(1)

- Distributed storage is a key abstraction
- What should be the interface?
- How should it work?

(3)

Why is it hard?

- High performance => sharding
- Many servers => constant faults
- fault tolerance => Replication
- Replication => Potential inconsistencies
- Consistency => network chit chat
  => lower performance

(1)

What do we mean by consistency?

- Ideally, same behavior as a single server
- writes one at a time (even if concurrent)
- reads latest write.

Consistency

(2)

* Don't want to talk about internals of storage
* Judge by behavior seen by clients.

Create an illusion of single node storage.

Example

(2)

C1: ⊢ wx10 ⊣

C2: ⊢ wx20 ⊣

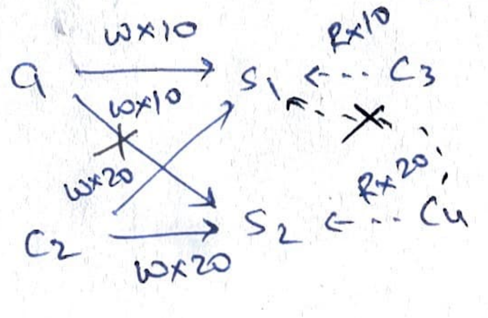
C3: ⊢ Rx? ⊣

C4: ⊢ Rx? ⊣

- Either 10 or 20, but same
- Single server has poor FT/scalability

Bad replication design

(1)



- C_1 crashes before sending to S_2
- S_1 crashes after C_3 reads, before C_4 reads.

Suddenly x=20.

Consistency problems

C_1 ⊢ $W_x 10$ ⊣

C_2 ⊢ $W_x 20$ ⊣

⊢ $R_x 10$ ⊣ C_3

⊢ $R_x 20$ ⊣ C_4

$C_1 \rightarrow S_1 \rightarrow C_3$

C_2 ⤨ $S_2 \rightarrow C_4$

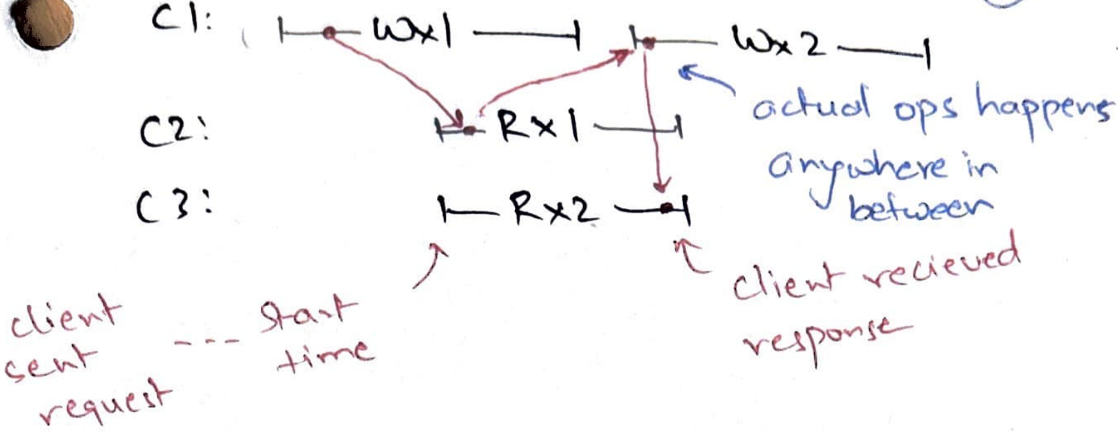
CR Goals -

- ~~Multi...~~ FT Key value store
- Data fits in memory / disk
- Many concurrent clients
- Consistent view of storage.

→ we want that all execution histories are linearizable.

