Fault Tolerance (Again?)

Zhaoguo Wang

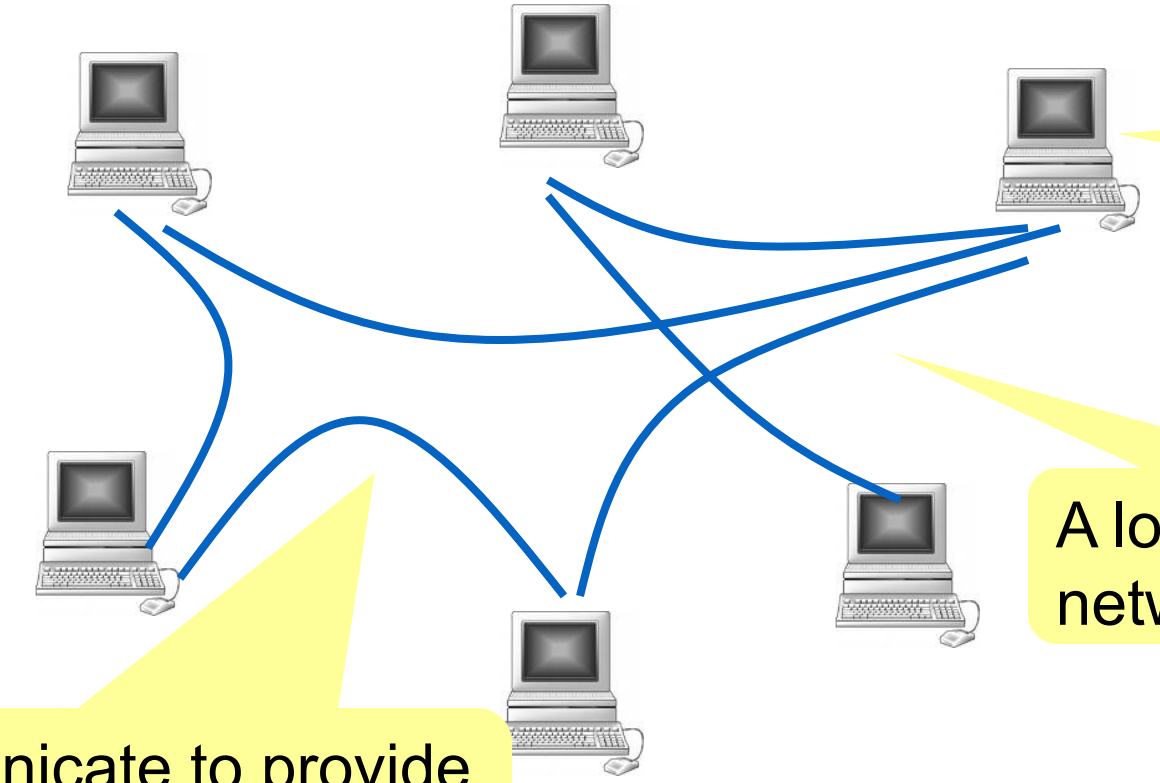
Some slides are adapted from Prof. Jinyang Li’s DS class

CSE – Distributed System Part

CSE – Distributed System Part

What are Distributed Systems?

What are Distributed Systems?



Multiple hosts

A local or wide area network

Machines communicate to provide some service for applications

CSE – Distributed System Part

Why Distributed System?

Why distributed systems? for ease-of-use

Handle geographic separation

Provide users (or applications) with location transparency

Examples:

- Web: access information with a few “clicks”
- Network file system: access files on remote servers as if they are on a local disk, share files among multiple computers

Why distributed systems? for availability

Build a reliable system out of unreliable parts

- Hardware can fail: power outage, disk failures, memory corruption, network switch failures…
- Software can fail: bugs, mis-configuration, upgrade …
- How to achieve 0.99999 availability?

Examples:

- Amazon's S3 key-value store

Why distributed systems?
for scalable capacity

Aggregate the resources (CPU, bandwidth, disk) of many computers

Examples?

- CPU: MapReduce, Spark, Grid computing
- Bandwidth: Akamai CDN, BitTorrent
- Disk: Google file system, Hadoop File System

Why distributed systems?
for modular functionality

Only need to build one service to accomplish one single task well.

- Authentication server
- Backup server.

Compose multiple simple services to achieve sophisticated functionality

- A distributed file system: a block service + a meta-data lookup service

The downside

Much more complex

> “A distributed system is a system in which I can’t do my work because some computer that I’ve never even heard of has failed.”
>
> -- Leslie Lamport

#1 Abstraction & Interface

What kind of API to offer applications?

An example, a storage service's API:

- File system: mkdir, readdir, write, read
- Database: create tables, select, insert...
- Disk: read block, write block
- Key-value store: put(key, value), get(key)

Conflicting goals:

- Simple to use
- Flexible (Suitable for many applications)
- Efficient to implement

#2 System architecture

Where to run the system?

- Data center or wide area?

How is system functionality spread across machines?

- Peer-to-peer? Different nodes w/ different roles?
- A single point of management/coordination?

#3: Fault Tolerance

How to keep the system running when some machine is down?

Dropbox operates on ~10,000 servers, how to read files when some are down?

- Replication: store the same data on multiple servers.
- Does the system still give “correct” data?

#4: Consistency

Contract with apps/users about meaning of operations.

Difficult due to:

- Failure, multiple copies of data, concurrency

E.g. how to keep 2 replicas “identical”

- If one is down, it will miss updates
- If net is broken, both might process different updates

#5 Performance

Latency & Throughput

To increase throughput, exploit parallelism

- Many resources exist in multiples
  - CPU cores, IO and CPU
- Need ways to divide the load

To reduce latency,

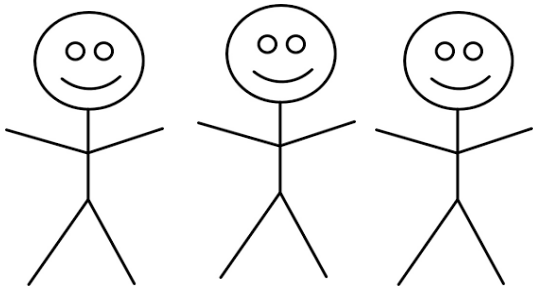
- Figure out what takes time: queuing, network, storage, some expensive algorithm, many serial steps?

More on performance: latency vs. throughput

Starbucks coffee shop

10 sec to make
a cup of coffee

1 sec to handle
a request



To get a cup of coffee:

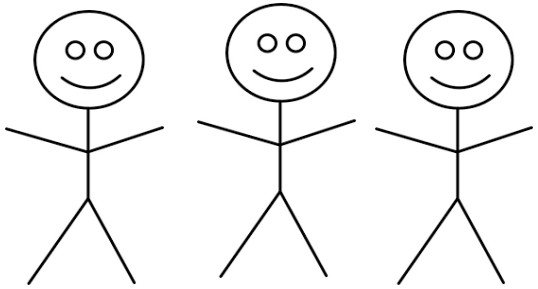
1. Cashier collects payment, writes down order
2. Barista makes coffee

More on performance: latency vs. throughput

Starbucks coffee shop

10 sec to make
a cup of coffee

1 sec to handle
a request



What's the minimal latency experienced by customer?

