

CPSC 416 – Distributed Systems

Lecture 15: Chain Replication & CRAQ

UBC Vancouver

2026-03-03

From Last Time

Turn to your neighbor. 2 minutes.

Flexible Paxos lets Phase 1 and Phase 2 quorums differ in size.

What is the *only* constraint that must hold between them?

Why does that constraint exist?

(Pairs: 2 minutes)

Checkpoint 0

Cross-phase intersection: every Phase 1 quorum must overlap every Phase 2 quorum.

This ensures a new leader can always discover any value that was committed.

But notice what we've been doing:

L10–L14 all used *voting*. Multiple nodes compare ballots, promise, accept.

Is voting the only way to get strong consistency?

Failure-First Question

Quorum-based consensus is expensive.

Multiple rounds. Multiple messages. Every node participates in decisions.

When can we get strong consistency cheaper?

What if we could avoid voting entirely—and still never lose a write?

The Idea: Replace Voting with Structure

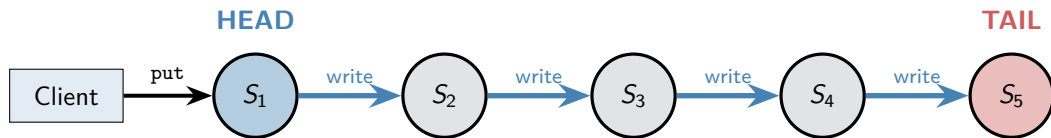
In Paxos, any node can propose. The protocol resolves conflicts.

What if we eliminate conflicts by design?

- Writes go to **one** designated node (the **head**)
- The head forwards to its successor, who forwards to the next, ...
- The last node (the **tail**) *commits* the write

No ballots. No quorums. No voting. Just a chain.

Chain Replication (van Renesse & Schneider, 2004)

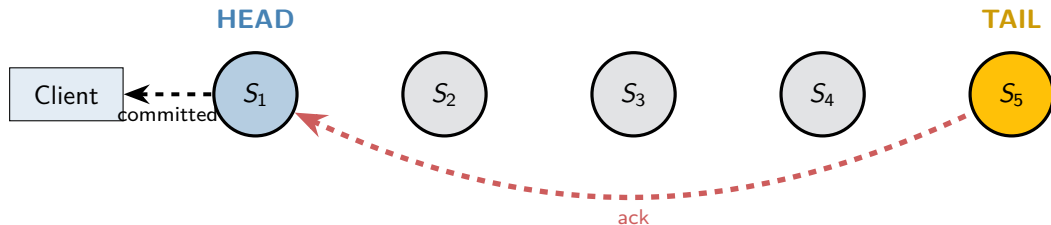


Write path: Client $\rightarrow$ Head $\rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow$ Tail

Every node applies the write to its local state, then forwards.

The tail is the last to see the write. That's the point.

The Ack Path



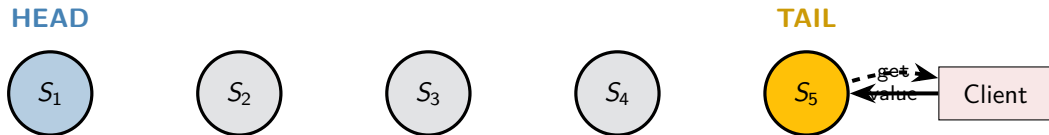
Commit path: Tail sends ack $\rightarrow$ Head $\rightarrow$ Client

When does the write become visible?

Only after the tail has applied it and acknowledged.

The tail is the commit point. Nothing is “chosen” until the tail says so.

Reads Go to the Tail



All reads go to the tail.

The tail has only committed writes—it never holds a “pending” value.

Consistency guarantee:

- Reads see exactly the committed state
- Writes are serialized through the chain
- Read-after-write: if write is committed, tail has it

Strong consistency. No voting. One round-trip for reads.

Checkpoint 1

In Paxos, writes contact a quorum (multiple nodes in parallel).

In chain replication, writes propagate sequentially through the chain.

Which has lower write latency? Which has higher write throughput?

What about read latency?

