Co-Designing Distributed Systems with Networks

CS5229 Guest Lecture

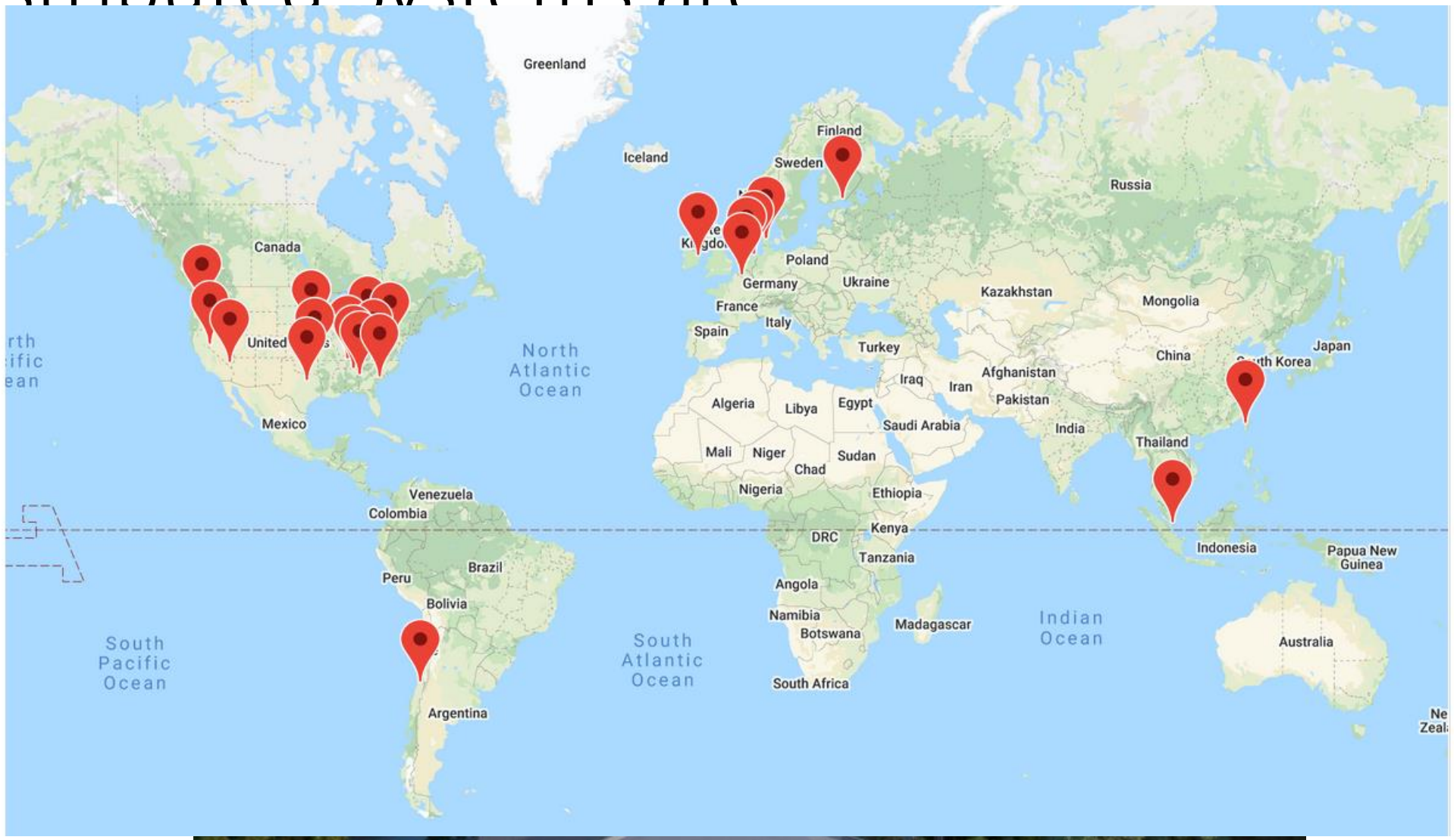
Jialin Li

NUS National University of Singapore | Computing

About me

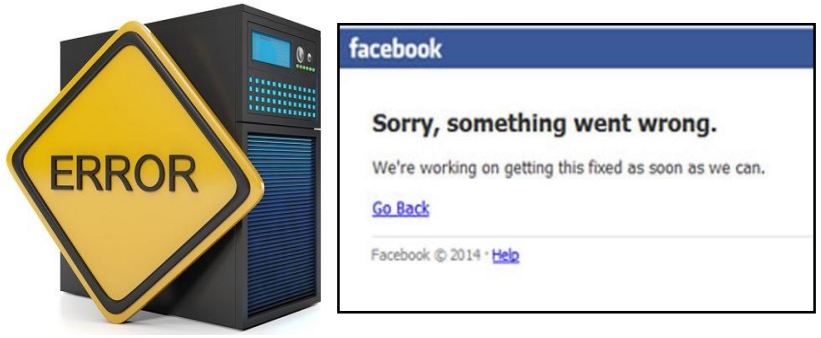
- B.S.E at the University of Michigan
- Ph.D. at the University of Washington
- Started faculty job at NUS in 2020
- Research interests: distributed systems, operating systems

Distributed Systems are E



Building Distributed Systems is **Challenging**

Remain **available** despite failures



Meet stringent **performance** requirements

Keep data strongly **consistent**

Failures and Availability

- Tens of thousands of servers
- Many types of failures
  - Power failures
  - Disk failures
  - OS crashes
  - Network failures
  - …
- At data center scale, failures happen all the time!
- Services need to remain available!