What's throughput if processing is sequential?

What's the best throughput when processing is parallelized?

CSE – Distributed System Part

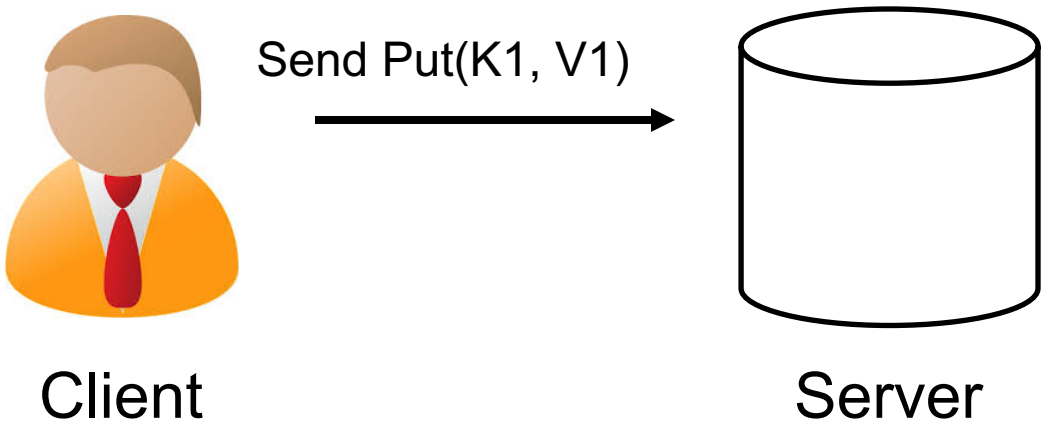
Part I: Fault Tolerance

Part II: Consistency

Part III: Distributed Transaction

A Simple Example – Key Value Store

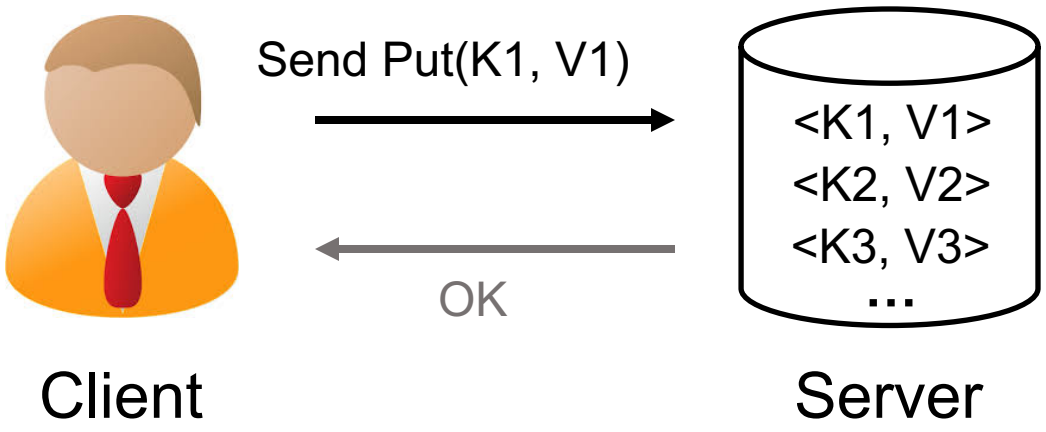
Interface: Put(Key, Value), Get(Key)



A client sends Put(K1, V1) to a server

A Simple Example – Key Value Store

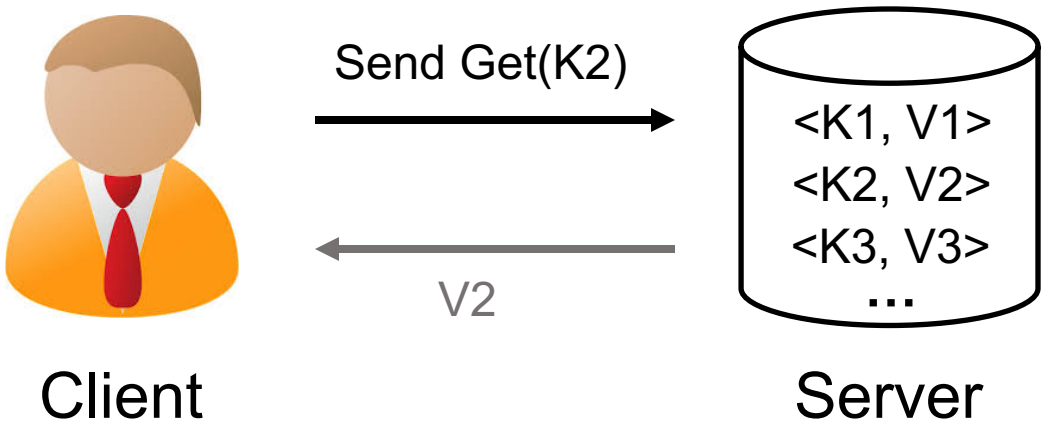
Interface: Put(Key, Value), Get(Key)



Server receives the request:
1. Record the tuple <K1, V1>
2. Reply the client with OK

A Simple Example – Key Value Store

Interface: Put(Key, Value), Get(Key)



Server receives the request:
1. Search tuple of K2
2. Reply the client with V2

CSE – Distributed System Part

Part I: Fault Tolerance

Part II: Consistency

Part III: Distributed Transaction

Why need to tolerate fault?

Particularly important at scale.

- Suppose a typical server crashes every month
- How often can some server crashes in a 10,000-server cluster?
  - 30*24*60/10000 = 4.3 minutes

Fault tolerance via replication

- One must replicate data on >1 servers.

Consistency: Correctness of replication

How to replicate data “correctly”?