(Pairs: 2 minutes)

The Trade-off

Paxos:

- Write latency ≈ 2 round-trips (Prepare + Accept), parallel
- Reads: must ensure serialization (leader lease or read quorum)

Chain replication:

- Write latency $\approx \text{chain length} \times \text{hop latency}$ (sequential)
- Reads: one hop to tail
- But: write throughput can be high (pipelining!)

Pipelining: The head can send write w_2 before w_1 reaches the tail.

Every node in the chain is always busy forwarding some write.

Latency is worse. Throughput can be comparable or better.

Why Is This Strongly Consistent?

Safety argument:

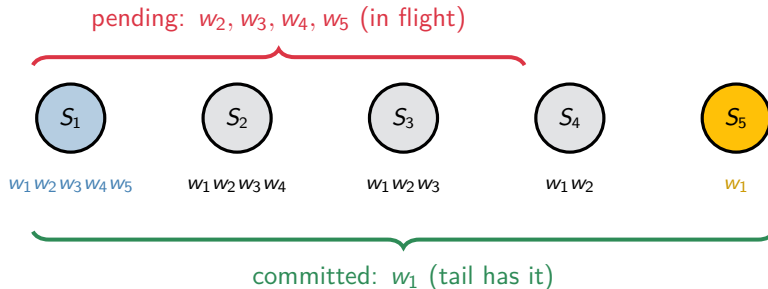
- ① All writes enter through the head—**total ordering by construction**
- ② Each node applies writes in the same order (the chain is FIFO)
- ③ The tail is the commit point—**only committed writes are visible**
- ④ Reads go to the tail—**readers see exactly the committed prefix**

Compare to Paxos:

- Paxos needs ballots to establish ordering among competing proposals
- Chain replication has no competition—the head is the sole entry point

Structure provides what voting provides: a total order on updates.

The Chain Invariant



Invariant: Each node's state is a *prefix* of its predecessor's state.

$$\text{state}(S_5) \subseteq \text{state}(S_4) \subseteq \text{state}(S_3) \subseteq \text{state}(S_2) \subseteq \text{state}(S_1)$$

The head has the longest prefix. The tail has the shortest (committed) prefix.

The “pending suffix” shrinks as you move toward the tail.

Now Break It

“Structure replaces voting.” Fine.

What happens when a node in the chain dies?

Three cases, each different:

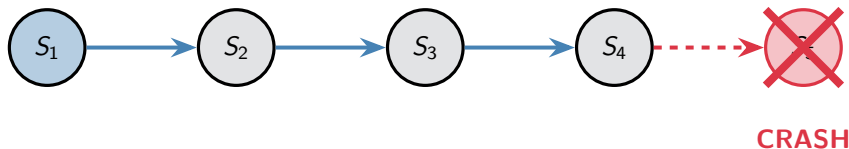
- ① Head failure
- ② Tail failure
- ③ Middle node failure

We need an external service to detect failures and reconfigure the chain.

Van Renesse & Schneider call this the **master** (or **configuration service**).

In practice: Paxos or Raft manages the chain membership.

Case 1: Tail Failure

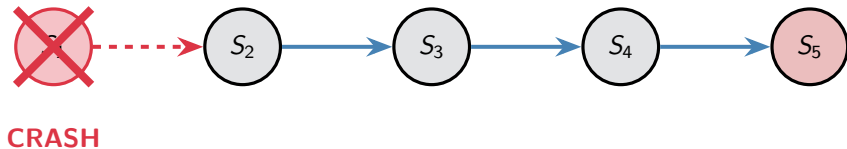


Recovery: S_4 becomes the new tail.

- S_4 's state is a **superset** of S_5 's state (prefix invariant)
- Any write that S_5 committed, S_4 already has
- Writes that S_4 has but S_5 didn't are now committed (they were pending)

Tail failure is the simplest case: promote the predecessor.

