 3
y: 9
z: 0
②
②
③
Log
x←3
y←1
y←9
···

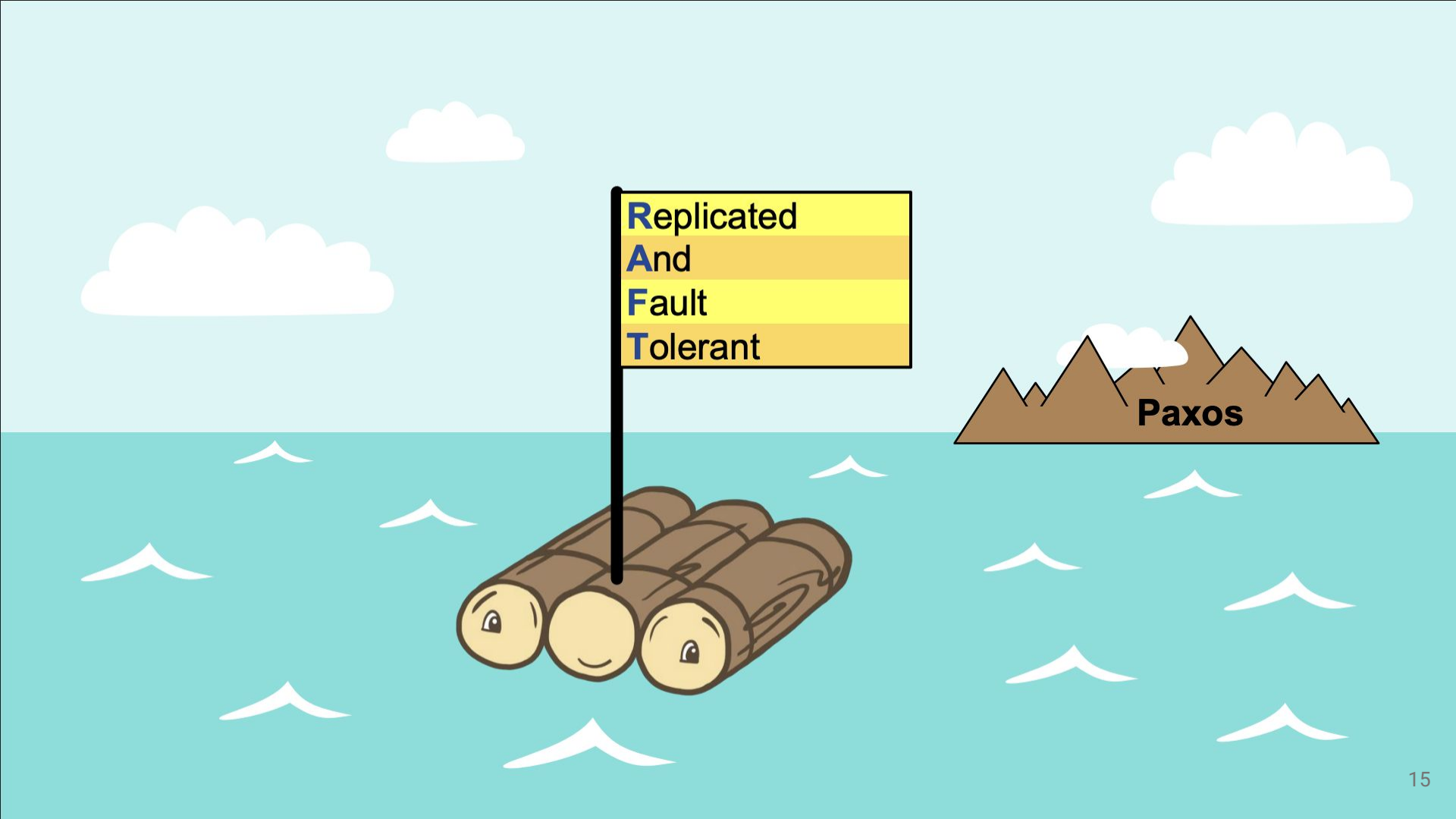
Problem : Paxos

- Exceptionally difficult to understand
- Poor foundation for building practical systems

- “There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. . . . the final system will be based on an unproven protocol” - Chubby implementers

Requirement

- A consensus algorithm that is easy to understand
- An intuition can be built around it.
- Extensions are inevitable in real-world implementations


Replicated
And
Fault
Tolerant
Paxos

Raft : Design Approach

- **Problem Decomposition:** divide problems into separate pieces
  - Leader election, log replication, and safety
- **State space reduction :** eliminate non-determinism wherever possible.
- Make design decisions based on **understandability**
- Randomized approaches introduce nondeterminism, but they tend to reduce the state space by handling all possible choices in a similar fashion

Raft : Core Idea

- Implement consensus by first electing a distinguished leader
- Give that leader complete responsibility for managing the replicated log.
- Tell servers when it is safe to apply log entries to their state machines
- **State Machine Safety**: If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

Problem : Raft

- How to elect a Leader?

Requirement

- Elect a leader from a set of servers
- Ensure only one leader emerges in an election

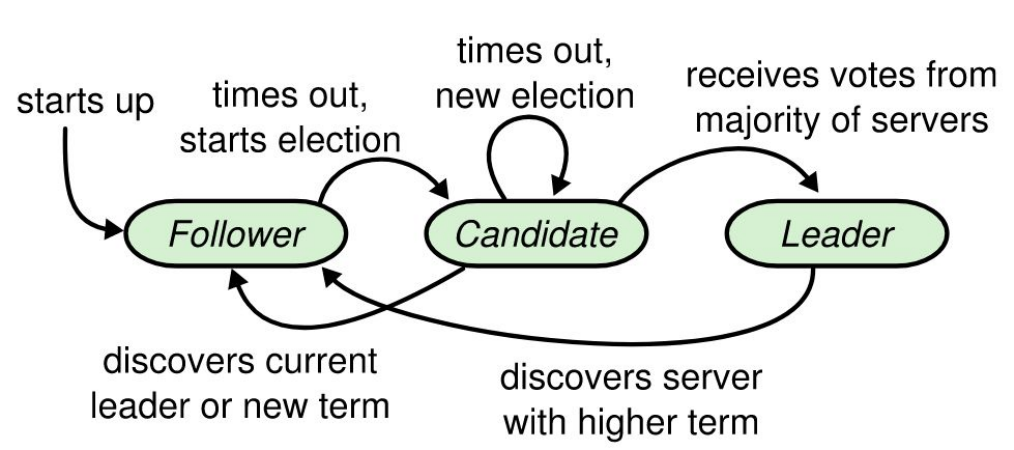
Raft Concepts

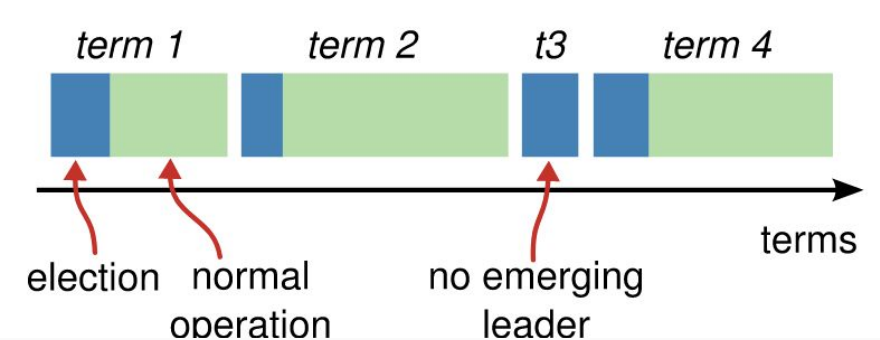
- A server can be in three states
  - **Leader** : handle client requests
  - **Candidate** : can compete in an election to become leader, if no leader is present
  - **Follower** : passive, issues no requests on their own. Responds to requests from leaders and candidates.
- **Terms** : Divide time into terms of arbitrary length, numbered with consecutive integers

Elections

- Each term begins with an election
- One or more candidates attempt to become leader
- If a candidate wins the election by getting **majority vote**, then it serves as leader for the rest of the term
- If an election results in no leader, a new election is initiated with a higher term
- Each server maintains an election timeout
- If timeout is triggered, term is increased and election begins
- Follower becomes a Candidate

Server States and Terms





Election Implementation : State of Servers

Persistent State : (Updated on stable storage before responding to RPCs)

- **currentTerm** : latest term server has seen (initialized to 0 on first boot, increases monotonically)
  - Need to remember when the server last voted, so it can vote again on a new term, and abstain if currentTerm is still active

- **votedFor** : candidateId that received vote in current term (or null if none)
  - A server can vote for more than one candidate if after recovering from a crash it forgets that it has already voted in the current term

Election Implementation : AppendEntries RPC

Used for heartbeats

Arguments :

- **term :** leader's term
- **leaderId** : for follower to redirect clients

Results :

- **term :** currentTerm, for leader to update itself
- **success** : false if term < currentTerm, else true

Election Implementation : RequestVote RPC

Invoked by candidates to gather votes

Arguments :

- **term :** candidate's term
- **candidateId** : candidate requesting vote

Results :

- **term :** currentTerm, for candidate to update itself
- **voteGranted** : reply false if term < currentTerm. If votedFor is null or candidateId(same candidate asking for vote in same term), grant vote

Election Implementation : Rules for All Servers

- If RPC request or response contains term T > currentTerm:
  - set currentTerm = T, convert to follower

Election Implementation : Rules for Followers

- Respond to RPCs from candidates and leaders
- If **election timeout elapses without receiving AppendEntries** RPC from current leader or **granting vote** to candidate: convert to candidate

Election Implementation : Rules for Candidates

On conversion to candidate, start election:

- Increment currentTerm
- Vote for self
- Reset election timer
- Send RequestVote RPCs to all other servers
- If votes received from majority of servers:
  - become leader
- If AppendEntries RPC received from new leader:
  - convert to follower
- If election timeout elapses: start new election

Election Implementation : Rules for Leaders

- After getting elected, send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts

Election Safety

Guarantee : At most one leader can be elected in a given term.

How?

- Each server will vote for **at most one** candidate in a **given term**, on a **first-come-first-served** basis
- A candidate wins an election if it receives votes from a **majority** of the servers in the full cluster for the **same term**.

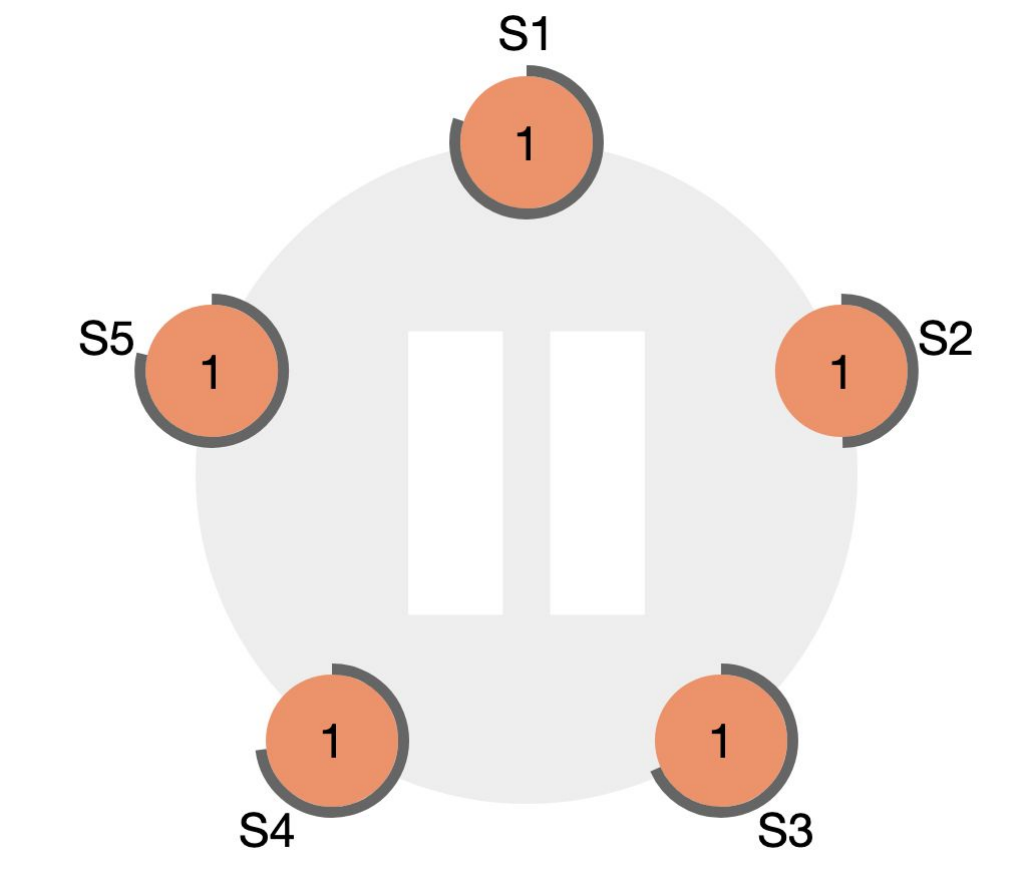
New Leader sends heartbeat messages to establish authority and prevent new elections

Election Safety

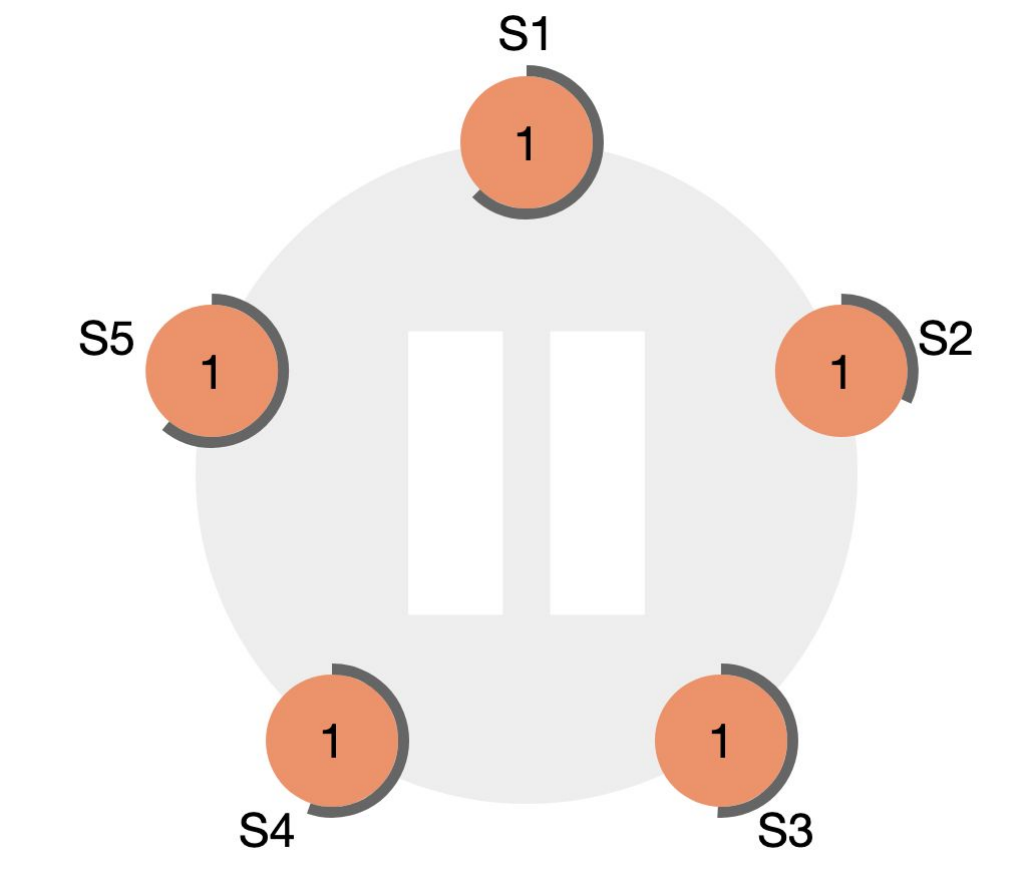
At most one leader can be elected in a given term

- Multiple leaders can exist at same point in time, belonging to different terms.
- Network Partition causes older leaders to not get info about new leaders
- Partitioned candidates that cannot get majority keep on increasing terms, if no heartbeat is received.
- When partition ends, this candidate because of its high term, can disrupt the cluster, by making current leader turn into follower, and trigger a new election.