Some informal notion of correctness:

(We’ll discuss formal notion in Part II)

- **Copies** of the same data should (eventually) **be the same**
- Replicated system should “**behave similarly” to its un-replicated system**

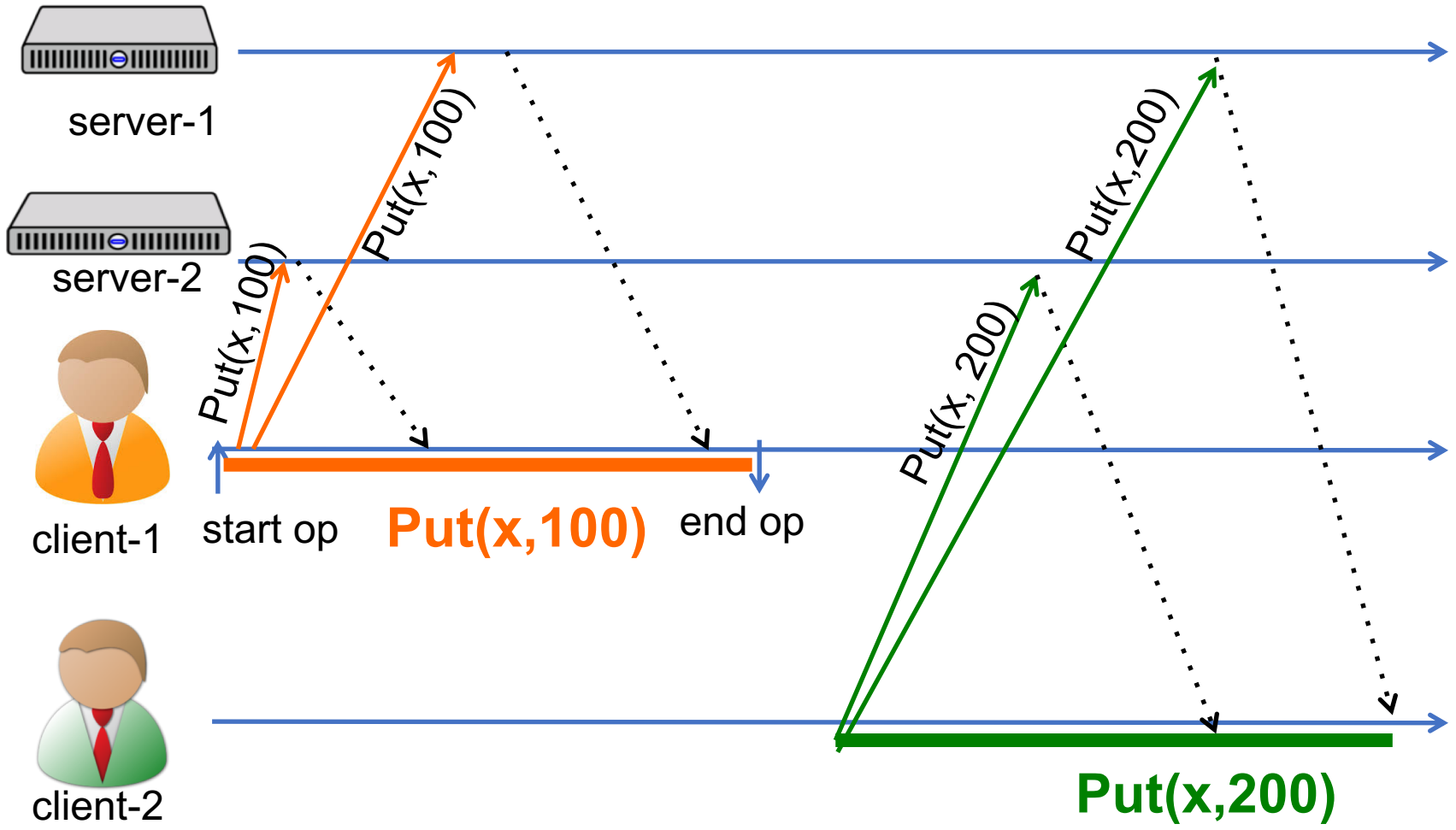
Challenges in achieving correctness

1. Concurrency
2. Machine failure
3. Network failure (e.g. network partition)

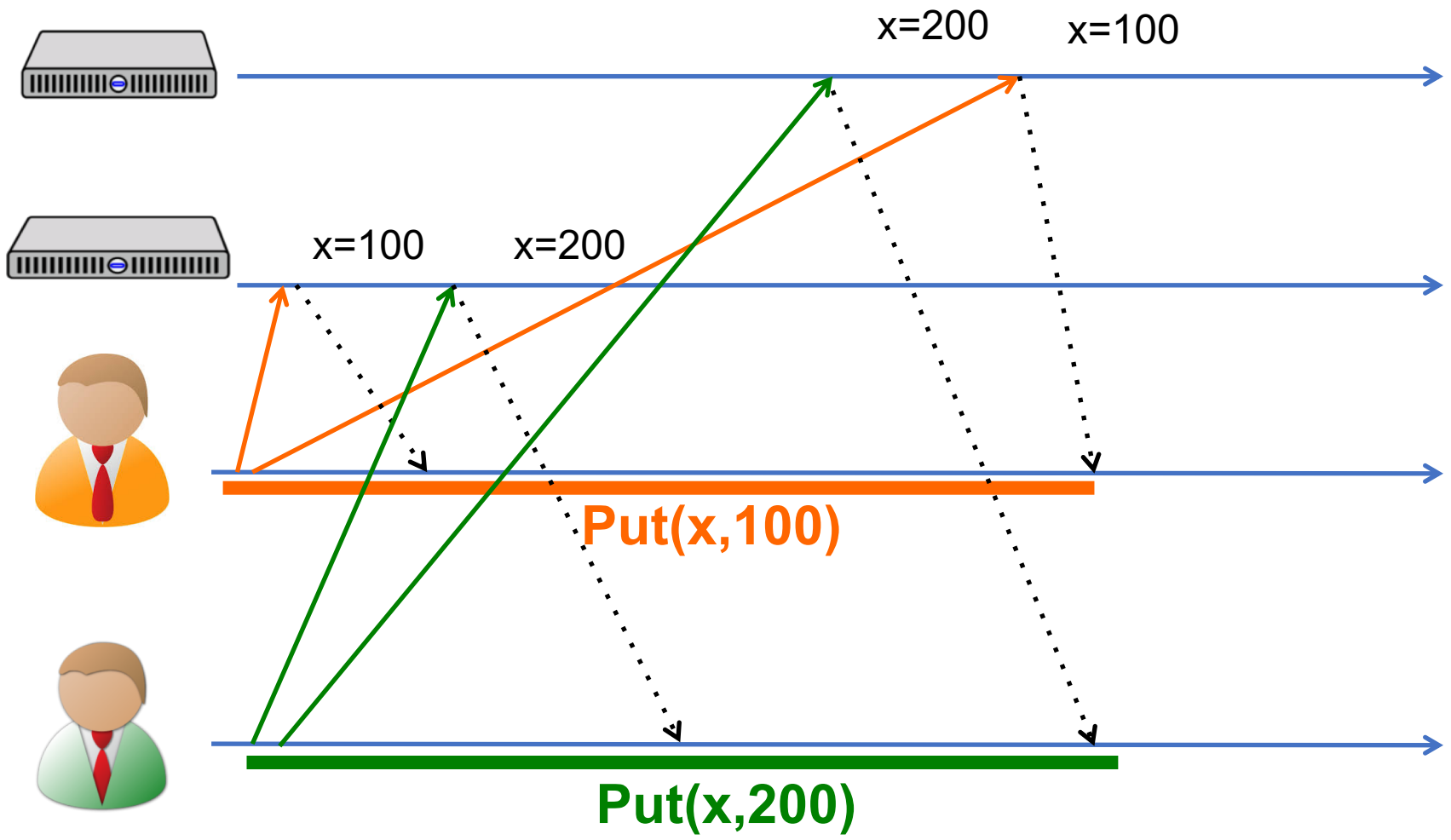
Particularly tricky because

- one might mistake “slowness” for “failure”
- one cannot tell 2 from 3.