Facebook crippled by global outage across apps, employee systems

Downdetector said the Facebook outage is the largest it has seen, with more than 10.6 million reports worldwide. PHOTO: REUTERS

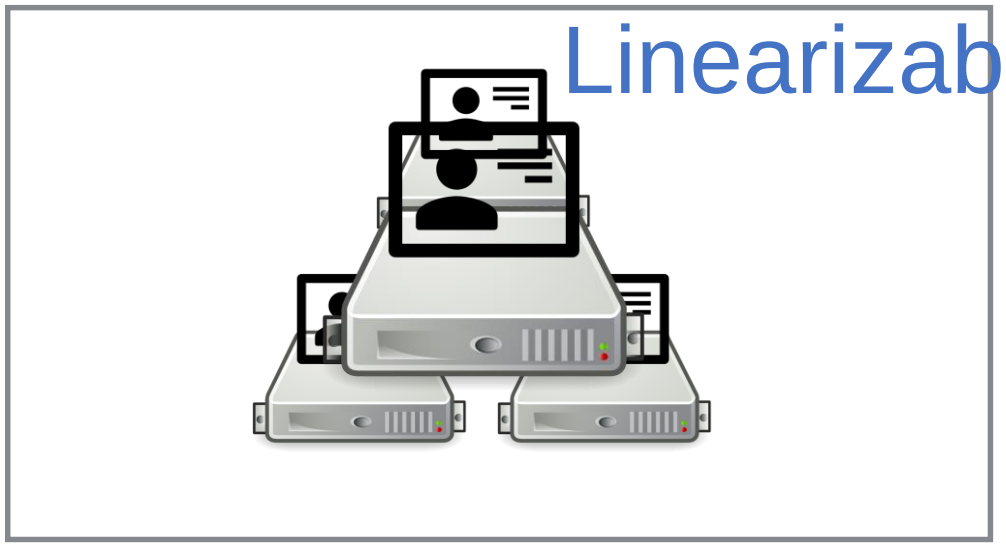
Consistency

- User data is distributed across servers
  - Caching
  - Scalability
  - Tolerating failures
- Need to maintain data **consistency** and **persistence**
  - Data loss
  - Unexpected behaviors
- Very challenging in the presence of failures!

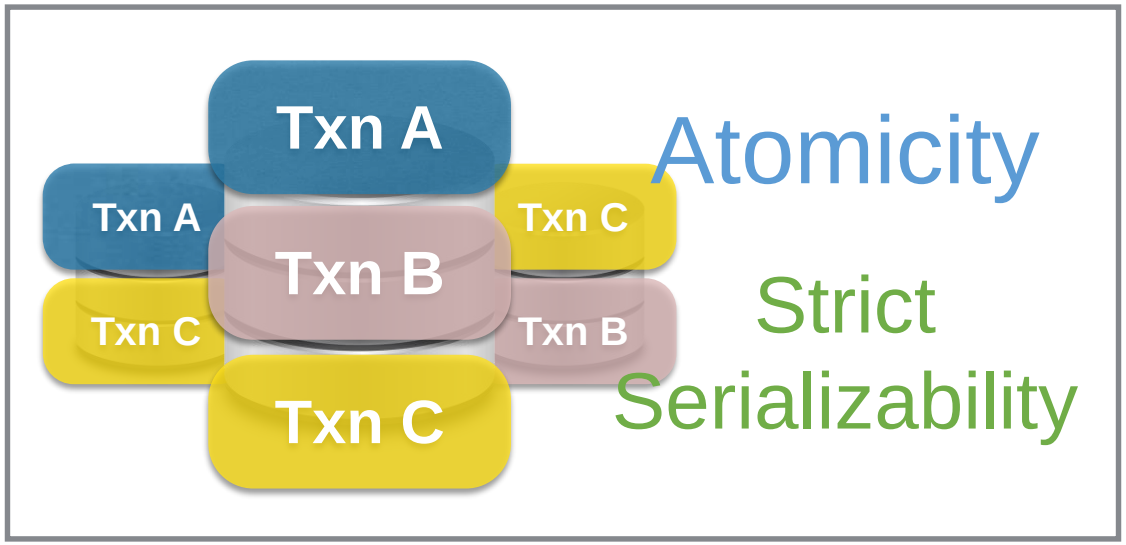
Performance is Critical!

- ⬆ service response time
  ⬇ revenue
- Each user request spawns into many sub-requests
- Response time dependent on the slowest subcomponent
- Tail latency matters!

We Build System Software to Provide **Fault Tolerance** and **Strong Consistency**