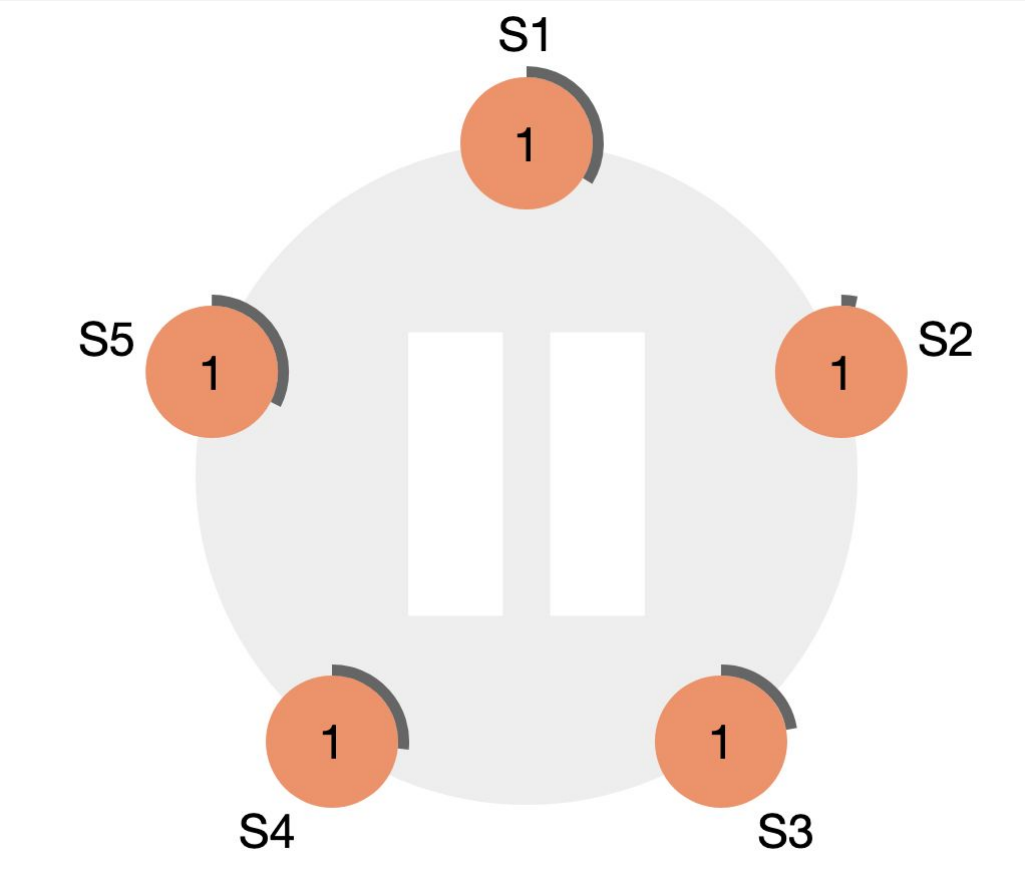
Election Example



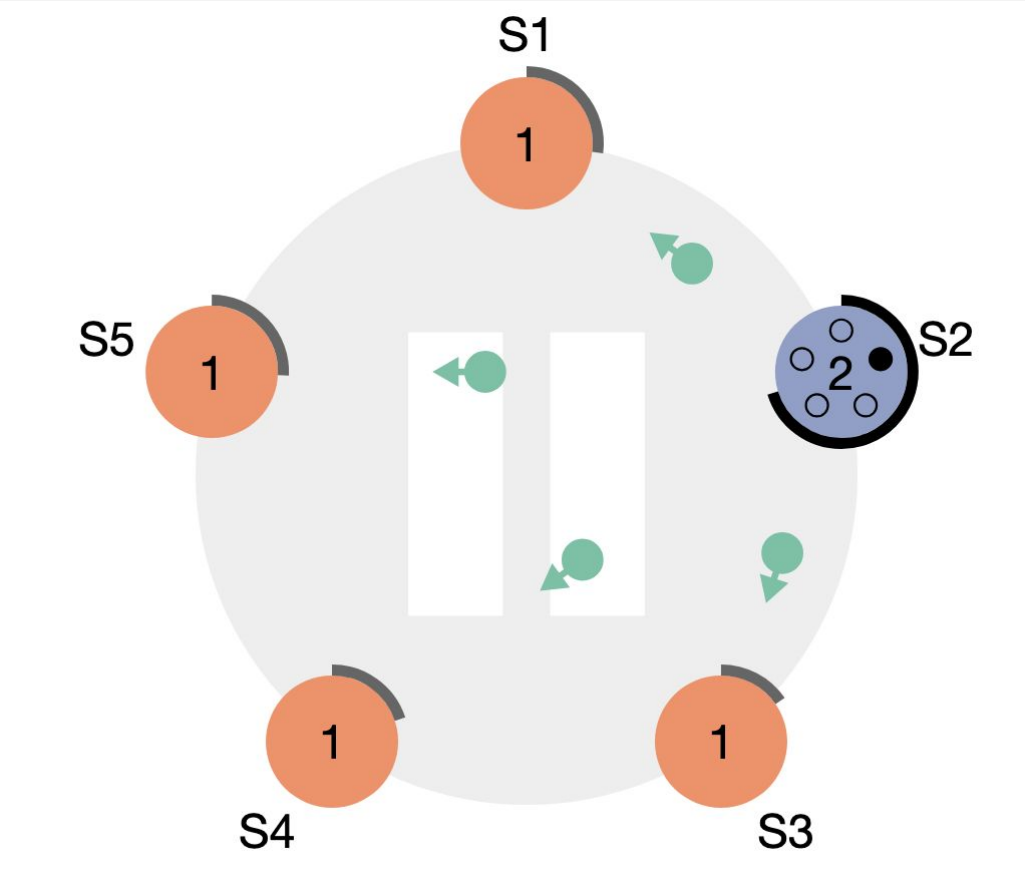
Election Example



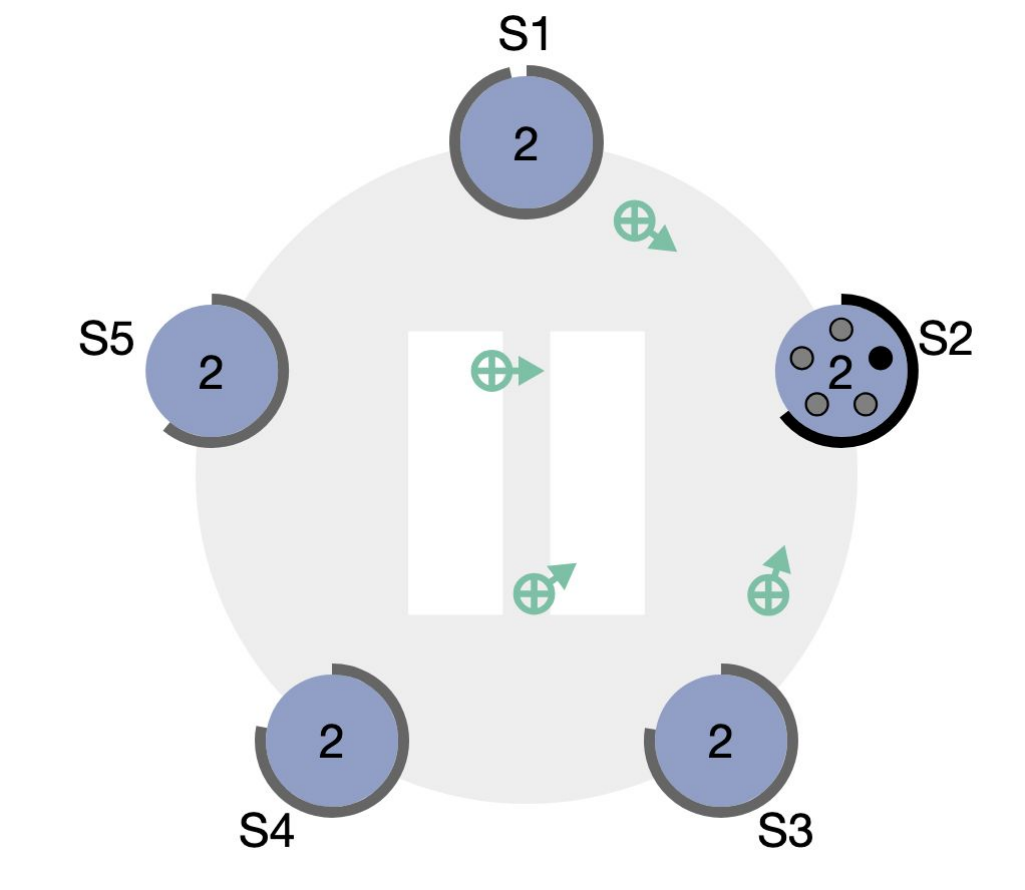
Election Example



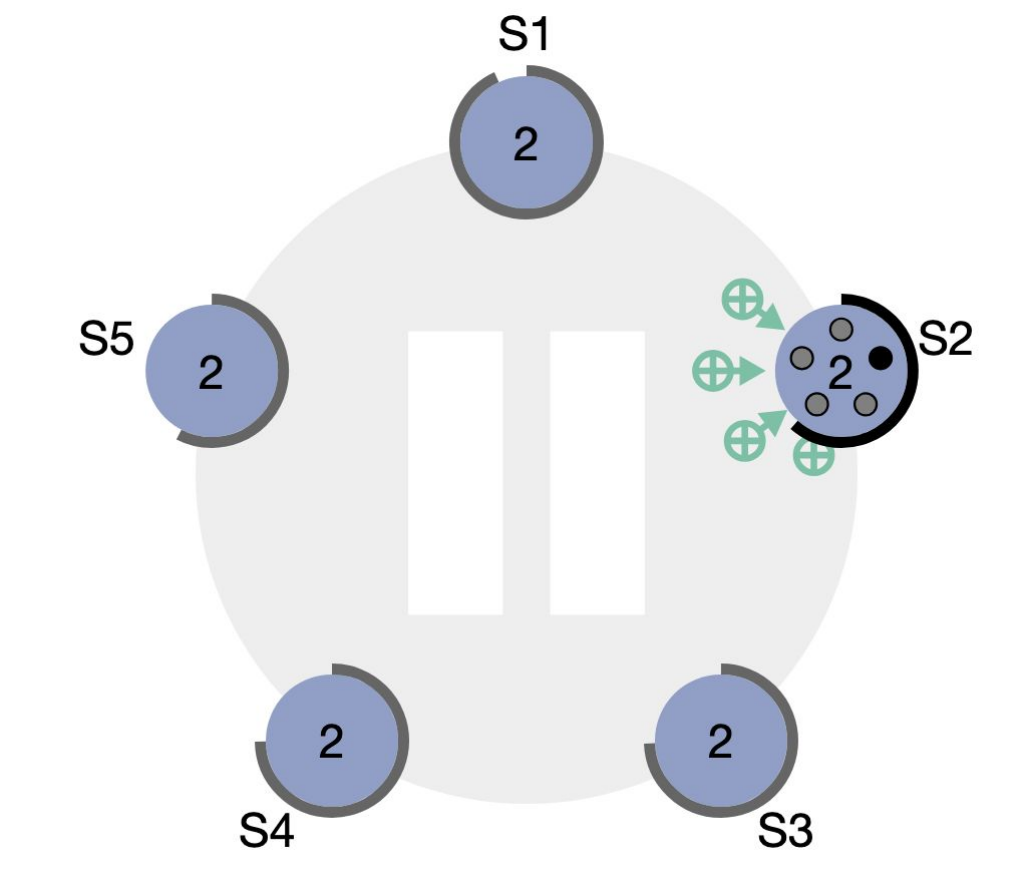
Election Example



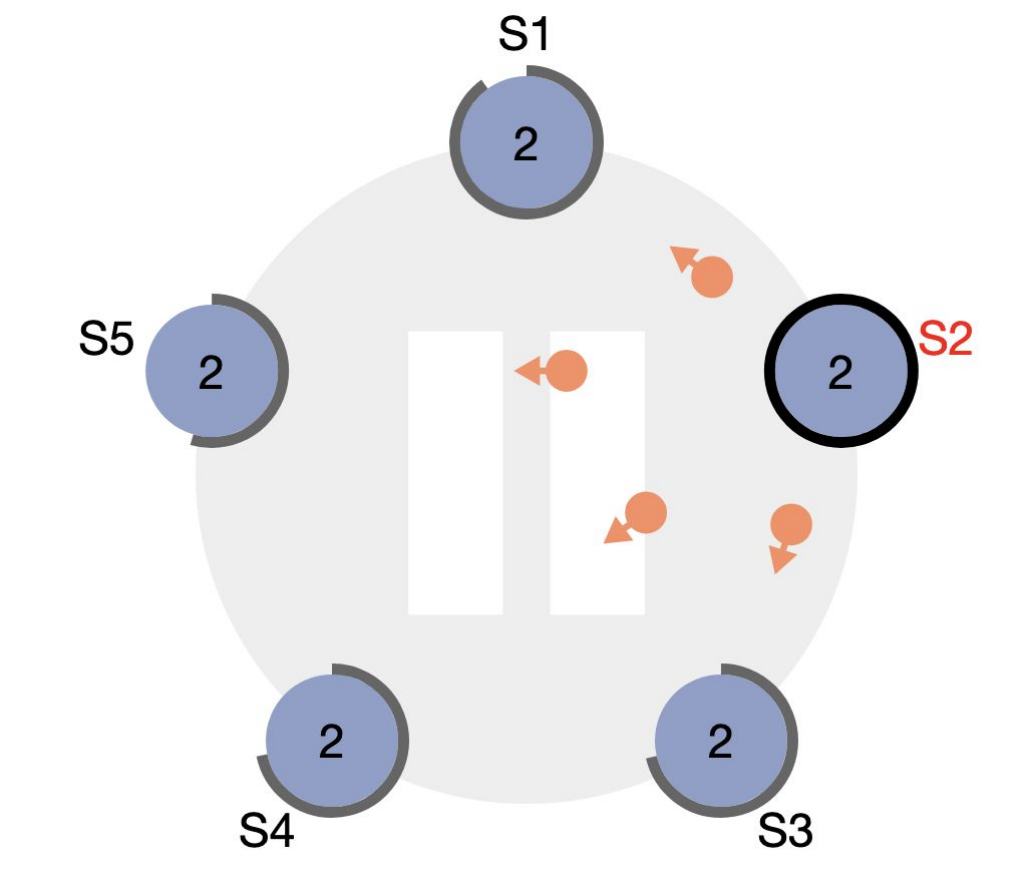
Election Example



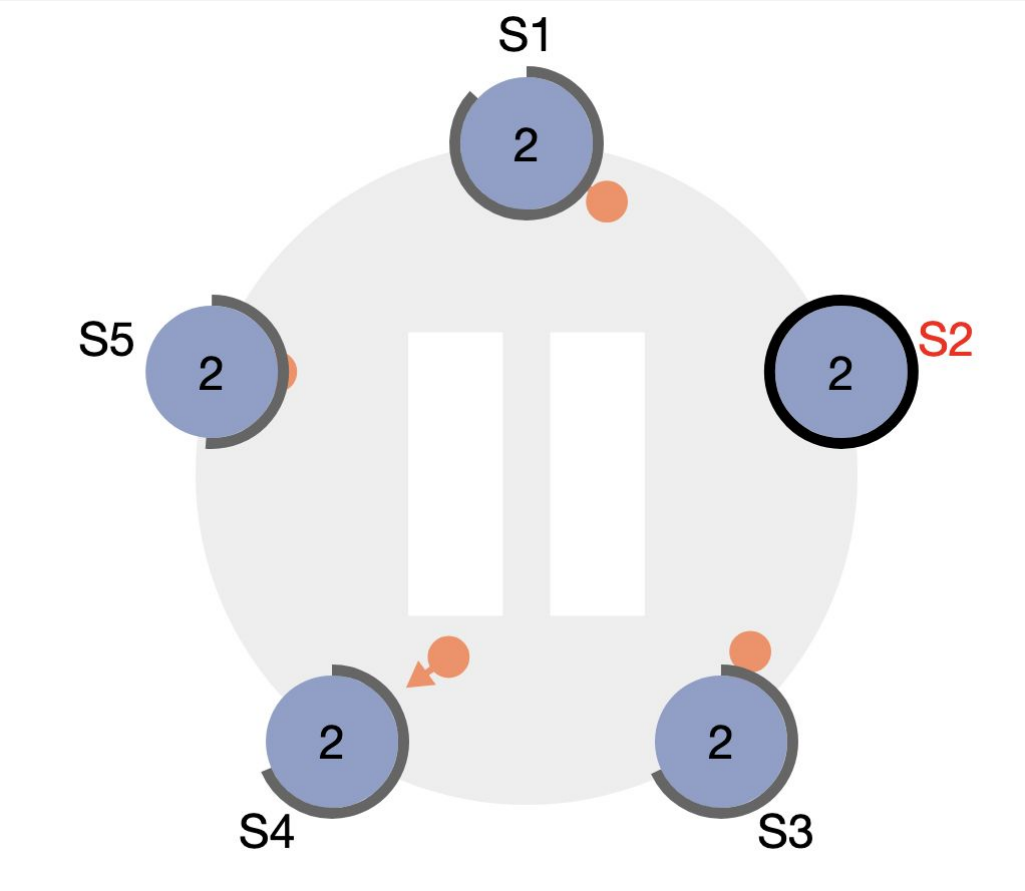
Election Example



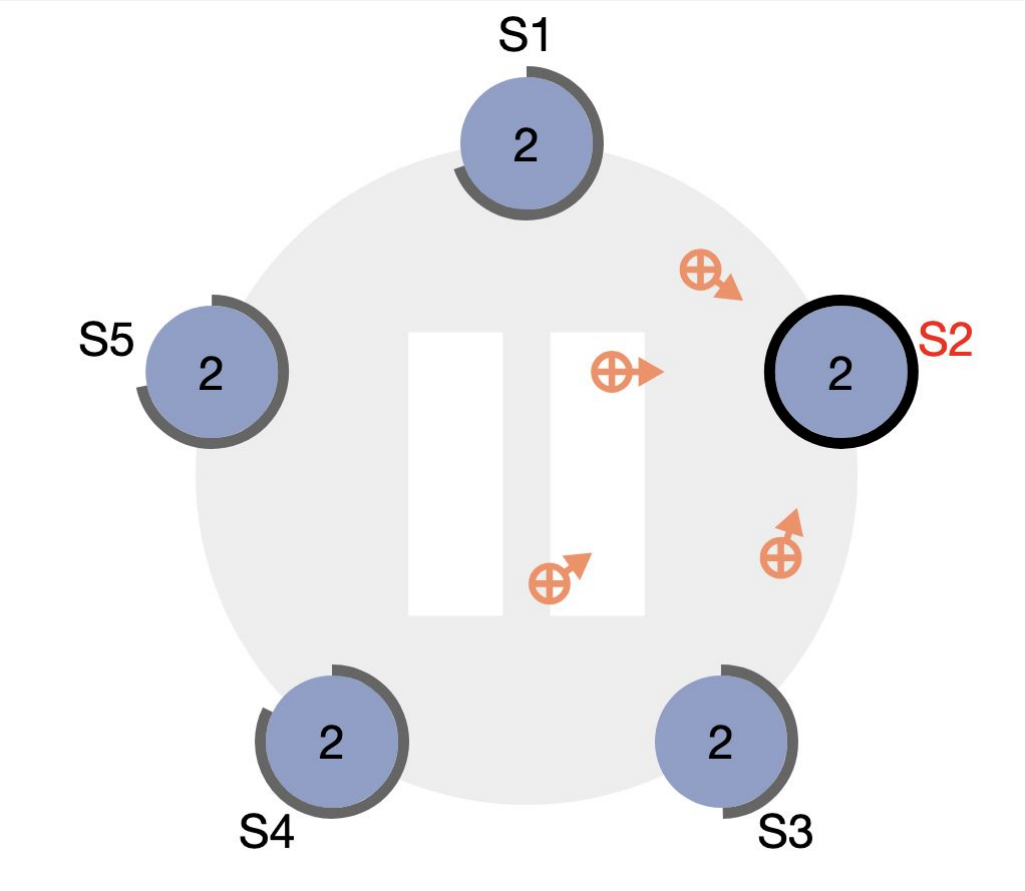
Election Example



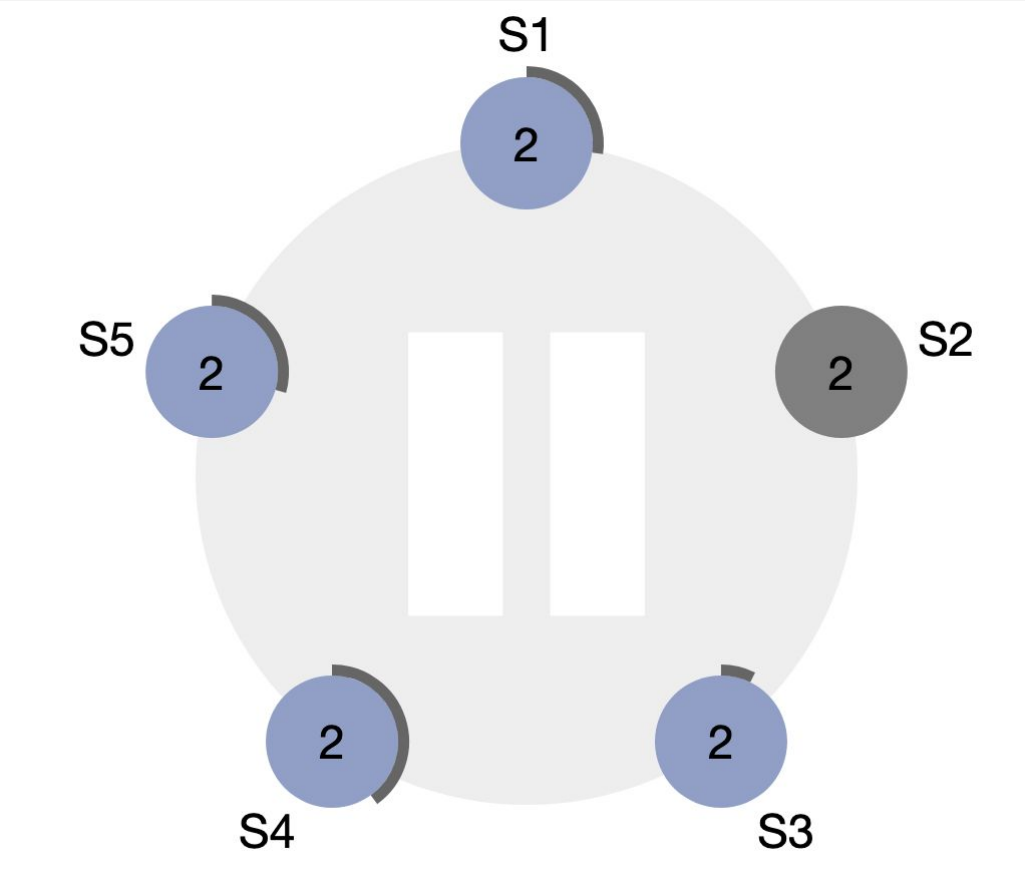
Election Example



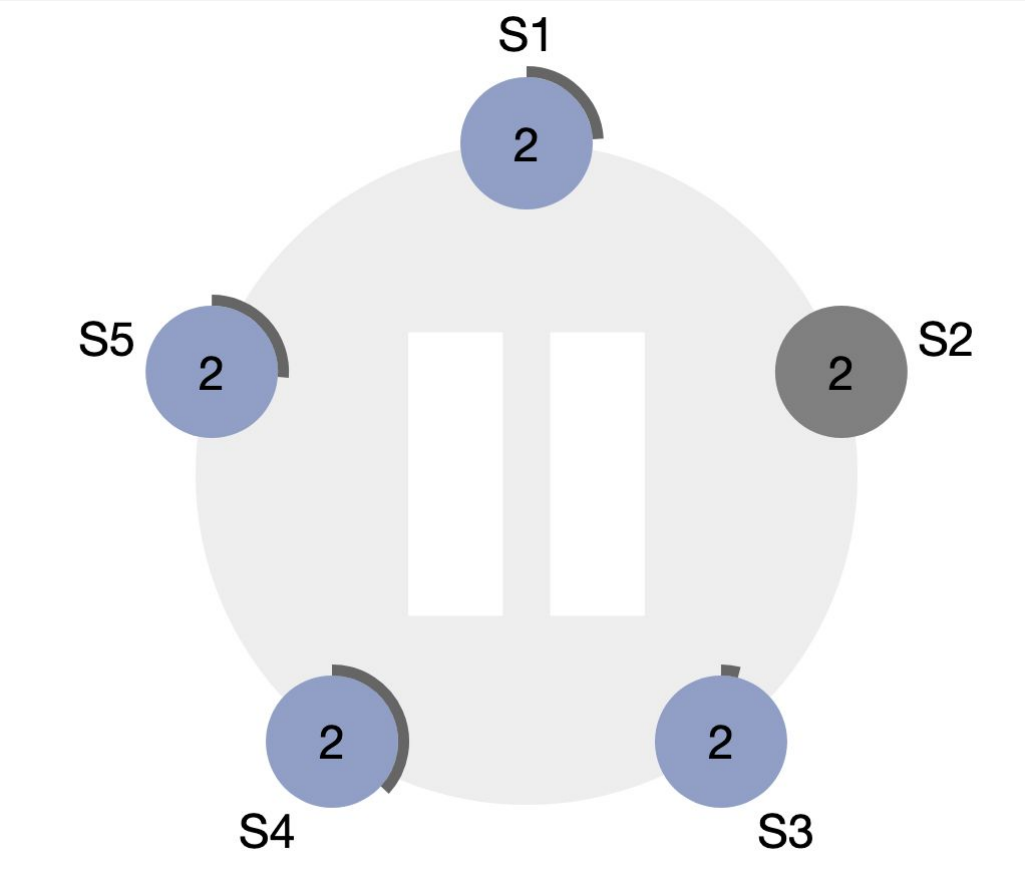
Election Example



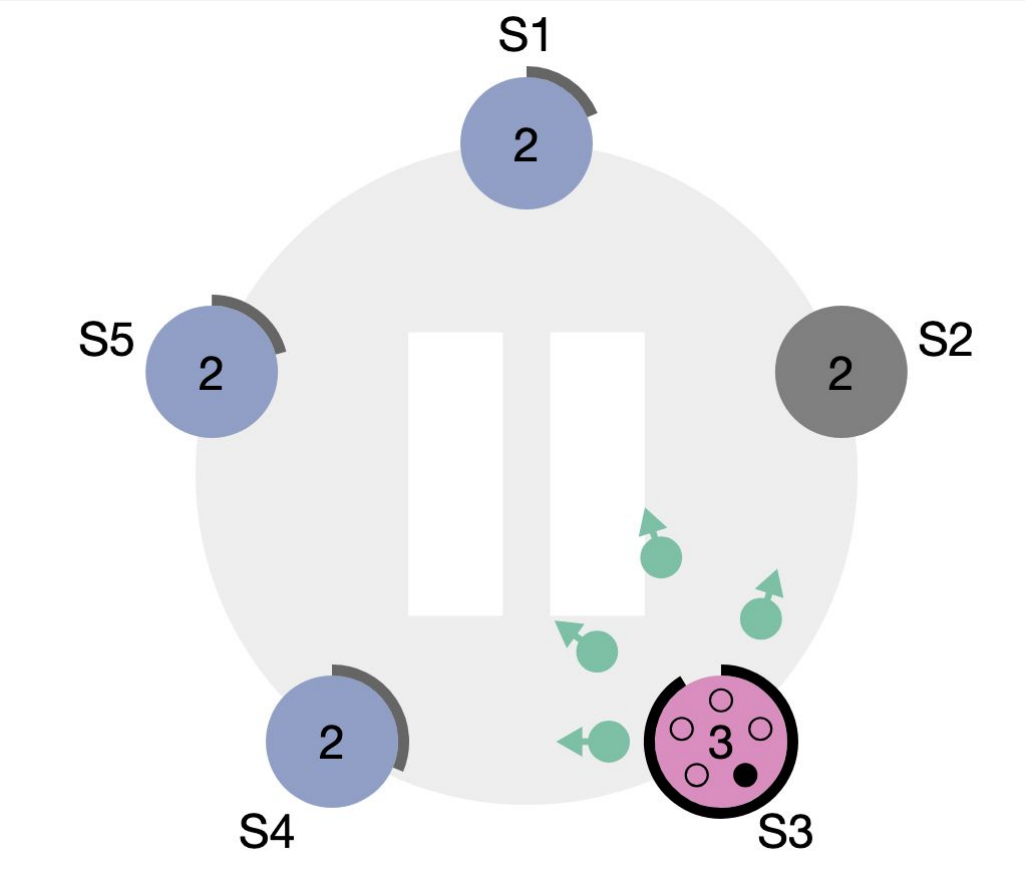
Election Example : Leader crash



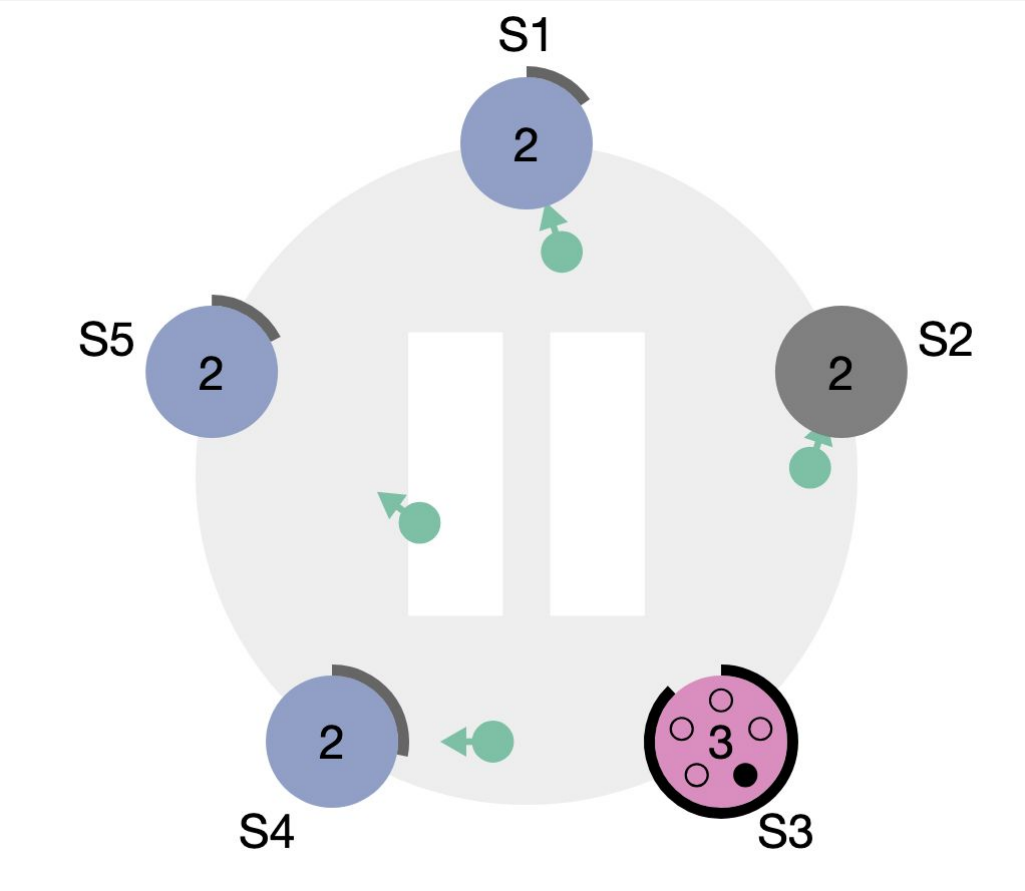
Election Example : Leader crash



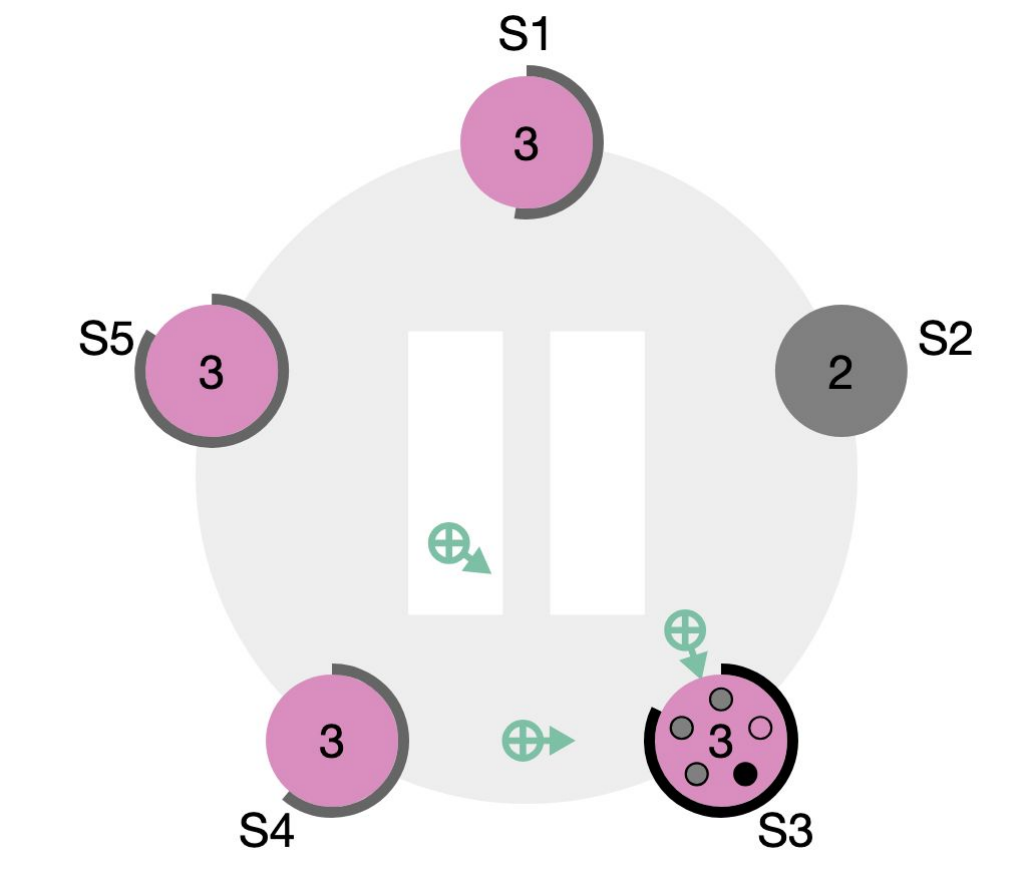
Election Example : Leader crash



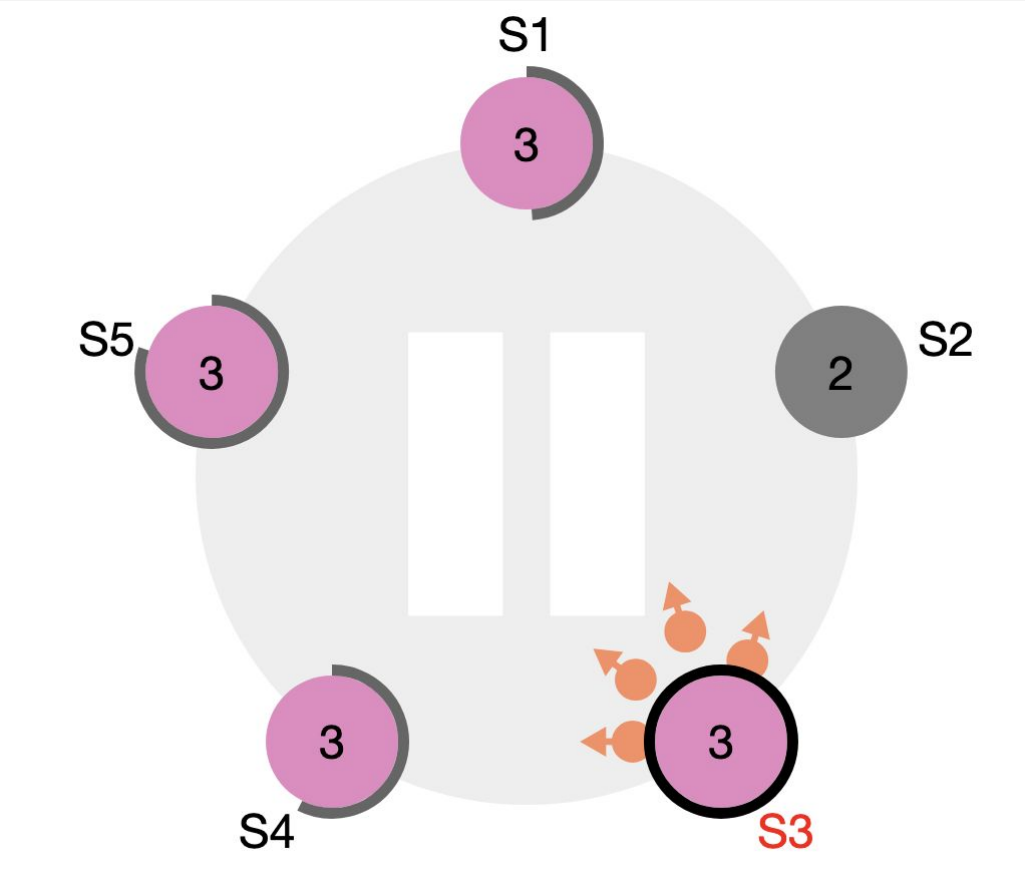
Election Example : Leader crash



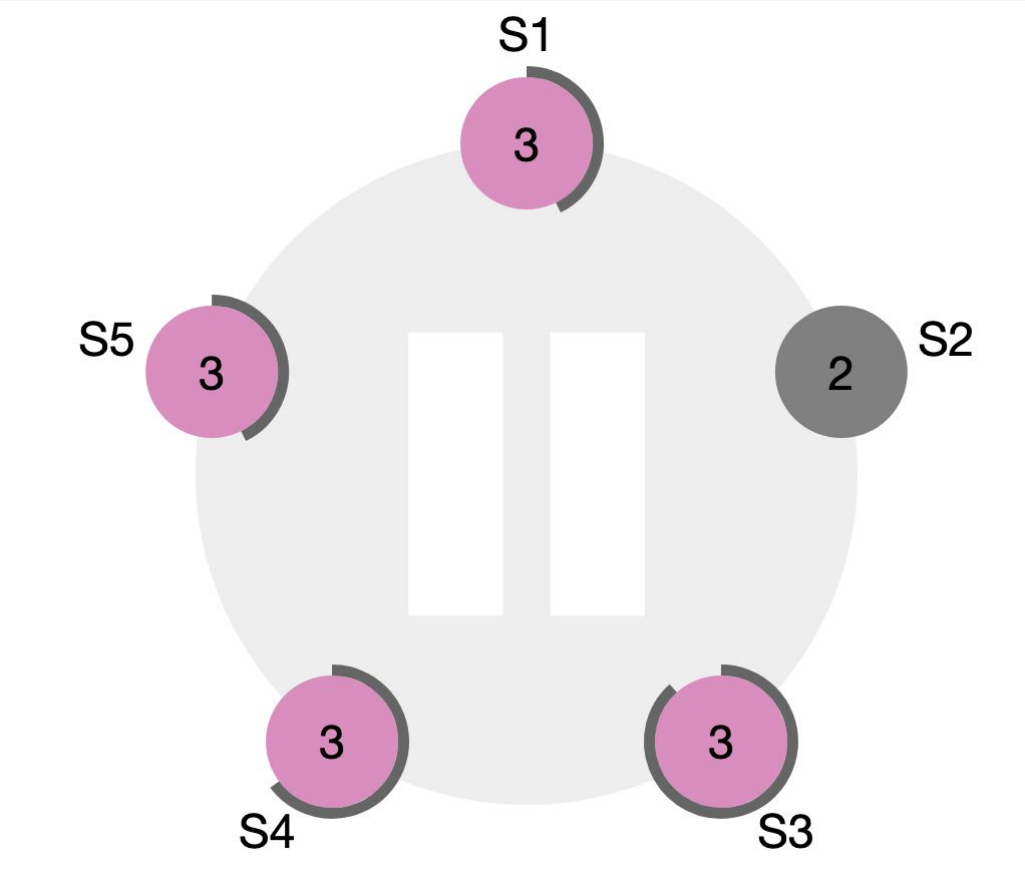
Election Example : Leader crash



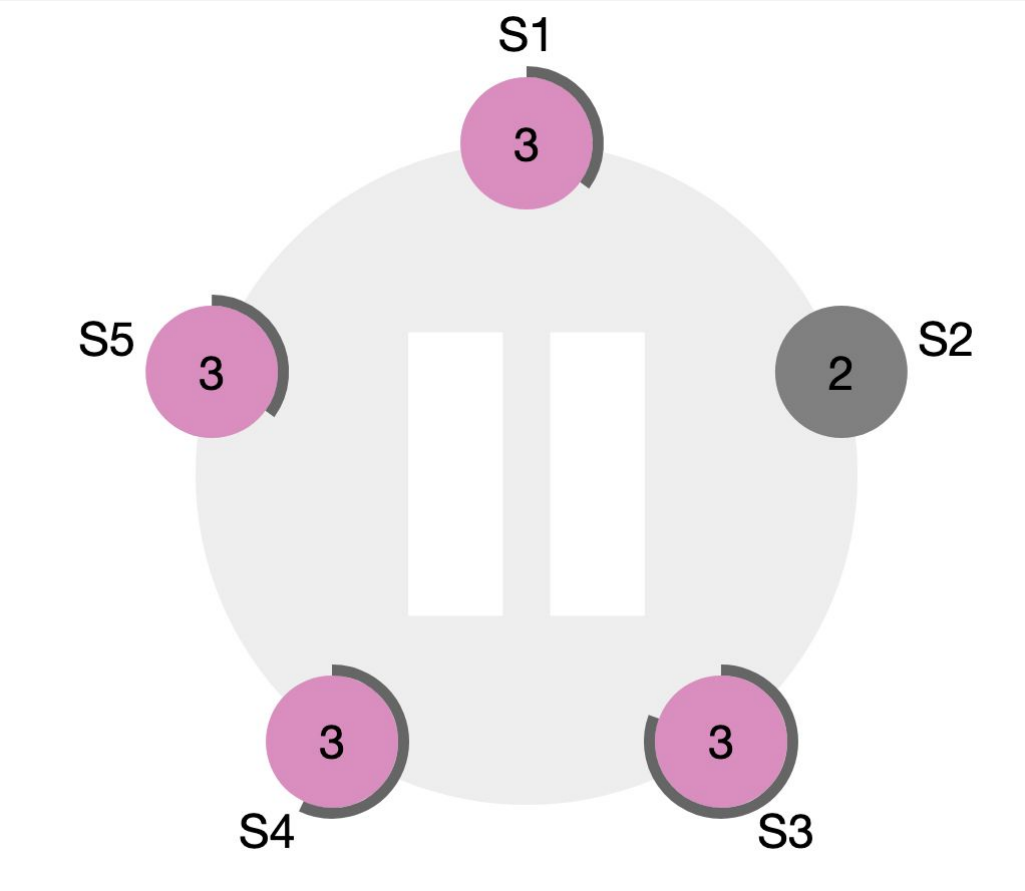
Election Example : Leader crash



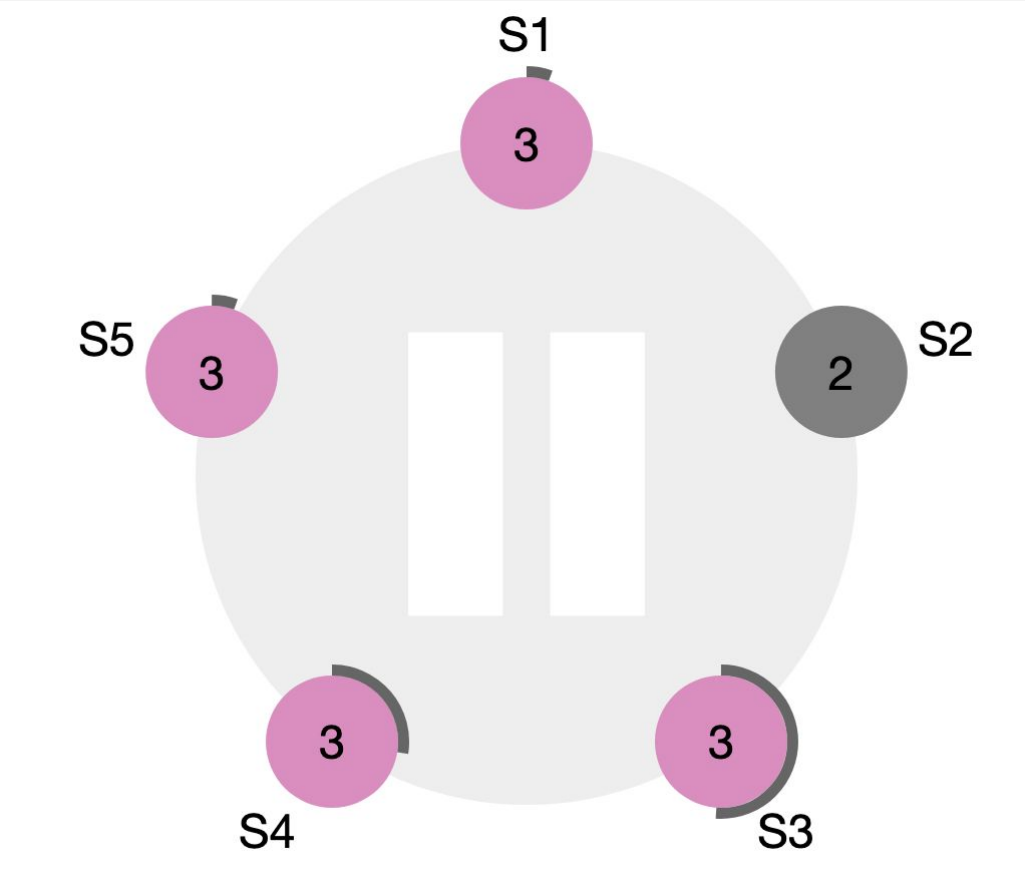
Election Example : Split Voting



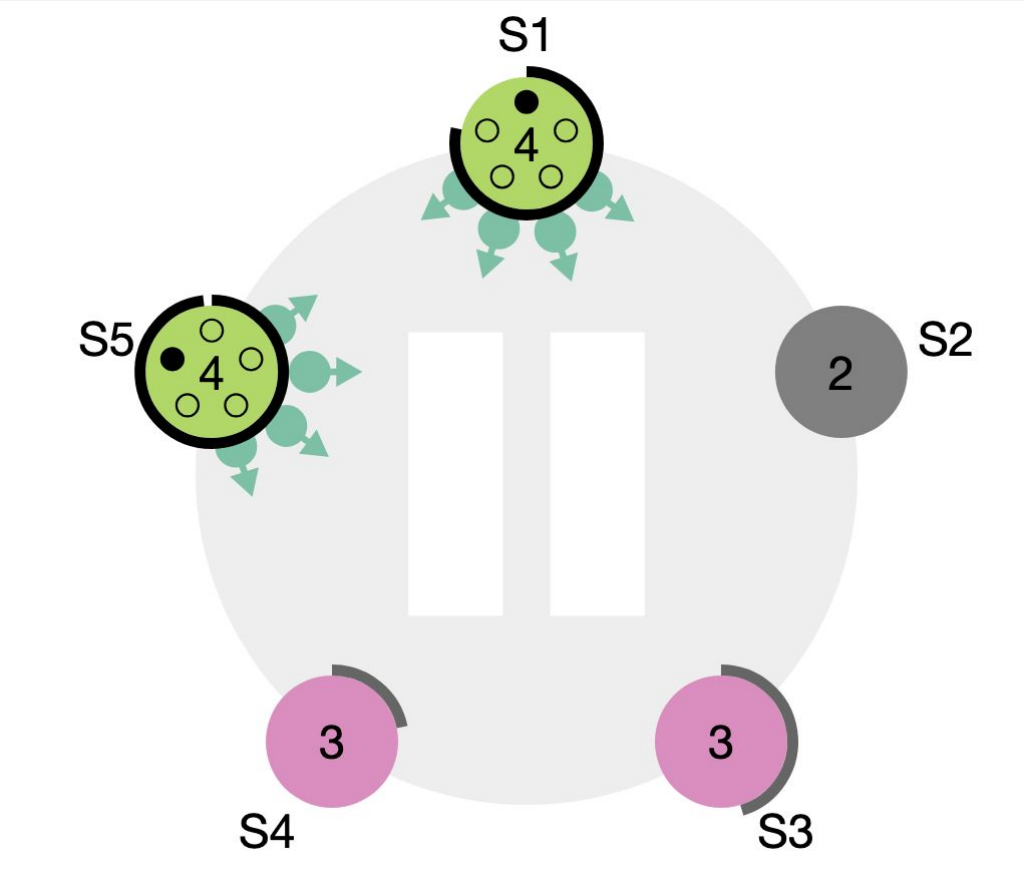
Election Example : Split Voting



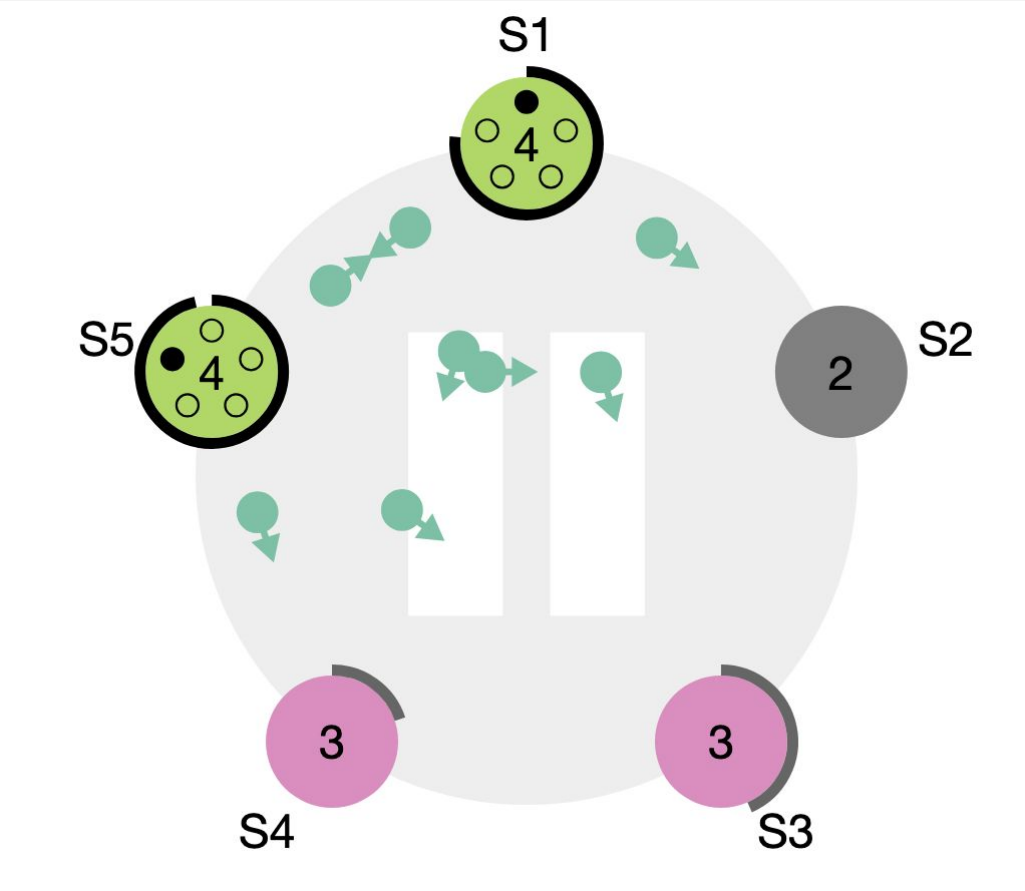
Election Example : Split Voting



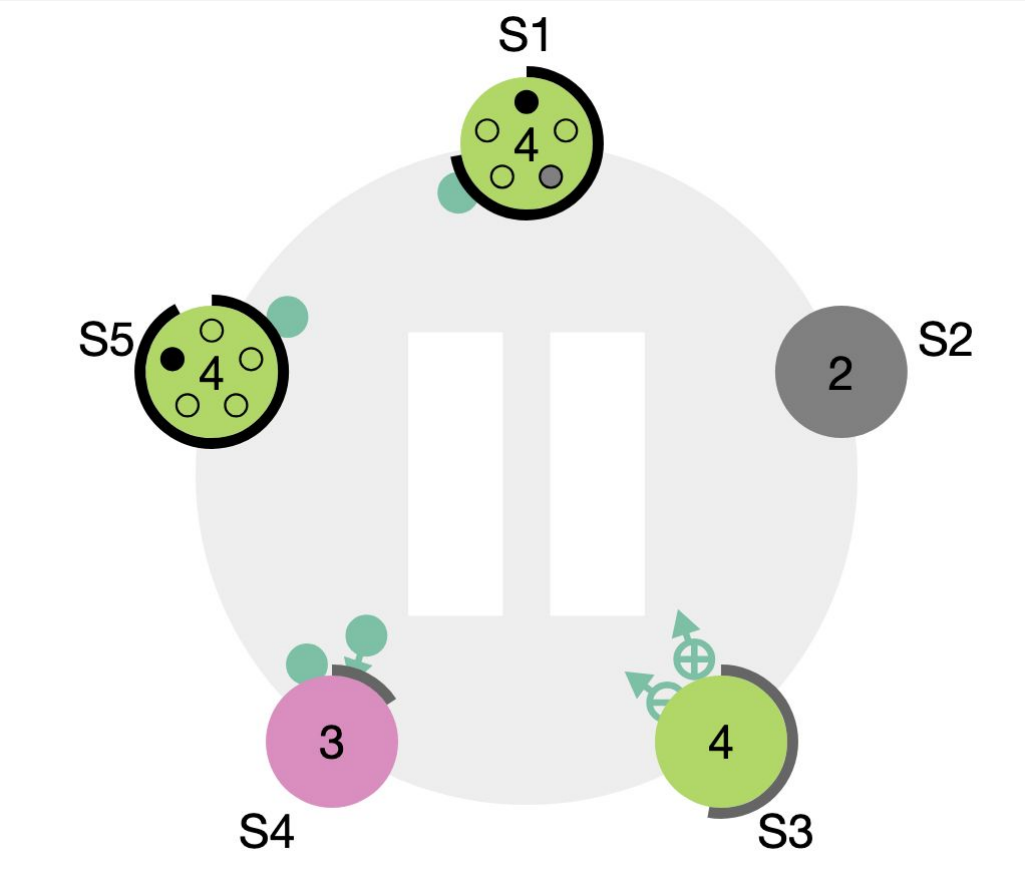
Election Example : Split Voting



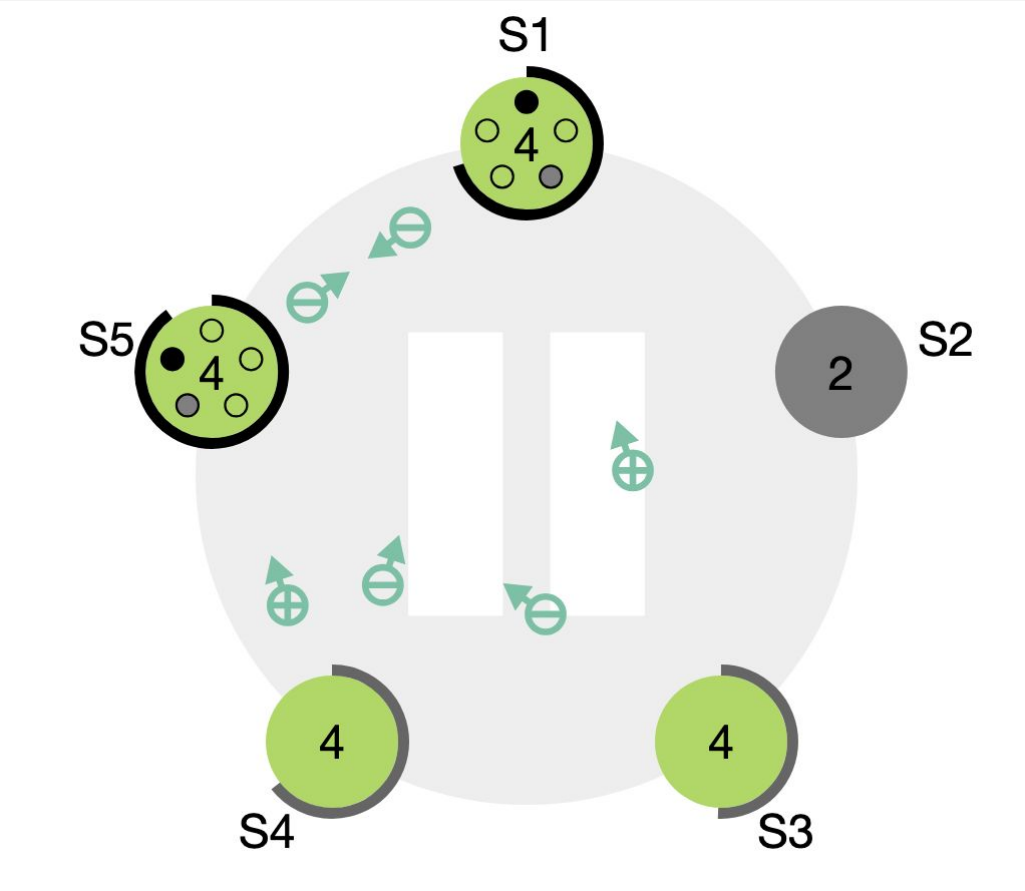
Election Example : Split Voting



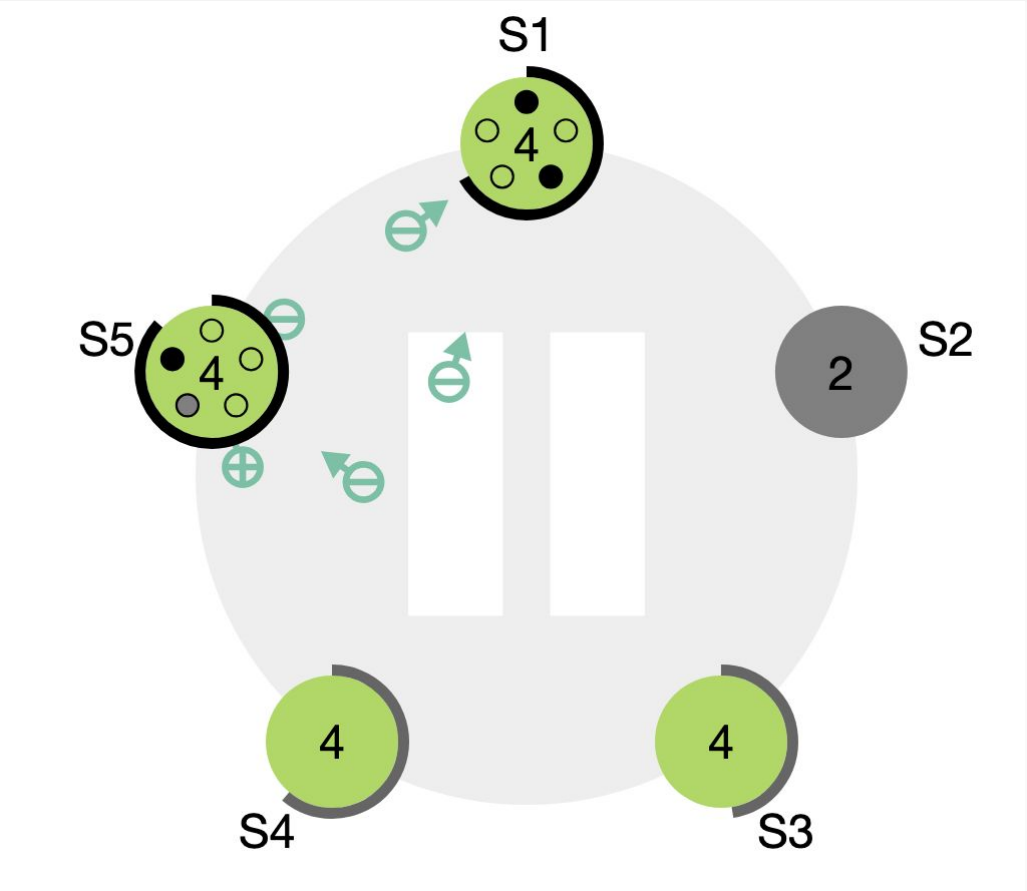
Election Example : Split Voting



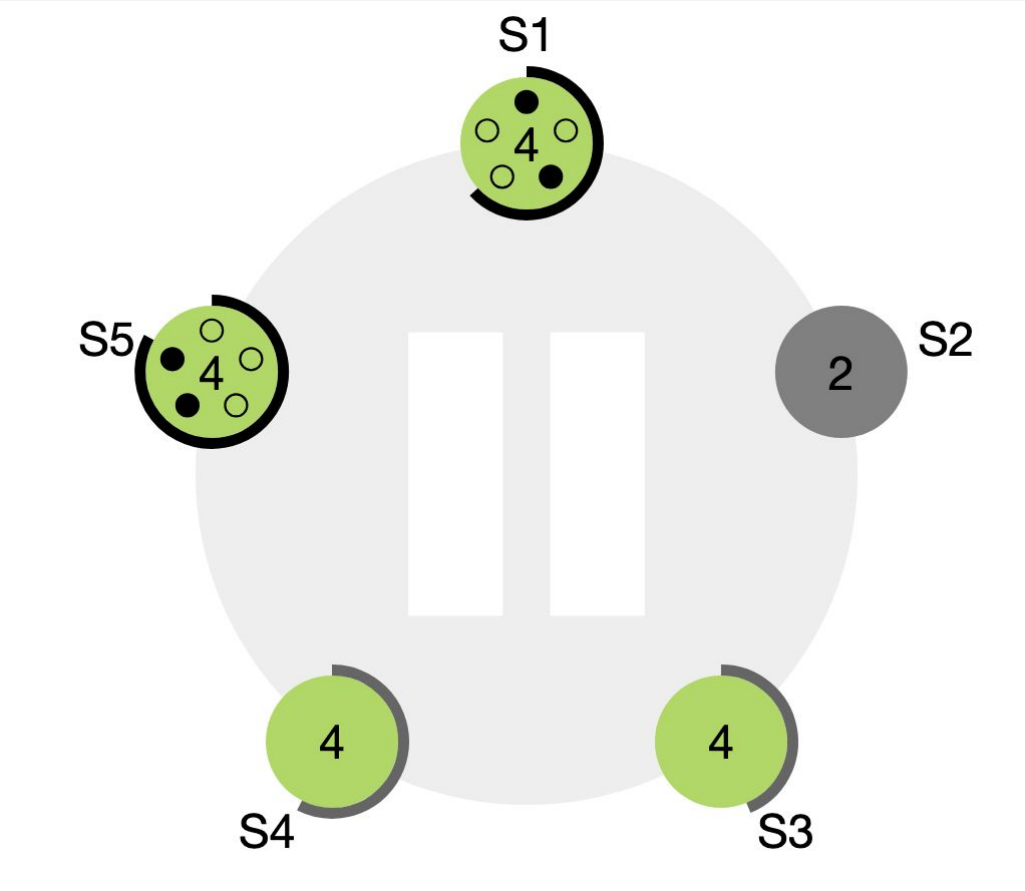
Election Example : Split Voting



Election Example : Split Voting



Election Example : Split Voting



Split Voting

- Use randomized election timeouts
- Choose random timeout every time, not just once at the start

Problem : Raft

- How to process write requests?