Replication: a strawman



Strawman fails under concurrency

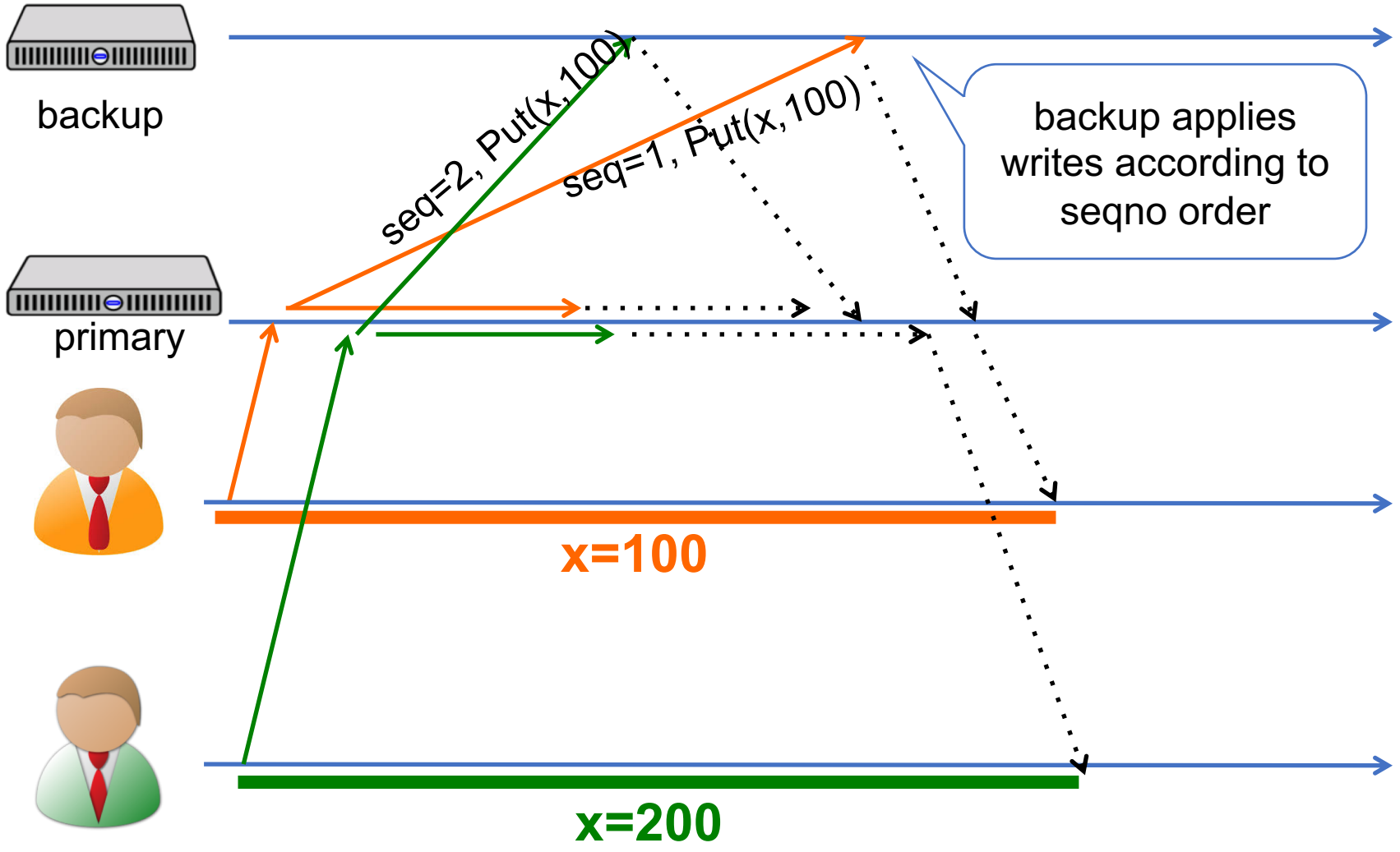


Order updates via primary

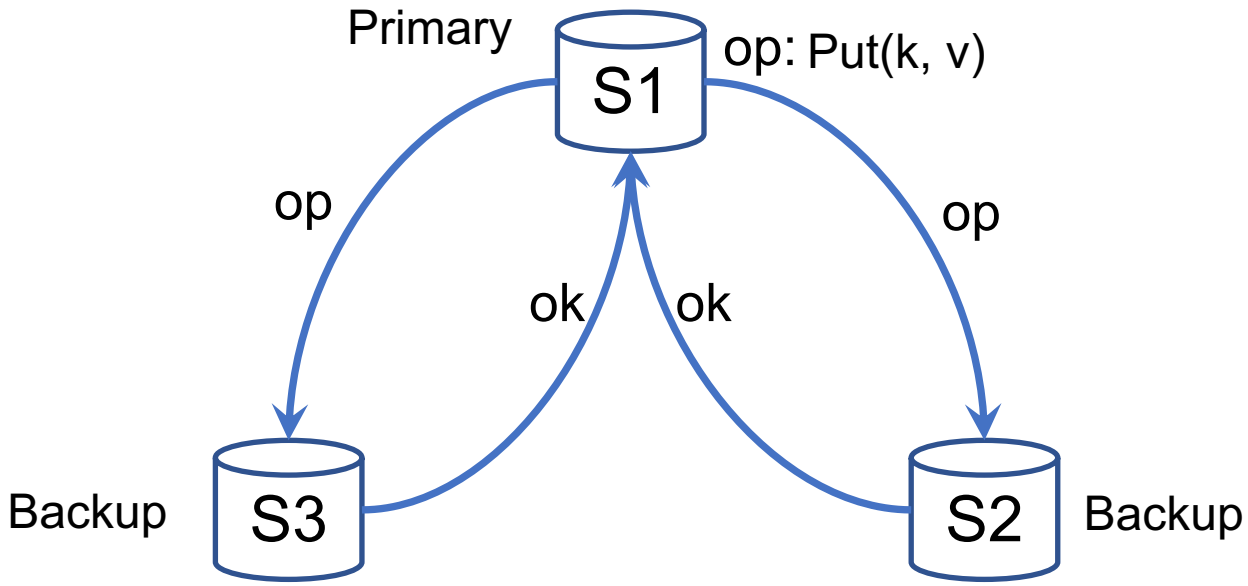
To keep replica in sync, writes must be done in the same order