Fault-tolerant systems



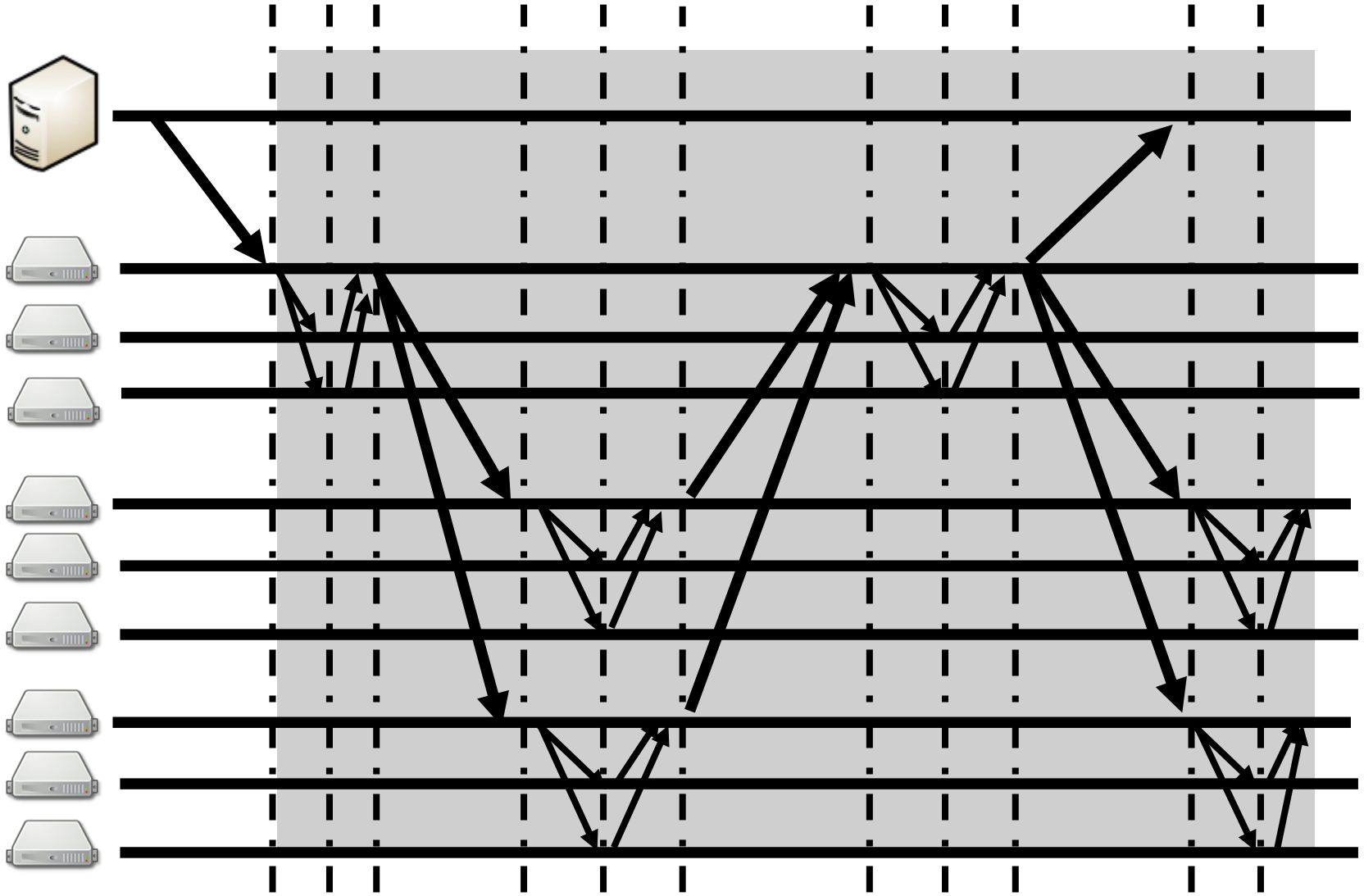
ACID transactions

But… High Performance Cost!

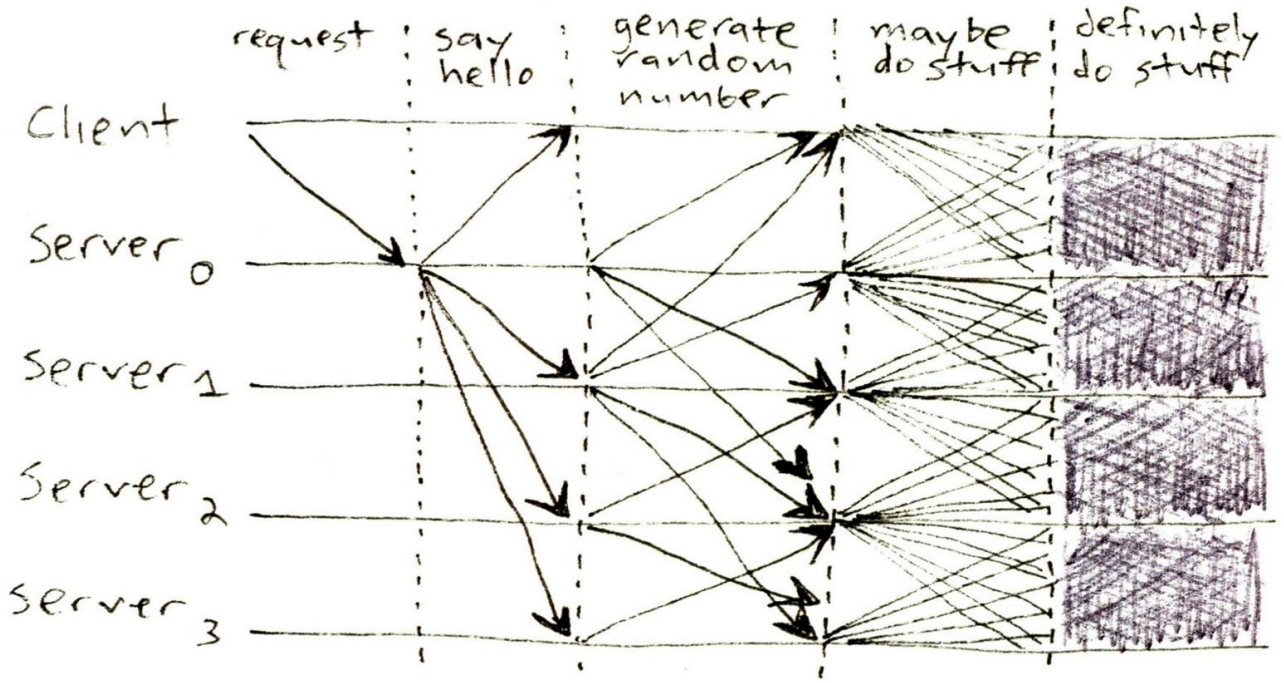
Fault-Tolerant

Distributed

Transactions



But… High Performance Cost!



[Mickens ’13]

Or… Weaker Guarantees for Better Performance

burden on application programmers

Amazon DynamoDB

We need **strong guarantees**!

Google Spanner

Can we build distributed systems with
both strong guarantees and **high performance**?