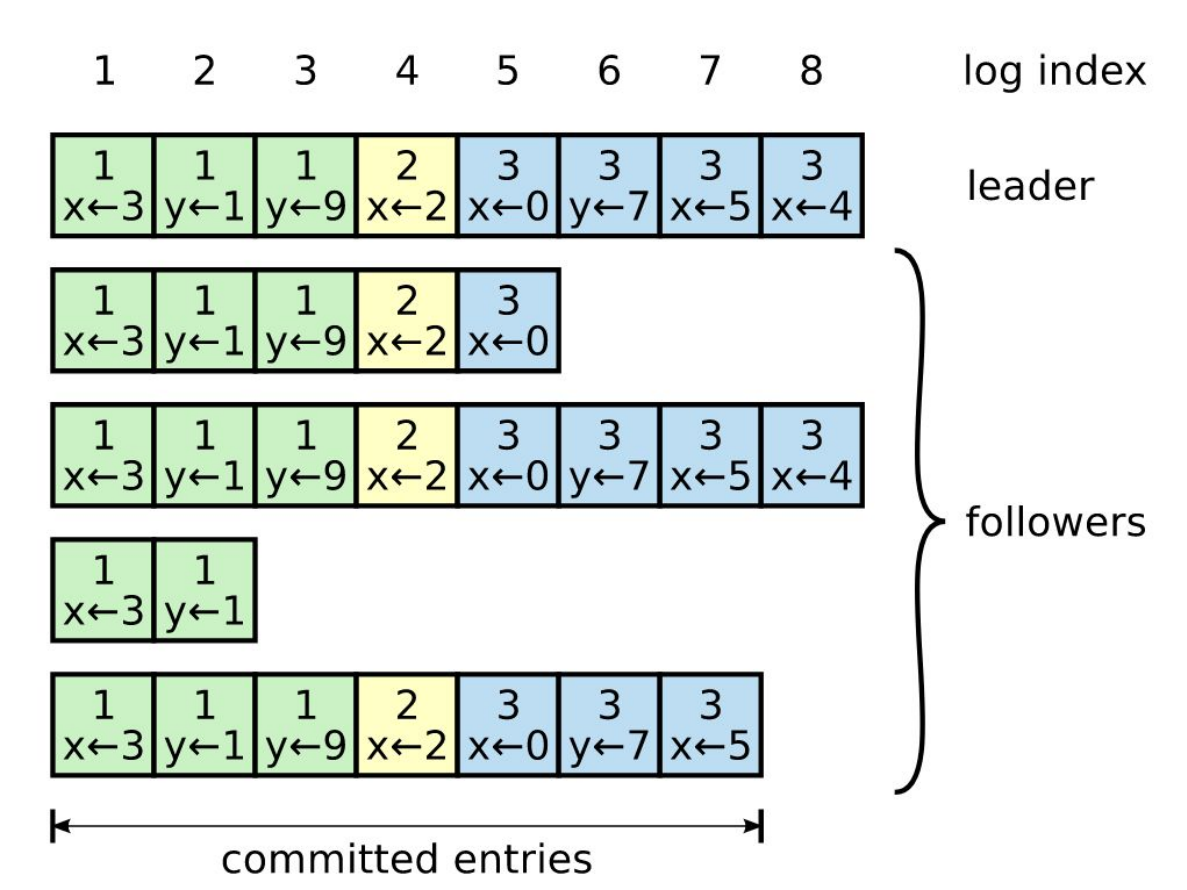
Requirement

- Allow writes to the cluster
- Ensure replicated log used to change state machines on servers is consistent

Log Replication : Idea

- Leader services client requests
- Client request contains command to be executed by the RSM
- Leader appends command to its log as a new entry
- issues AppendEntries RPCs in parallel to each of the other servers to replicate the entry.
- When the entry has been safely replicated(written on majority of servers), the leader applies the entry to its state machine
- returns the result of that execution to the client

Log entries


1 2 3 4 5 6 7 8
log index
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
leader
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
1 x←3 | 1 y←1
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5
followers
committed entries

Log entries

- Each log entry has an associated
  - term number
  - log index
- A log entry is committed once the leader that created the entry has replicated it on a majority of the servers
- A committed entry is safe to apply to the state machine
- Leader keep track of highest committed index, sends in AppendRPC msg
- Follower learns that an entry is committed, applies to local state machine in log order

Log Matching Safety Property

Guarantee 1: If two entries in different logs have the **same index** and **term**, then they **store** the **same command**

Proof

- If two entries in different logs have **index ‘i’**, and **term ‘t’**, then that entry was generated by a **leader in term ‘t’, at index ‘i’** in its log.
- Log entries **never change** their **position**
- A Leader only generates a single entry, containing a single command at **index ‘i’** and **term ‘t’**

Log Matching Safety Property

Guarantee 2: If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

Proof by Induction

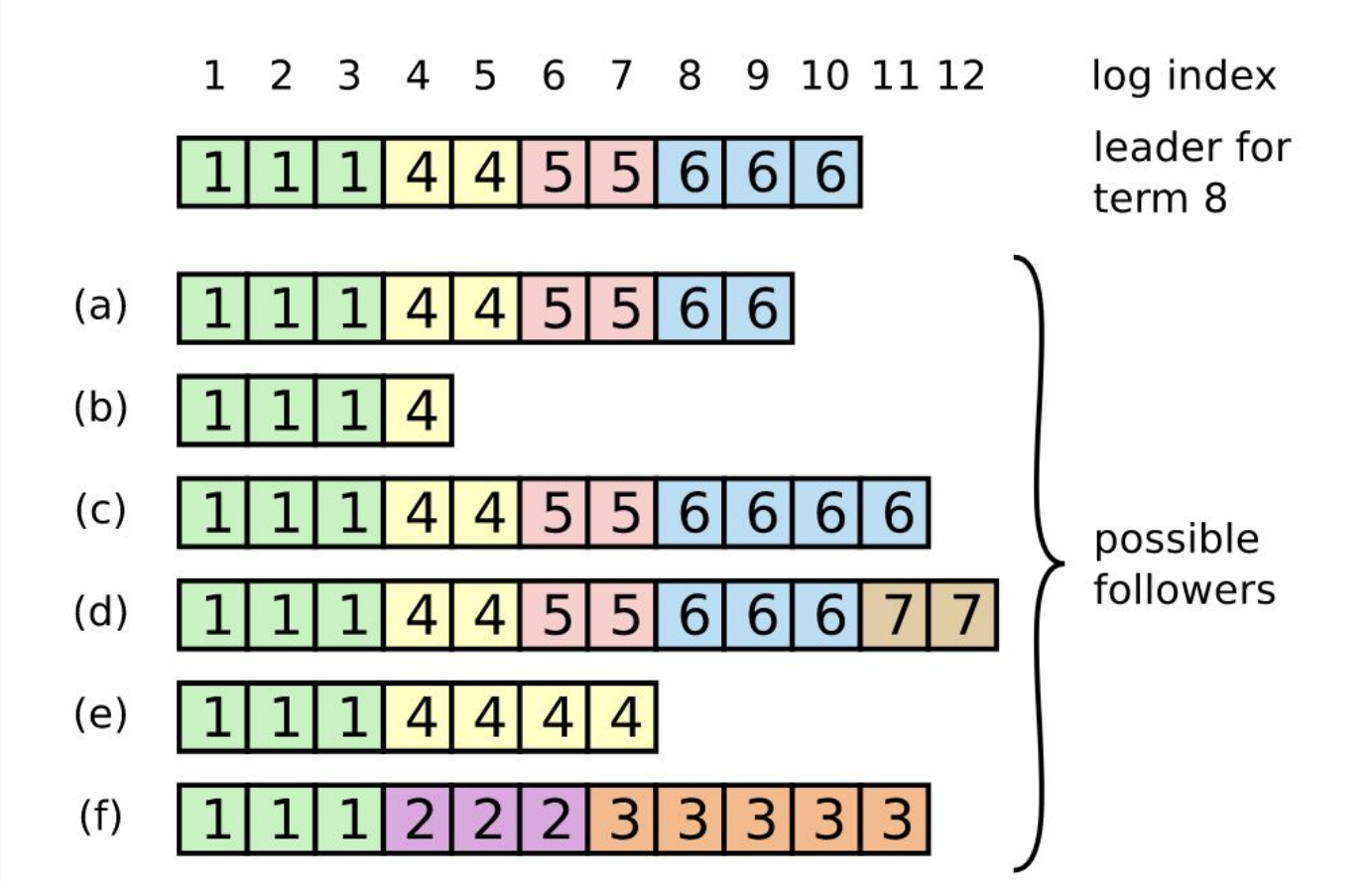
- **Base Case** : When both Leader and Follower logs are empty, Log Matching is satisfied
- **Inductive Case** : For each follower log entry to be extended, an AppendEntries RPC consistency check is done, to check whether an entry with the same index(**prevLogIndex**) and term(**prevLogTerm**) sent by the Leader(for its previous entry) exists in its Log

  Whenever AppendEntries returns successfully, the leader knows that the follower's log is identical to its own log up through the new entries.

Log Inconsistencies

- A follower may have:
  - missing entries that are present on the leader
  - Extra entries not present on leader
  - Or both

Log Inconsistencies



Log Inconsistencies

- Leader handles inconsistencies by forcing the followers' logs to duplicate its own
- **Conflicting entries** in **follower logs** will be **overwritten** with entries from the leader's log
  - Find latest log entry where two logs agree
  - Delete entries in follower after that point
  - Send the follower all of the leader's entries after that point
- The leader maintains a ***nextIndex*** for each follower, which is the index of the next log entry the leader will send to that follower.
- ***nextIndex*** initialized to index just after last one in leader's log

Log Inconsistencies : Consistency check

- Performed by AppendEntries RPC
- If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC.
- After a rejection, the leader decrements ***nextIndex*** and **retries** the **AppendEntries** RPC
- Eventually nextIndex will reach a point where the leader and follower logs match => AppendEntries succeeds => follower's log is consistent with leader's log
- Adds leader's entries to follower

Leader Append-Only Safety Property

Guarantee : A Leader never overwrites or deletes entries in its log, but only appends new entries.

How?

- **Requests** received from **clients** are **appended** in the **Leader's log**.
- If a **follower's log** is **inconsistent** with the **Leader's**, the **AppendEntries**(RPC) **consistency check** will fail.
- **Follower log** entries are **deleted** till the **point** where **leader** and **follower logs match**

No need for Leader to trim/delete its own entries for consistency with followers

Log Replication : State of Servers(Old)

Persistent State :

- **currentTerm :** latest term server has seen (initialized to 0 on first boot, increases monotonically)
- **votedFor :** candidateId that received vote in current term (or null if none)

Log Replication : State of Servers(Updated)

Persistent State :

- **currentTerm** : latest term server has seen (initialized to 0 on first boot, increases monotonically)
- **votedFor** : candidateId that received vote in current term (or null if none)
- **Log[] :** each log entry contains command for state machine and term when command was received.(Index is implicit)

Volatile state on all servers :

- **commitIndex :** index of highest log entry known to be committed (initialized to 0, increases monotonically)
- **lastApplied :** index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders: (Reinitialized after election)

- **nextIndex[] :** for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
- **matchIndex[] :** for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

Log Replication : RequestVote RPC(No change)

Invoked by candidates to gather votes

Arguments :

- **term :** candidate's term
- **candidateId** : candidate requesting vote

Results :

- **term :** currentTerm, for candidate to update itself
- **voteGranted** : reply false if term < currentTerm. If votedFor is null or candidateId(same candidate asking for vote in same term), grant vote

Log Replication : AppendEntries RPC (Old)

Arguments :

- **term :** leader's term
- **leaderId** : for follower to redirect clients

Results :

- **term :** currentTerm, for leader to update itself
- **success** : Reply false if term < currentTerm, else True

Log Replication : AppendEntries RPC (Updated)

Arguments :

- **term :** leader's term
- **leaderId** : for follower to redirect clients
- **prevLogIndex** : index of log entry immediately preceding new ones
- **prevLogTerm** : term of prevLogIndex entry
- **Entries[]** : log entries to store (empty for heartbeat)
- **leaderCommit** : leader's commitIndex

Results :

- **term :** currentTerm, for leader to update itself
- **success** : true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1) Reply false if term < currentTerm
2) Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3) If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4) Append any new entries not already in the log
5) If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

Log Replication : Rules for All Servers(Old)

- If RPC request or response contains term T > currentTerm:
  - set currentTerm = T, convert to follower

Log Replication : Rules for All Servers(Updated)

- If RPC request or response contains term T > currentTerm:
  - set currentTerm = T, convert to follower
- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine

Log Replication : Rules for Followers(No change)

- Respond to RPCs from candidates and leaders
- If **election timeout elapses without receiving AppendEntries** RPC from current leader or **granting vote** to candidate: convert to candidate

Log Replication : Rules for Candidates(No change)

On conversion to candidate, start election:

- Increment currentTerm
- Vote for self
- Reset election timer
- Send RequestVote RPCs to all other servers
- If votes received from majority of servers:
  - become leader
- If AppendEntries RPC received from new leader:
  - convert to follower
- If election timeout elapses: start new election

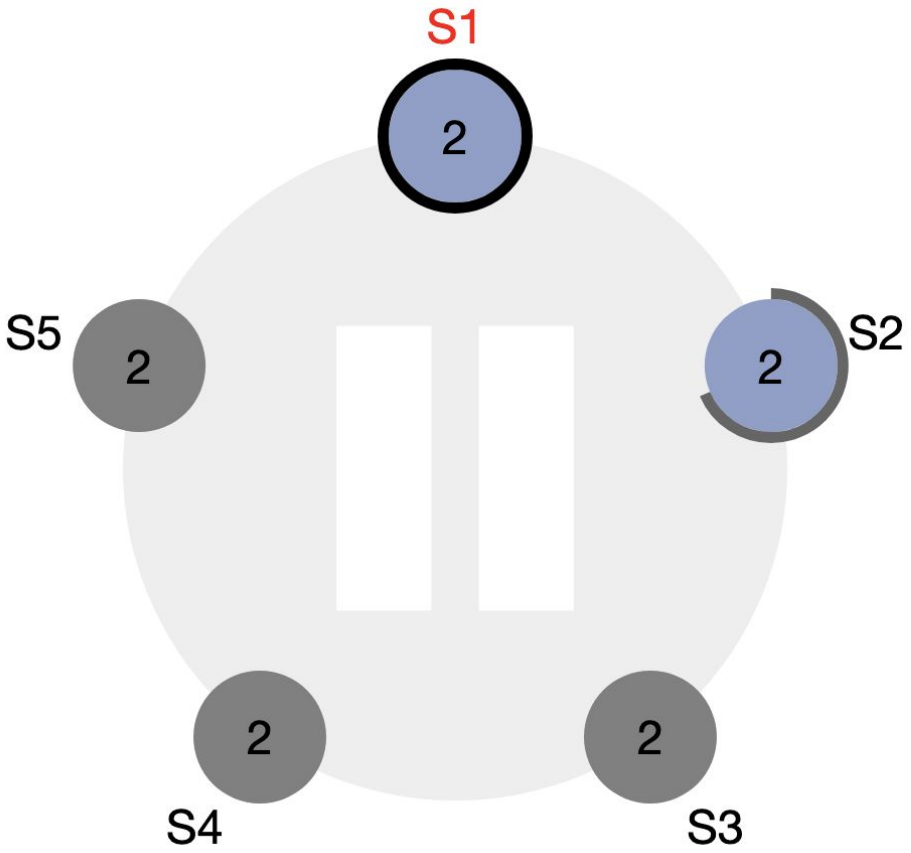
Log Replication: Rules for Leaders(Old)

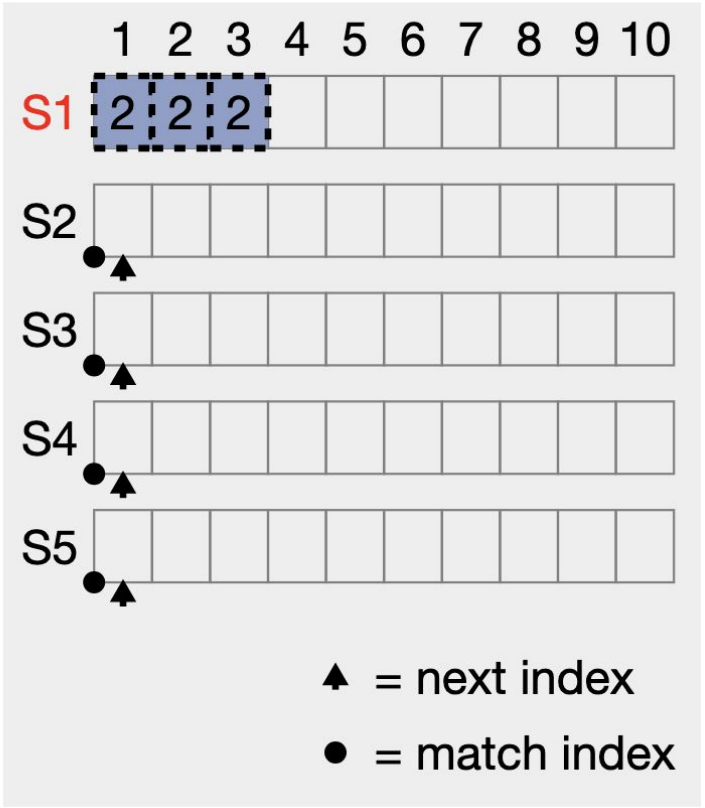
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts

Log Replication: Rules for Leaders(Updated)

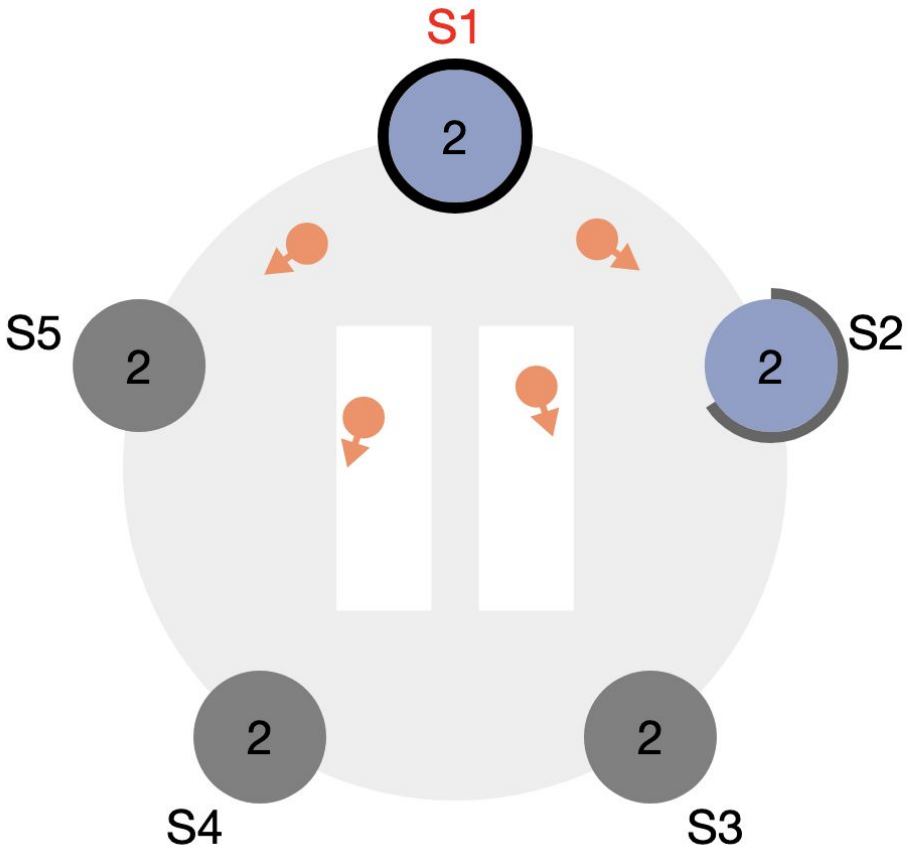
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts
- If command received from client:
  - append entry to local log,
  - respond after entry applied to state machine
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, set commitIndex = N