Idea: use a designated server (primary) to determine the order of updates, others follow order

Primary determines order of updates

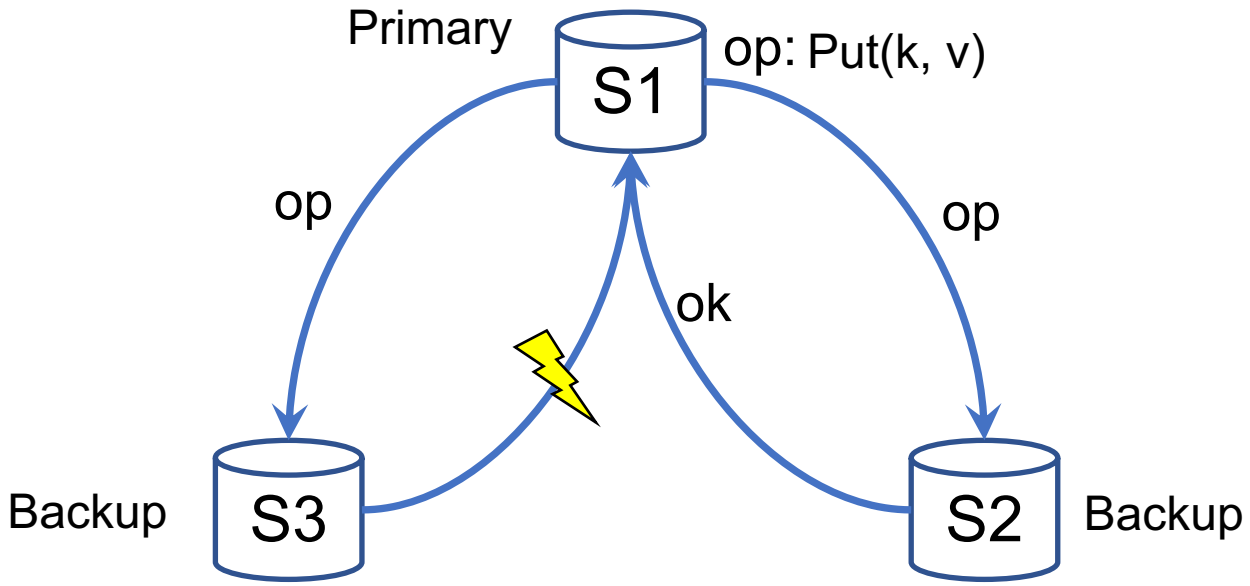


Challenges in handling failure



1. Receive op from client
2. Send op to all backups
3. Wait for all backups' reply
4. Execute op and return to client

Challenges in handling failure

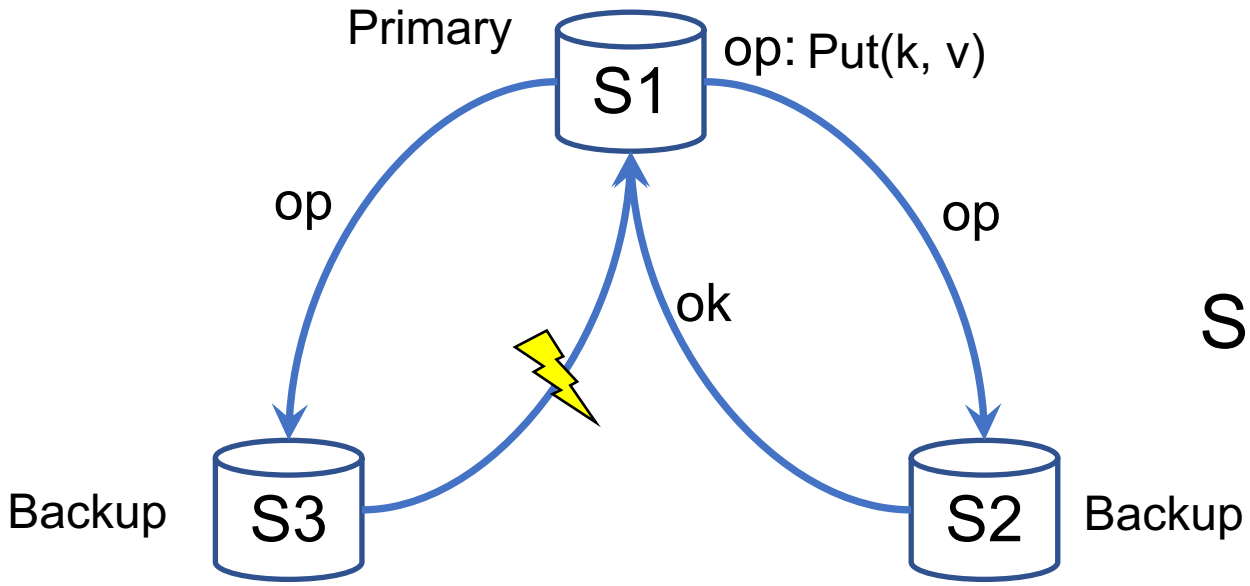


Challenge I:
What if a backup timed out
in acknowledging the replication?

Solution I: Primary re-tries?