Existing distributed systems designs consider the network

as a **black box**

Current Network Model

Asynchrony: packets may
arbitrarily

- Dropped
- Reordered
- Duplicated

vers messages
rom A to B


SDN
Google | google.com/datacenters

Trend: data center switches becoming more **programmable**

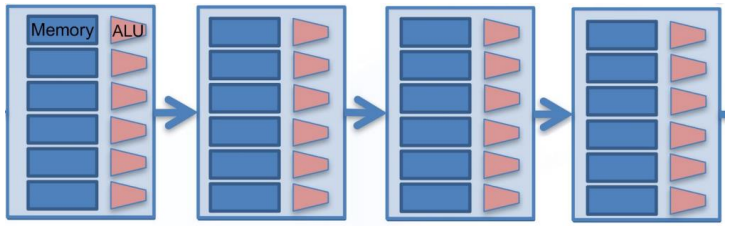
Programmabl
packet head

```
30 header_type ipv
31     fields {
32         versio
33         ihl : 
34         diffse
35         totalL
36         identi
37         flags
38         fragOf
39         ttl : 
40         protoc
41         hdrChec
42         srcAddr : 32;
43         dstAddr : 32;
44     }
45 }
```

urpose
emory

application-specific computation at **line rate**

Flexible packet processing pipeline

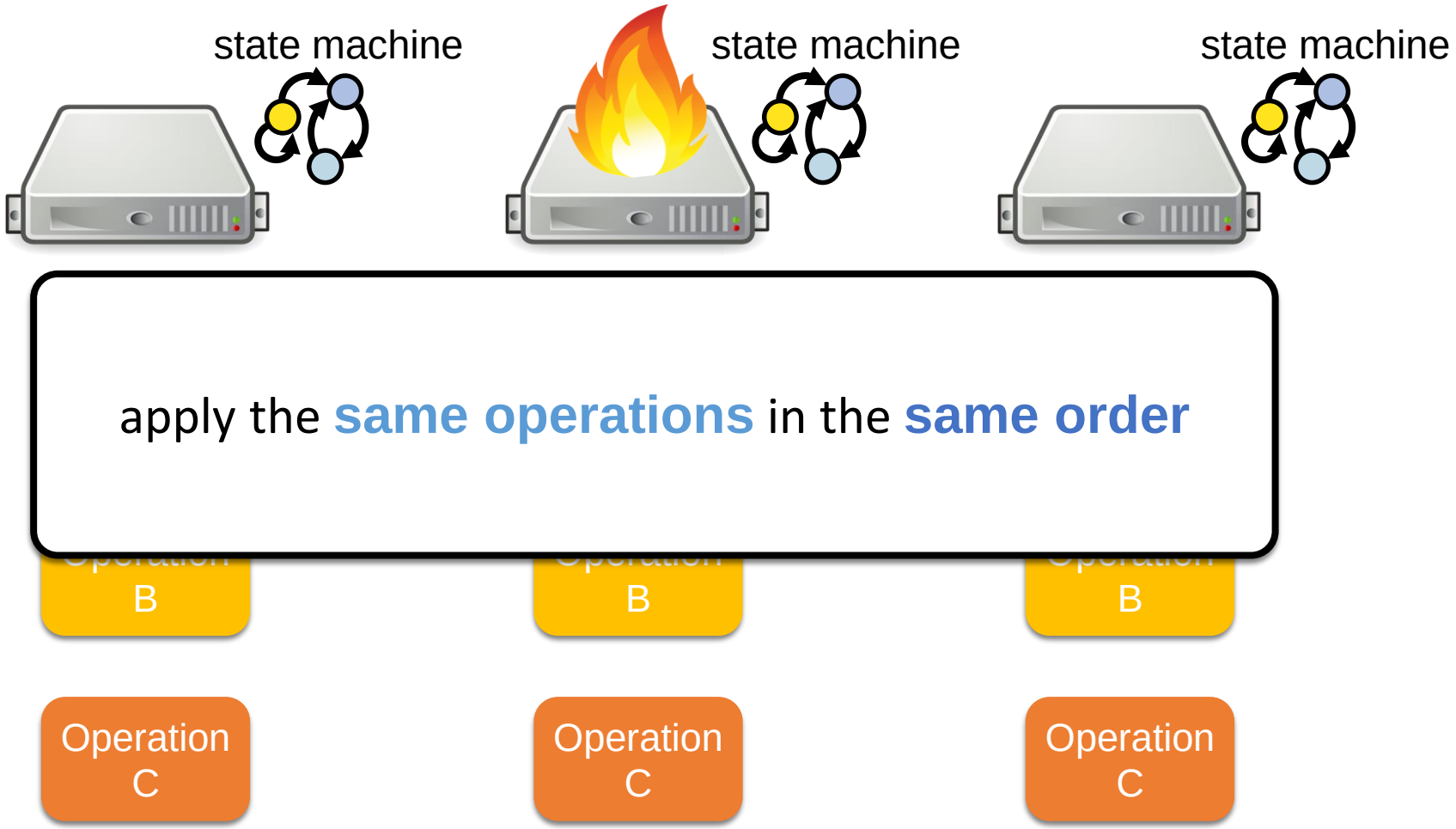


New approach: co-designing distributed systems with the network

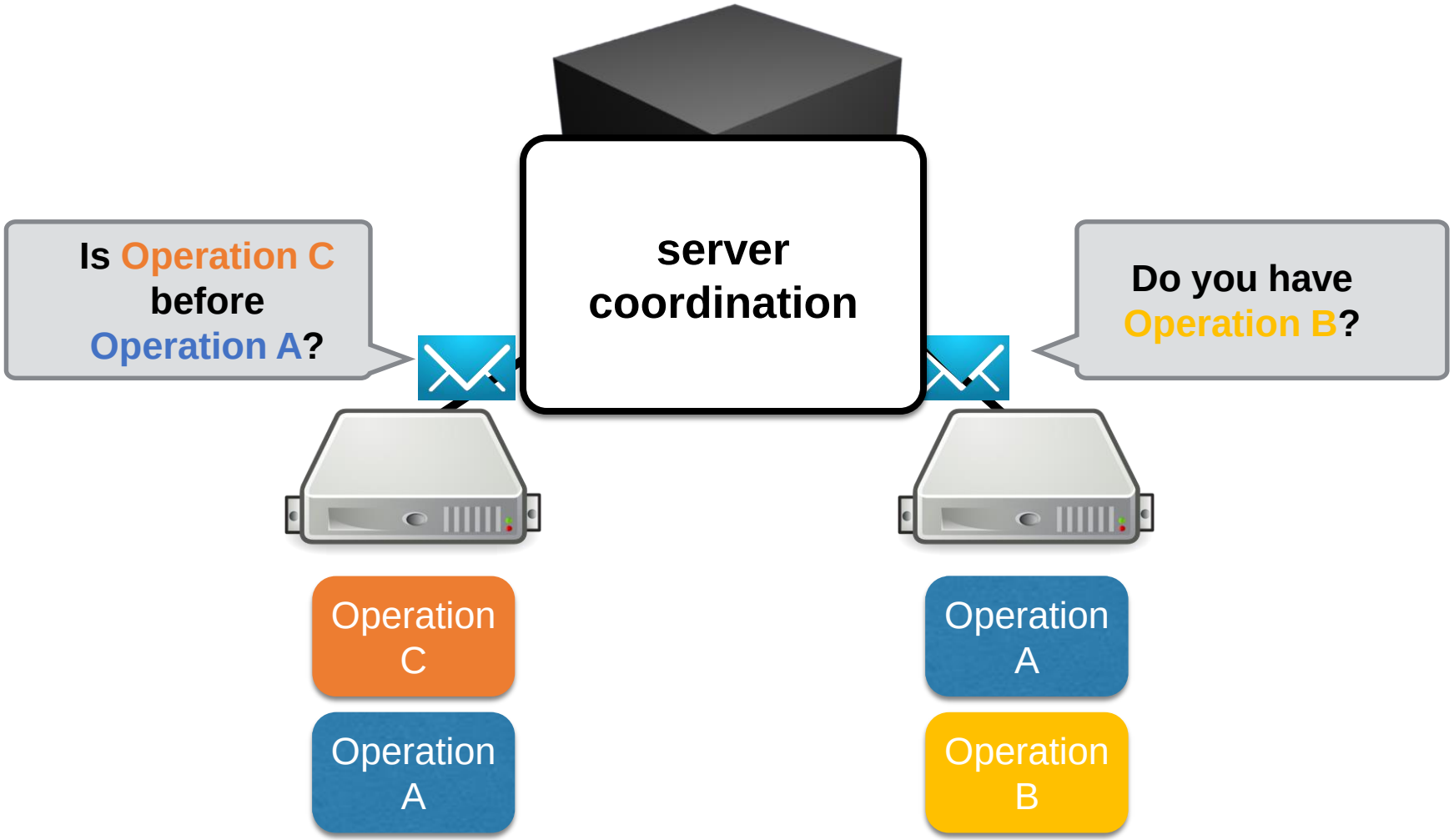
- No longer treat network as a black box
- Leverage programmable network devices to design stronger **network primitives**
- Build new **distributed protocols** around these network primitives
- Result: practical distributed systems with both **strong guarantees** and **high performance**
- No more weak consistencies!