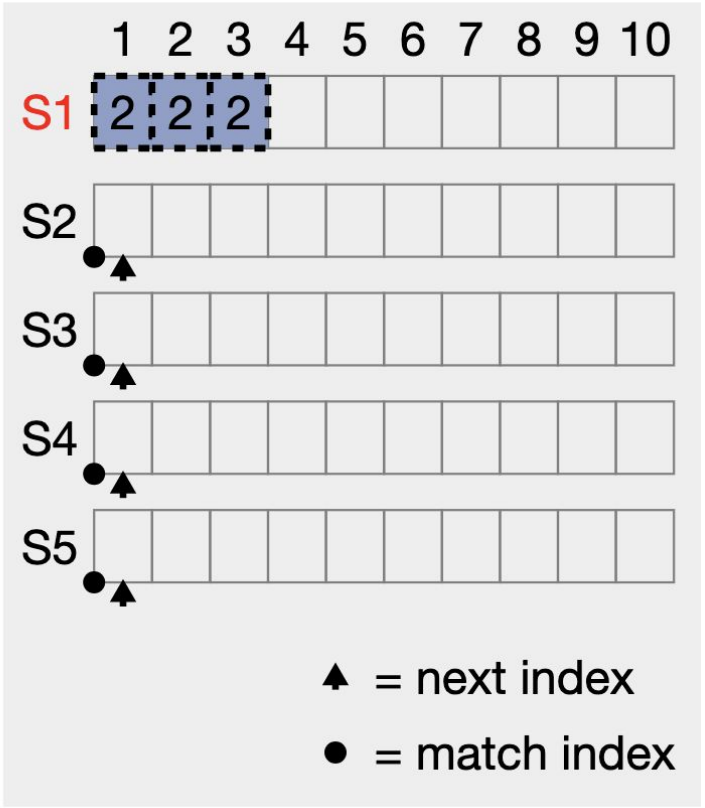
Log Replication: Example



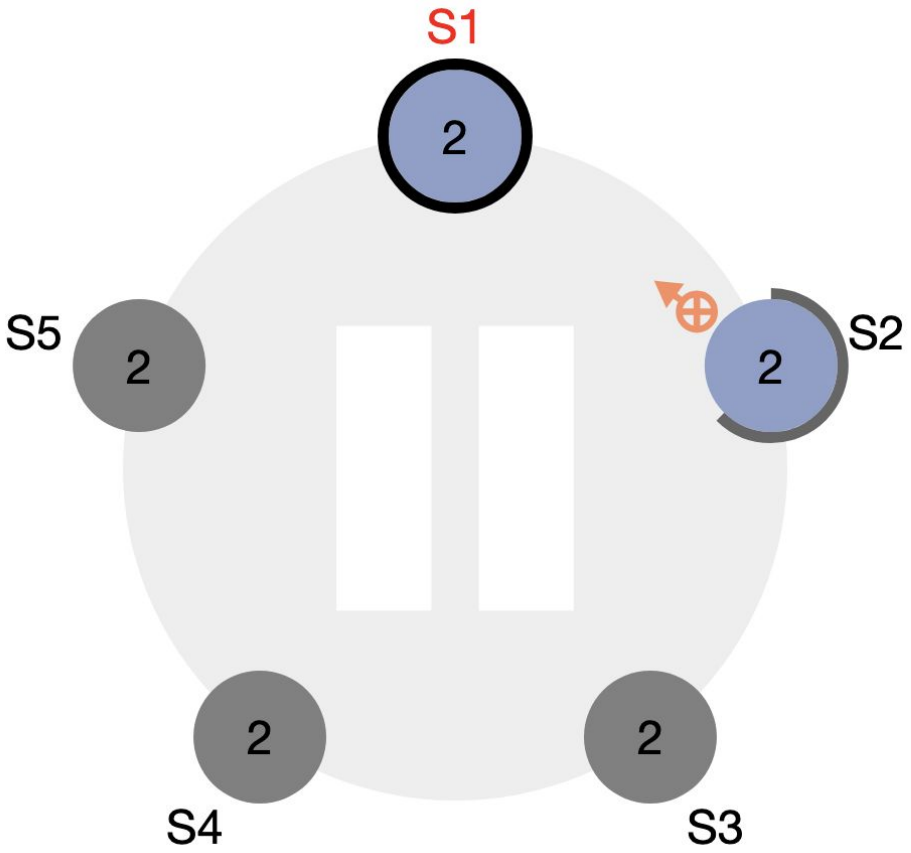


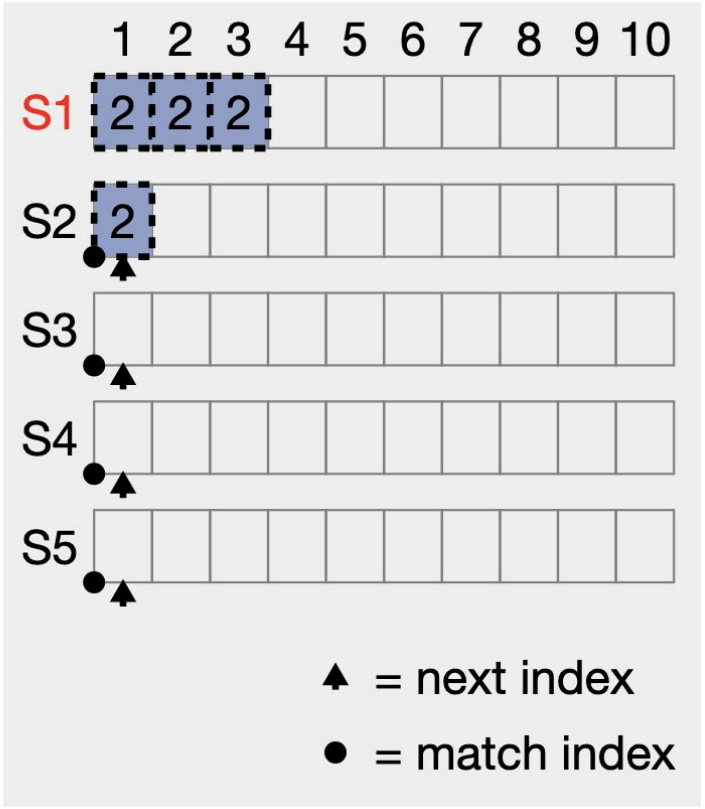
Log Replication: Example



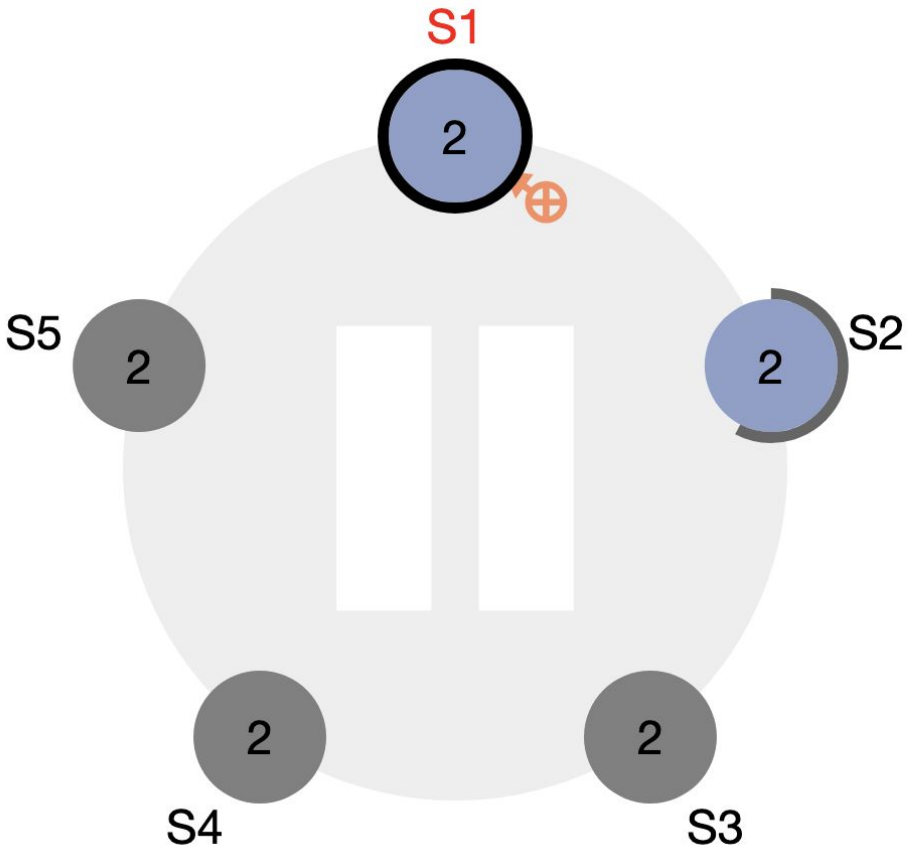


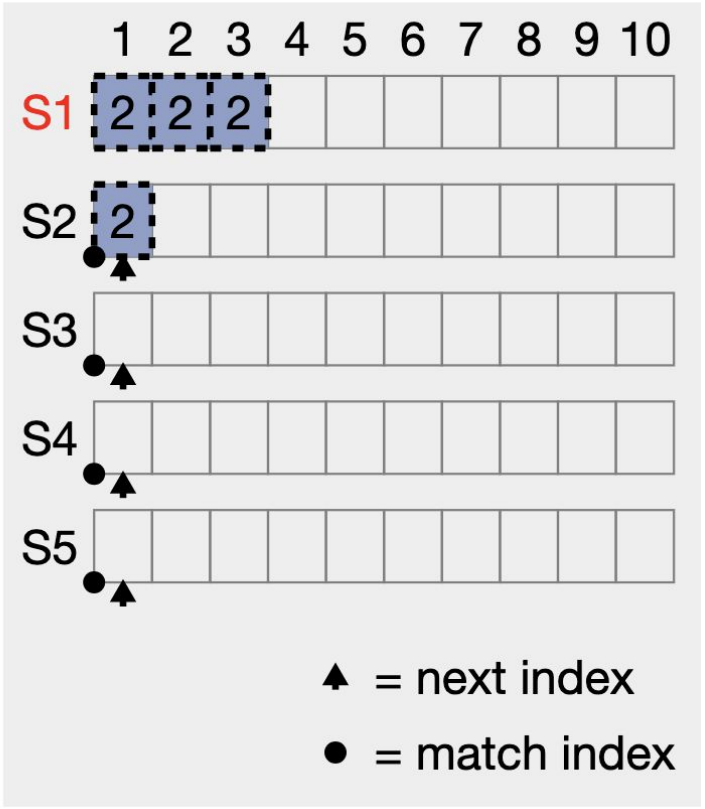
Log Replication: Example



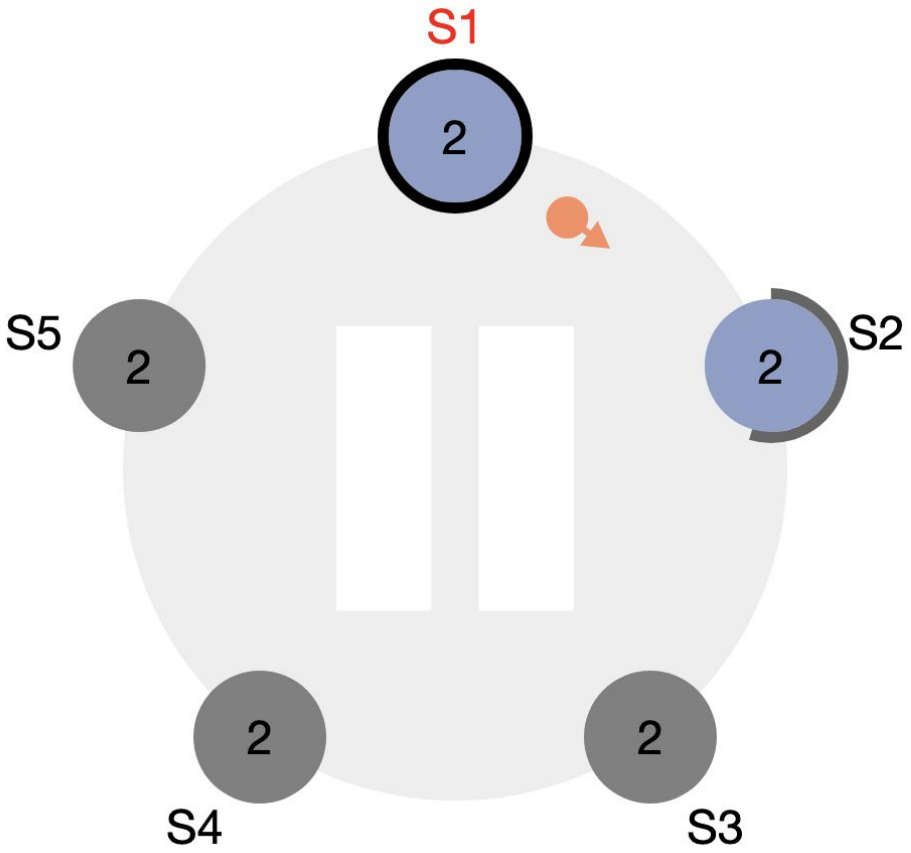


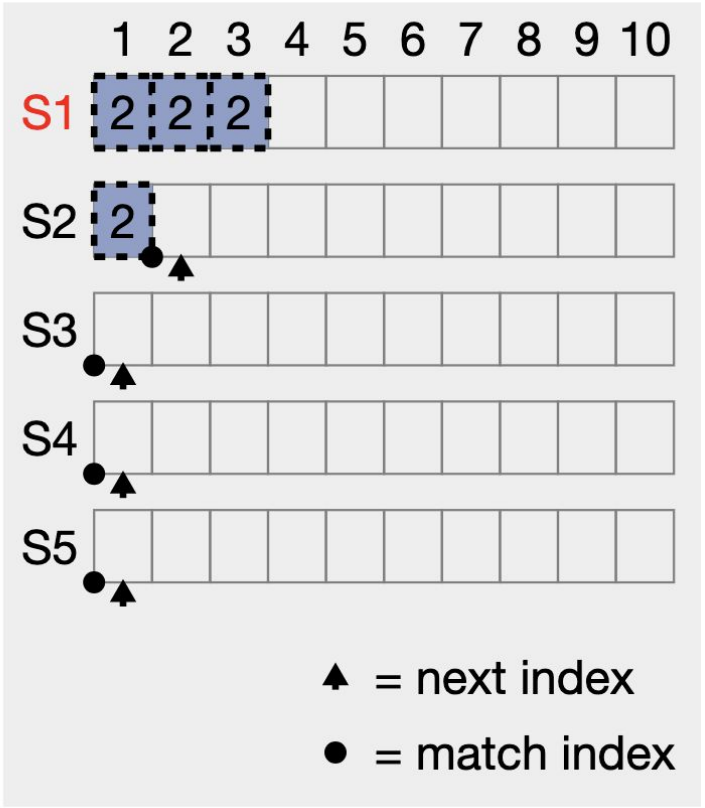
Log Replication: Example



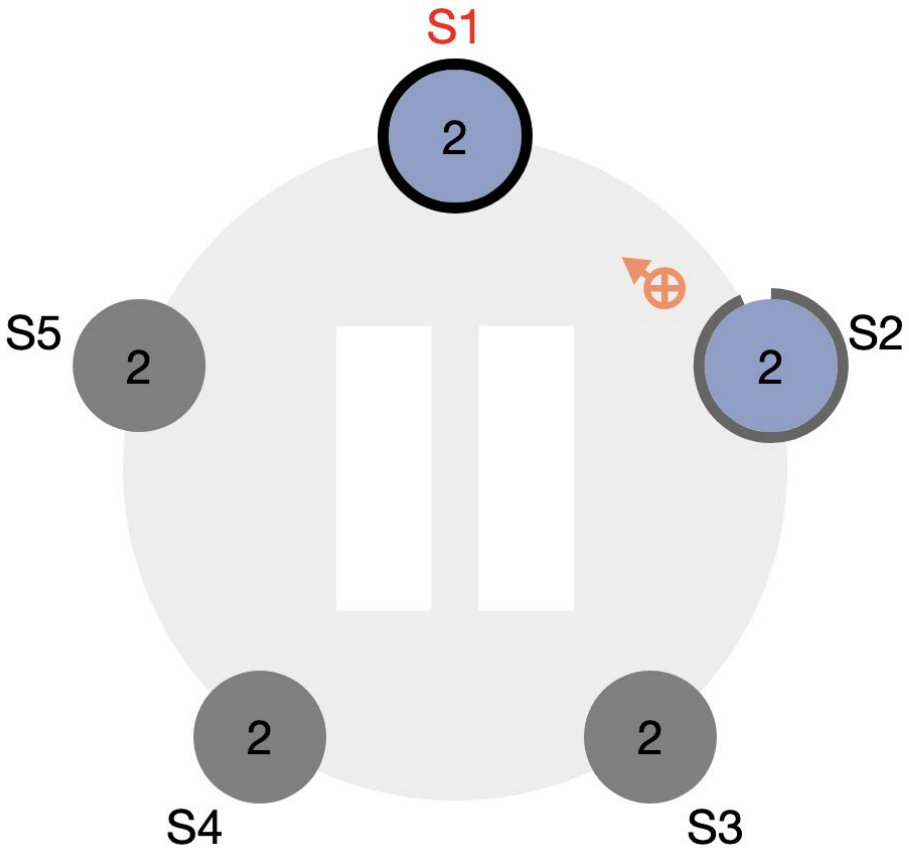


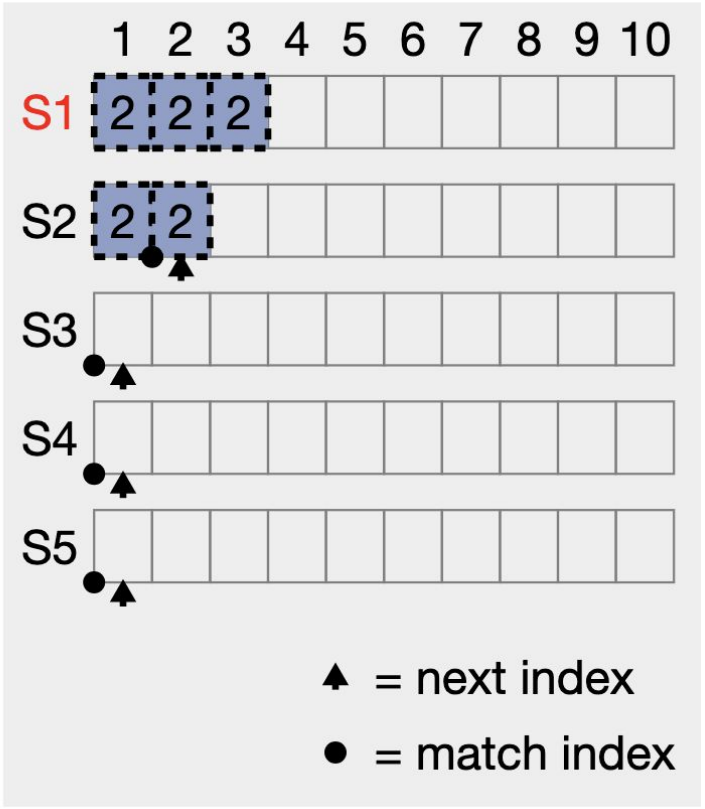
Log Replication: Example



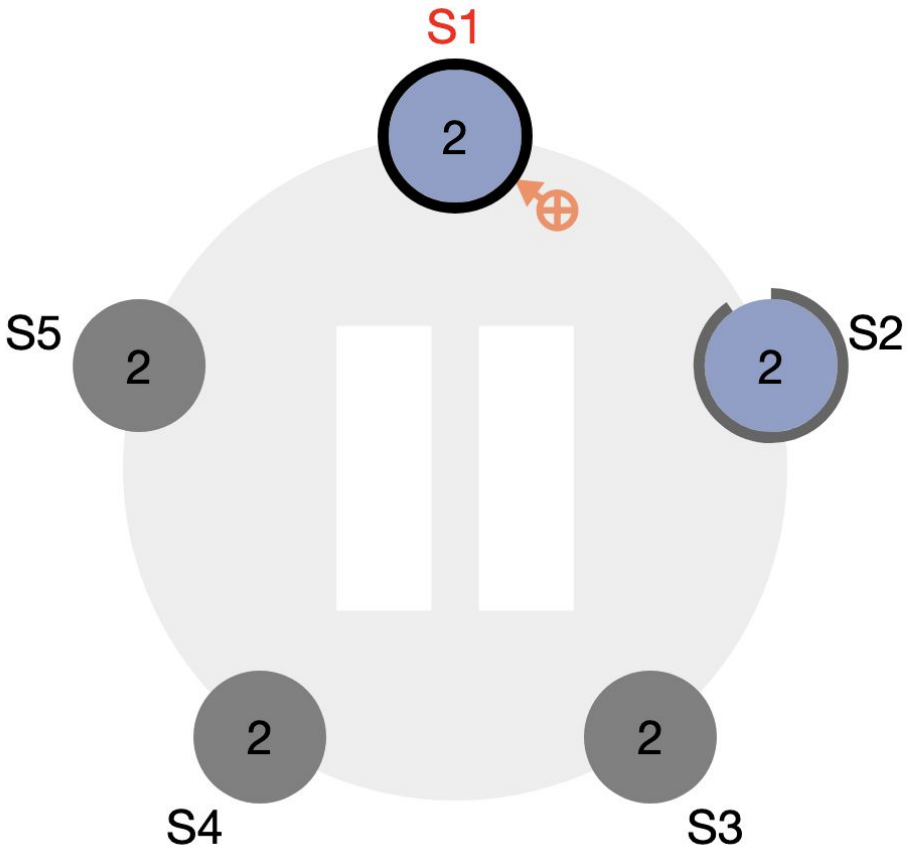


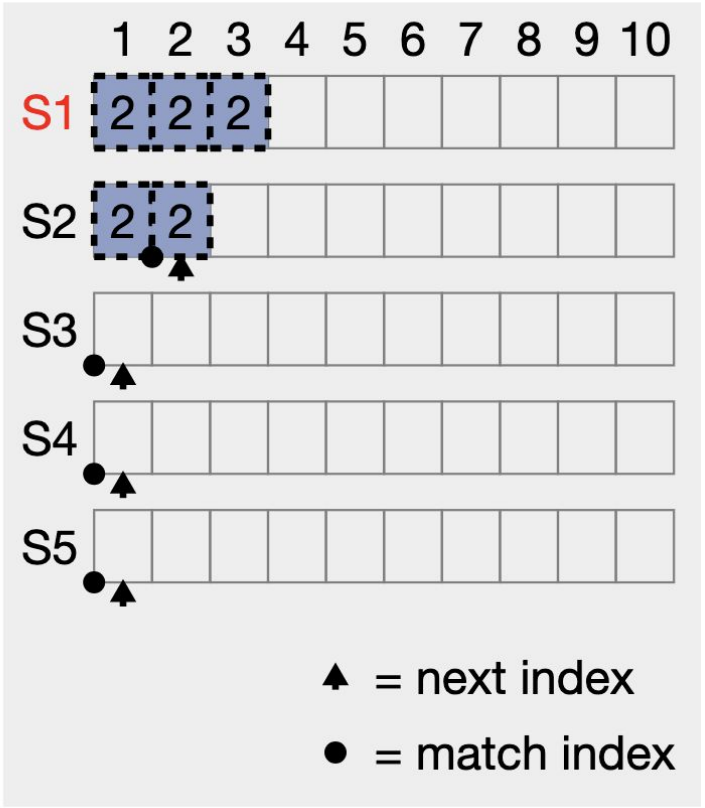
Log Replication: Example



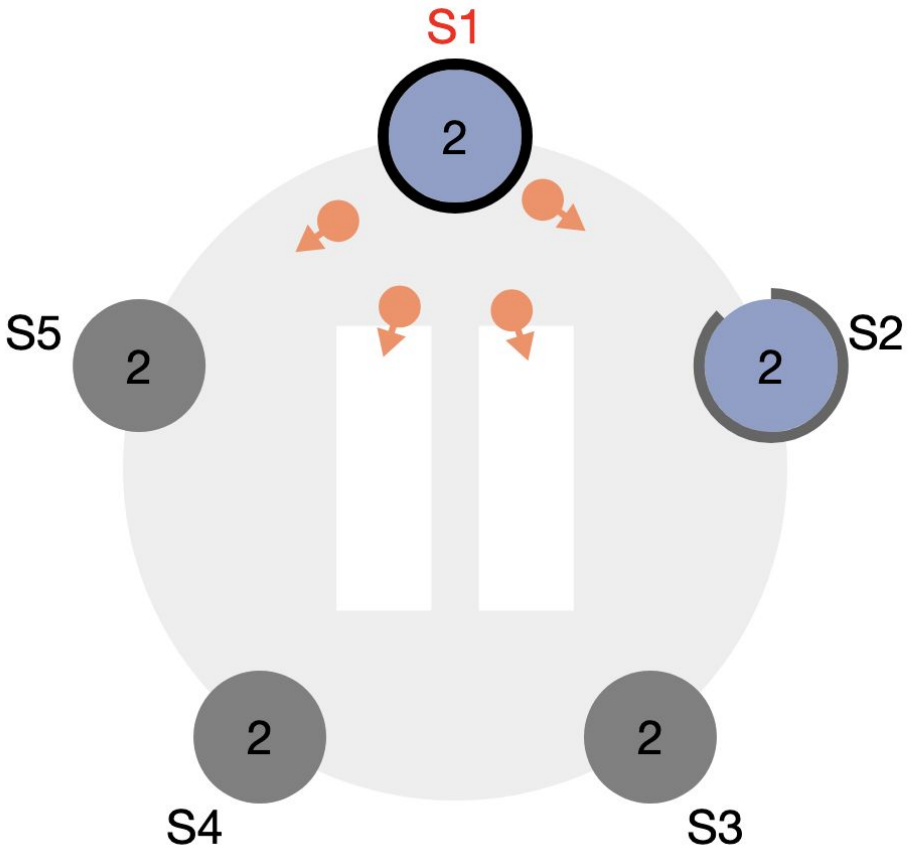


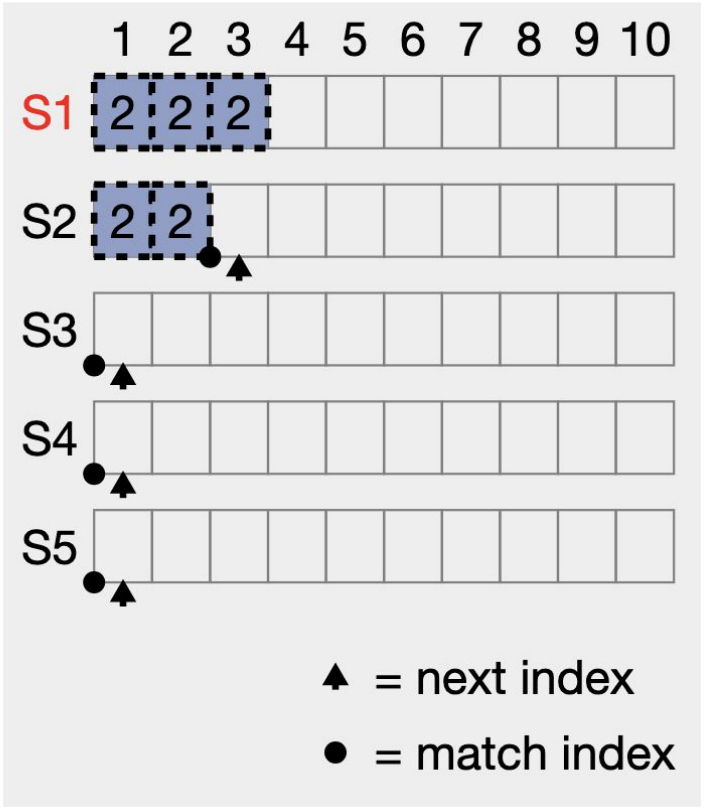
Log Replication: Example



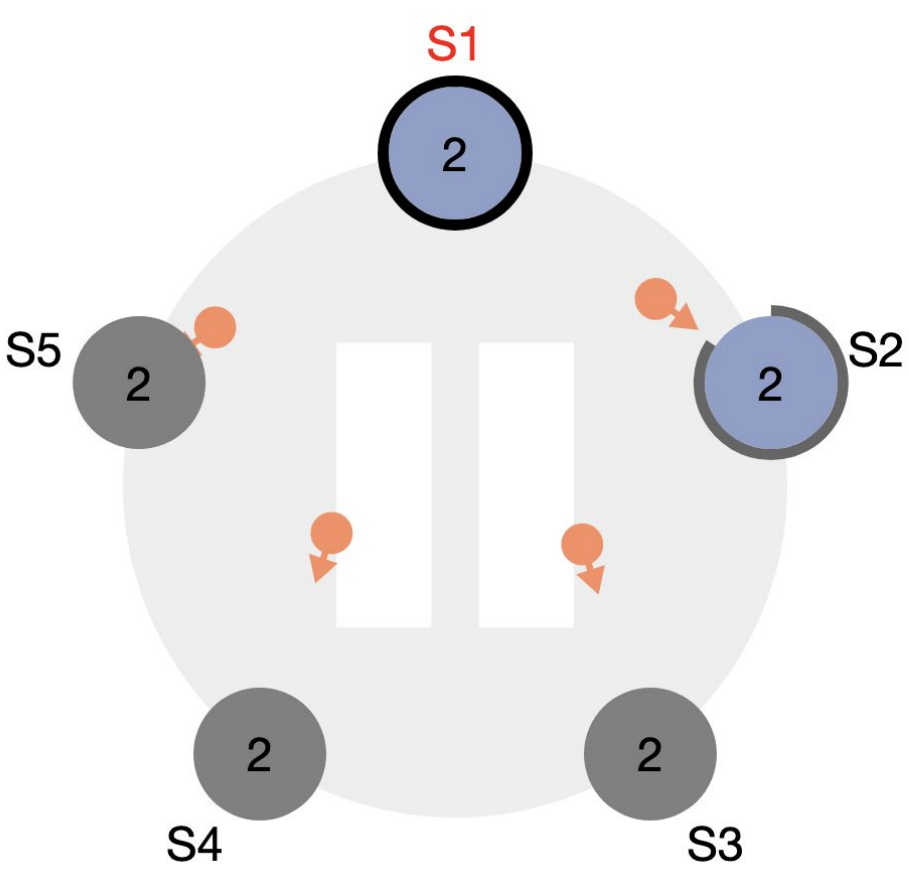


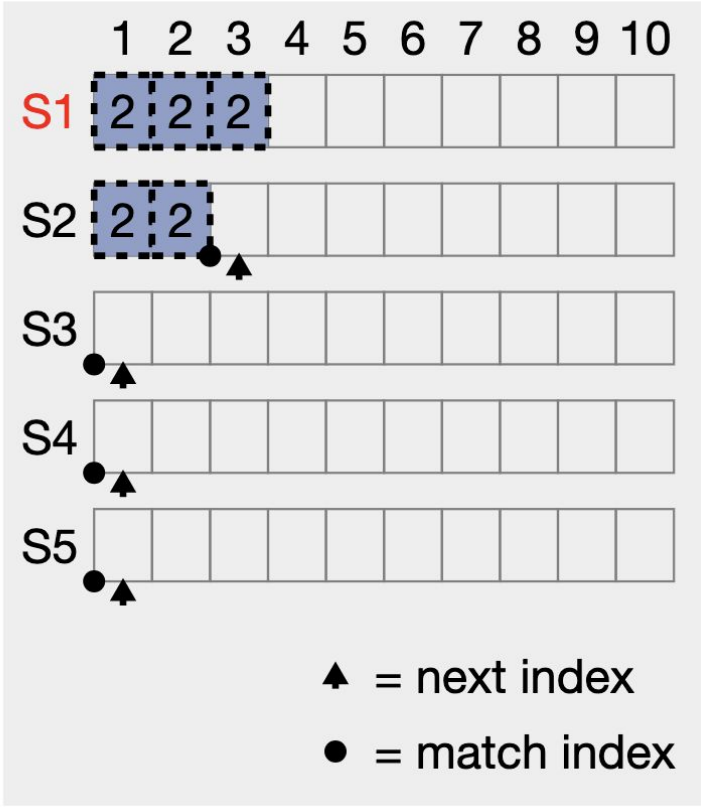
Log Replication: Example



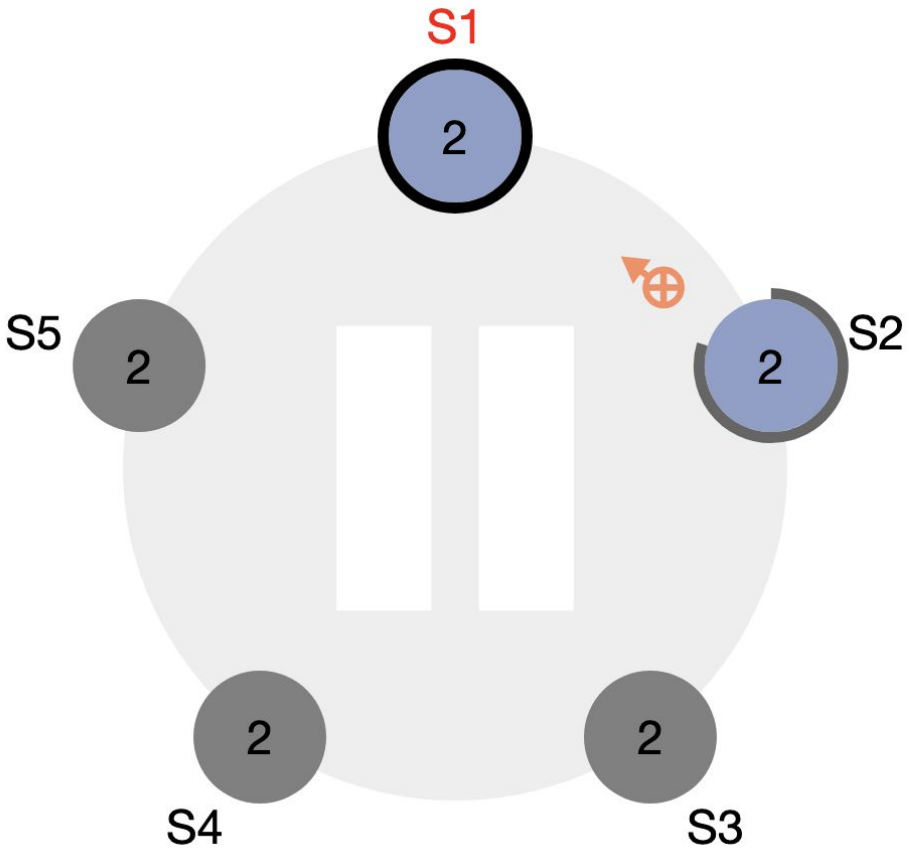


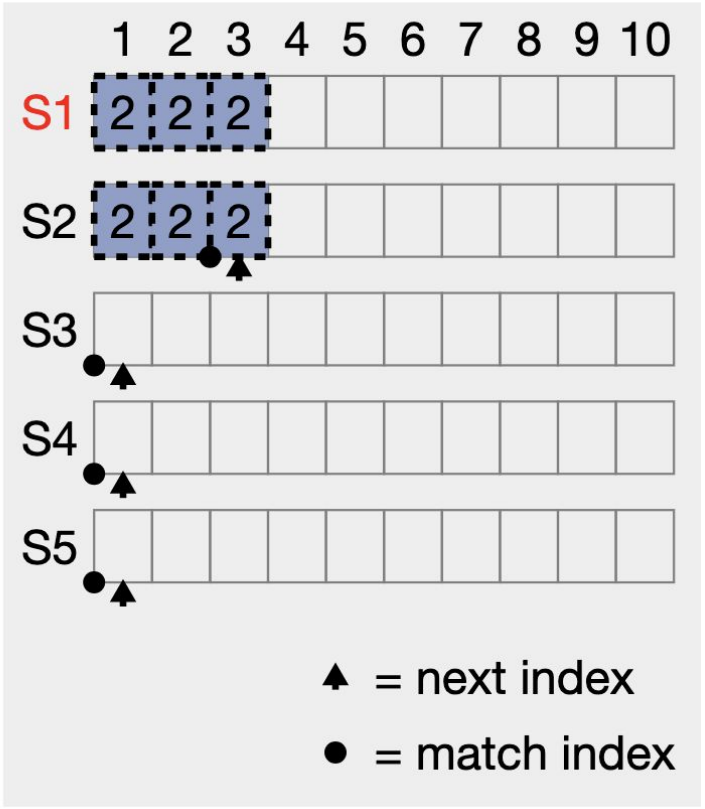
Log Replication: Example



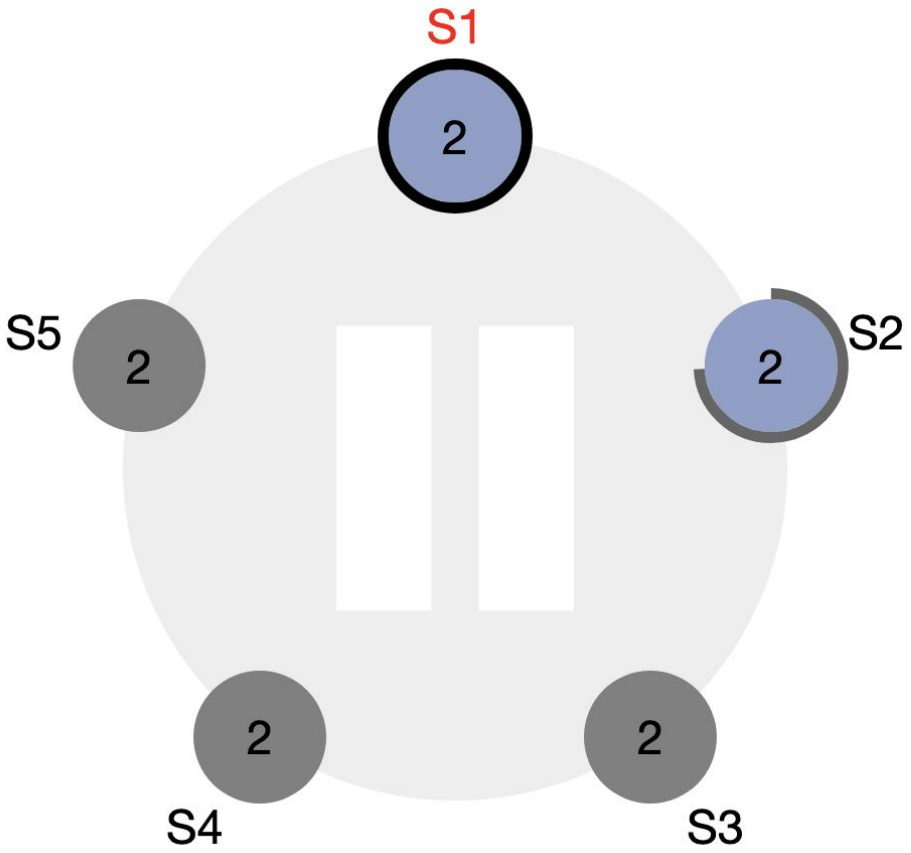


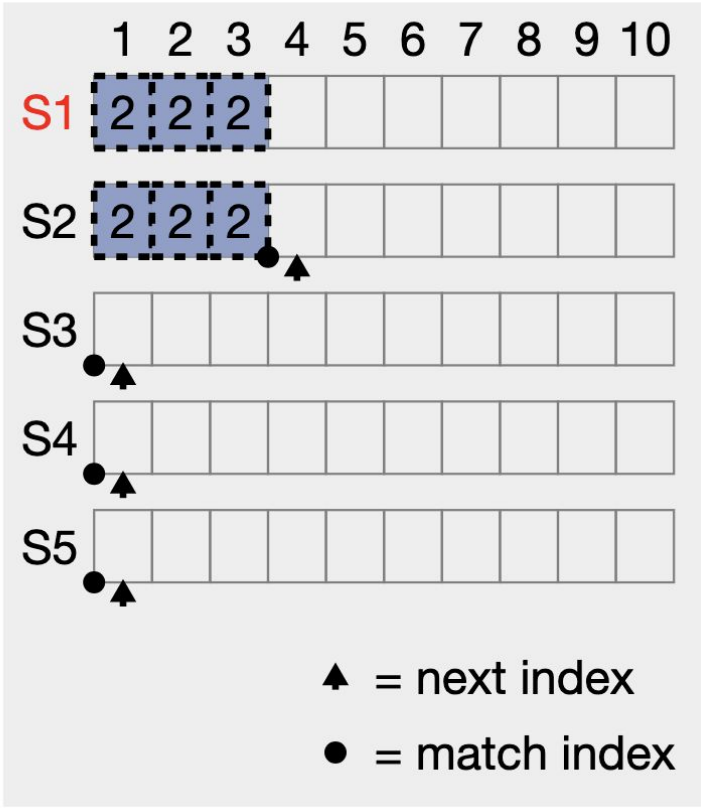
Log Replication: Example



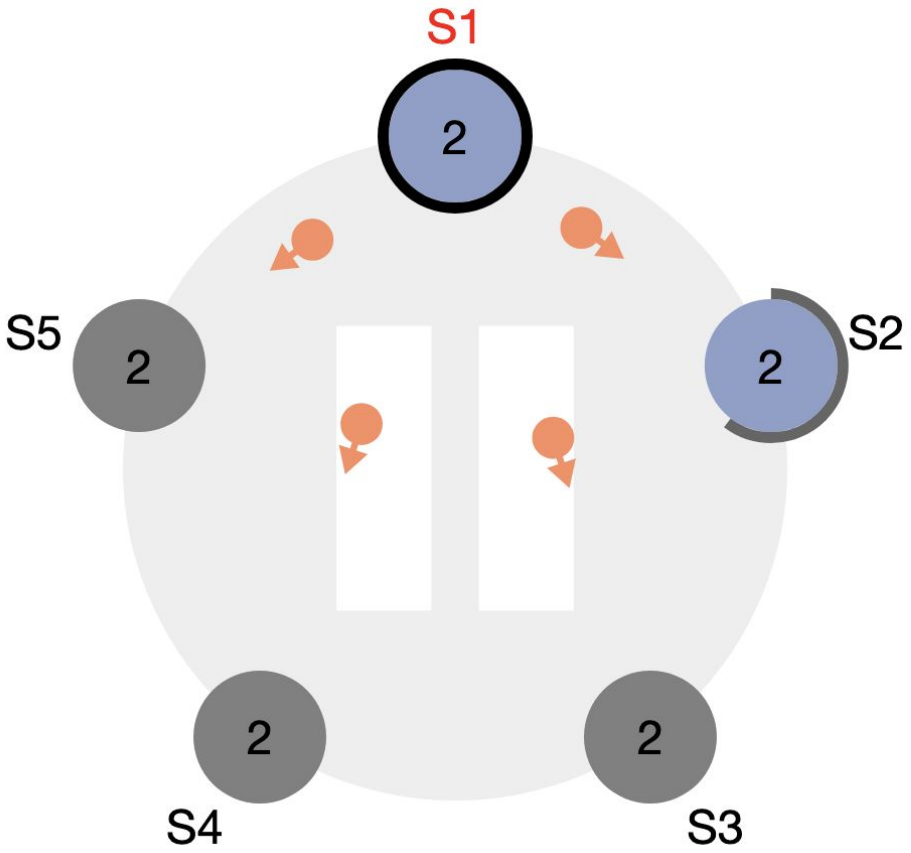


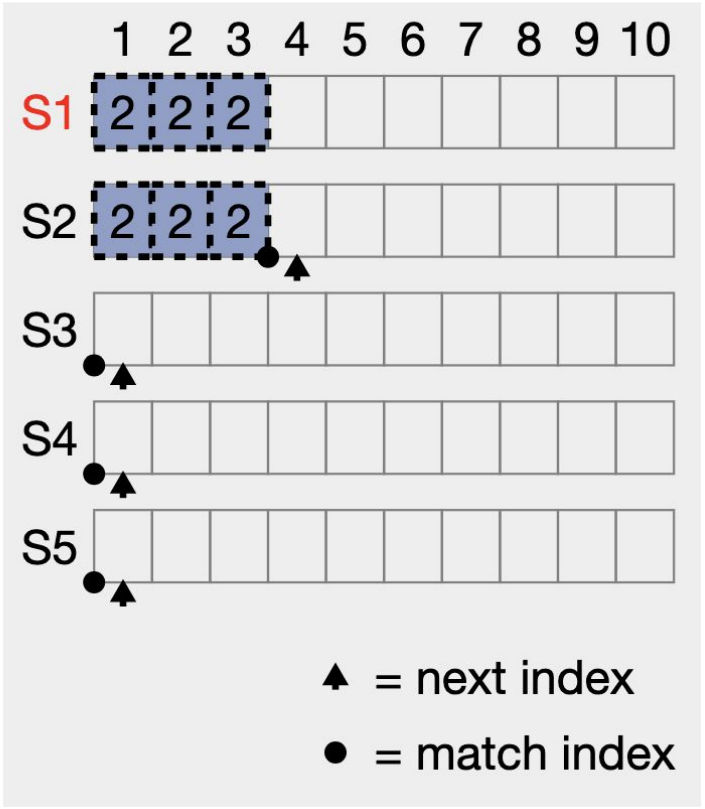
Log Replication: Example



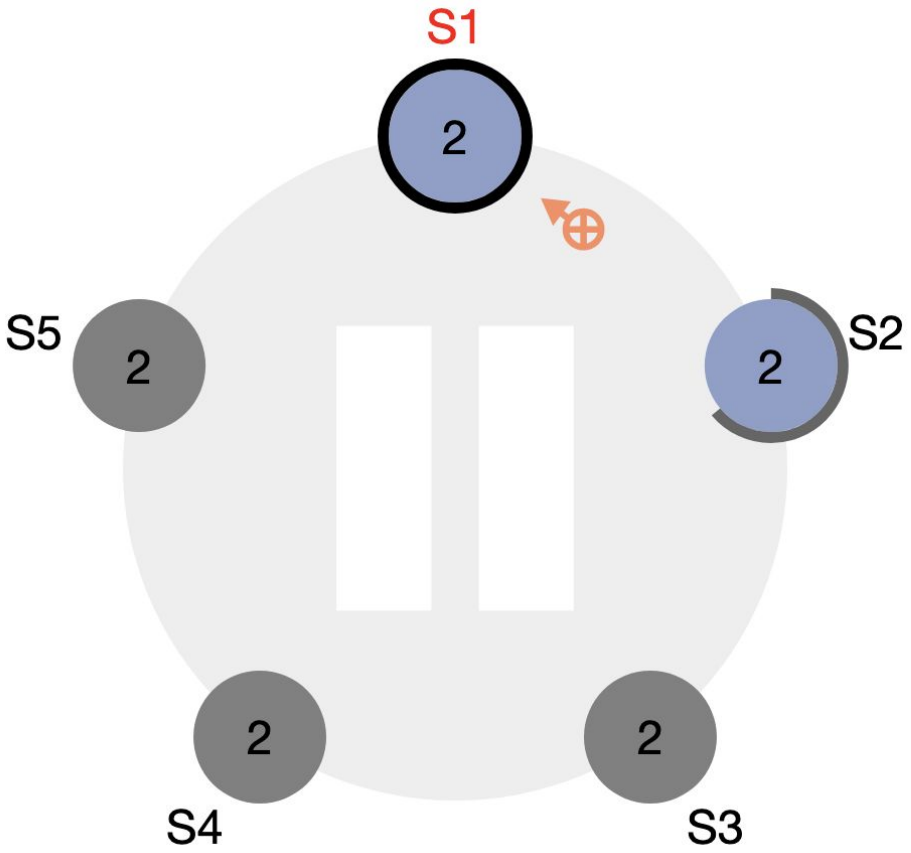