Works only if backup failure is transient

Challenges in handling failure

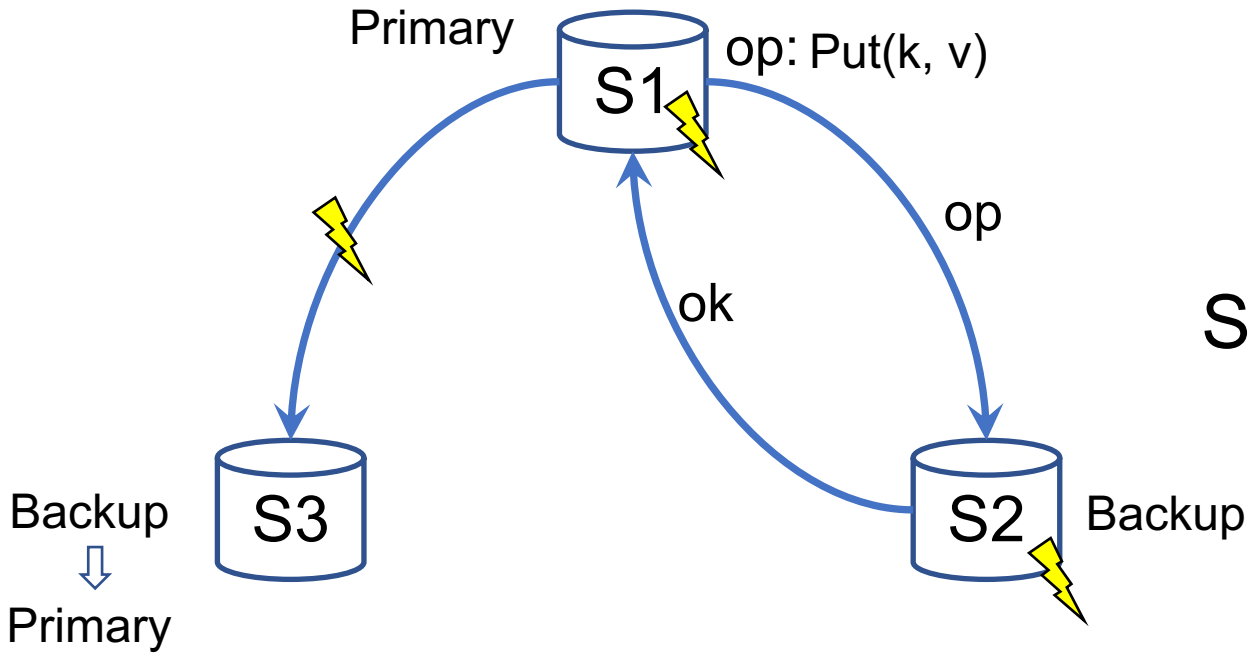


Challenge I:
What if a backup timed out
in acknowledging the replication?

Solution II: Ignore the failed replica

How many is safe to ignore?
How to make a backup catch up?

Challenges in handling failure

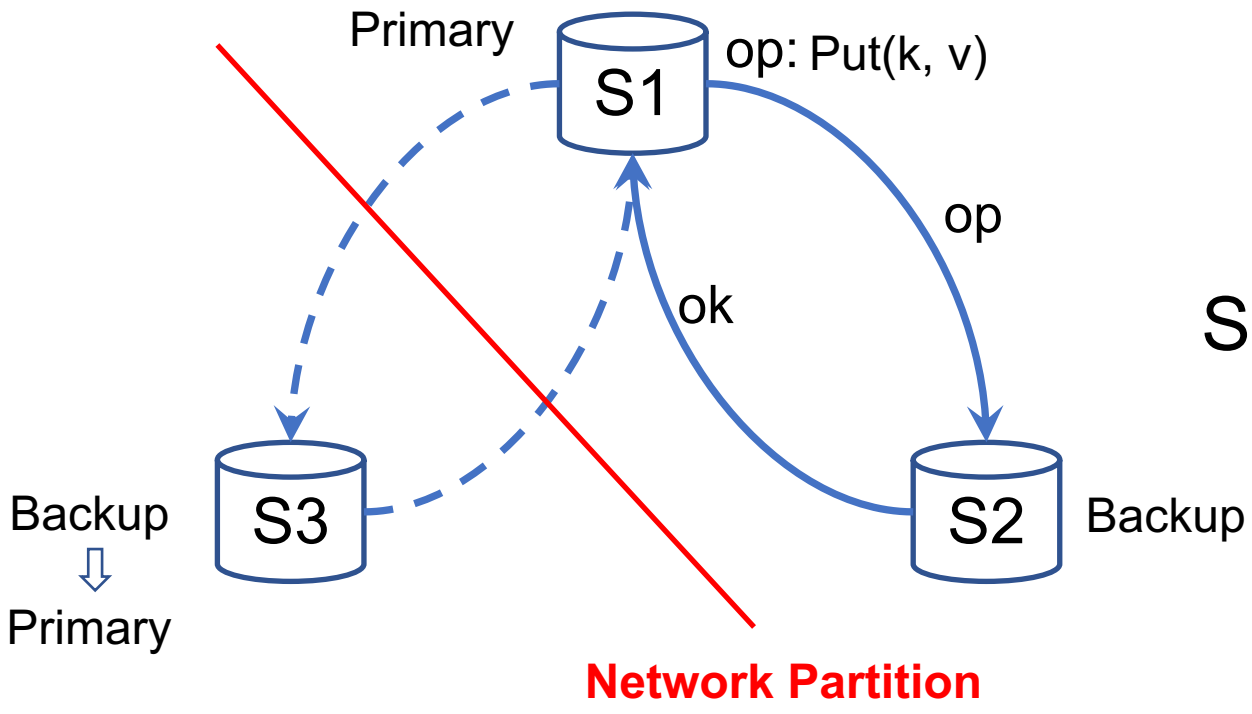


Challenge II:
What if a primary fails?

Solution: switch to another primary

How to ensure the finished op not lost after switch?

Challenges in handling failure



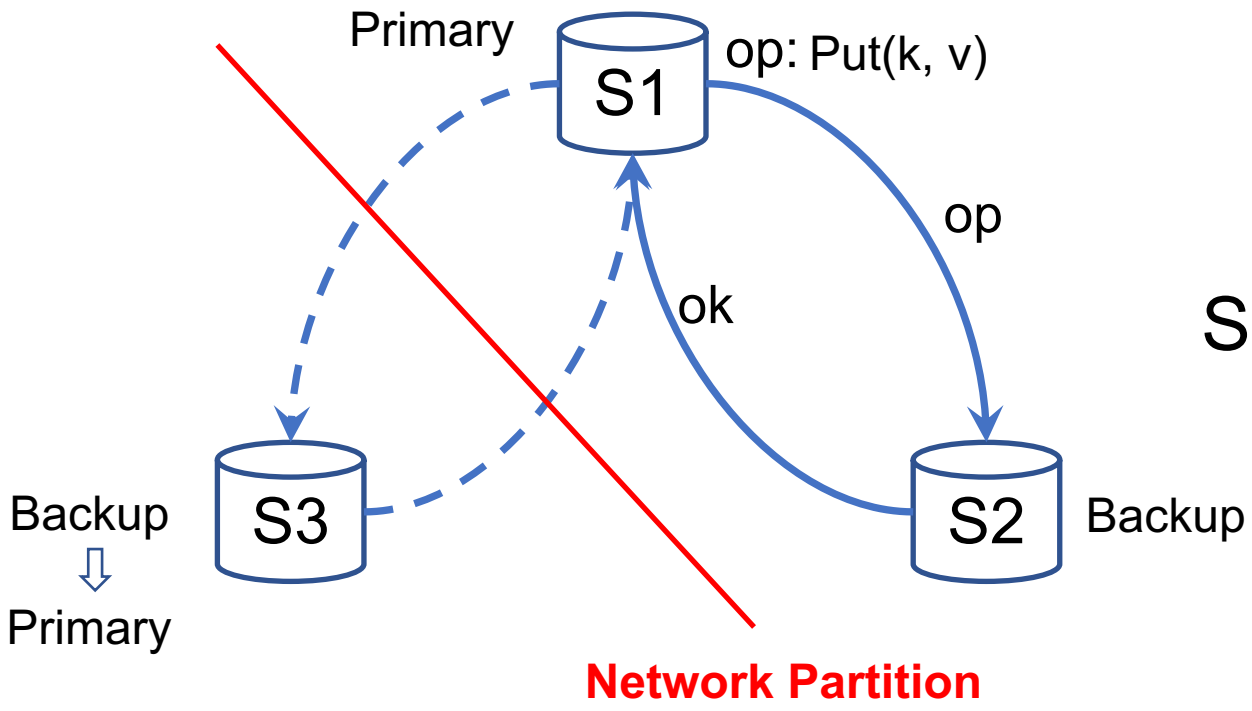
Challenge II:
What if a primary fails?