Co-Design Approach Applied to **State Machine Replication**

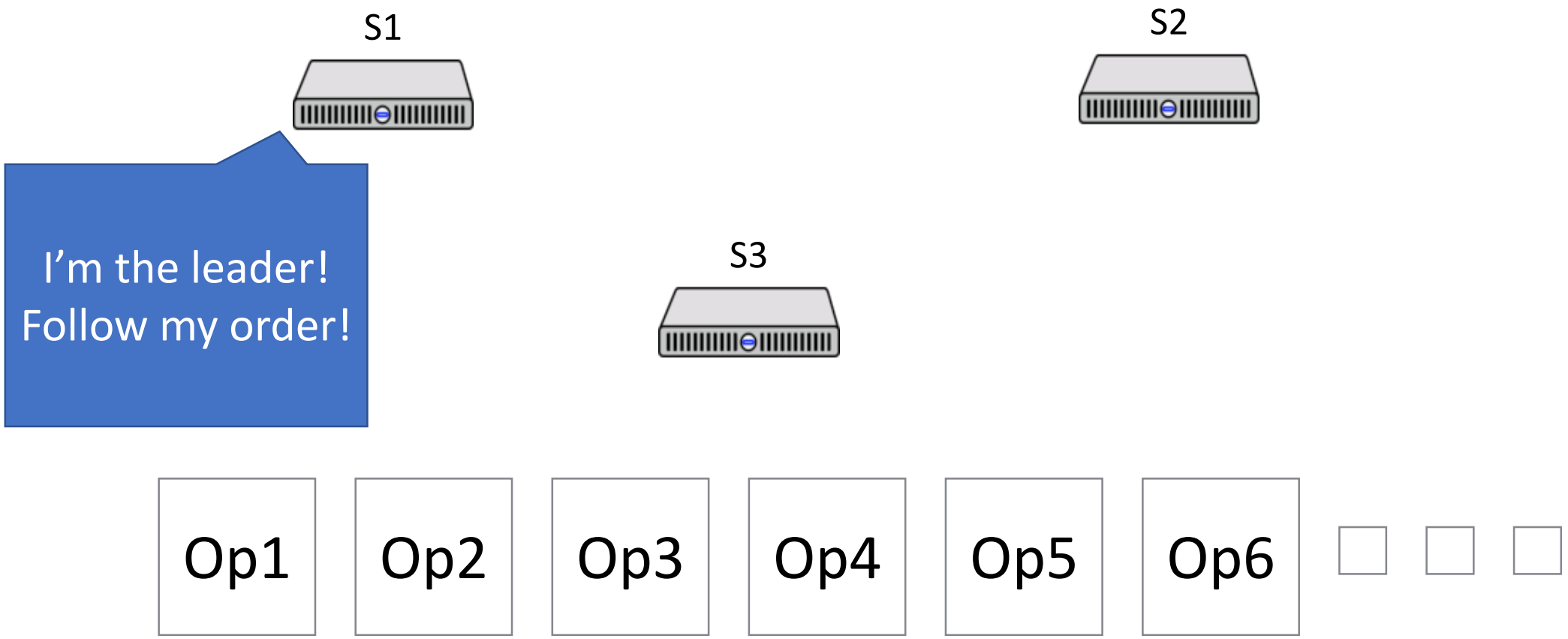
State Machine Replication Primer



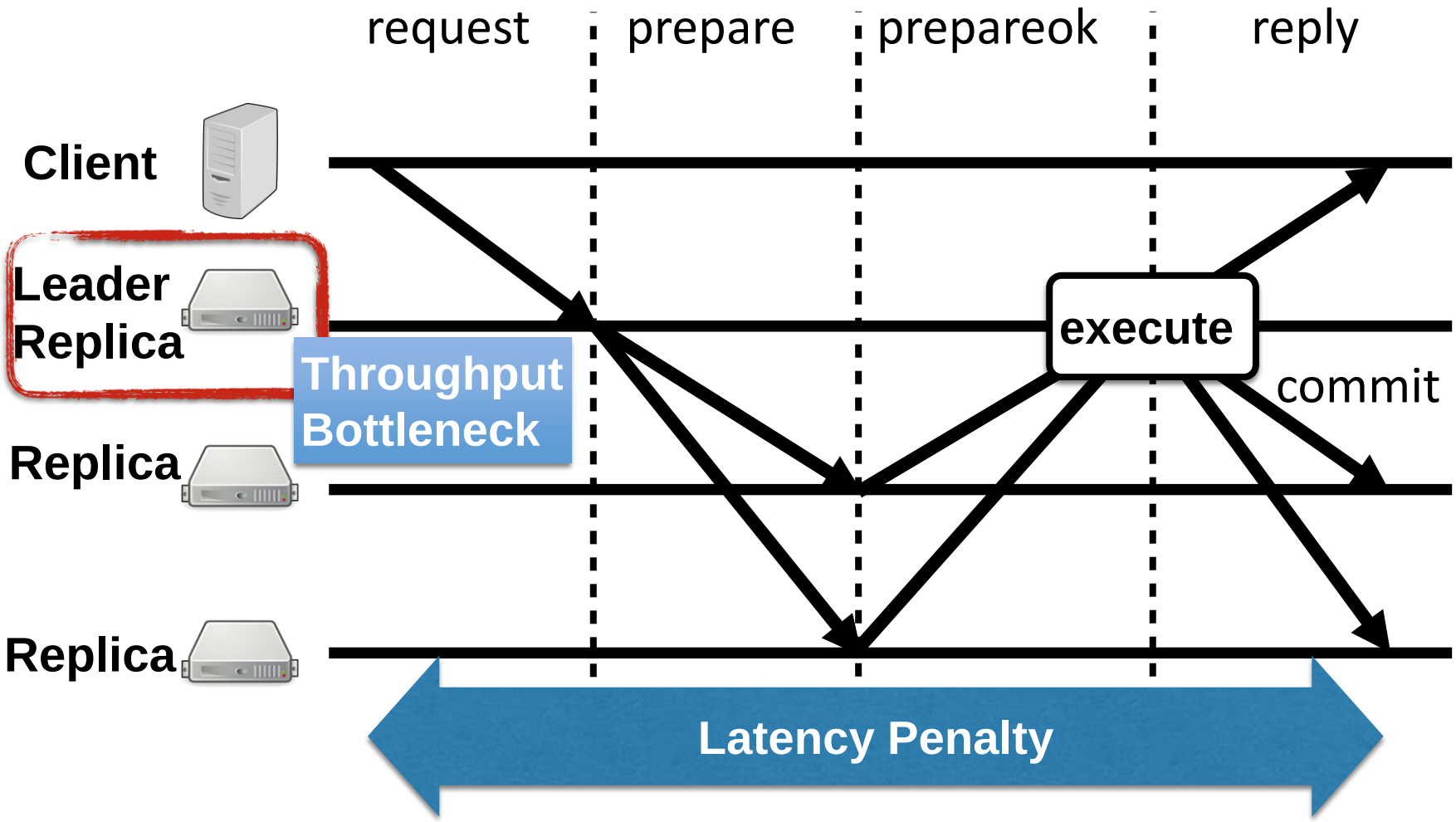