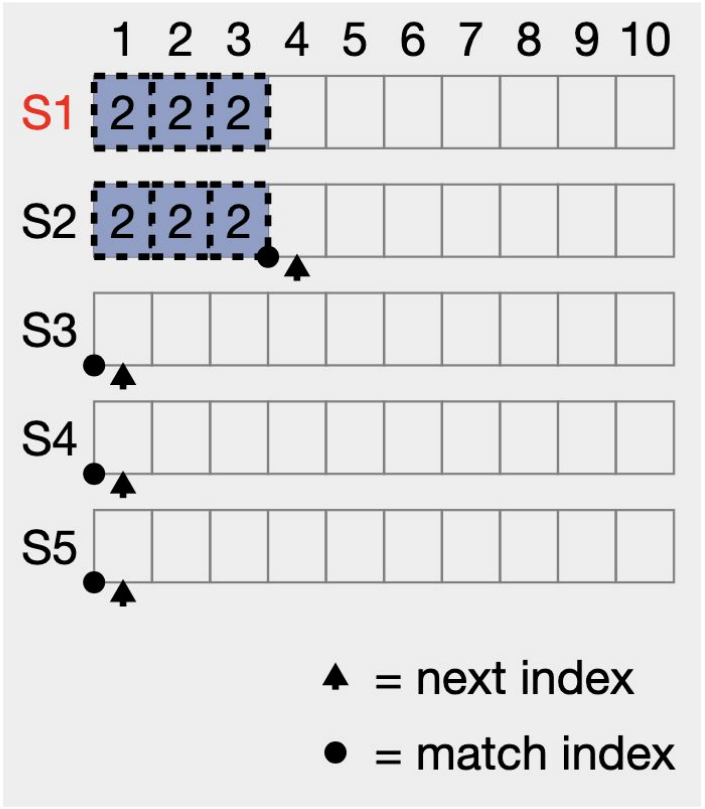
Log Replication: Example



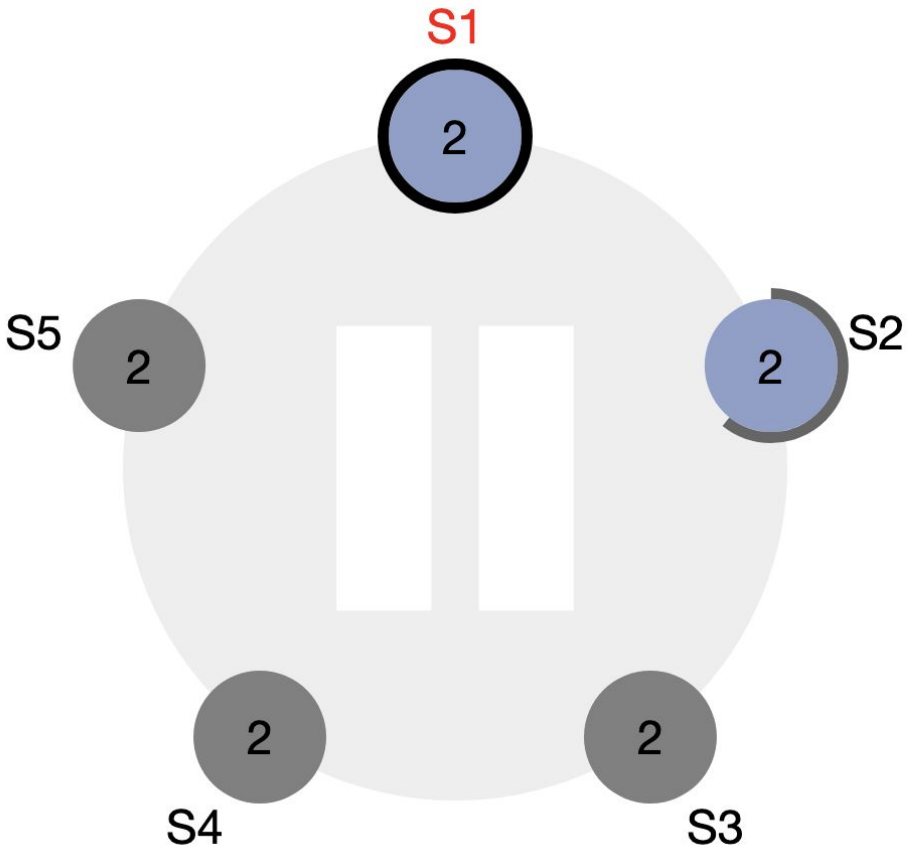


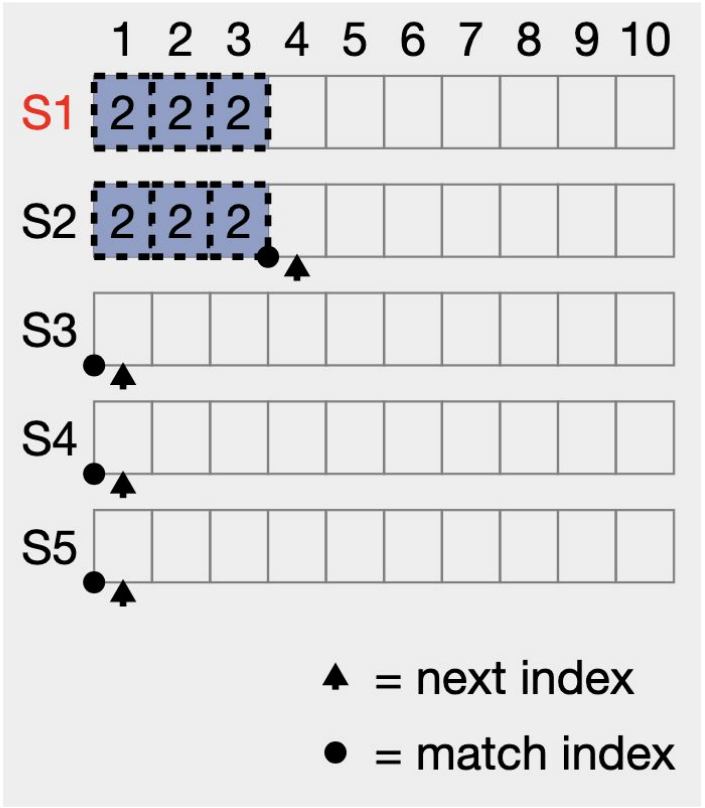
Log Replication: Example



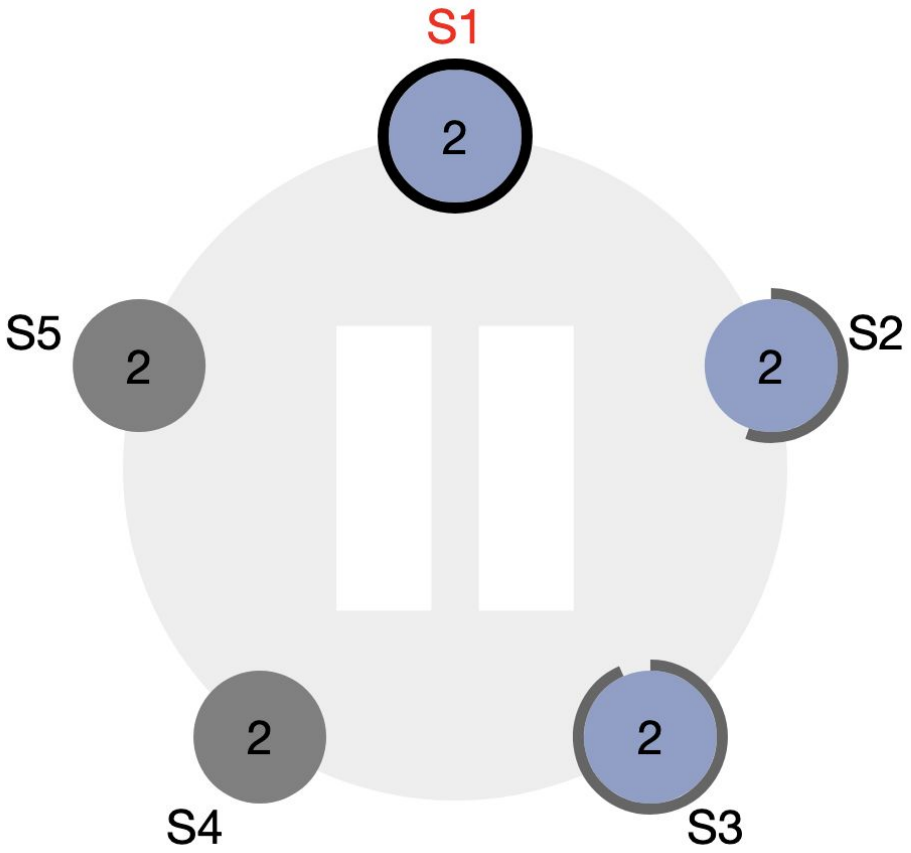


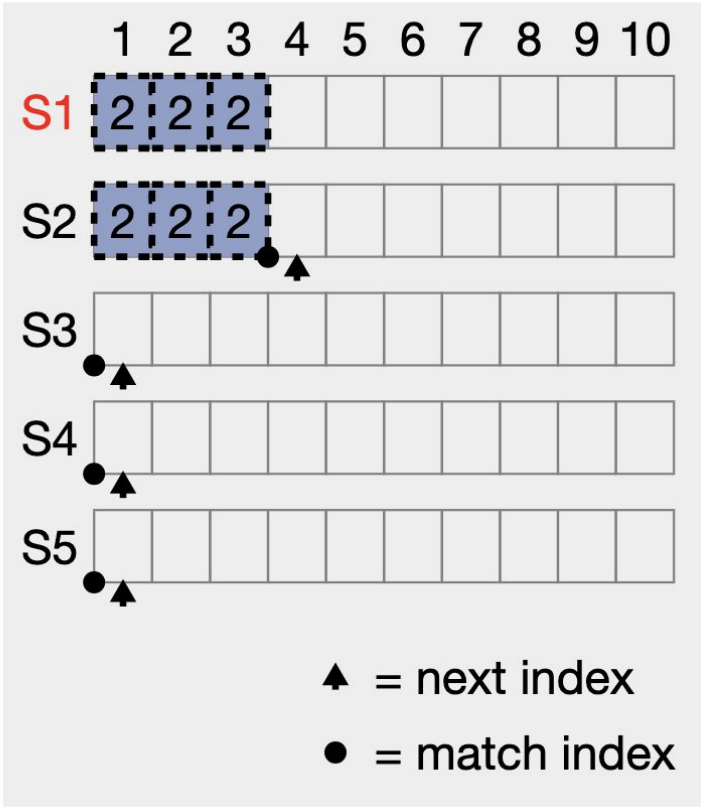
Log Replication: Example



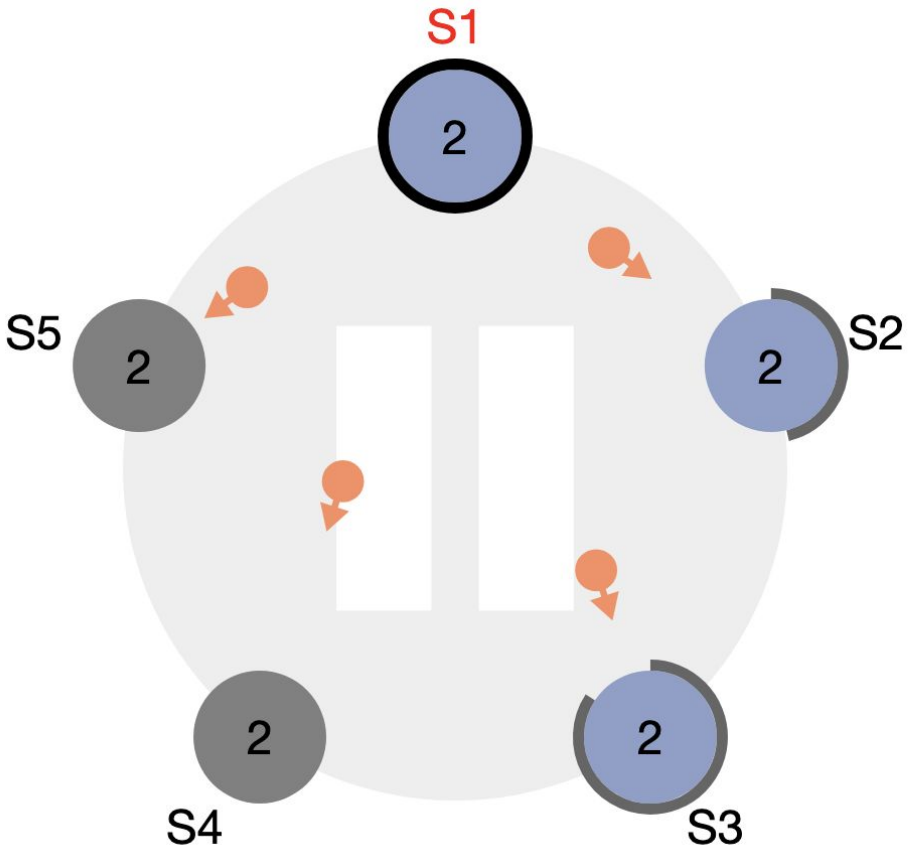


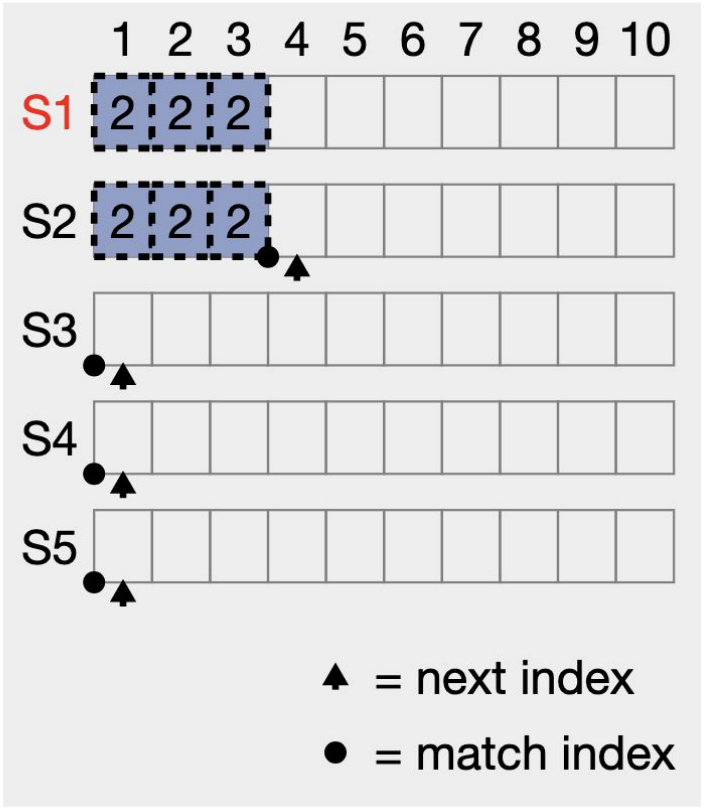
Log Replication: Example



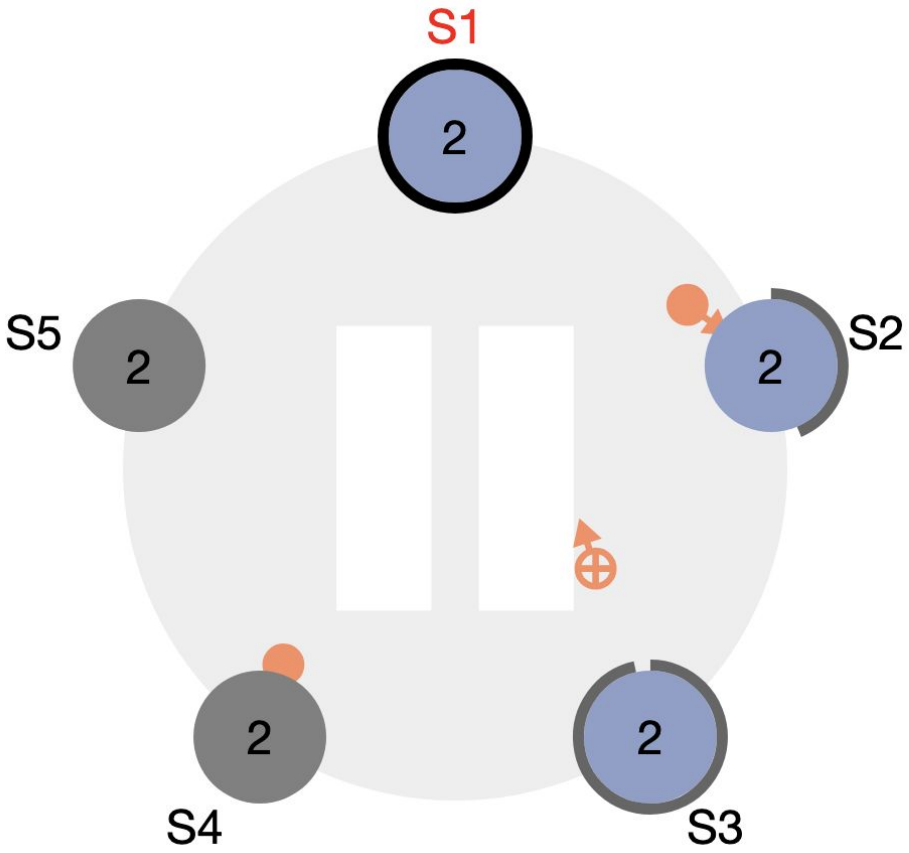


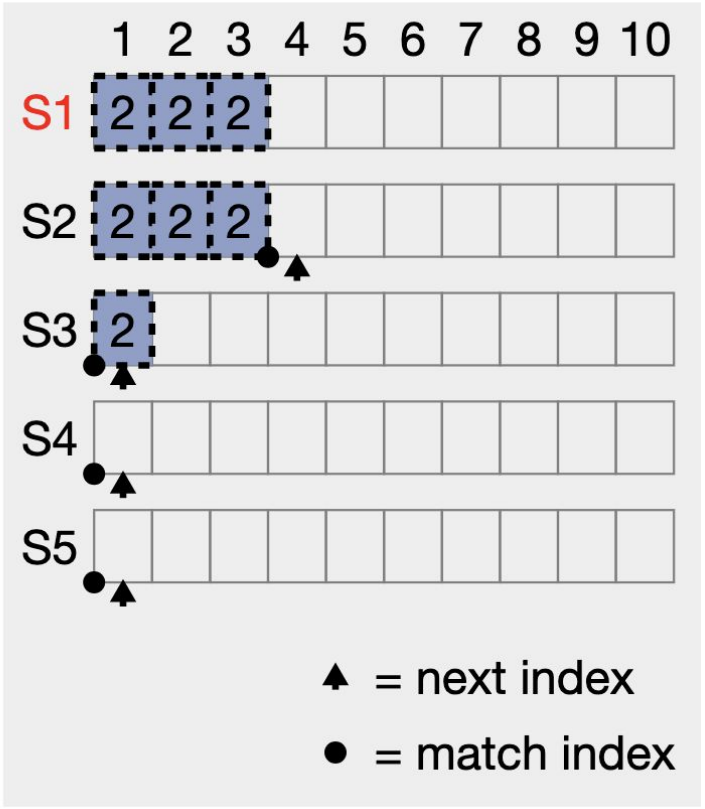
Log Replication: Example



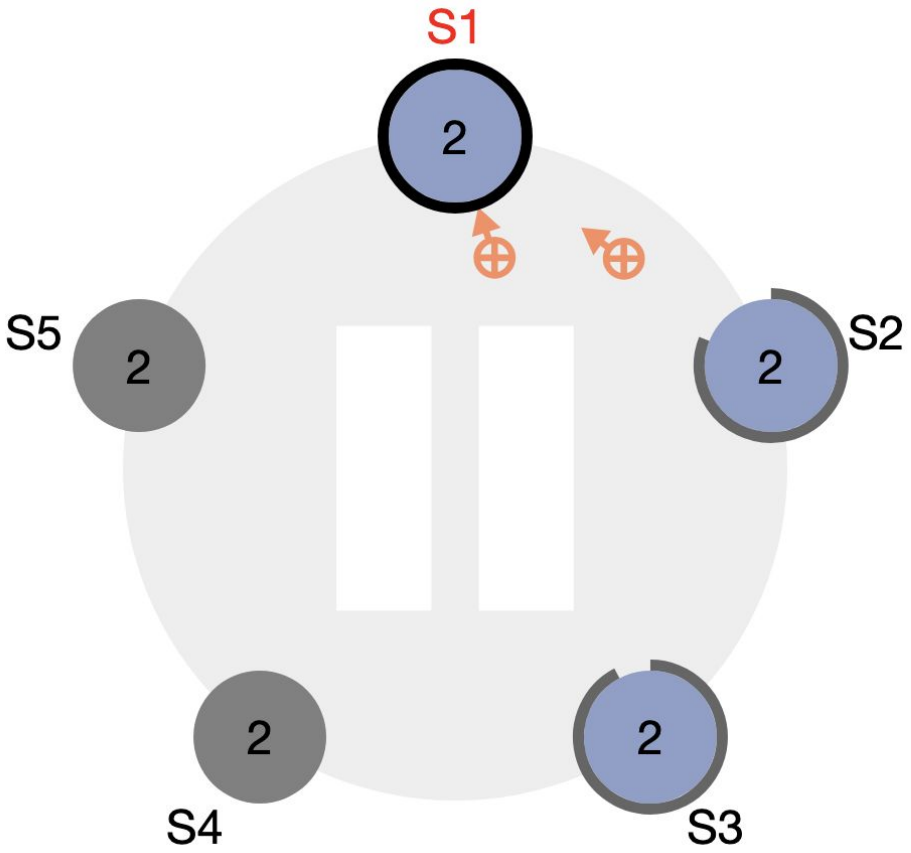


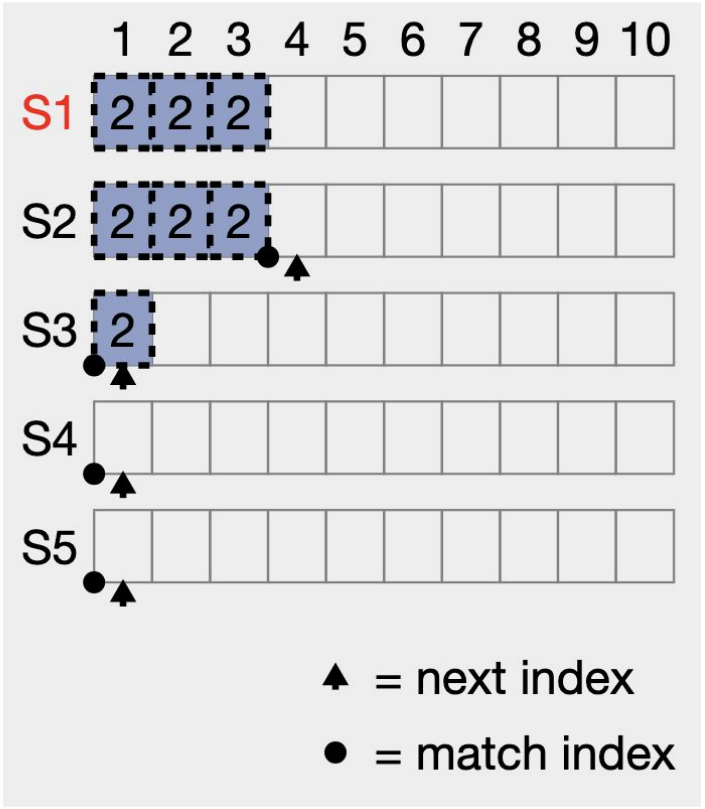
Log Replication: Example



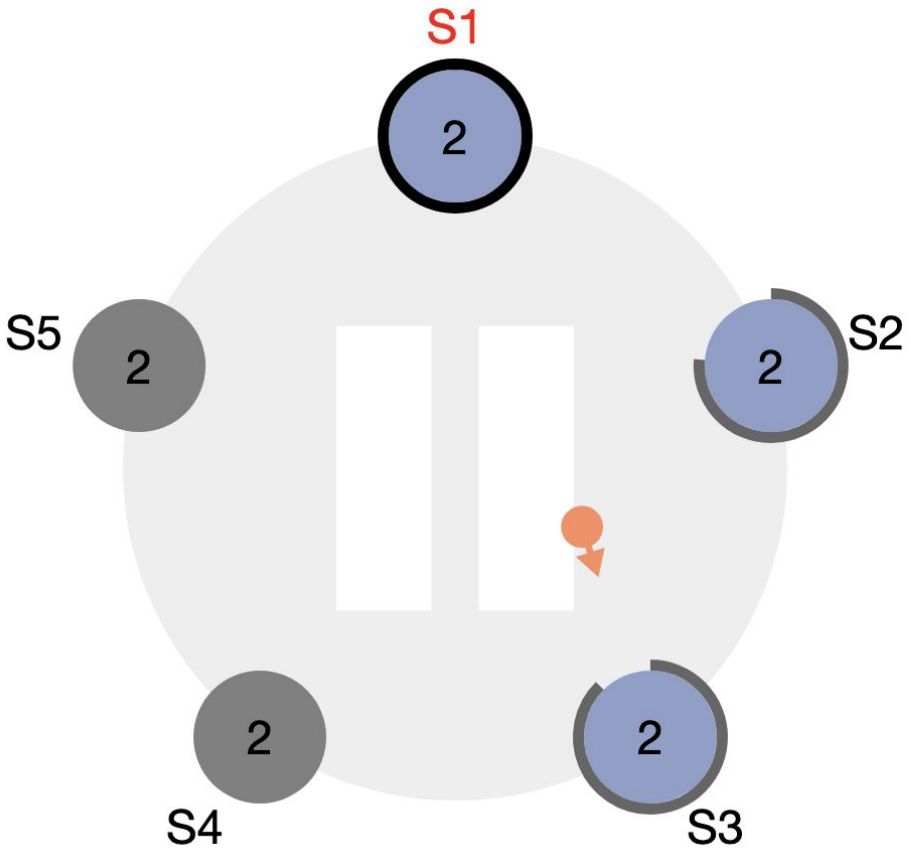


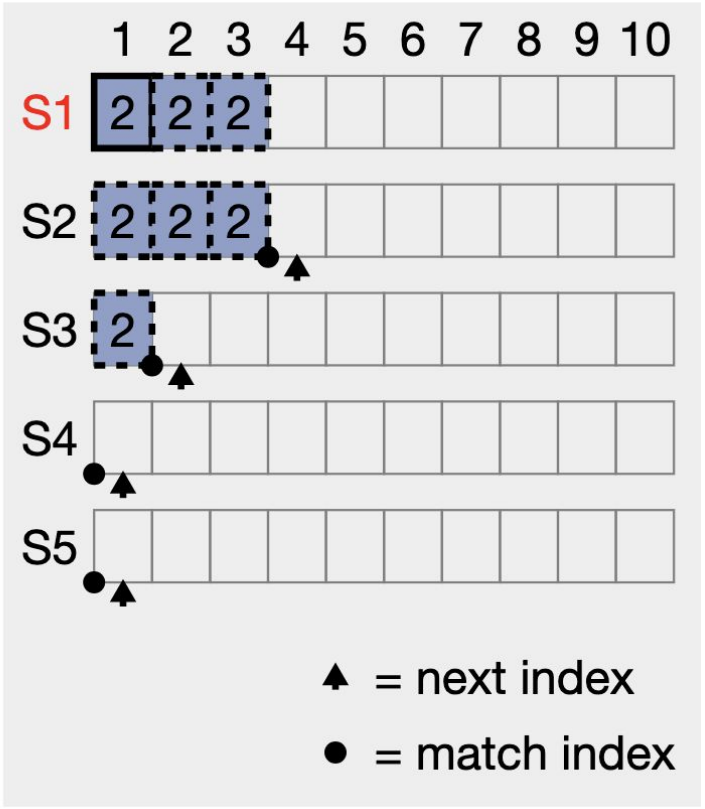
Log Replication: Example



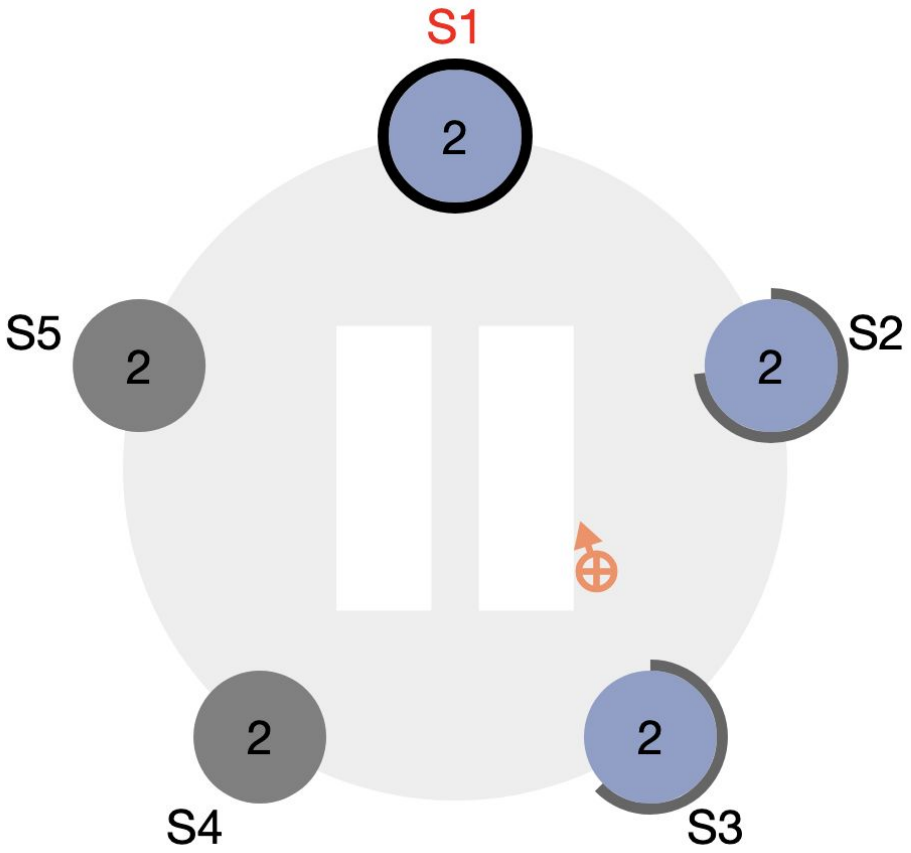


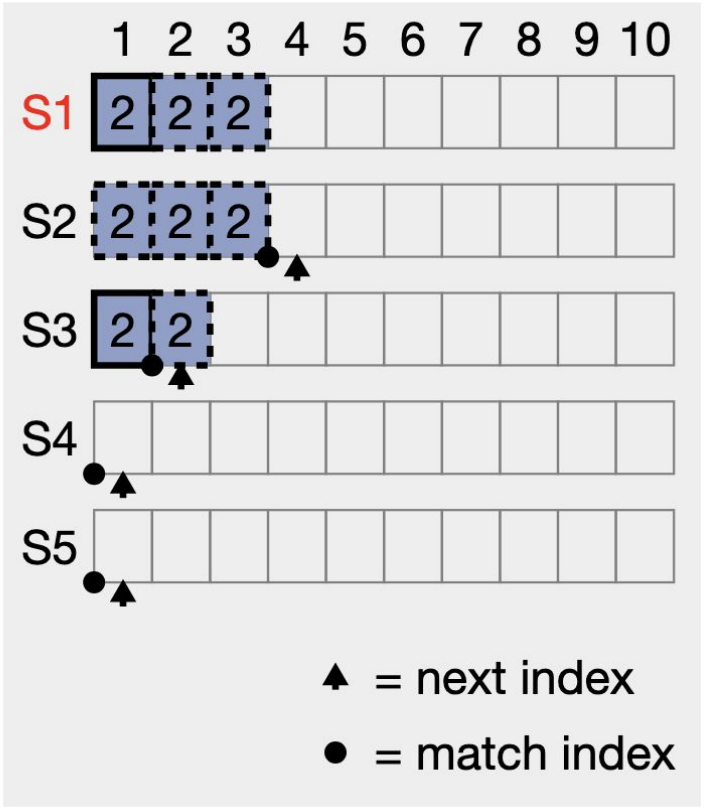
Log Replication: Example



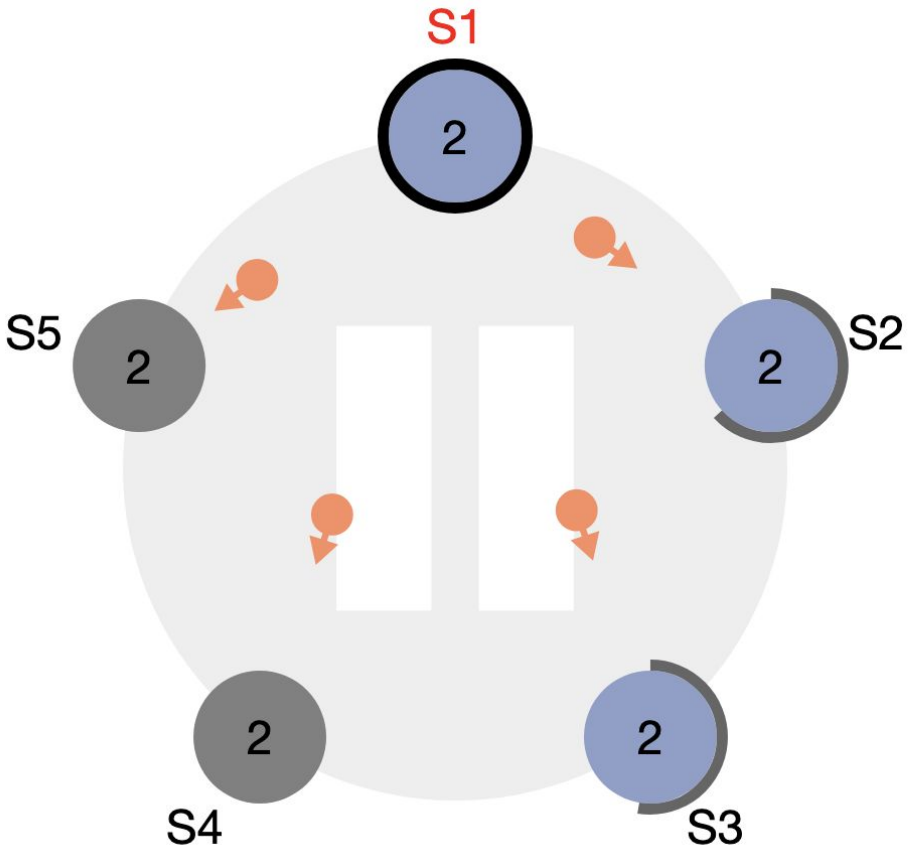


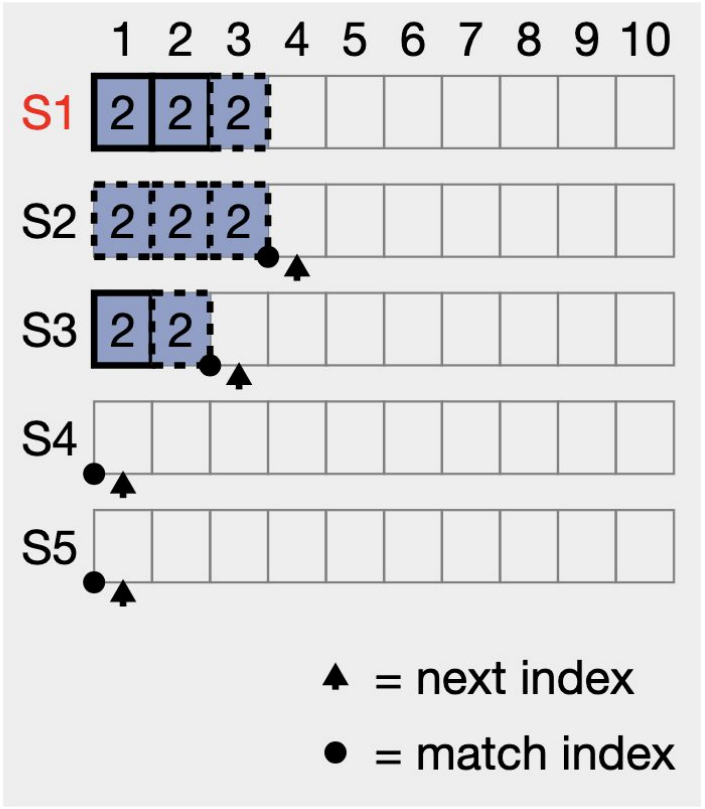
Log Replication: Example



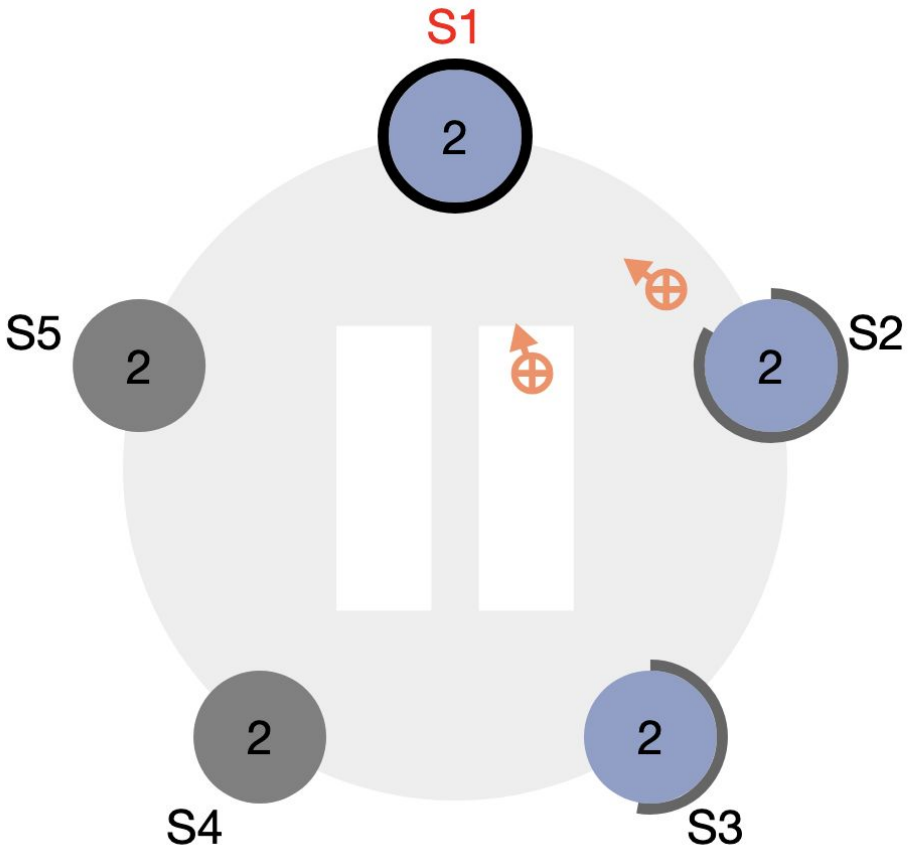


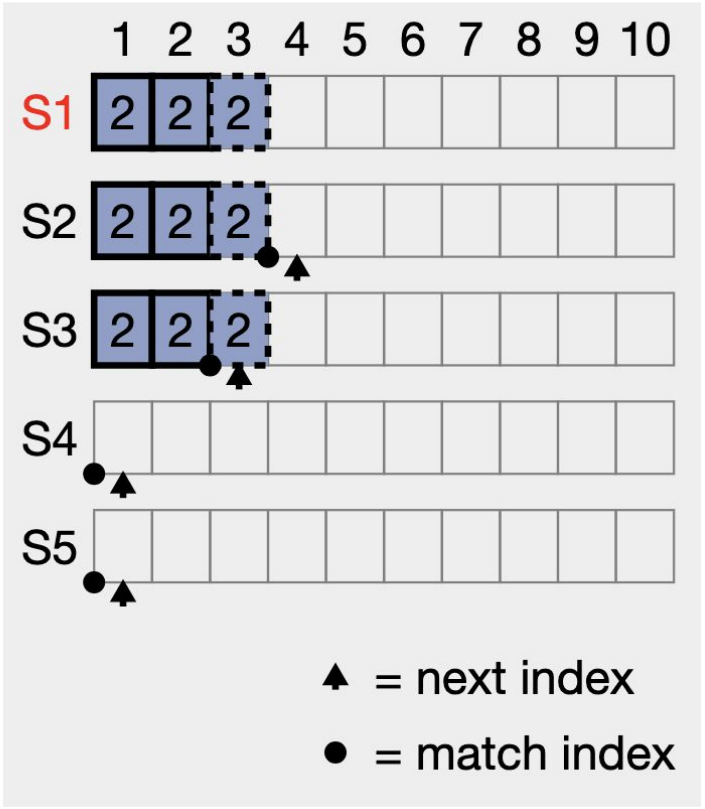
Log Replication: Example



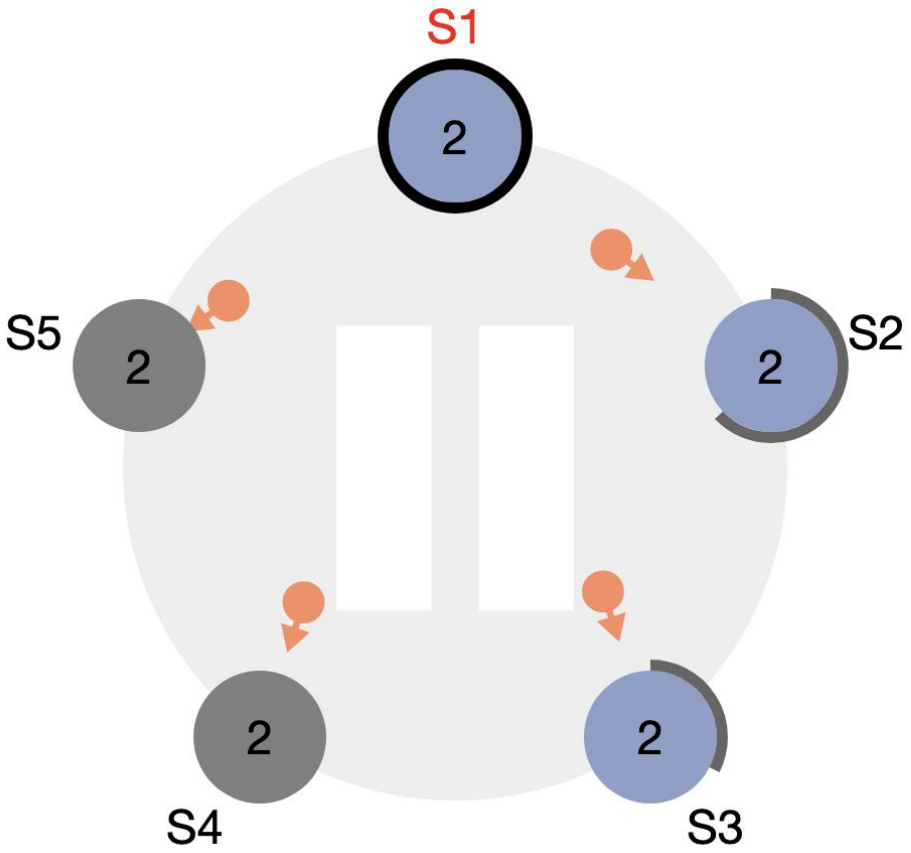


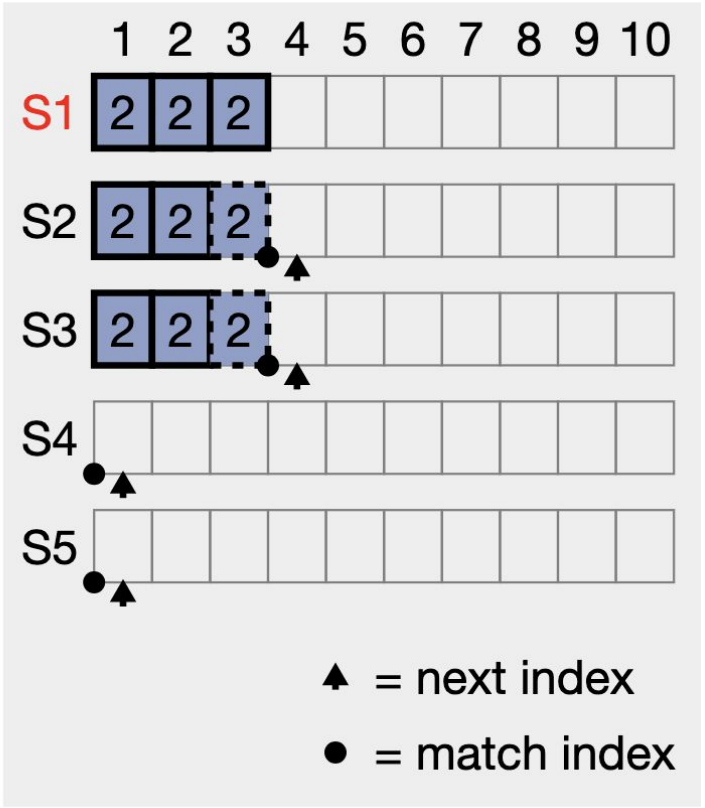
Log Replication: Example



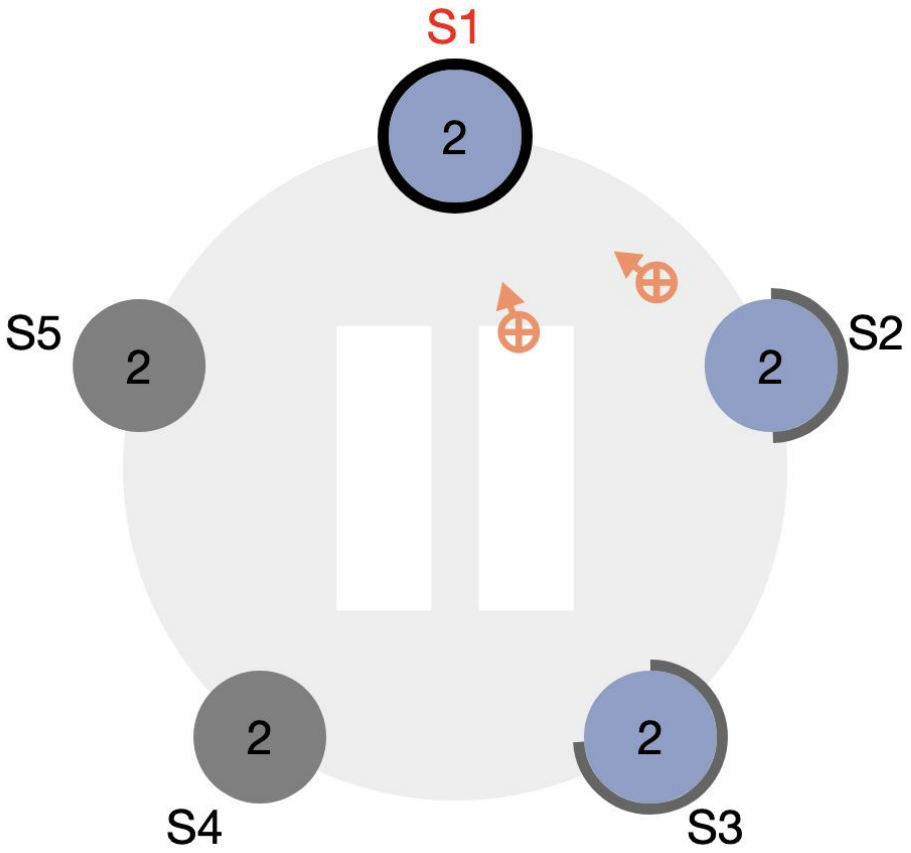