Case 2: Head Failure

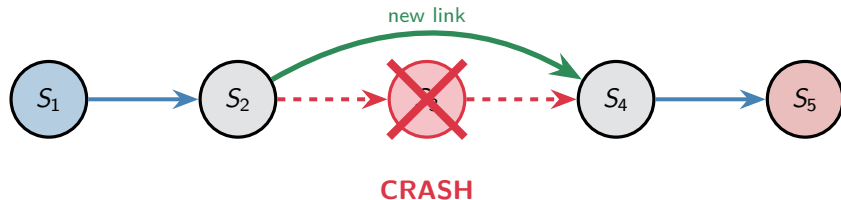


Recovery: S_2 becomes the new head.

- Writes that S_1 accepted but hadn't forwarded to S_2 : **lost**
- But those writes were never committed (tail never saw them)
- Client never received an ack $\rightarrow$ client can retry

Uncommitted writes can be lost. That's fine—the client knows.

Case 3: Middle Node Failure



Recovery: Link S_2 directly to S_4 . Skip the dead node.

Problem: S_2 may have writes that S_3 received but hadn't forwarded to S_4 .

- S_2 's state is a superset of S_4 's state (prefix invariant)
- S_2 re-sends the "gap" writes to S_4
- Once S_4 catches up, the chain is whole again

Middle failure is the hardest: we must close the gap.

Checkpoint 2

S_3 crashes. Writes w_7 and w_8 were in S_3 but hadn't reached S_4 .

Were w_7 and w_8 committed?

Does S_2 have them?

What does S_2 need to do?

(Pairs: 2 minutes)

The Role of the Configuration Service

The chain itself handles *data flow*.

But **who decides the chain membership?**

- A **configuration service** (master) monitors chain health
- Detects failures via heartbeats/timeouts
- Publishes the new chain configuration
- Nodes reconfigure themselves accordingly

What is this configuration service?

Typically: a Paxos or Raft cluster.

Chain replication doesn't eliminate consensus. It limits consensus to configuration changes—a rare event.

The Read Bottleneck

In vanilla chain replication, all reads go to the tail.

The tail is a bottleneck for reads.

5-node chain: 4 nodes do no read work at all.

For a read-heavy workload, most of your hardware is idle on reads.

Can we spread reads across the chain without losing strong consistency?

CRAQ: Chain Replication with Apportioned Queries

Terrace & Freedman, 2009

Key idea: allow reads from *any* node, with a version check.

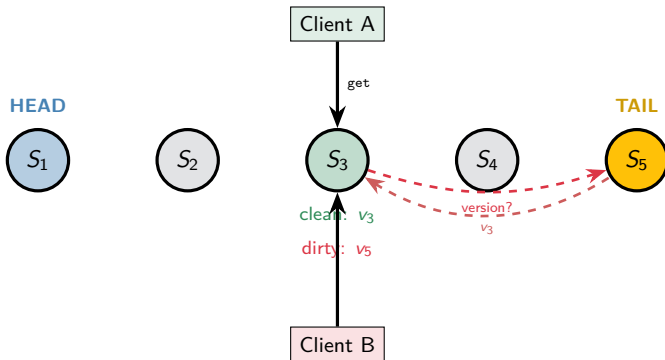
Each node stores two kinds of state for each object:

- **Clean version:** the latest version known to be committed
- **Dirty version:** a newer version that may or may not be committed yet

Read logic at node S_i :

- 1 If the latest version is **clean**: return it immediately
- 2 If the latest version is **dirty**: ask the tail for the latest committed version number, then return that version

CRAQ: Read From the Middle



Client A: Latest is clean $\rightarrow$ return immediately. **Client B:** Latest is dirty $\rightarrow$ ask tail for committed version $\rightarrow$ return that.

Most reads are clean and served locally. Dirty reads add one hop.

CRAQ: When Does a Version Become Clean?

A write propagates down the chain:

- 1 Head receives write, marks it **dirty**, forwards to S_2
- 2 S_2 stores it **dirty**, forwards to S_3
- 3 ...
- 4 Tail receives write, marks it **clean** (committed)
- 5 Tail sends ack back up the chain
- 6 Each node, on receiving the ack, marks that version **clean**

In steady state with no writes in flight: all versions are clean.

During a burst of writes: nodes near the head have dirty versions.

Dirty versions are transient. The ack wave cleans them up.