Black-box network model requires **server coordination**



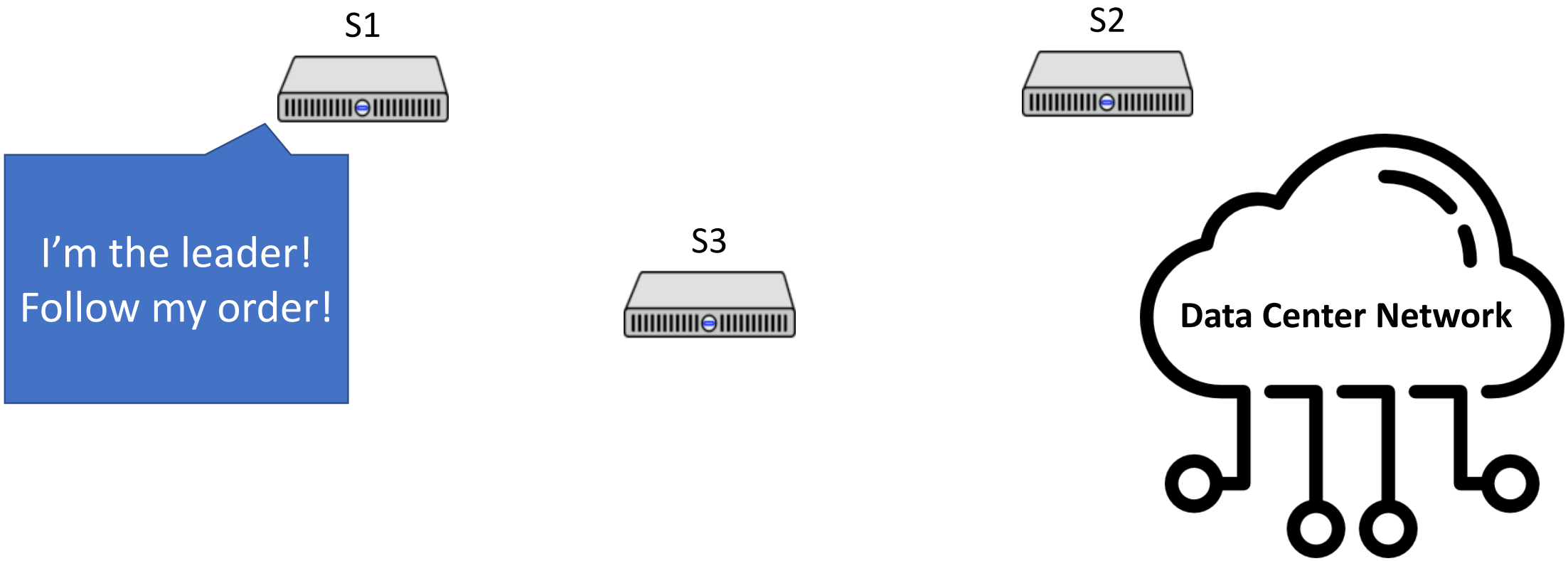
How do we ensure same order of requests?