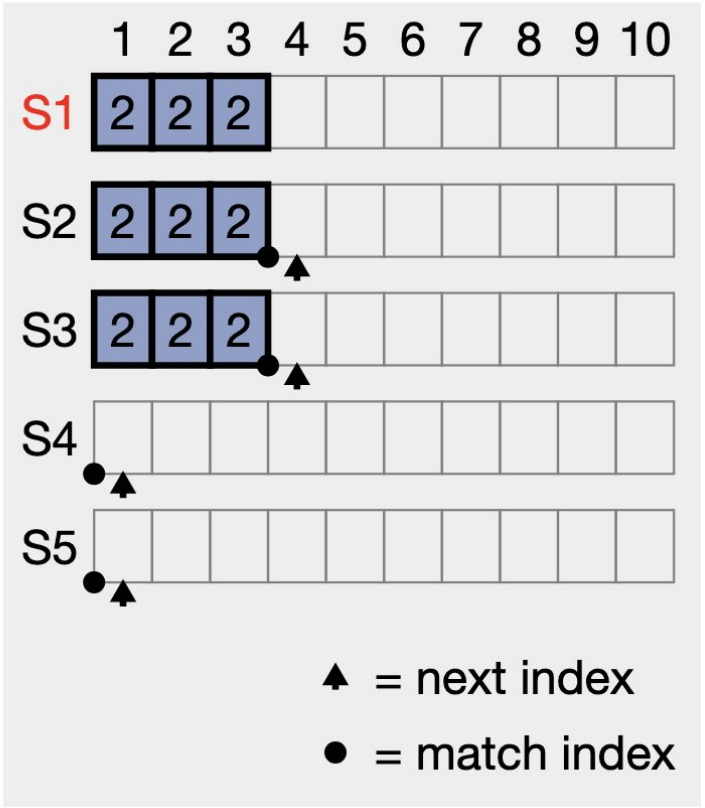
Log Replication: Example



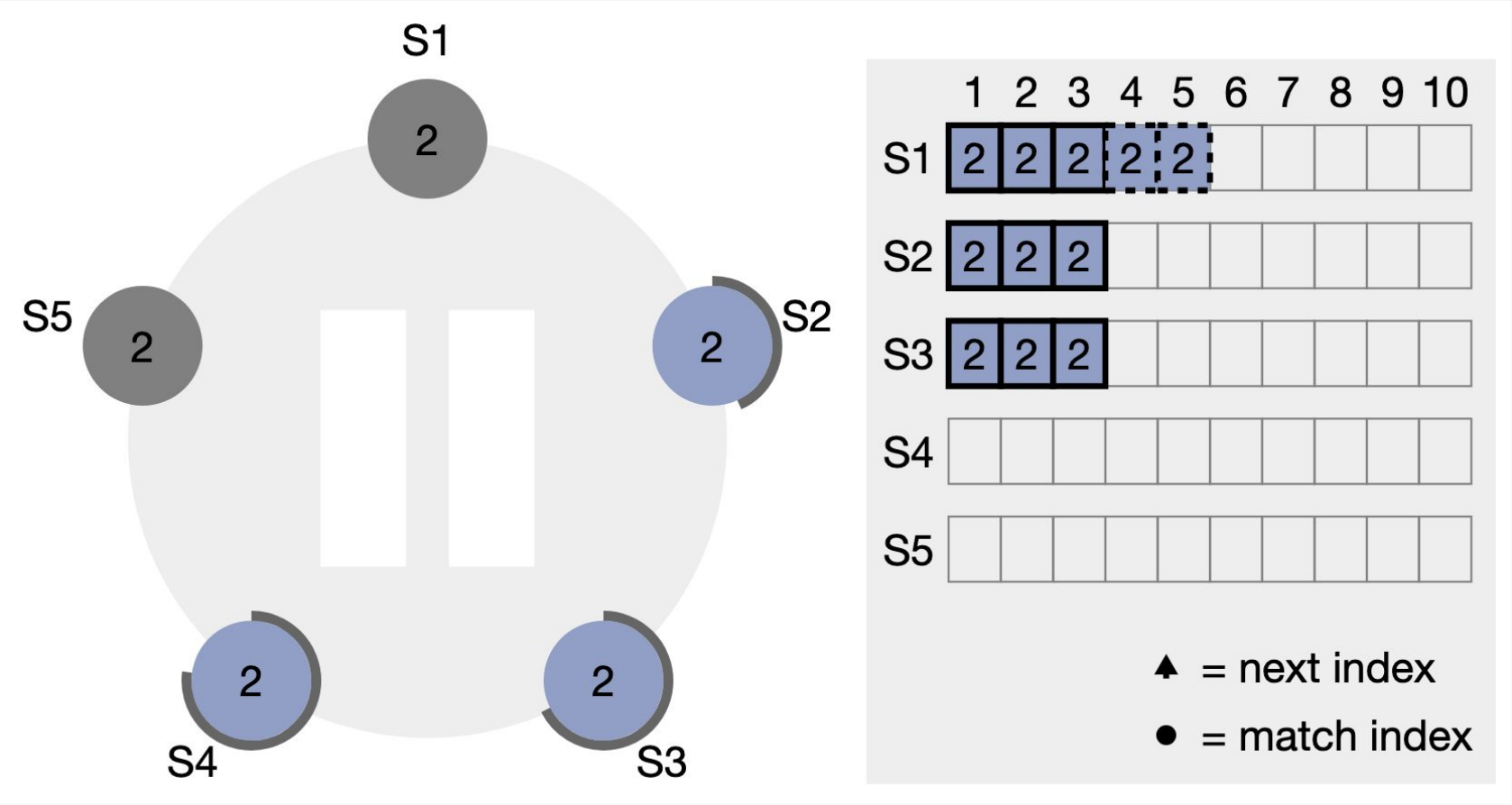


Log Replication: Example

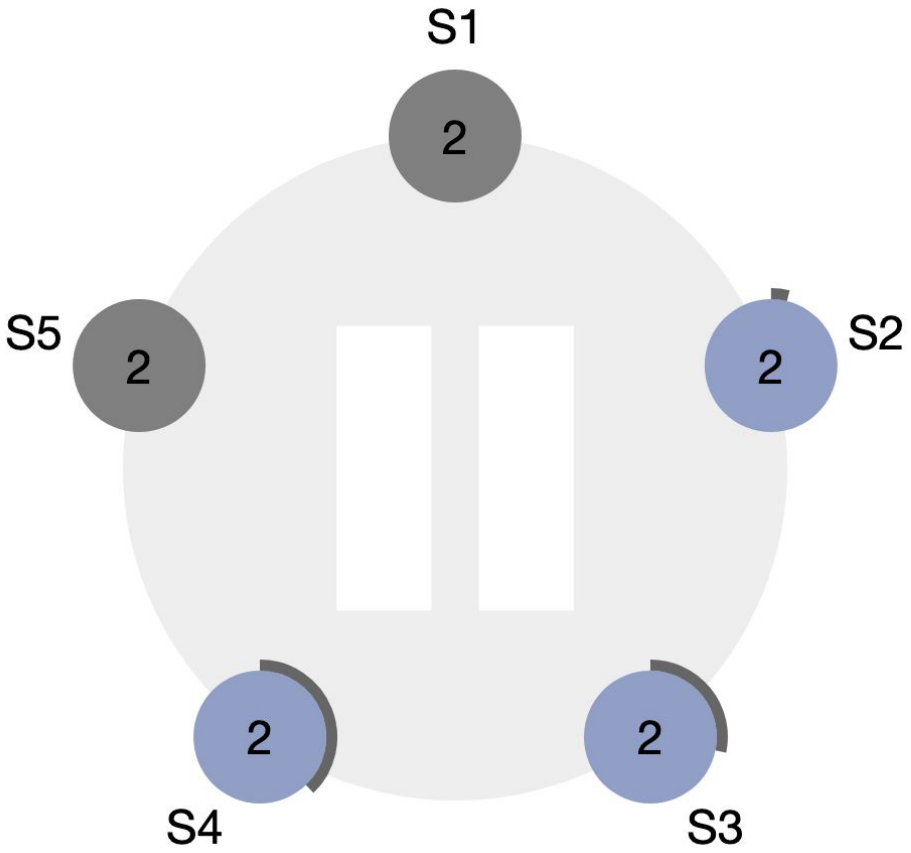


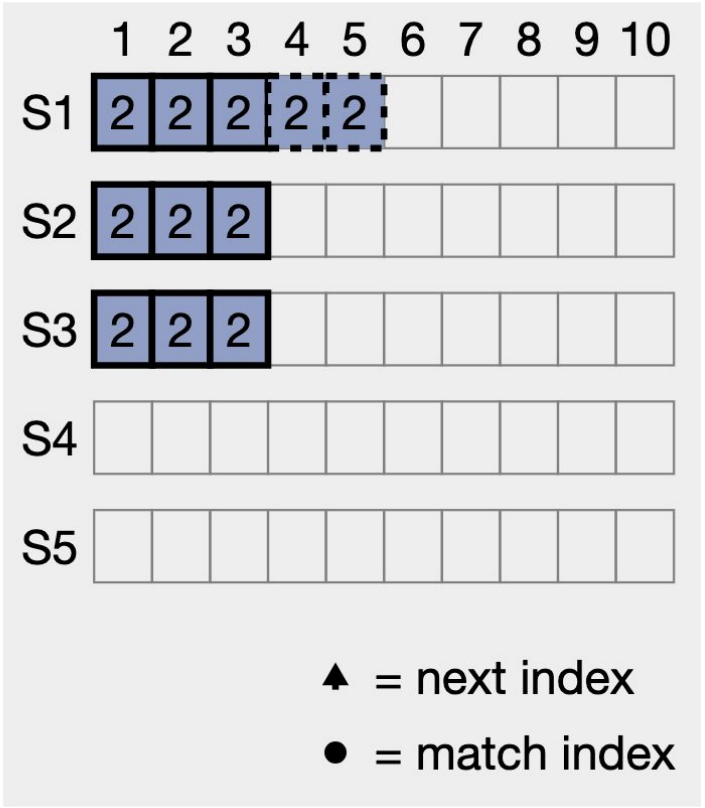


Log Replication: Handling log inconsistencies

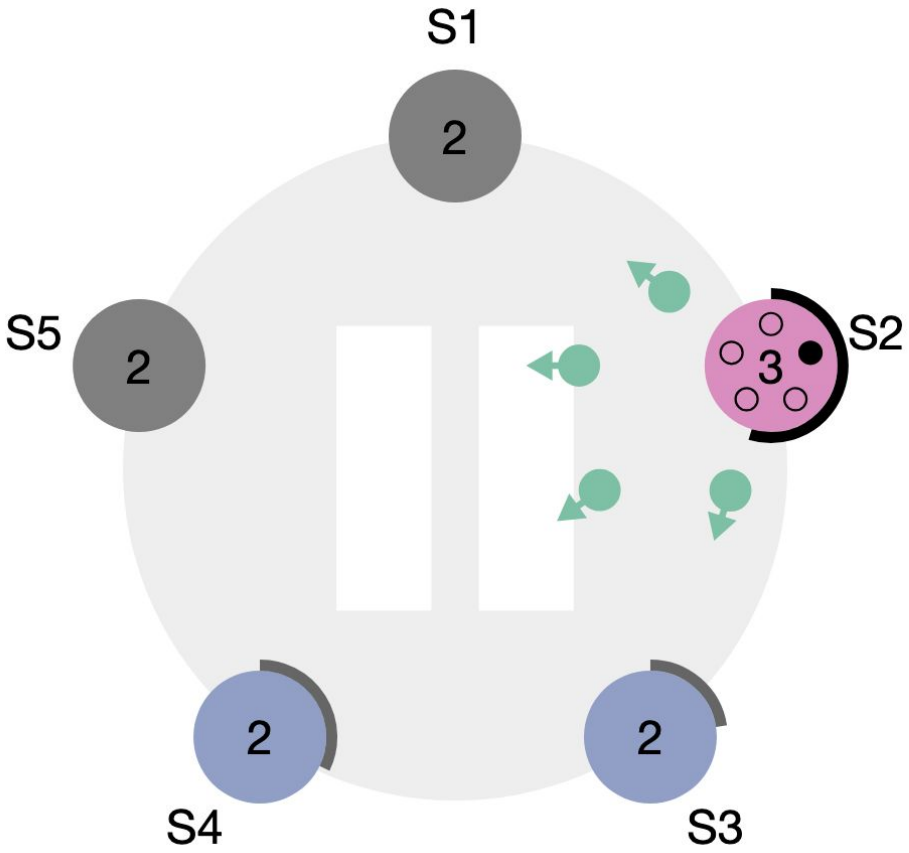


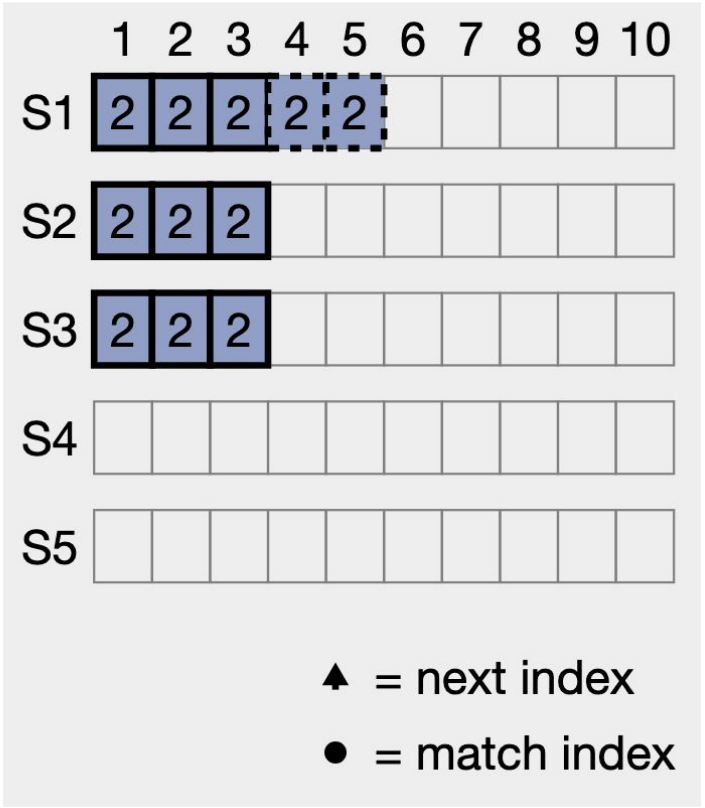
Log Replication: Handling log inconsistencies



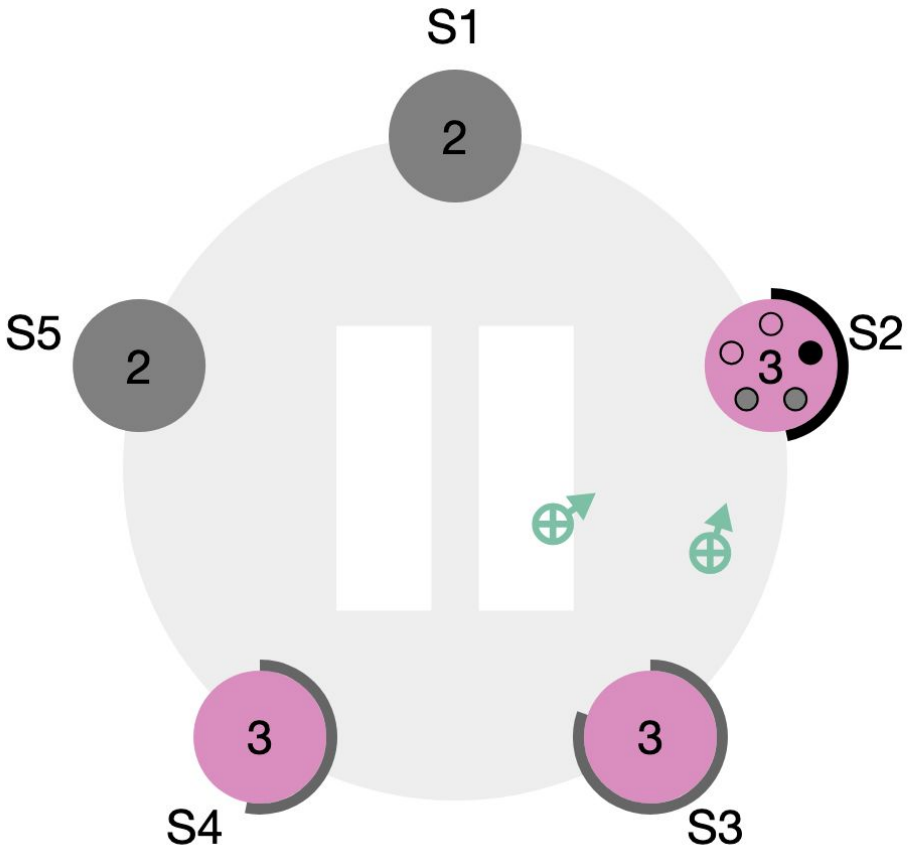


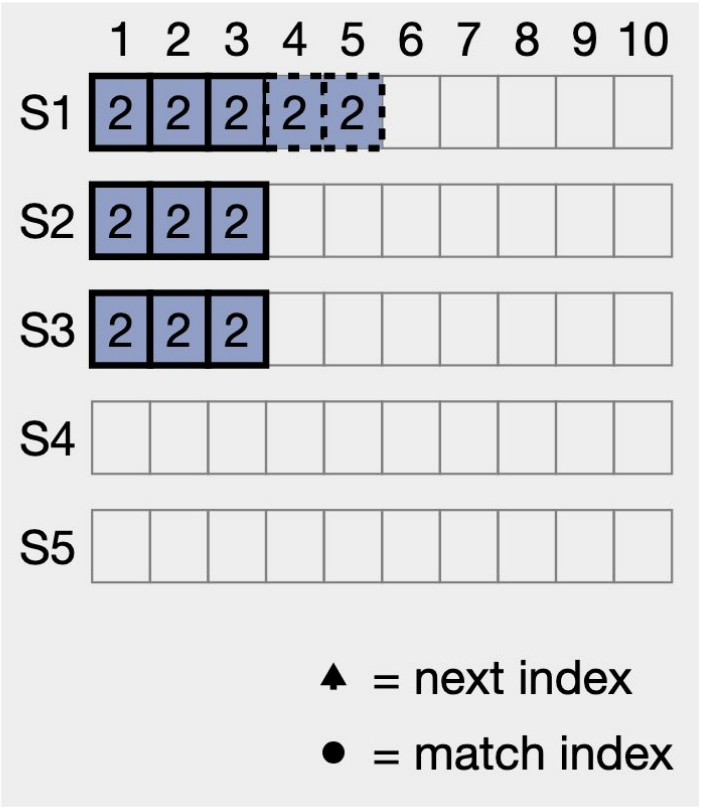
Log Replication: Handling log inconsistencies



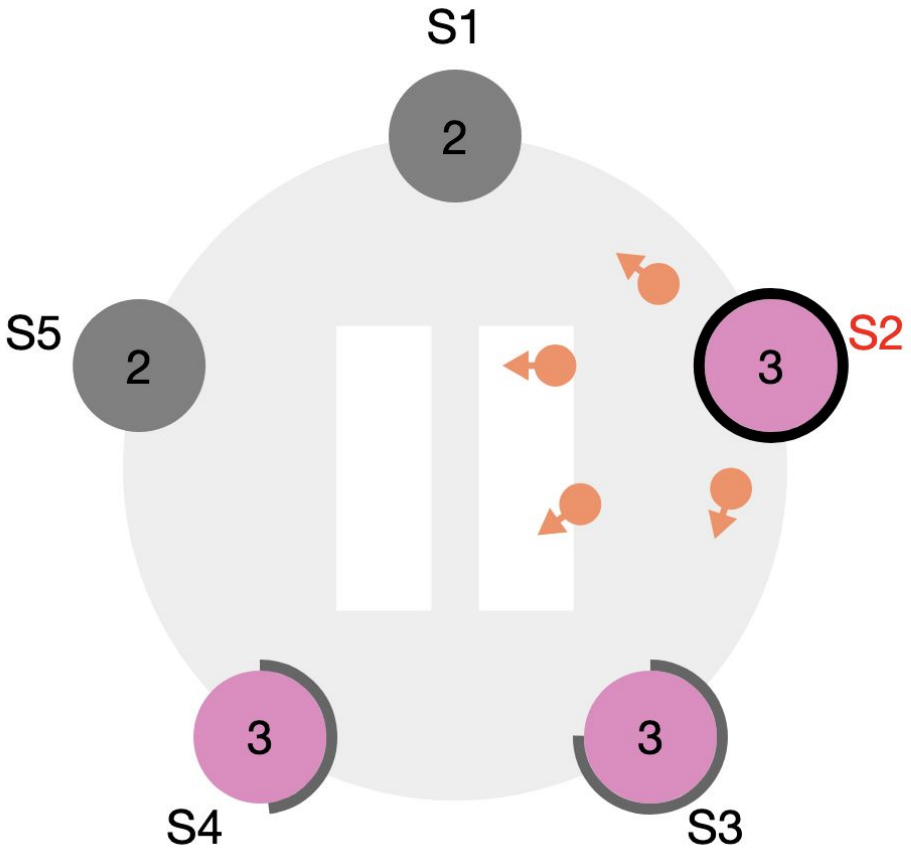


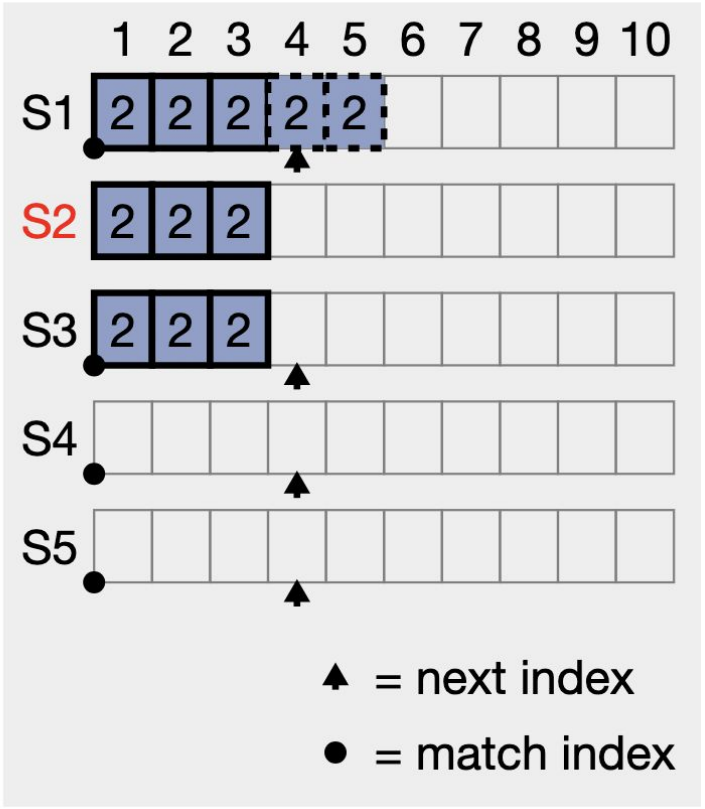
Log Replication: Handling log inconsistencies



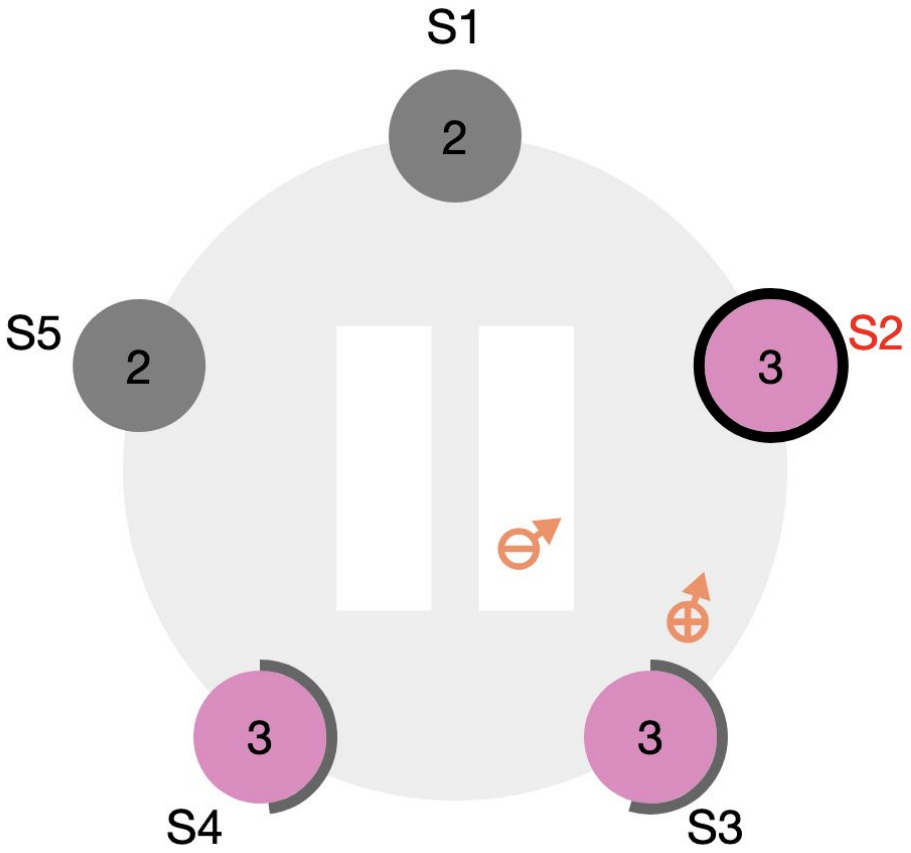


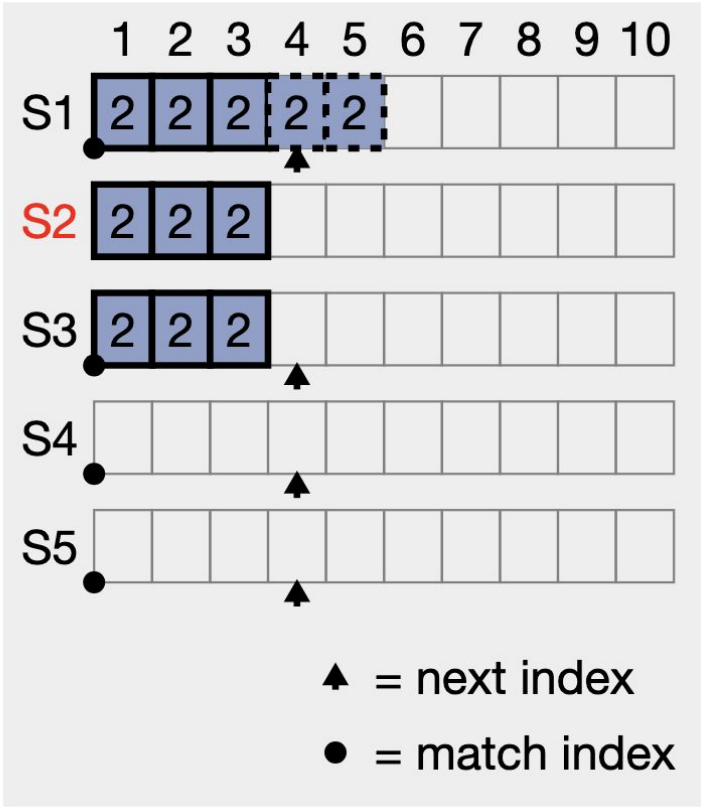
Log Replication: Handling log inconsistencies



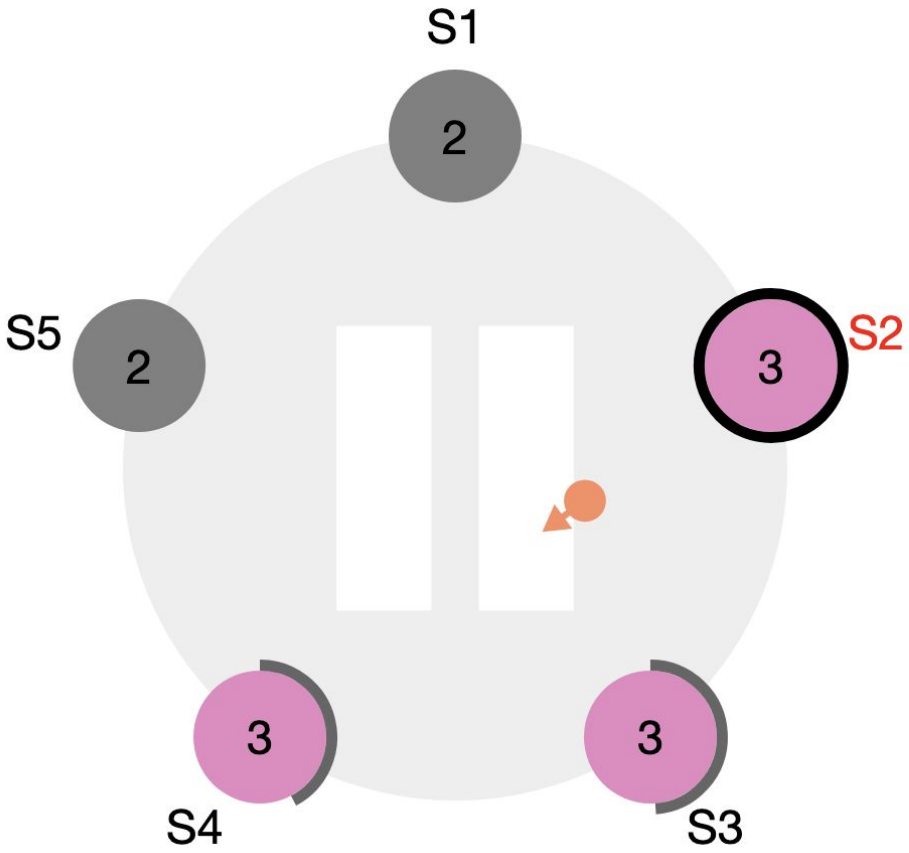


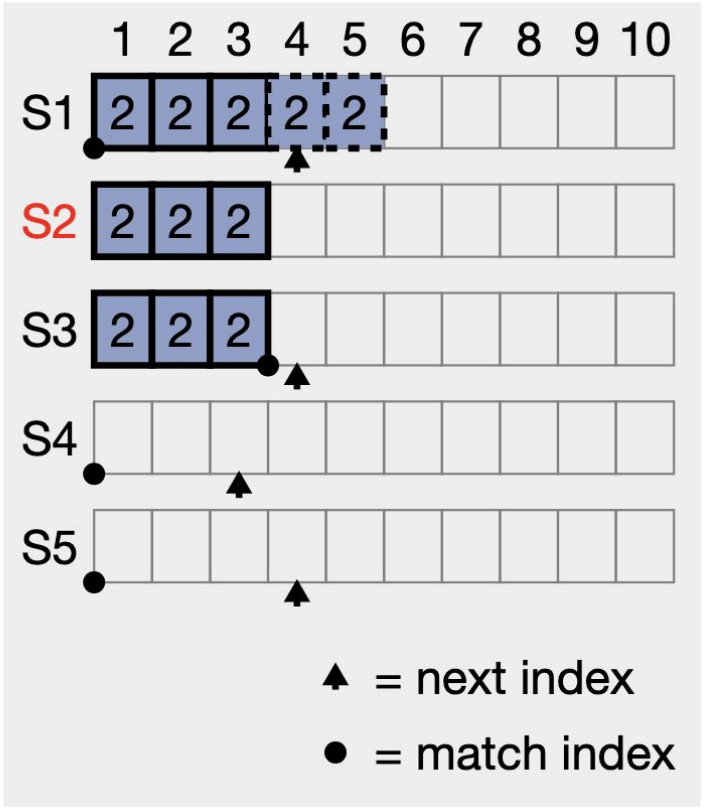
Log Replication: Handling log inconsistencies



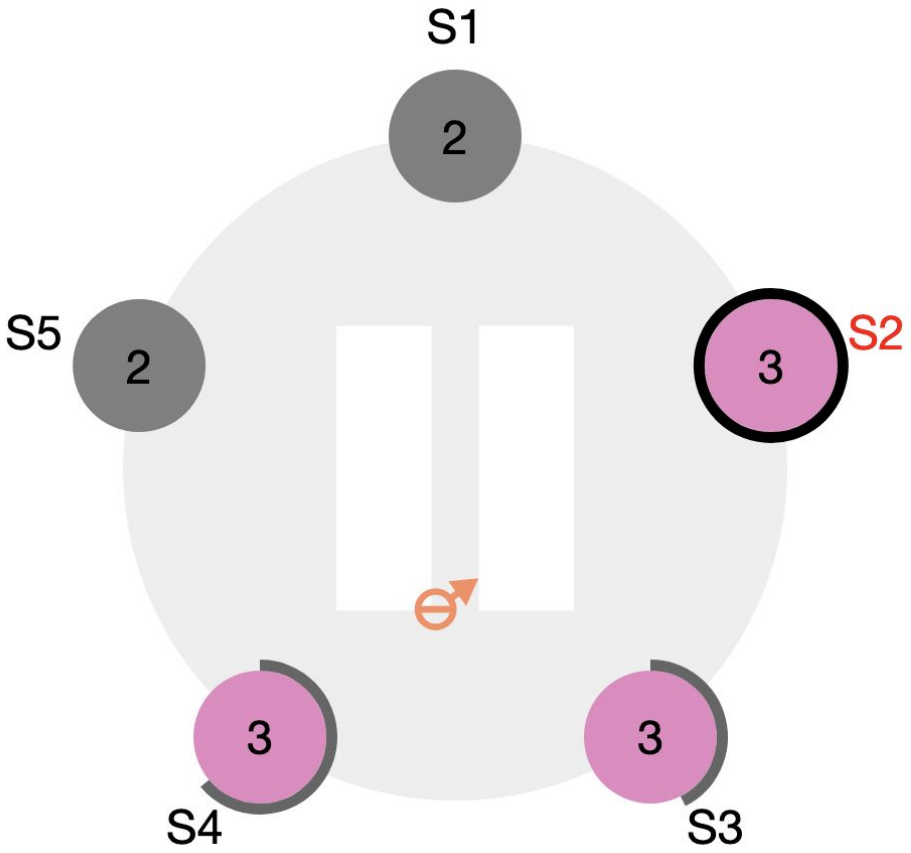


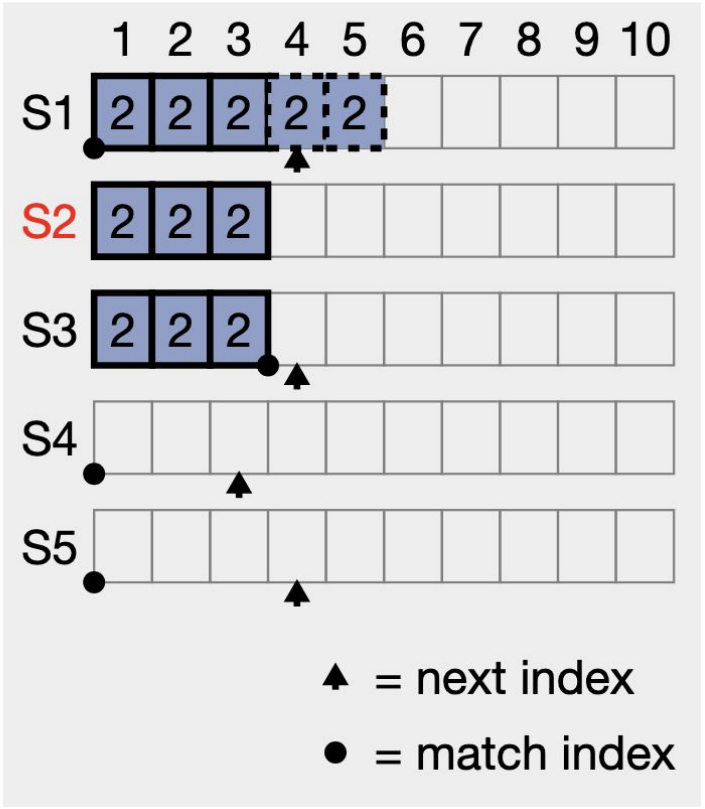
Log Replication: Handling log inconsistencies



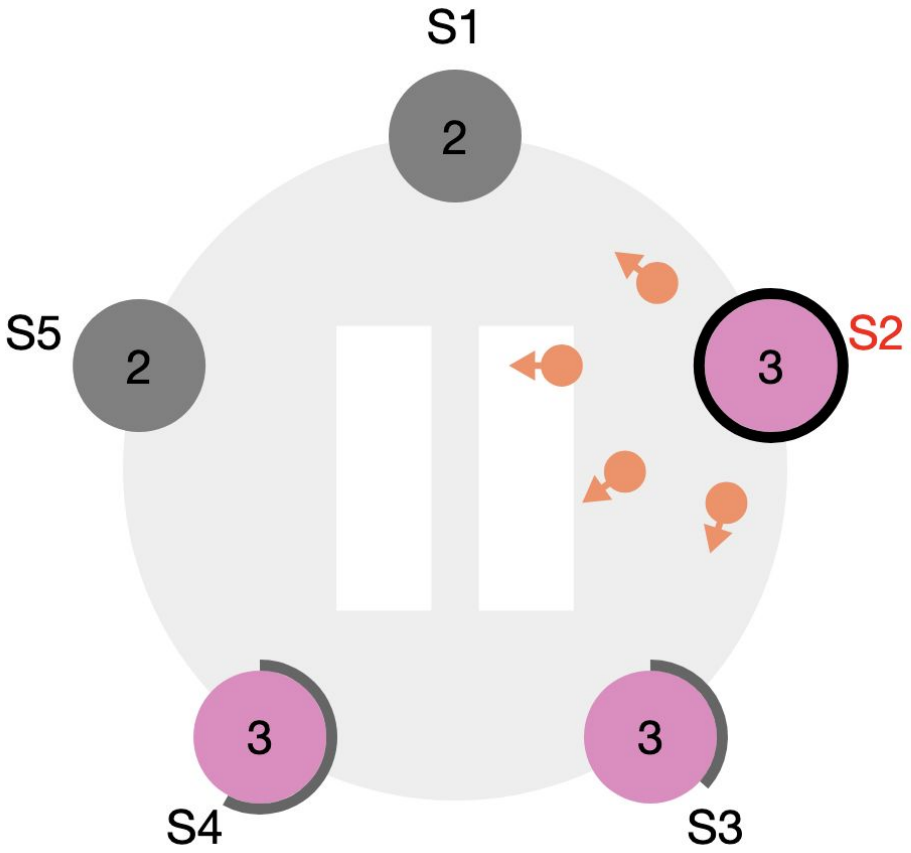


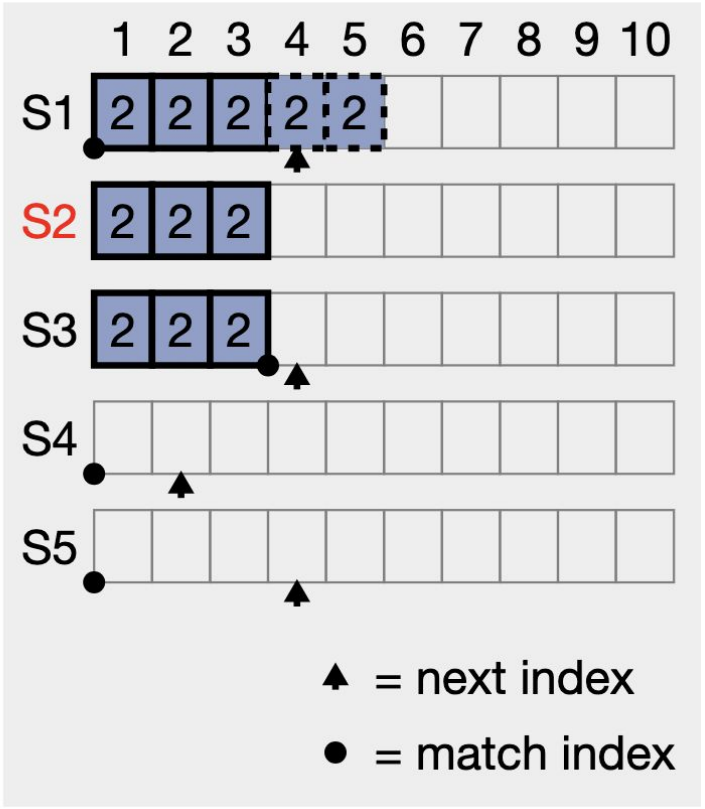
Log Replication: Handling log inconsistencies