Solution: switch to another primary

How to ensure the finished op not lost after switch?

Could there accidentally be two validate primaries?

Challenges in handling failure



Challenge II:
What if a primary fails?

Solution: switch to another primary

How to ensure the finished op not lost after switch?

Could there accidentally be two validate primaries?

What to re-integrate a recovered server?

Failure handling: the stone age

For a long time, people do it manually

(with no guaranteed correctness)

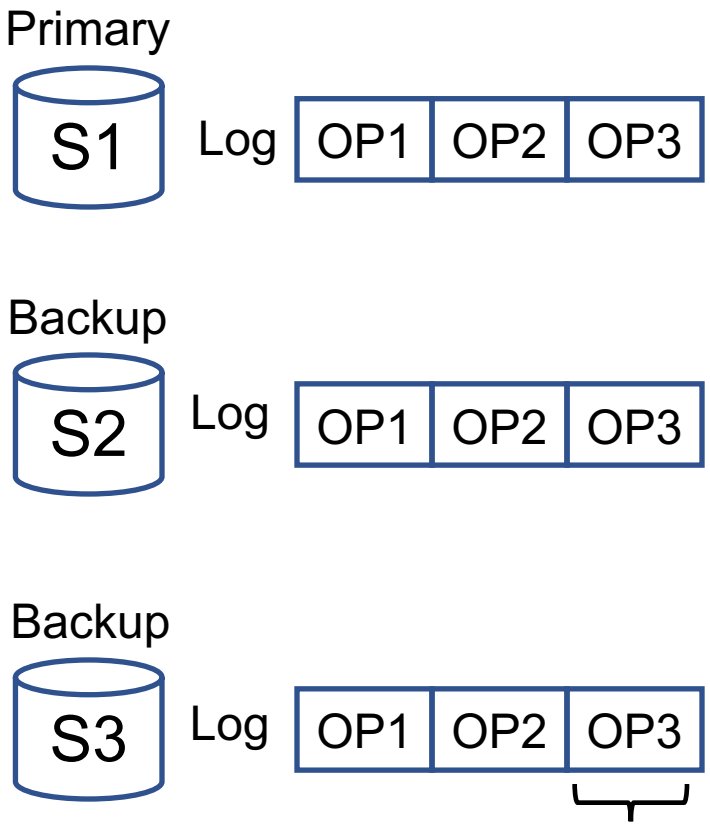
- One primary, one backup.
- Primary ignores temporary replication failure of a backup.
- If primary crashes, human operator re-configures the system to use the former backup as new primary
- some ops done by primary might be “lost” at new primary

Viewstamp replication

A landmark work: The first (together w. Paxos) to handle failure correctly.

- Original paper Oki and Liskov, 1988
- Viewstamp revisited: Liskov and Cowling, 2012

VR overview: state machine replication



Servers replicate a log of operations (instead of directly modifying state in-place)

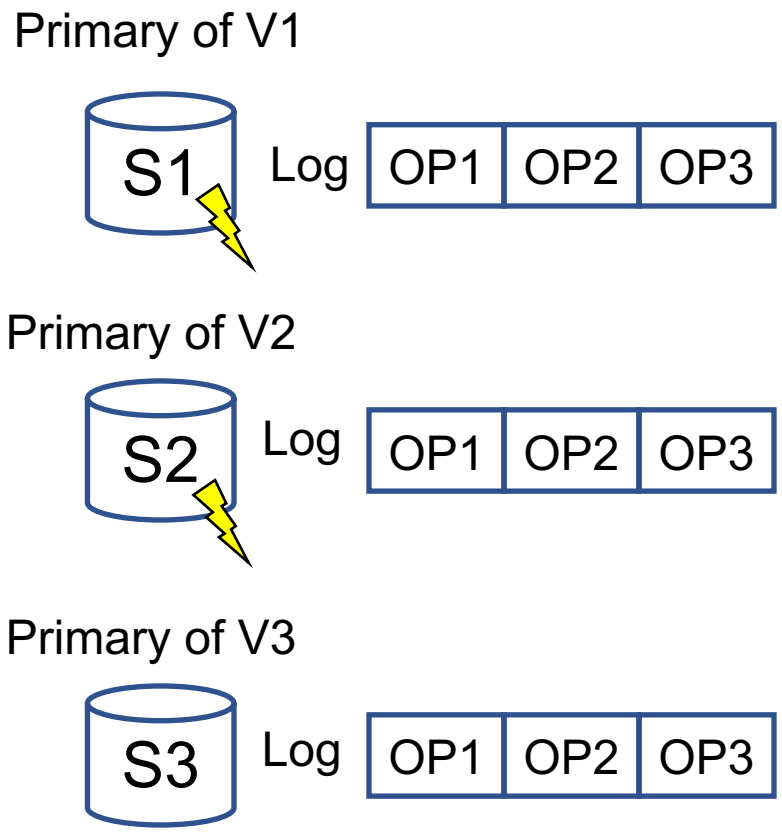
- Efficient for comparing state among servers, sync-ing out of date servers

Correctness: servers execute the same sequence of log

- Operations must be deterministic

General: a log of operations may be:
[put key=x, data="..."] [put key=y, data="..."] ...
[create /zhaoguo/x] [mv /zhaoguo/x, /zhaoguo/y]....
[update T set grade=10 where uid=123] [insert into T values ...] ...

VR overview: primary-backup



VR assumes a static configuration of servers, e.g. S1, S2, S3

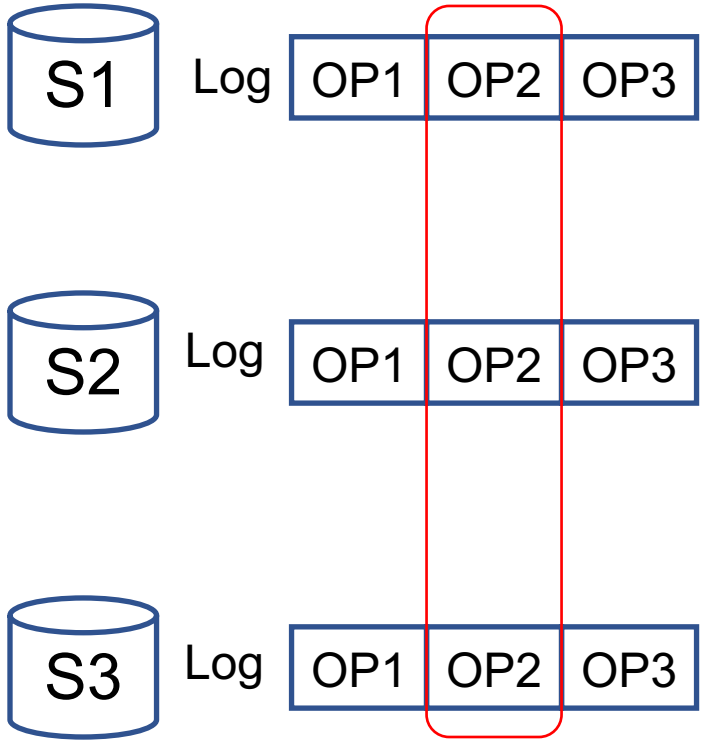
To handle primary failure, VR moves through a sequence of "views"