- One can find a total order of all ops
  - * matches real-time for non-overlapping ops
  - * read sees last write

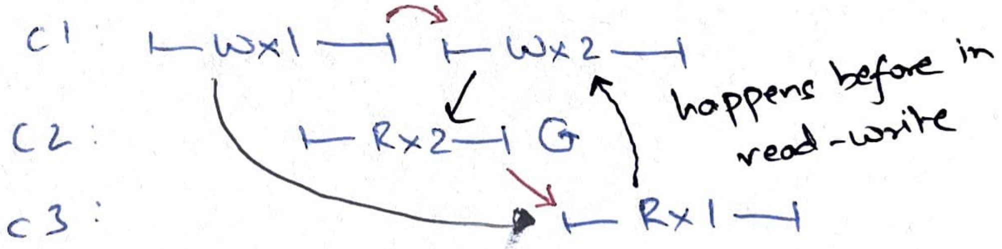
4



⇒ linearizable

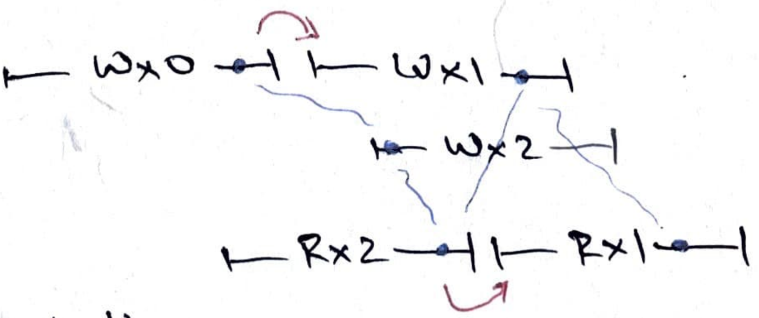
4

happens before in real time



cycle ⇒ NOT linearizable

4



⇒ linearizable

In concurrent writes, write that started later can be executed first.

Bad things never happen

- Linearizability is a safety property

C1: ⊢— Rx — . . . .

C2: ⊢— Wx —— . . . .

} Trivially satisfied by a storage system that does nothing!

=) We also want liveness (even w/ faults)

↳ Good things eventually happen

- Liveness ②

  - Good things eventually happen.

Research. safety-

* will not publish incorrect results/ low quality research

Liveness

* will eventually publish

- Safety, liveness in concurrency. ②

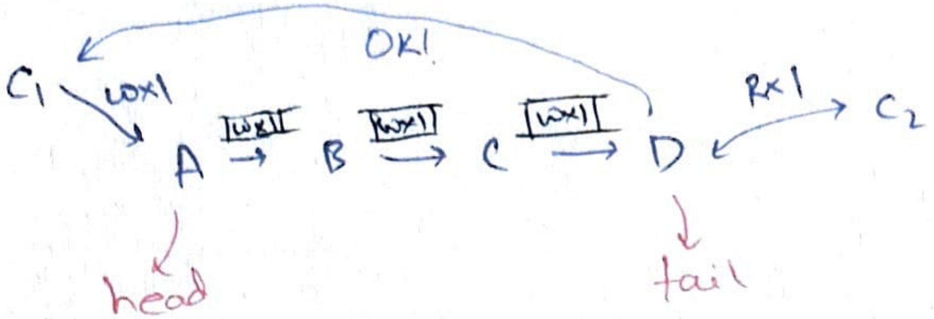
Safety:

- Mutual exclusion
- No deadlocks

Liveness

- Each process will eventually get the shared resource (starvation freedom)

- Chain Replication



Clearly linearizable: in terms of tail server

- Why TLA+ (Temporal logic of actions)

- As storage systems grow in complexity, reasoning that they behave like a single system becomes very difficult
- Adding optimizations to existing systems often can break guarantees.

- Days of debugging can save hours of writing specifications.

- TLA+ lets us formally prove than one system implements another. Eg. CR implements single machine storage (linearizable)

- Prove safety / liveness

- BIG IDEA:

TLA+ helps convert implementation into mathematical implication

$CR \Rightarrow SS$

- * First 1/3rd of course
  - Learn TLA+. Prove that CR implements SS.
  - Very beautiful concepts - stuttering, refinement, actions, ...
- * Second 1/2
  - Discuss distributed storage systems.

Groups of 2.