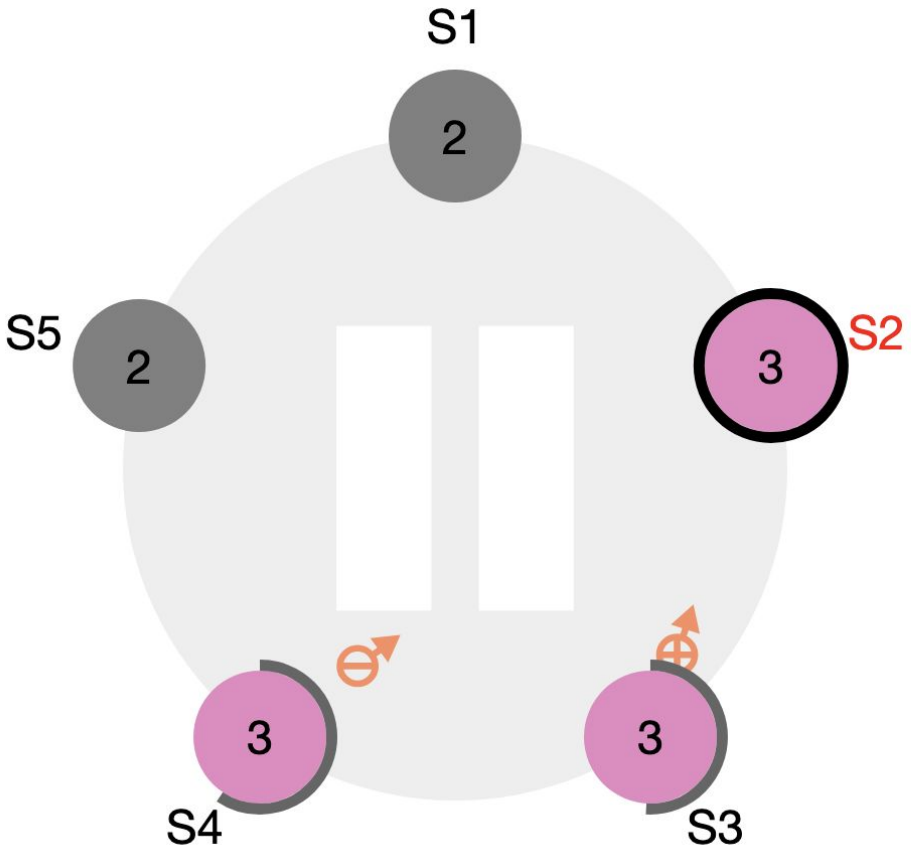


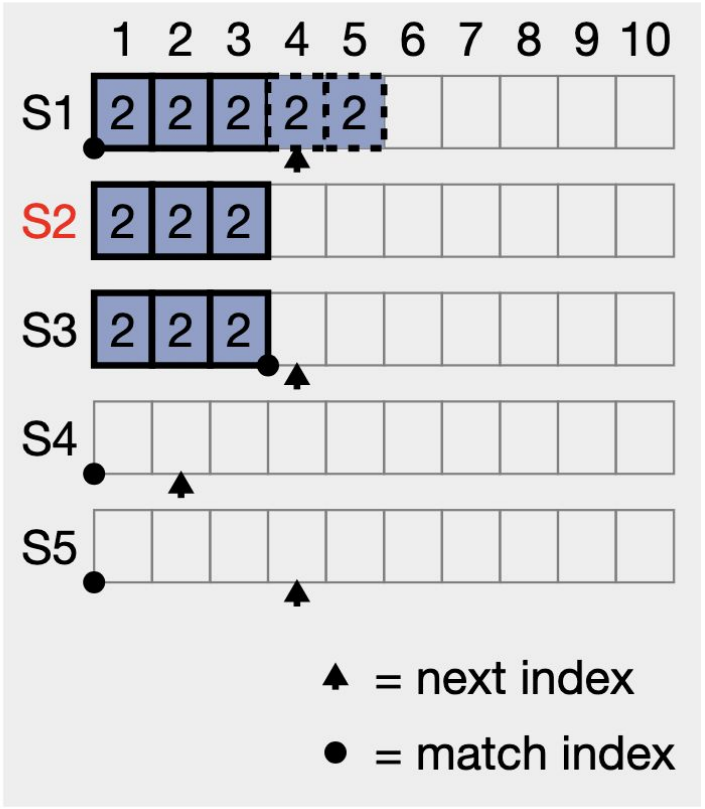
Log Replication: Handling log inconsistencies



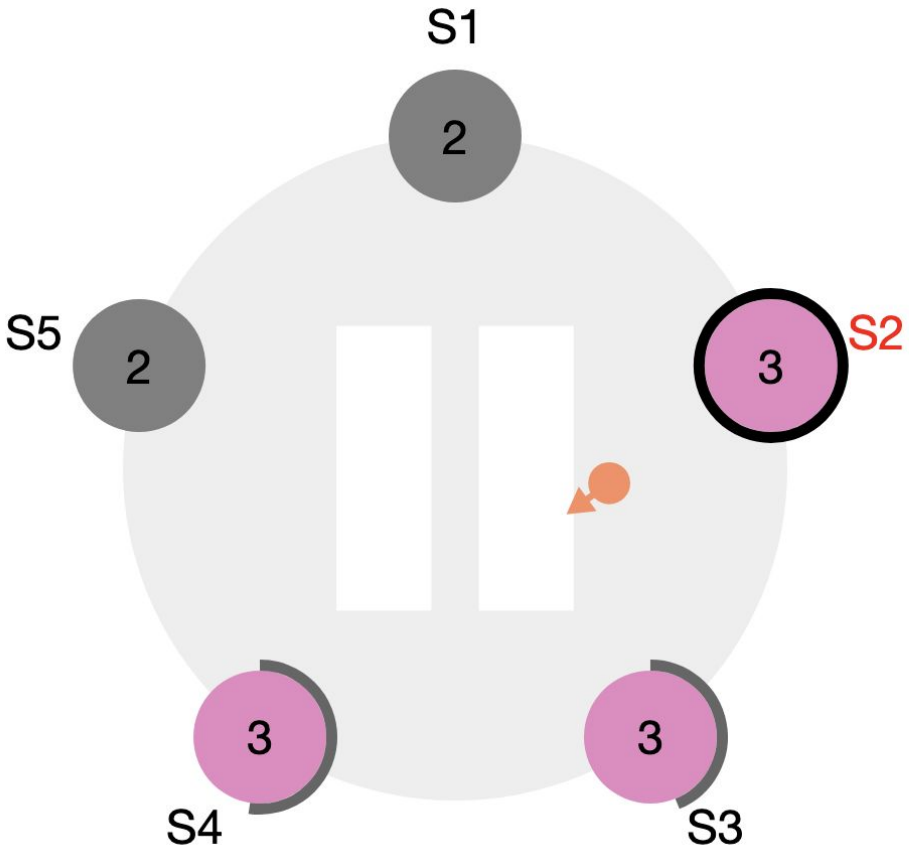


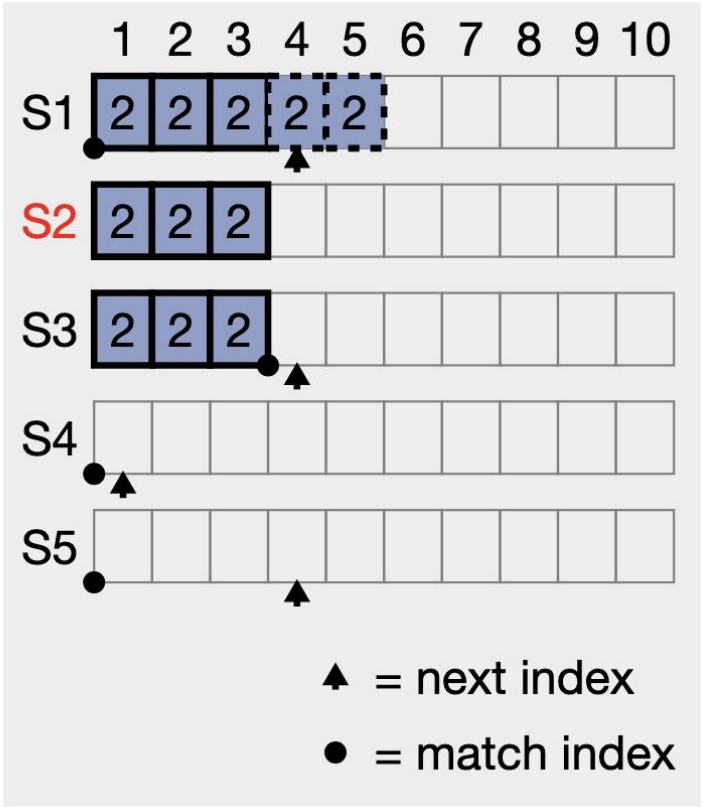
Log Replication: Handling log inconsistencies



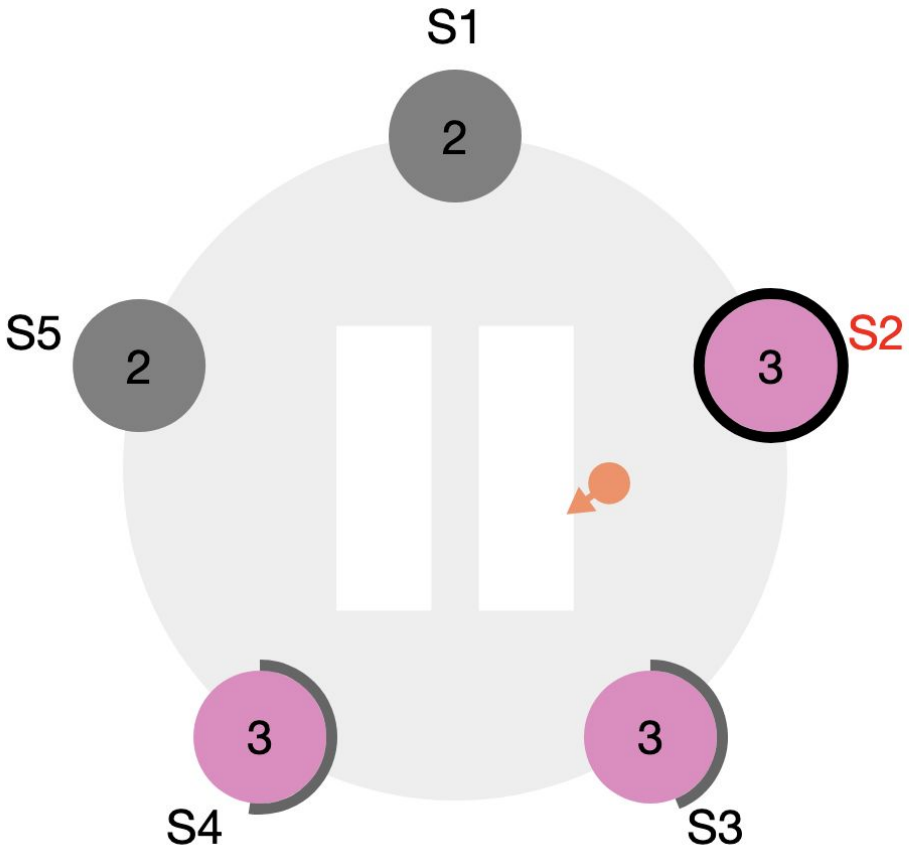


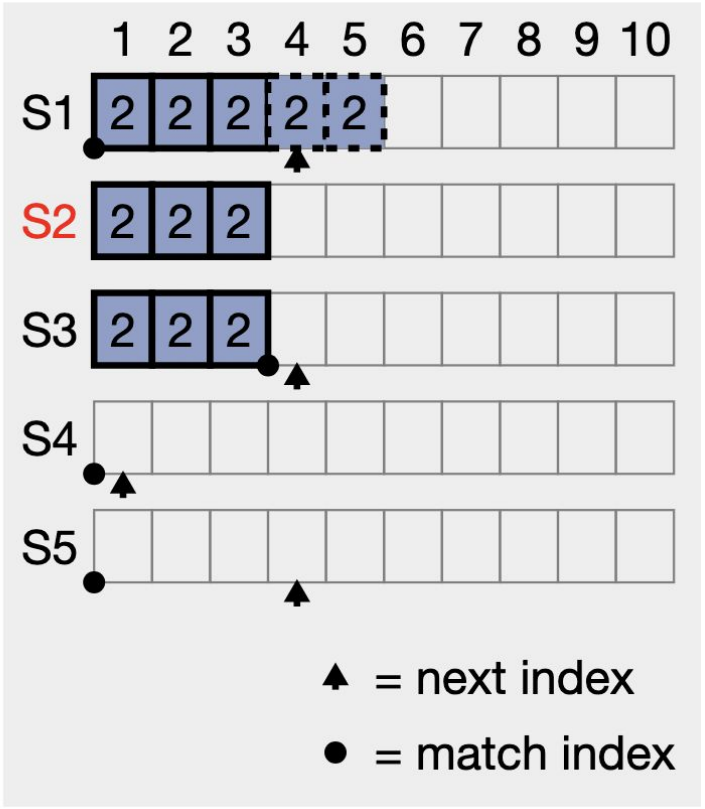
Log Replication: Handling log inconsistencies



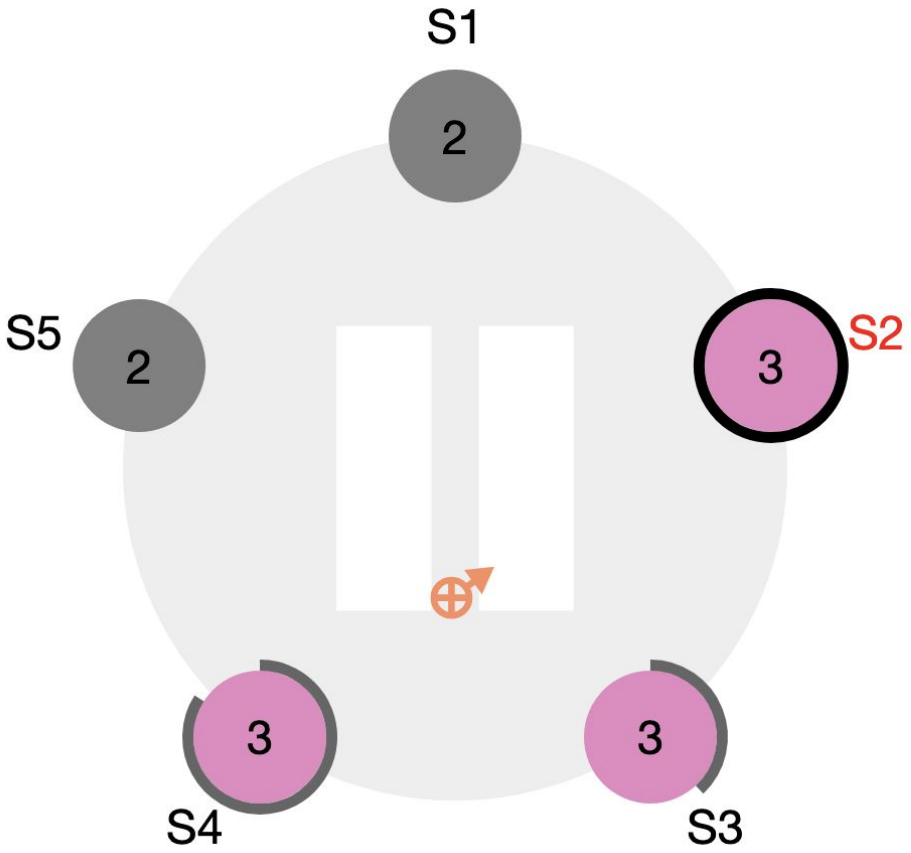


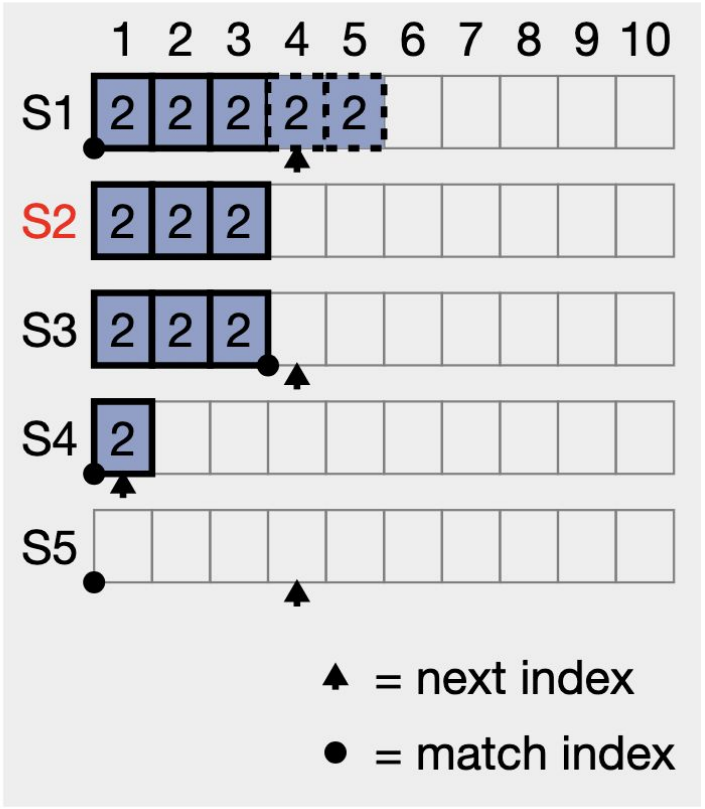
Log Replication: Handling log inconsistencies



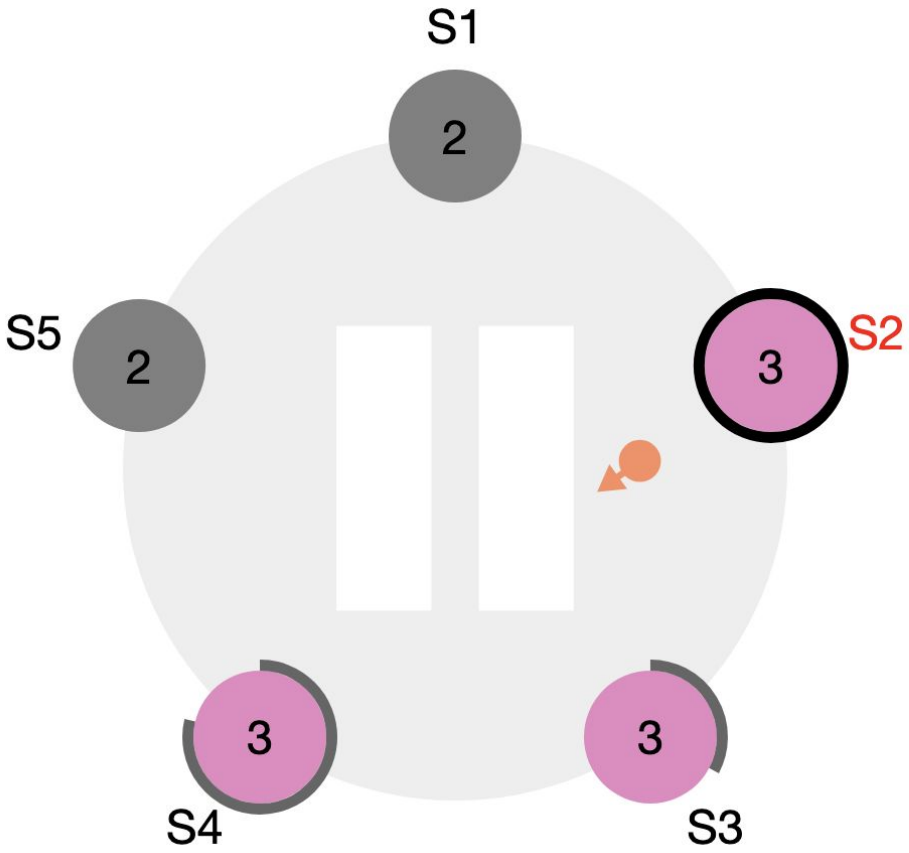


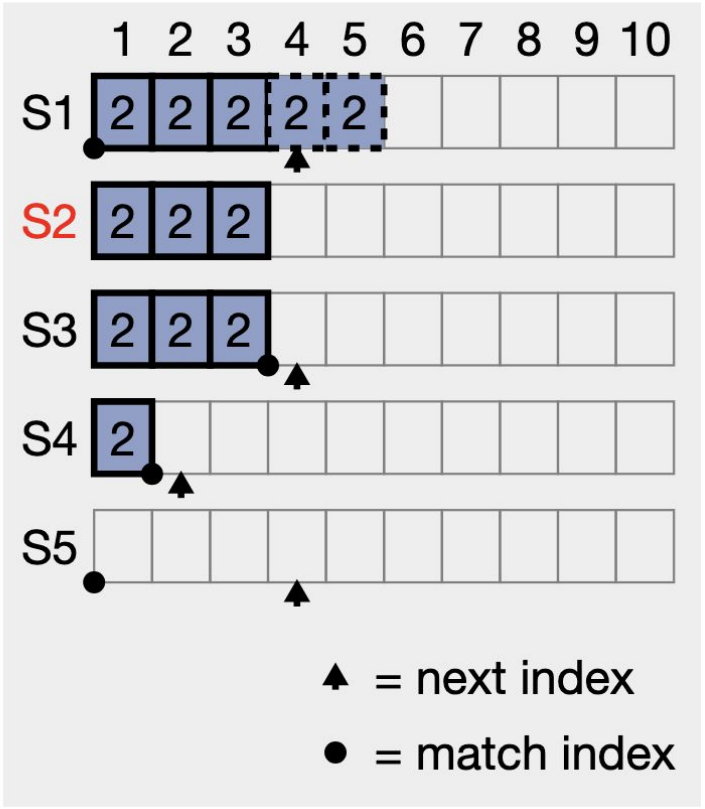
Log Replication: Handling log inconsistencies



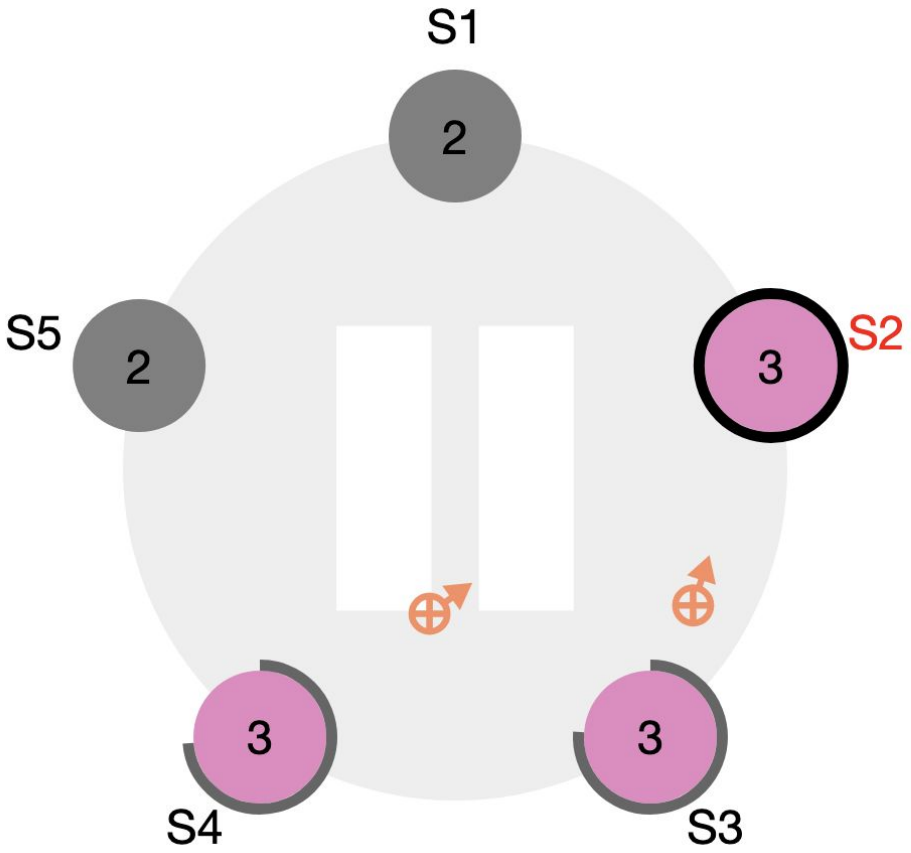


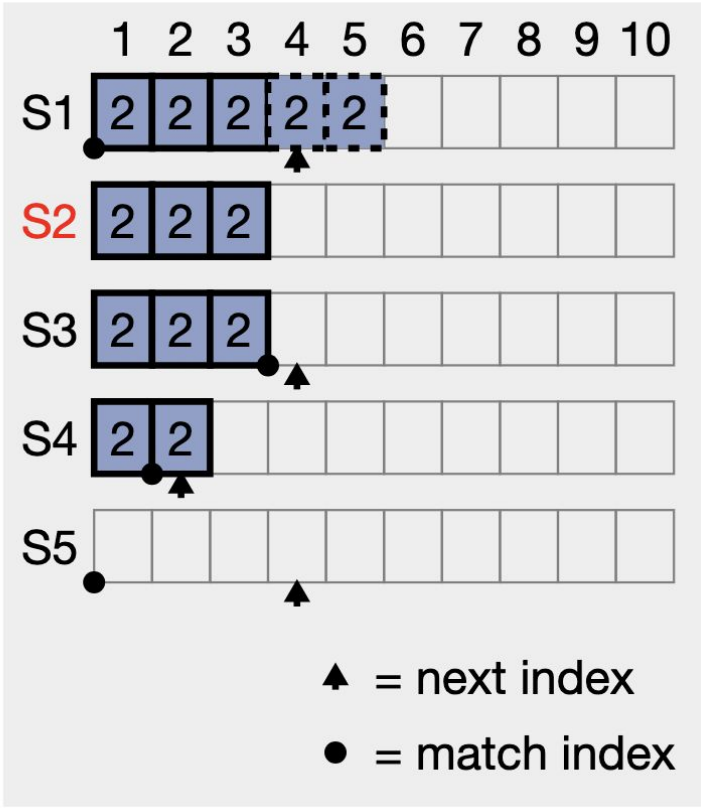
Log Replication: Handling log inconsistencies



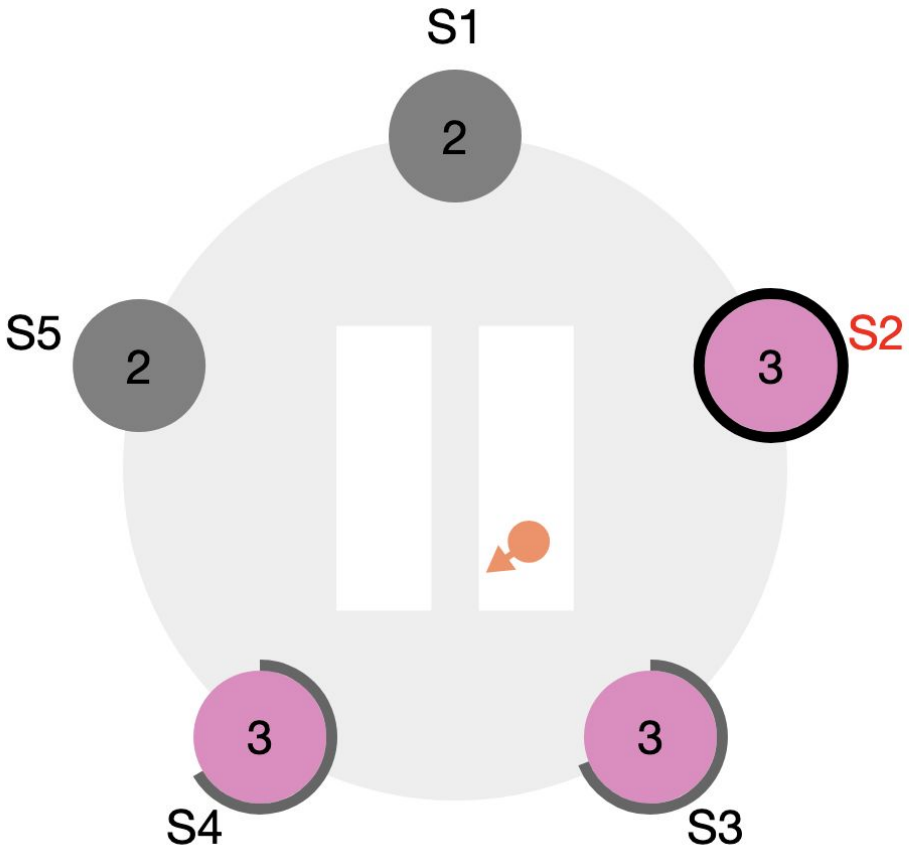


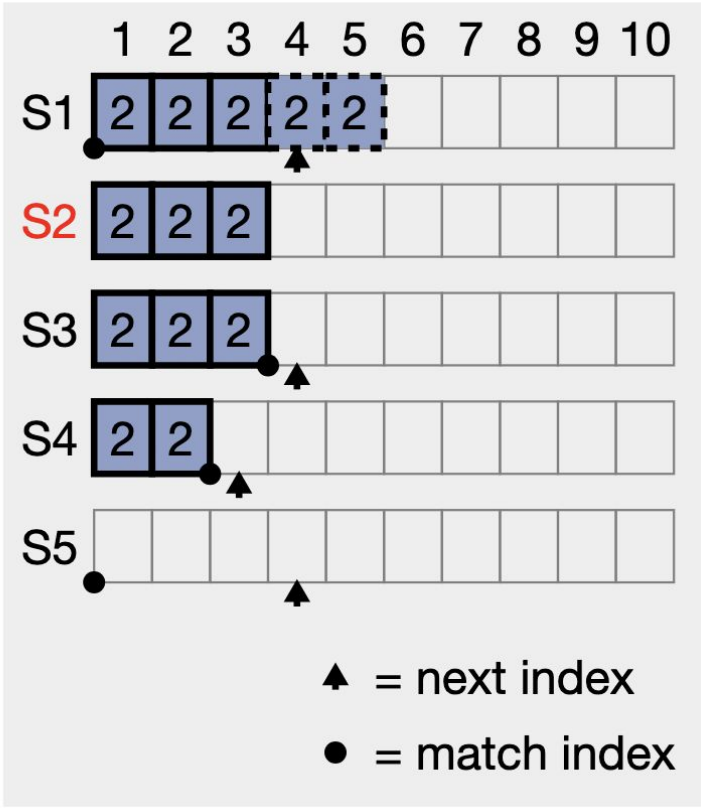
Log Replication: Handling log inconsistencies



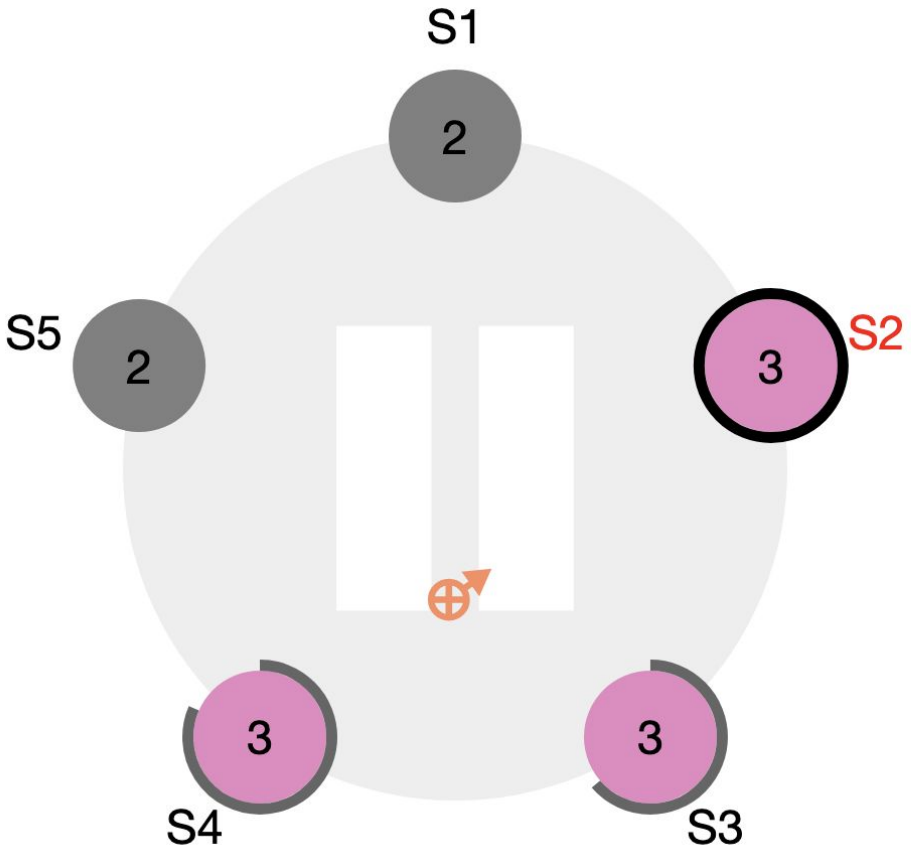


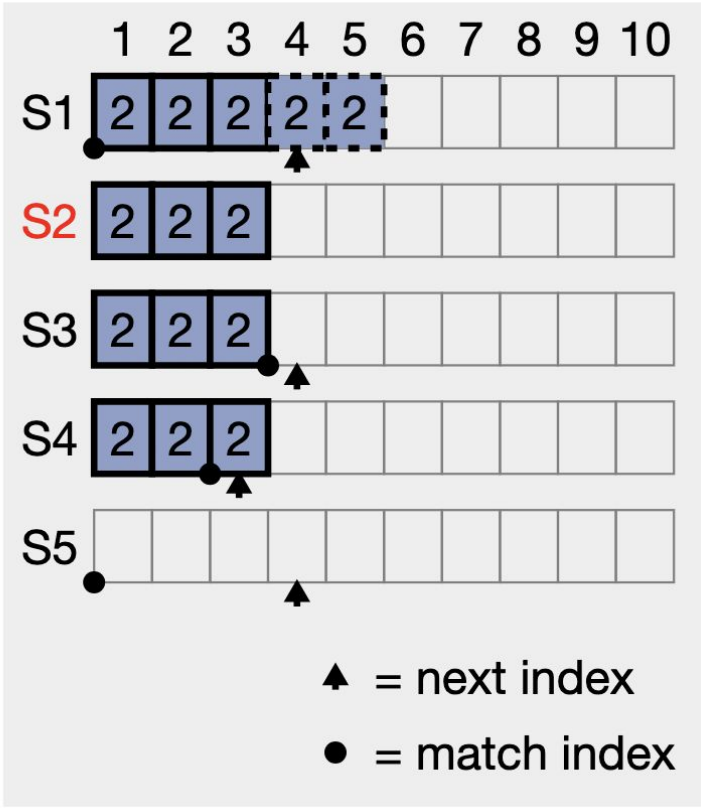
Log Replication: Handling log inconsistencies



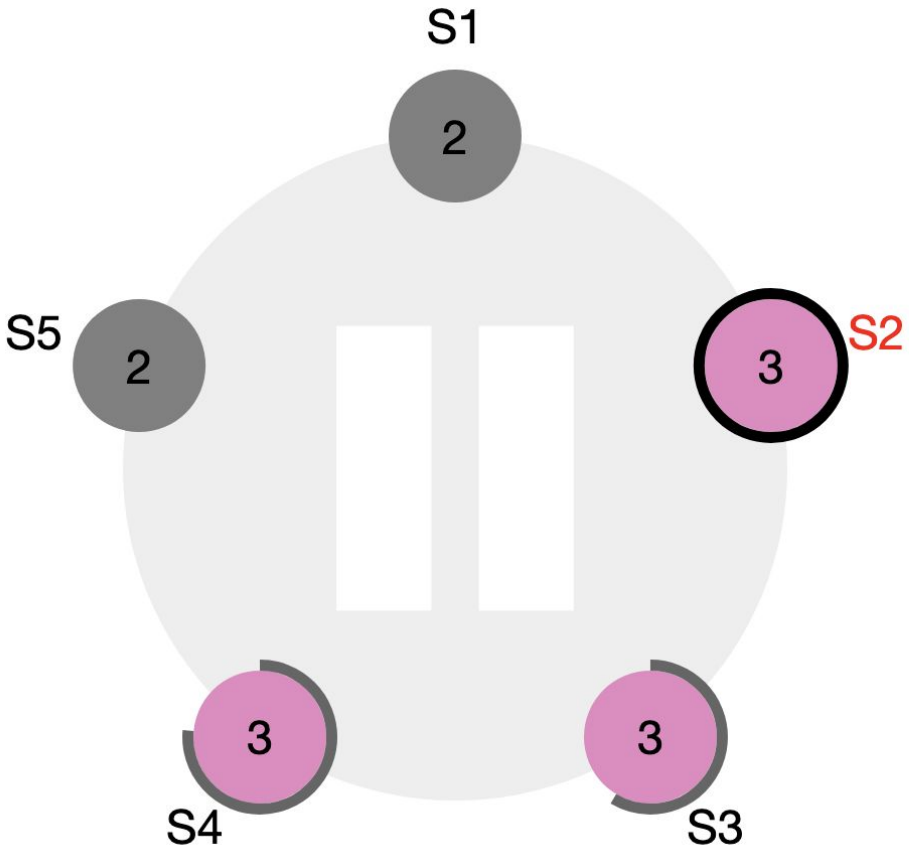


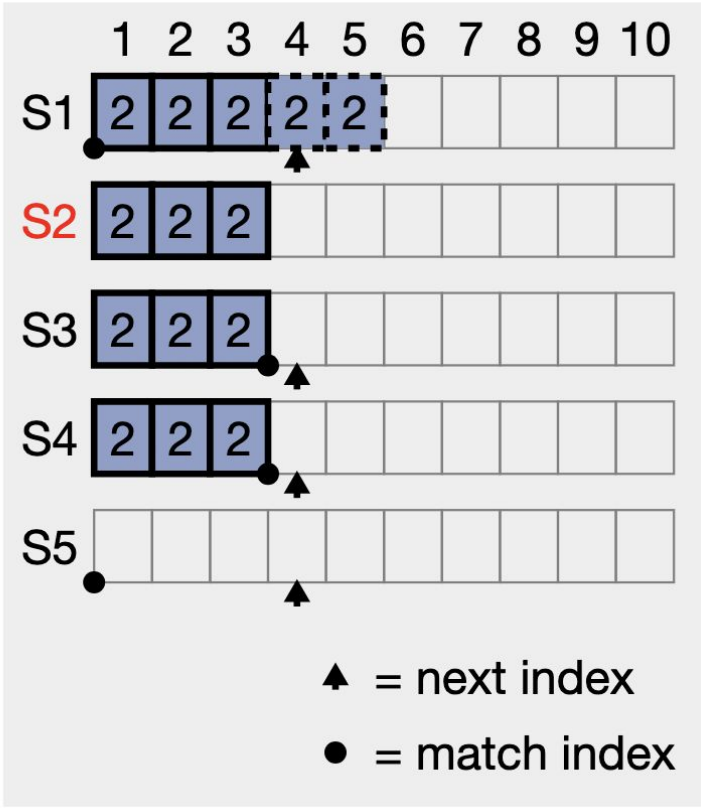
Log Replication: Handling log inconsistencies