- Deterministic mapping from view-number to primary:
  primary = view-number % total_servers

VR correctness conditions

An op is “committed” if it is replicated by a threshold number of servers

- Once committed, an op’s position in log is fixed



Correctness

No two different ops are committed at the same log position

VR correctness conditions

An op is “committed” if it is replicated by a ~~threshold number~~ ***majority*** of servers (Quorum)

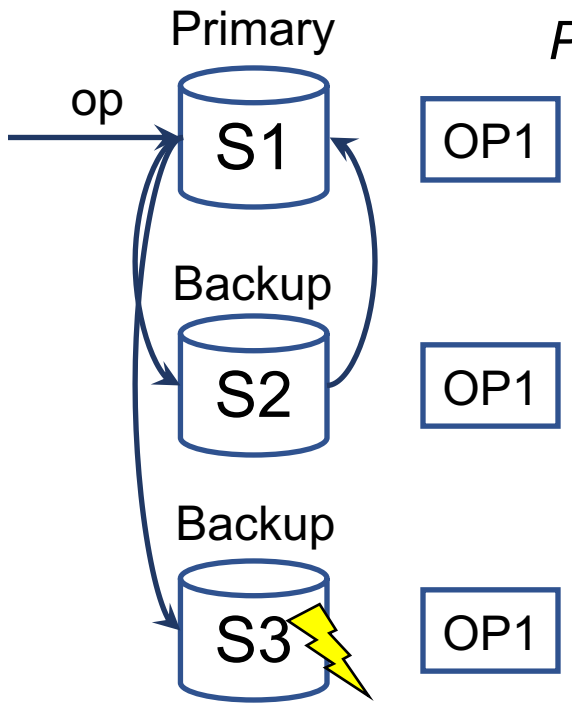
- Once committed, an op’s position in log is fixed

Correctness→

- No two different ops are committed at the same log position

Key mechanism for correctness

Key mechanism for correctness: quorum intersection



Primary considers the op committed.

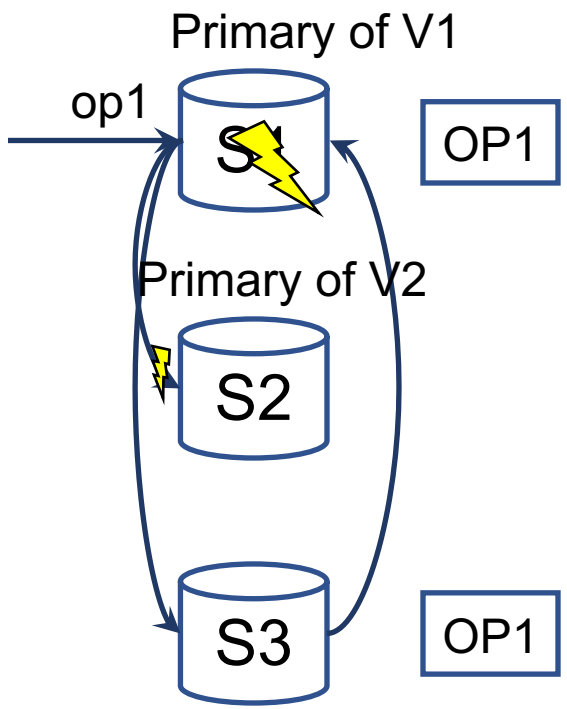
What is the quorum size? 2

How to prevent two primaries from committing different ops at same position?

Primary waits for quorum before considering an op committed.

- If backup is slow (or temporarily crashed), the primary can still commit as usual.

Key mechanism for correctness: quorum intersection



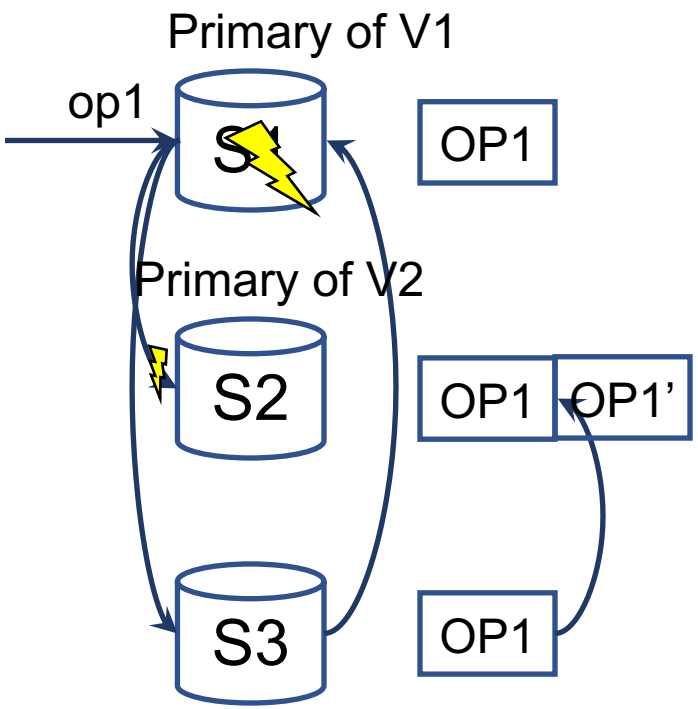
How to prevent two primaries from committing different ops at same position?

1. S1 becomes primary of V1
2. S1 commits *OP1* at pos=1 on S1, S3
3. S1 crash, S2 becomes primary of V2
4. S2 tries to commit *OP1'* at pos=1
5. S3 also will receive request *OP1'* at pos=1

Solution: S3 could ensure OP1 and OP1' are not replicated at the same position.

Correctness condition: all committed ops in view V - 1 must be known to primary in view v.

Key mechanism for correctness: quorum intersection



Correctness condition: all committed ops in view V - 1 must be known to primary in view v.

But how?

Solution: View v is only active after v's primary has learned the log state of majority of nodes (at earlier views).

S2 needs to learn log state form S3 before becoming V2's primary

Reference

- Liskov, Barbara, and James Cowling. "Viewstamped replication revisited." (2012).