Standard approach: **Paxos/VR/Raft**



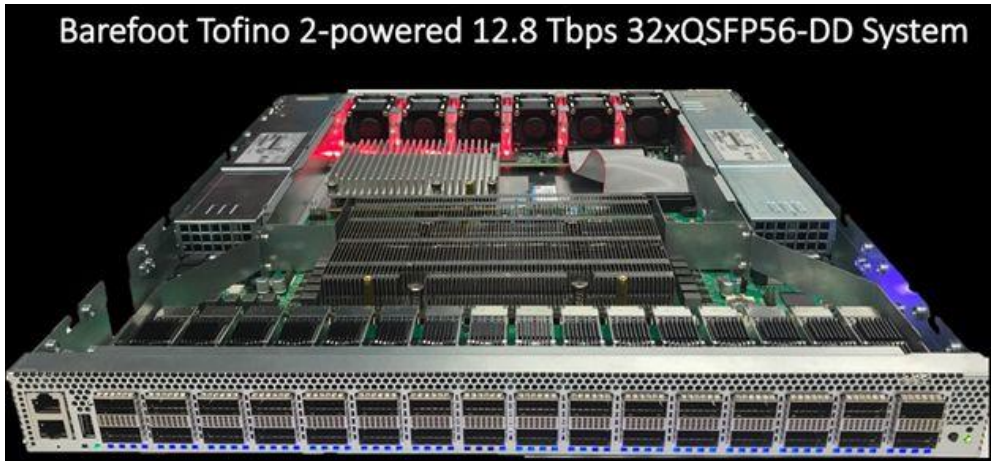
What if…
the network **orders** requests for us?



New network primitive: Ordered Unreliable Multicast

- **Multicast:** client sends a single message. network delivers the message to all replicas
- **Ordered multicast**: all replicas receive multicast messages **in the same order**
- **Unreliable delivery**: no guarantee that messages will be delivered
- **Drop detection**: replicas receive **notifications** for dropped messages

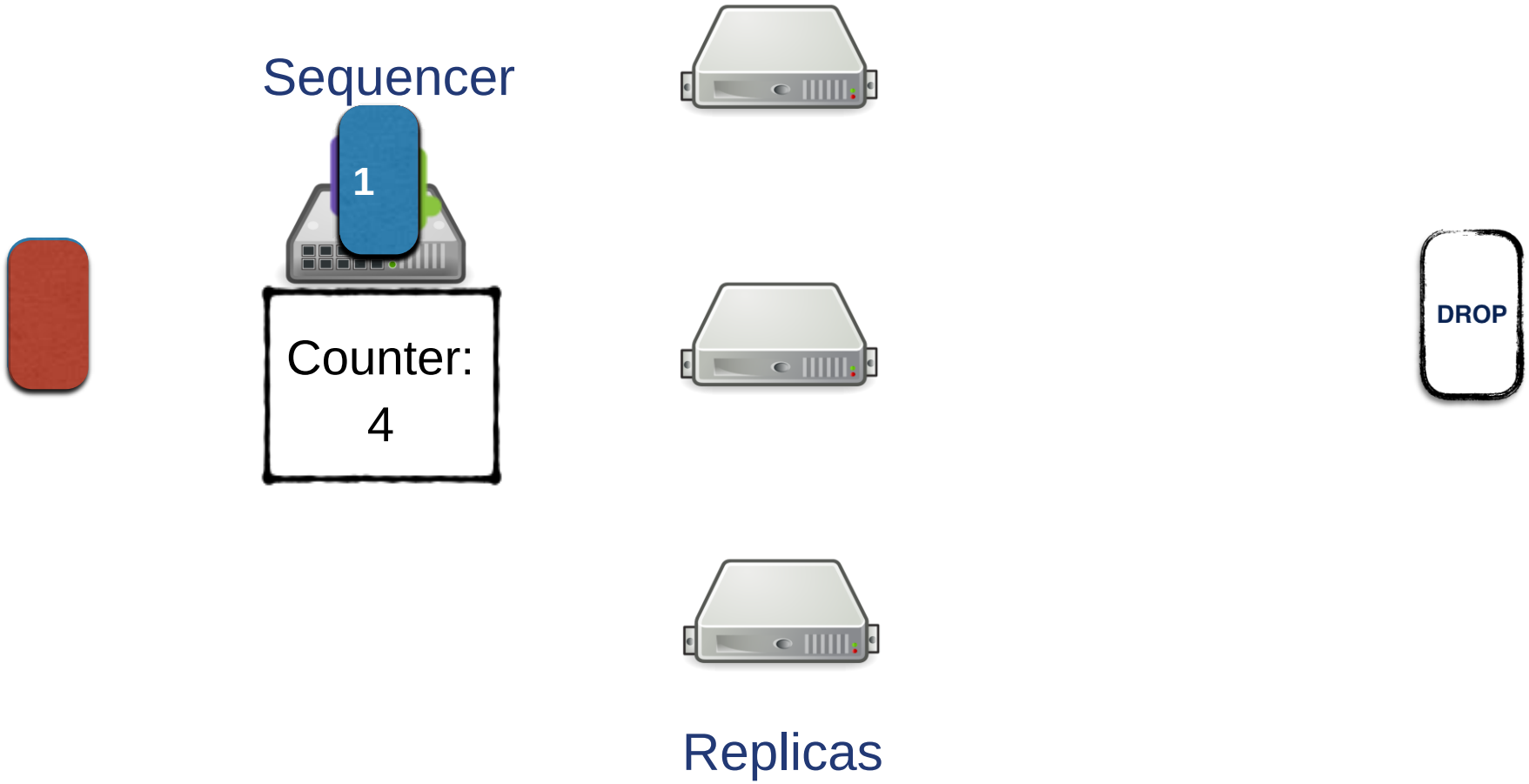
How to implement ordering in the network??