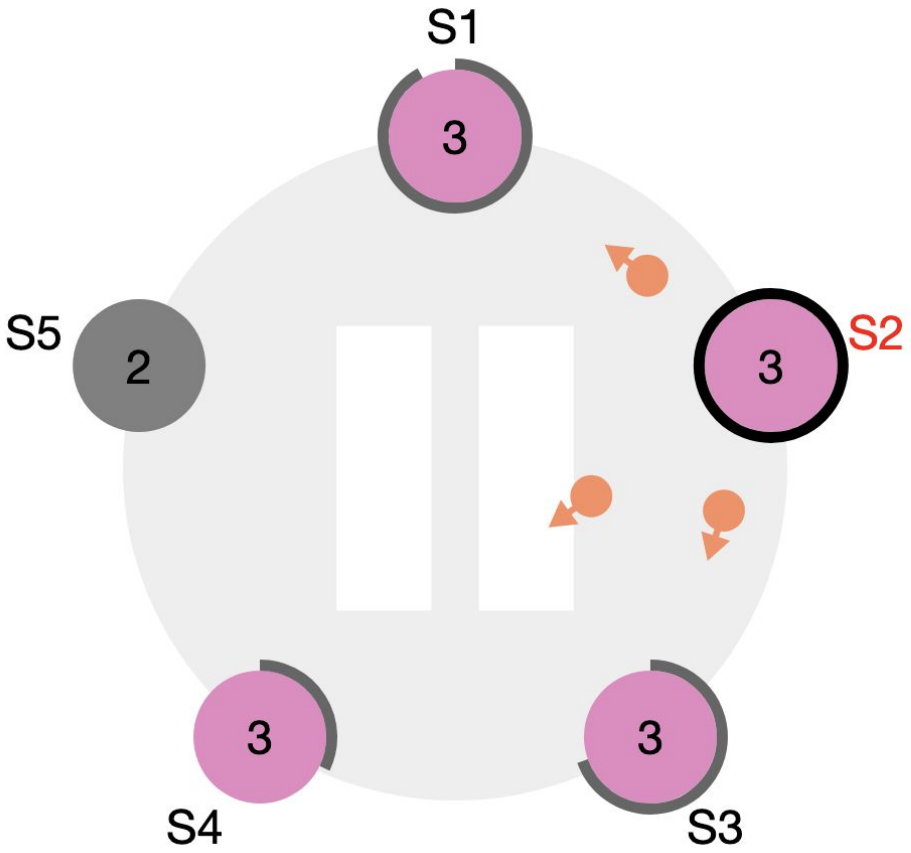


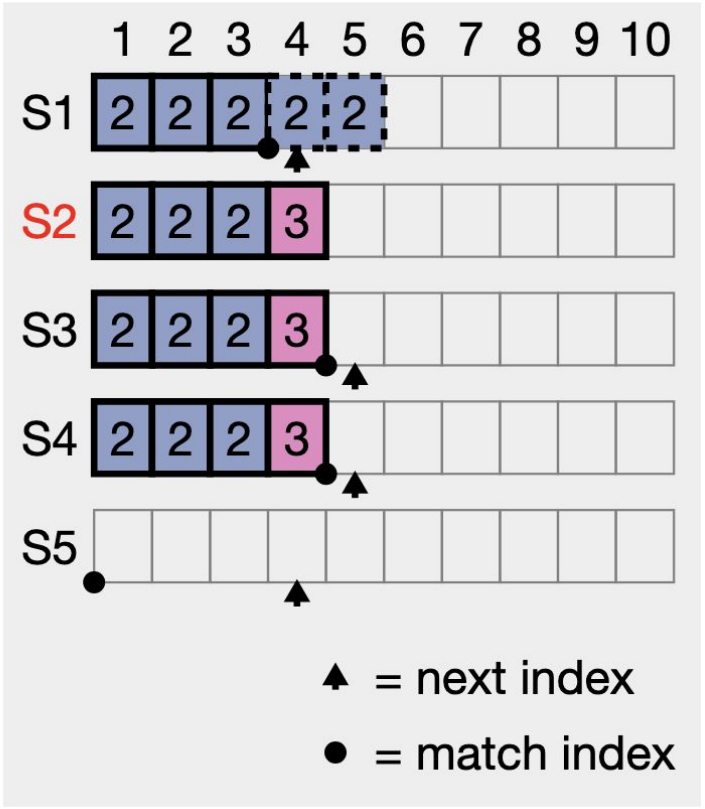
Log Replication: Handling log inconsistencies



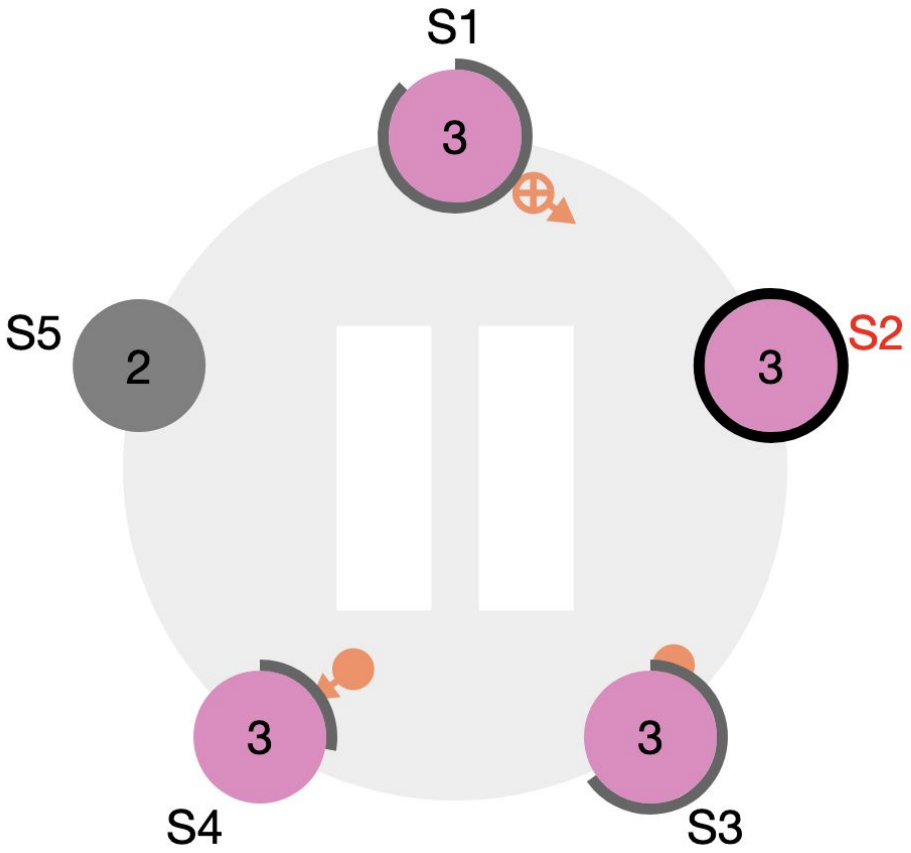


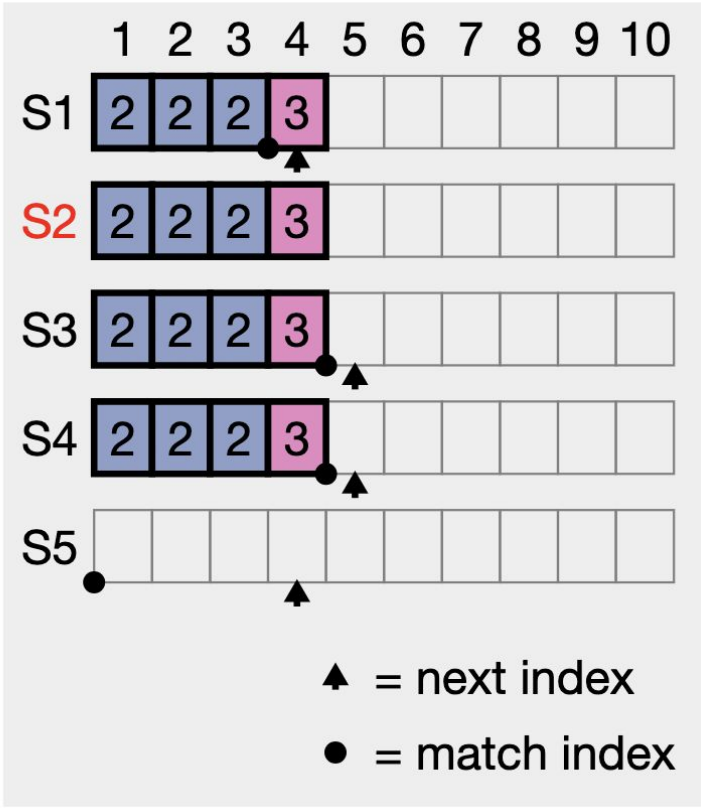
Log Replication: Handling log inconsistencies(overwrite)





Log Replication: Handling log inconsistencies(overwrite)





Problem

- Raft supports
  - Election Safety
  - Leader Append-Only
  - Log Matching
- Still cannot ensure State Machine Safety
- a Follower might be unavailable while the leader commits several log entries, then it could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences.

Requirement

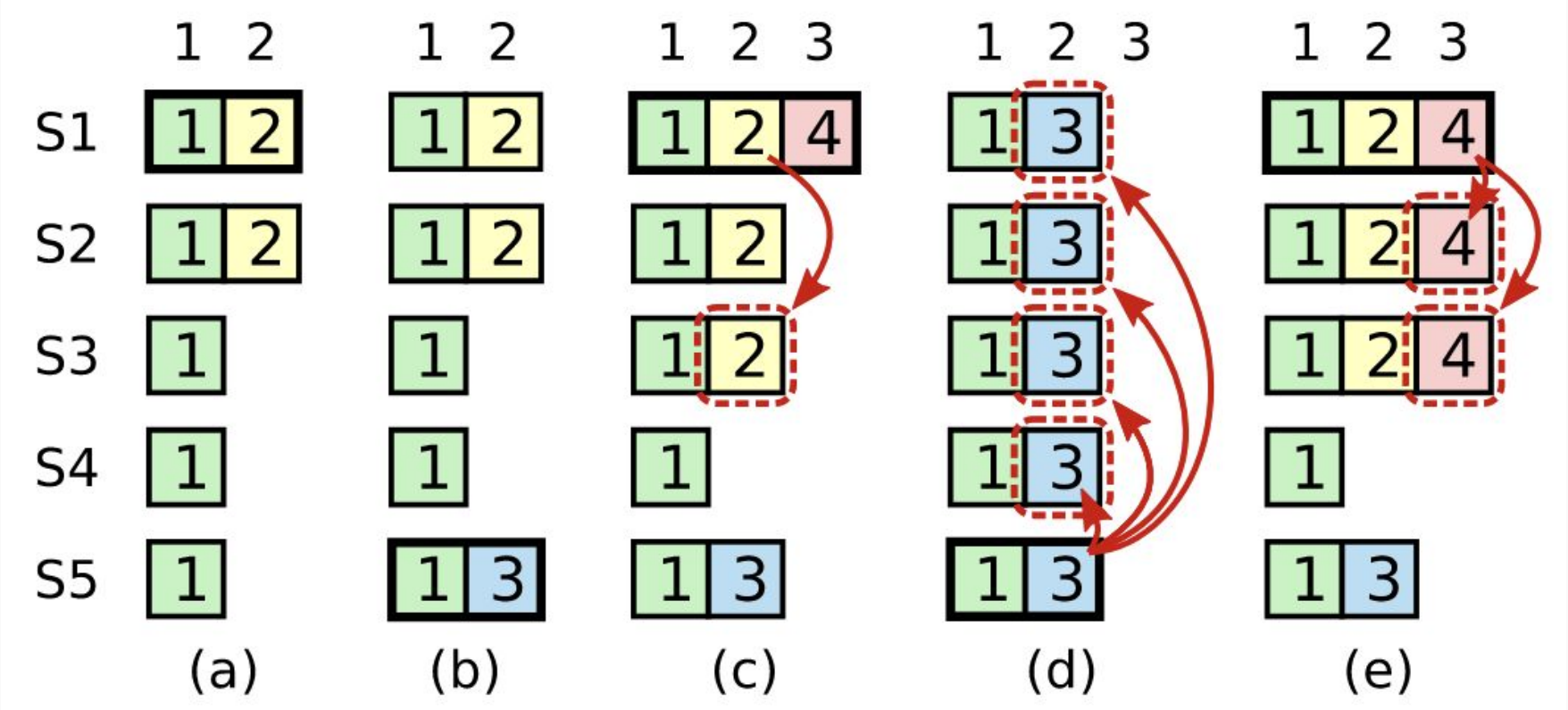
- Ensure **Leader Completeness**
  - Raft guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader.
  - Log entries only flow in one direction, from leaders to followers
- Ensure **State Machine Safety**

First Solution : Election Restriction

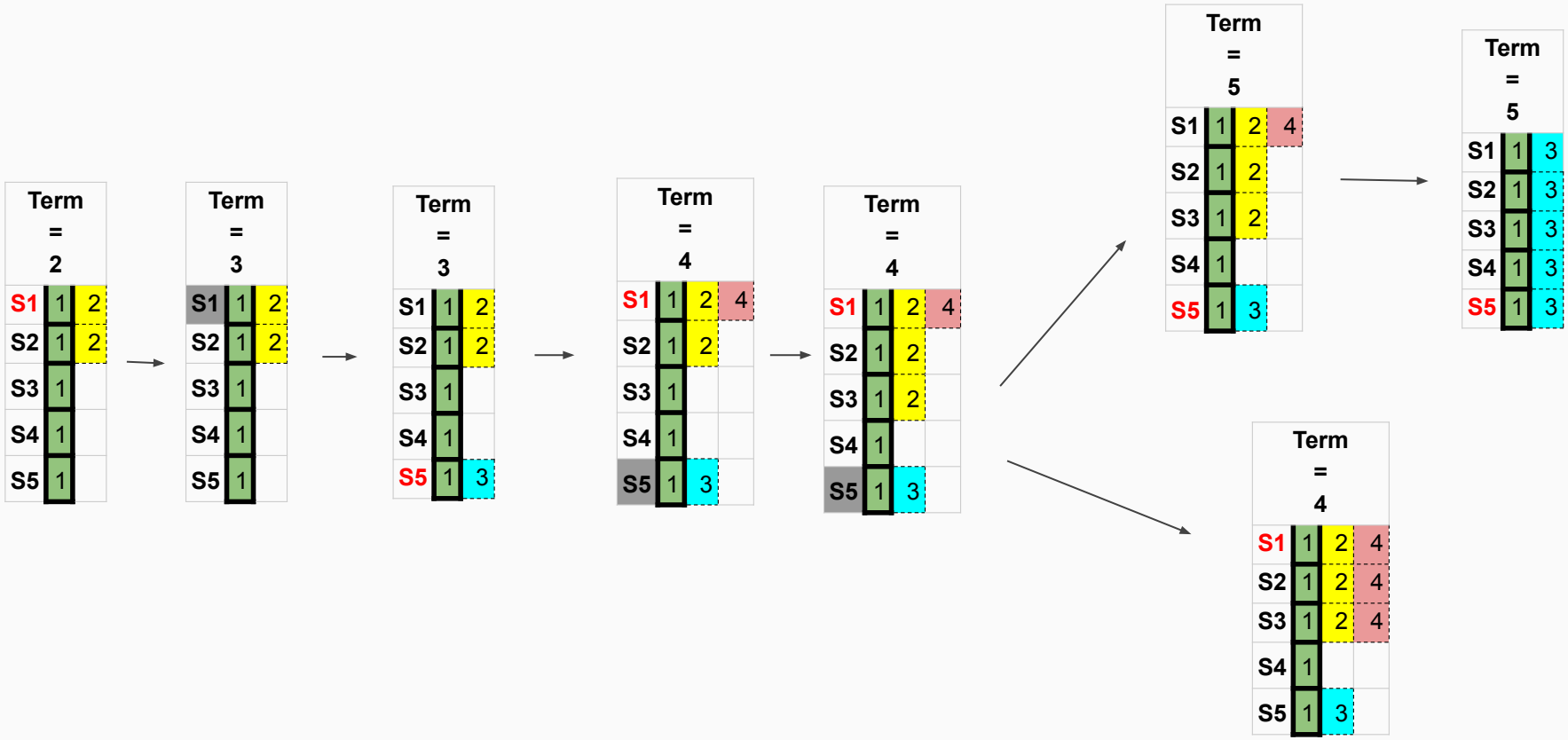
Prevent a candidate from winning an election unless its log contains all committed entries

- Every committed entry must be stored on **at least one server in any majority**.
- To win, a candidate must gather votes from a majority.
- If its log is missing a committed entry, some server in that majority will detect the candidate is not “up-to-date” and refuse to vote. Therefore, the candidate cannot win unless it has **all committed entries**.
- If two logs have last entries with different terms, then the **log** with the **later term** is **more up-to-date**.
- If the logs end with the **same term**, then whichever **log** is **longer** is **more up-to-date**.

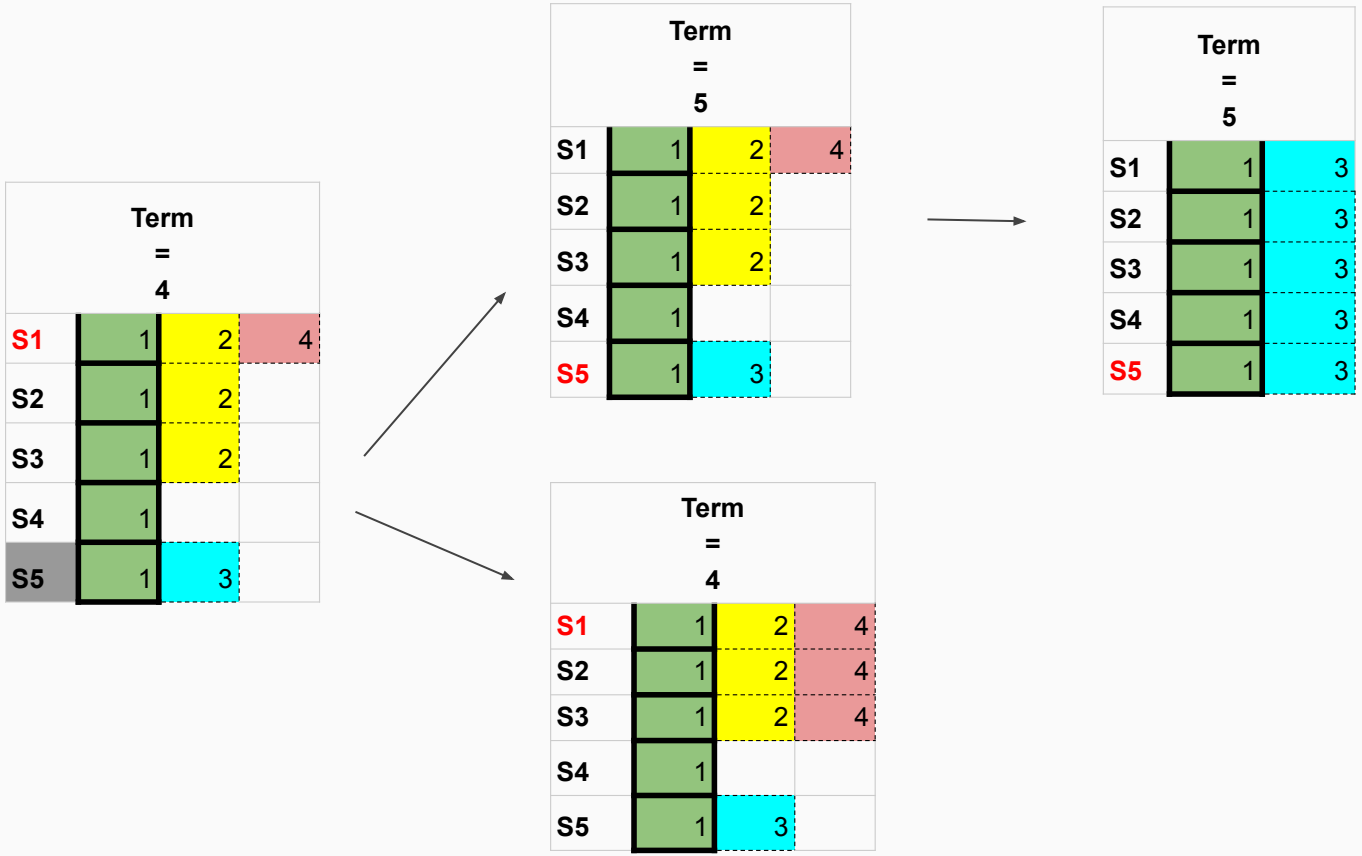
Problem : Committing entries from previous terms



Problem : Committing entries from previous terms



Problem : Committing entries from previous terms



Second Solution : Only commit entries from current term

Solution

- A Leader only commits entries for its current term.
- Implicitly, all prior entries are committed indirectly because of the Log Matching Property

Leader Completeness : Proof by Contradiction

Assumption

- There are 2F + 1 servers
- A leader in term ‘T’ has committed an entry ‘e’ at index ‘i’ in the log
- A leader in term ‘T+n’(also named as ‘U’) does not have the entry ‘e’ in their log. It is the smallest term U > T whose leader ($leader_U$) does not store the entry ‘e’.

T … T+1 … T+2 ….. T+(n-1) …… U = T + n

Analysis

- The $leader_T$ committed an entry to a minimum of F servers(majority). For all terms t > T, a minimum of F+1 servers will have the committed entry ‘e’ in their log.
- When $leader_U$ obtained votes from a majority, there was at least one Voter, that had the entry ‘e’ in its log, and voted for $leader_U$ (Majorities overlap)

Leader Completeness : Proof by Contradiction

- U > T. The Voter accepted the committed entry from $leader_T$ **before** voting for $leader_U$. Else, it would have rejected AppendEntries request from $leader_T$
- If Voter granted its vote to $leader_U$, means log of $leader_U$ must have been at least as up-to-date as the Voter's. This leads to two scenarios:

1) When **$lastlogterm(leader_U) = lastlogterm(Voter)$** and **$loglength(leader_U) >= loglength(Voter)$**

    - This implies that log of U contained every entry already present in log of Voter, meaning log of U contained 'e'
    - Contradiction : U was assumed to not have 'e' in its log

Leader Completeness : Proof by Contradiction

2) When **lastlogterm(leader$_U$) > lastlogterm(Voter)**
- lastlogterm(Voter) >= T
- lastlogterm(leader$_U$) ∈ {T+1, T+2…T+n-1}
- By assumption, the earlier leader that created leader$_U$'s last log entry must have the entry 'e' itself.
- By Log matching property, leader$_U$'s log must also contain the committed entry 'e' => Contradiction

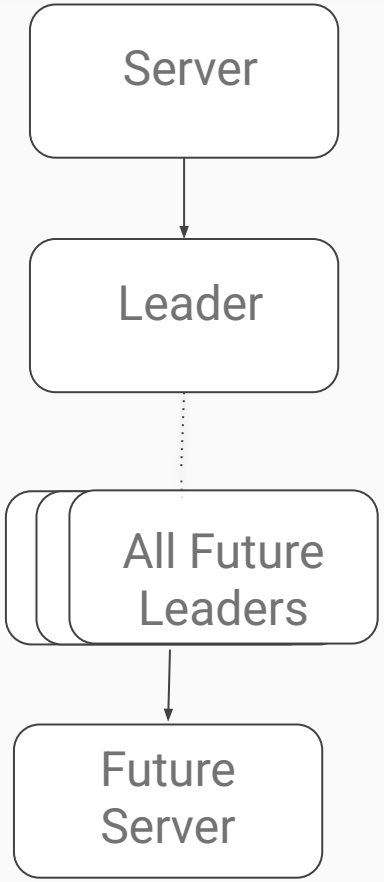
Conclusion
- Leaders of all terms greater than T must contain all entries from term T that are committed in term T.
- The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly.

State Machine Safety

Prove : If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

- At the time a server applies a log entry to its state machine, its log must be identical to the leader's log up through that entry and the entry must be committed.
- Let min_applied_term be the smallest term, in which any server applies a given log index 'i'
- Leader completeness guarantees that leaders for all higher terms will store that same log entry
- Servers that apply the index in later terms will apply the same value.

State Machine Safety



applies entry at index ‘i’ at term min_applied_term

Leader and follower have same log upto this entry, so leader also applies entry at index ‘i’ at term min_applied_term

By Leader completeness, all future Leaders have the same entry in their logs

Also applies same entry at index ‘i’ at a future term

State Machine Safety

- Raft requires servers to apply entries in log index order.

+

- Combined with the State Machine Safety Property,

- All servers will apply exactly the same set of log entries to their state machines, in the same order.

Problem

How to handle Follower and Candidate Crashes?

Follower and Candidate Crashes

- Retrying RequestVote and AppendEntries RPCs **indefinitely,** till servers are live again.
- If a server has applied an RPC, but crashed before responding, it will receive same RPC again after restart.
- **Raft RPCs are idempotent.**
- AppendEntries request is ignored if same log entry already exists on the follower

Timing and Availability

- **Safety** in Raft does **not depend** on **timing**
- **Availability** does **depend** on **timing**, specifically during **Leader Election** to elect and maintain a steady leader

- **broadcastTime** : average time it takes a server to send RPCs in parallel to every server in the cluster and receive their responses
- **electionTimeout :** period in which a follower receives no communication from a Leader, after which an election is triggered
- **MTBF(Mean Time between Failures):** average time between failures for a single server.

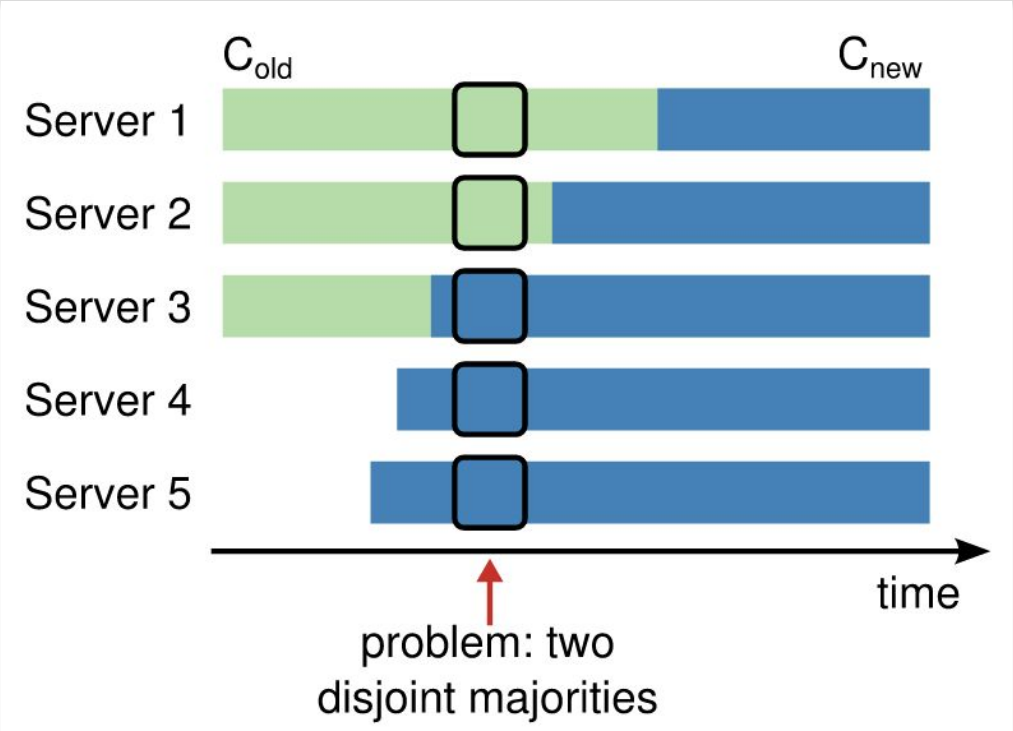
broadcastTime ≪ electionTimeout ≪ MTBF

Problem

How to add and remove servers from cluster?

- Need to **add** or **remove** servers **without** taking the **whole cluster offline.**
- **Not possible** for all servers to **atomically switch** to new configuration at the same time.
- Must be **no point** during **configuration change**, where it is **possible** for **two leaders** to be elected at the same time(Election Safety)

Overlapping configs



Server 1 and 2 form majority in old config
v/s
Server 3, 4, and 5 in new config

Cluster Membership Changes

Use majority consensus provided by Raft

- Use a **two phase** approach
- First **switch** to a **transitional configuration** called **joint consensus**
- Once the **joint consensus has been committed**, the system then **transitions** to the **new configuration**

Joint Consensus

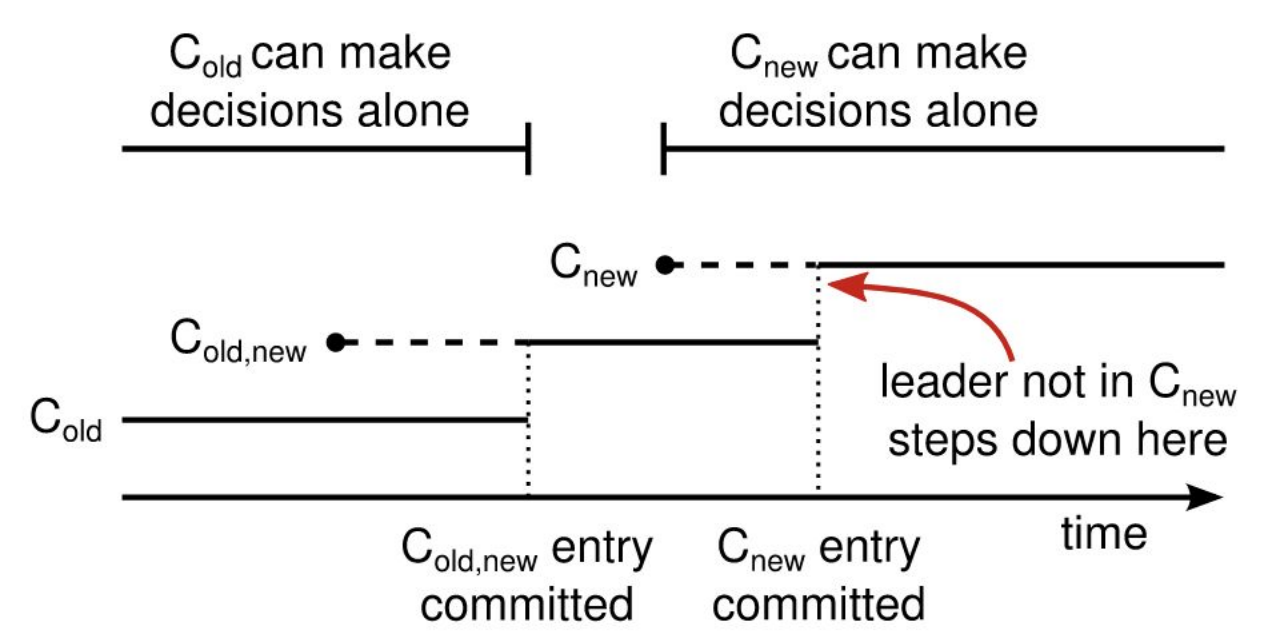
- Allows **individual servers** to **transition** between configurations at **different times** without compromising safety

- Allows the **cluster** to **continue servicing client requests throughout** the **configuration change**.

- **Cluster configurations** are stored and communicated using **special entries** in the **replicated log**

Steps for Membership Change

1) **Leader adds $C_{old,new}$** as a log entry and replicates to followers
2) **Followers add $C_{old,new}$** to their log, and use that config for all future decisions
3) Leader will use the rules of $C_{old,new}$ to determine when the log entry for $C_{old,new}$ is committed.
4) If the leader crashes, a new leader may be chosen under either C_{old} or $C_{old,new}$, depending on whether the winning candidate has received $C_{old,new}$.
5) **Once $C_{old,new}$ has been committed, neither C_{old} nor C_{new} can make decisions without approval of the other**.
6) Leader Completeness Property ensures that only servers with the $C_{old,new}$ log entry can be elected as Leader
7) **Leader creates** a **log entry** describing **C_{new}** and replicates it to the cluster.
8) When the **new configuration has been committed** under the rules of **C_{new}**, the **old configuration is irrelevant** and servers not in the new configuration can be **shut down**



No point in time in which Cold and Cnew can both make decisions independently.

Issues with Reconfiguration

New servers may not initially store any log entries.

- Could take quite a while for them to catch up, during which time it might not be possible to commit new log entries.
- Raft introduces an additional phase before the configuration change, in which the new servers join the cluster as non-voting members
- The Leader replicates log entries to them, but they are not considered for majorities.
- Once the new servers have caught up with the rest of the cluster, the reconfiguration can proceed as described above.

Issues with Reconfiguration

The Leader may not be part of the new configuration

- The Leader steps down (returns to follower state) once it has committed the C_{new} log entry.
- A period of time (while it is committing C_{new}) when the Leader is managing a cluster that does not include itself.
- The Leader replicates log entries but does not count itself in majorities.

Issues with Reconfiguration

Removed servers can disrupt the cluster.

- After transition to C_{new}, removed servers stop receiving heartbeats, leading to **timeouts** and **new elections** with higher terms.
- A newly elected leader may not be steady because of repeated timeouts, leading to poor availability

Solution

- Servers **ignore RequestVote RPCs** when they **believe** a **current leader exists**.
- If a **server receives** a **RequestVote RPC within** the **minimum election timeout** of **hearing from** a **current leader**, it **does not update its term or grant its vote**.

Evaluation: Understandability

Methodology

- A study was conducted with students at Stanford and U.C. Berkeley comparing Raft to Paxos
- Students watched video lectures and took quizzes on both algorithms

Results

- On average, participants scored 4.9 points higher on the Raft quiz than the Paxos quiz
- An overwhelming majority (33 of 41 participants) reported they felt Raft would be easier to implement and explain

Evaluation: Understandability

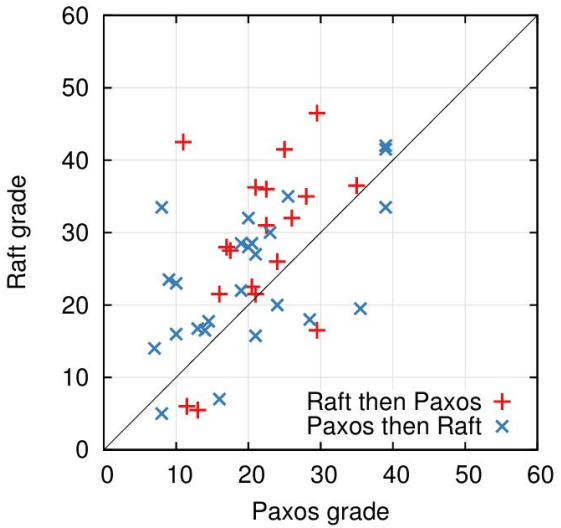


Figure 12: A scatter plot comparing 43 participants' performance on the Raft and Paxos quizzes. Points above the diagonal (33) represent participants who scored higher for Raft.

Evaluation: Correctness

Formal Specification

- A formal specification for Raft was developed using the TLA+ language.
- This specification is about 400 lines long and makes the algorithm completely precise.

Proof of Safety

- The **Log Completeness Property** was mechanically proven using the TLA proof system.
- The **State Machine Safety property** was proven with a complete and relatively precise informal proof (about 3500 words).

Evaluation: Performance

- Raft's performance is similar to other consensus algorithms like Paxos
- For **replicating log entries** raft uses minimal number of messages (a single round-trip from the leader to half the cluster)
- It easily supports batching and pipelining requests for higher throughput and lower latency

Leader Election: Convergence speed and downtime

Q. Does the election process converge quickly?

- **Without Randomness:** No. The process is slow, consistently taking over 10 seconds to converge due to repeated split votes.

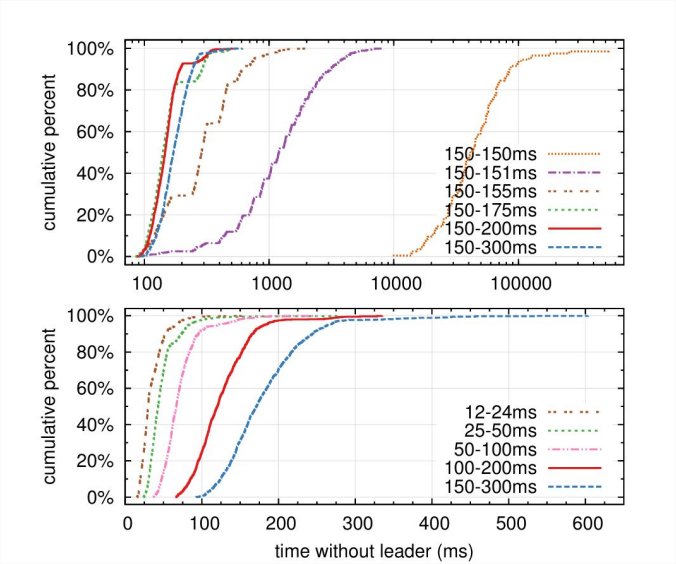
Evaluation: Performance

- **With Randomness:** Yes. Adding just 5ms of randomness to the election timeout allows elections to converge rapidly, with a median downtime of 287ms.

  Q. What is the minimum downtime after a leader crash?

- In the experiments, the minimum average downtime achieved was **35 ms** when using a short election timeout of 12-24 ms.
- **Caveat:** Lowering timeouts too much can cause instability, as leaders may not have time to send heartbeats before other servers start new elections.
- The paper recommends a conservative timeout like **150-300 ms** for a good balance of availability and stability

Time to detect a crashed leader



Thank You