Checkpoint 3

In CRAQ, a client reads from S_2 . The version is dirty.

S_2 asks the tail for the committed version and gets v_7 .

Between the time S_2 asks and the tail responds, a new write w_8 commits.

Is S_2 's response (v_7) still consistent?

What consistency model does this give us?

(Pairs: 3 minutes)

CRAQ Consistency

Is returning v_7 safe when v_8 exists?

Yes. The read was linearized at the point the tail answered.

v_7 was the committed version at that instant.

v_8 committed *after* the version check—it's a concurrent write.

CRAQ provides strong consistency (linearizability):

- Every read returns a value that was committed at some point during the read
- Writes are totally ordered through the chain
- The tail is the serialization point

The tail is still the authority. CRAQ just avoids asking it when it's unnecessary.

Lattice Placement

From L10–L14: Quorum intersection → ballot ordering → consensus.

Chain replication takes a different path:

- No quorums: topology determines who participates
- No ballots: the head is the sole entry point (ordering by construction)
- No voting: sequential propagation replaces parallel agreement

Same safety goal: all replicas agree on the order of updates.

Different mechanism: structure instead of voting.

Cost of this approach:

- Chain requires a configuration service (still needs consensus!)
- Write latency scales with chain length
- Failure recovery is topology-dependent

Chain replication doesn't replace consensus. It moves consensus to configuration, where it runs rarely.

In-Class Activity

Setup: A 5-node chain: $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5$.

Writes w_1 through w_6 have been sent. Current state:

	S_1	S_2	S_3	S_4	S_5
State	w_1-w_6	w_1-w_5	w_1-w_4	w_1-w_2	w_1

Now: S_3 crashes.

- (a) Which writes are committed? Which are pending?
- (b) What messages are in flight that will never arrive?
- (c) Draw the recovery: what does S_2 send to S_4 ?
- (d) After recovery, what is the new committed state?

(Groups of 3: 10 minutes. Draw on paper. Then present.)

Activity Debrief

(a) Committed writes: Only w_1 —only writes the tail (S_5) has.

(b) In-flight messages lost:

- $S_3 \rightarrow S_4$: write w_3 (and possibly w_4) in transit
- $S_2 \rightarrow S_3$: write w_5 in transit (but S_3 is dead)

(c) Recovery: S_2 links directly to S_4 .

- S_2 has w_1 – w_5 . S_4 has w_1 – w_2 .
- S_2 sends w_3, w_4, w_5 to S_4 (closing the gap)
- Once S_4 has caught up, normal forwarding resumes

(d) After recovery: New chain $S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5$.

As writes propagate to the new tail, committed state grows.

w_6 is safe in S_1 . It will propagate through $S_2 \rightarrow S_4 \rightarrow S_5$ and commit.

Evidence Question

You're running a chain-replicated key-value store.

A client reports: "My writes are succeeding, but reads return stale data."

Where in the system would you look?

What could cause this in vanilla chain replication? In CRAQ?

(Hint: think about what happens if the configuration service has a stale view.)

What We Learned

- **Not everything needs voting.** A chain topology provides total ordering by structure.
- **Writes:** head $\rightarrow$ tail (sequential propagation, pipelined throughput)
- **Reads:** tail only (vanilla) or any node with version check (CRAQ)
- **Commit point:** the tail. Nothing is visible until the tail has it.
- **Failure handling:** head, middle, tail are different cases; all rely on the prefix invariant
- **Configuration service:** chain replication moves consensus to membership management—a rare operation

The lattice connection:

L10–L14: safety through quorum intersection and ballot ordering.

L15: safety through sequential propagation and a single commit point.

Next: what does “consistent” even mean? L16 opens the consistency model landscape.

- **Reading for L16:** Dynamo (DeCandia et al., Amazon, 2007)
- **Optional:** “Consistency in Non-Transactional Distributed Storage” (Viotti & Vukolić, 2016)
- **Next lecture (L16):** Consistency models—what “consistent” means, and when you want less of it
- Self-assessment opens 13:00, due next class 11:30 (Pacific)
- Capstone milestone details on Canvas