Client requests first go through a programmable switch

Switch writes a **counter** value into request packet

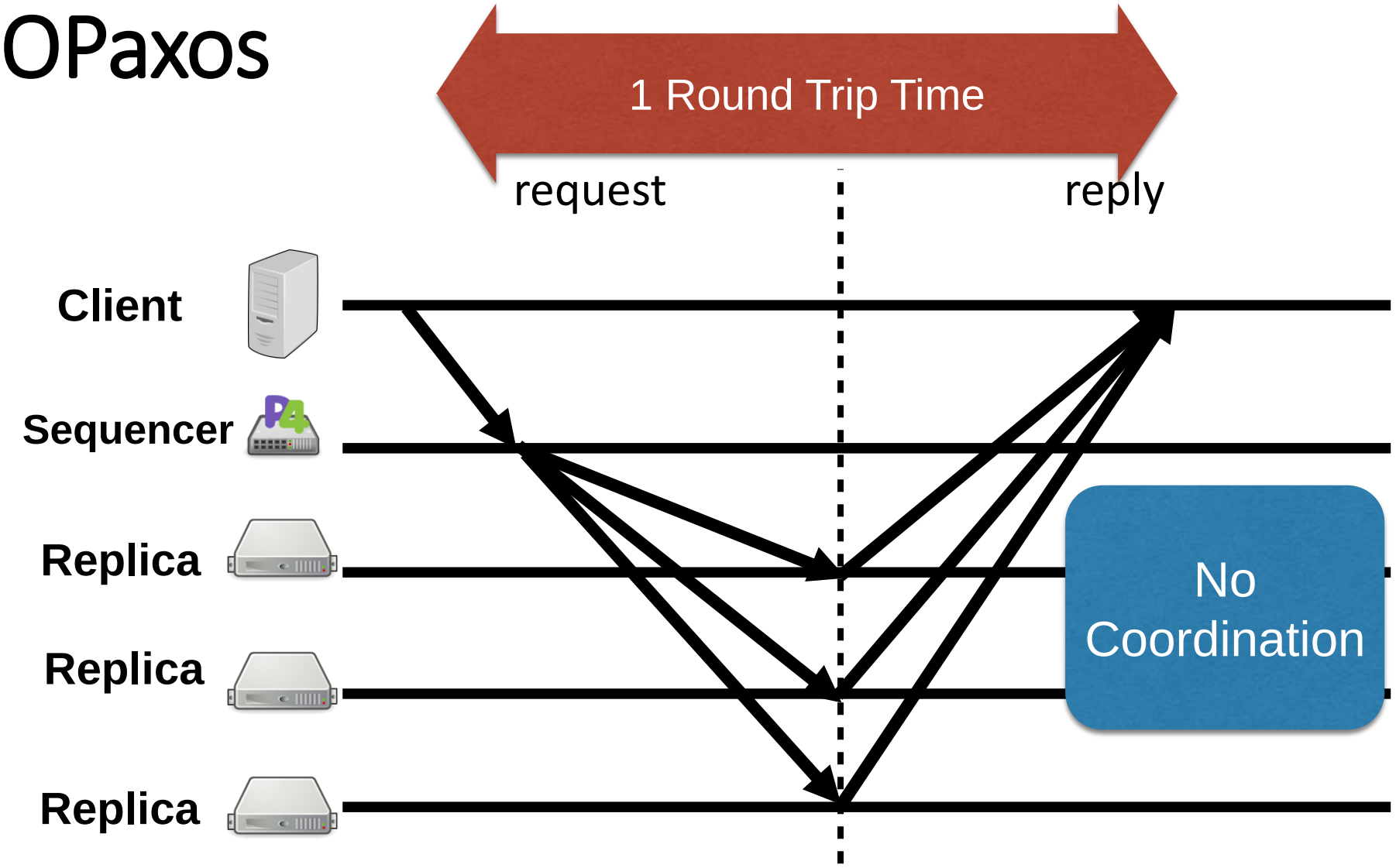
How to implement Ordered Unreliable Multicast (efficiently)?



How does Ordered Unreliable Multicast benefit state machine replication?

- All replicas receive messages in the **same order**
- If they receive all the messages
- Just execute and reply
  - they **know** other replicas also receive the messages in order
- All replicas are in **consistent state**
- **No coordination** at all!

Co-designing SMR with Network Ordering: **NOPaxos**

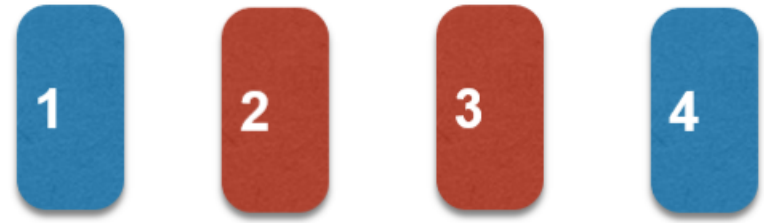


What happens when messages are **dropped** in the network?

Sequencer

Counter:
4

Replicas



Gap Agreement

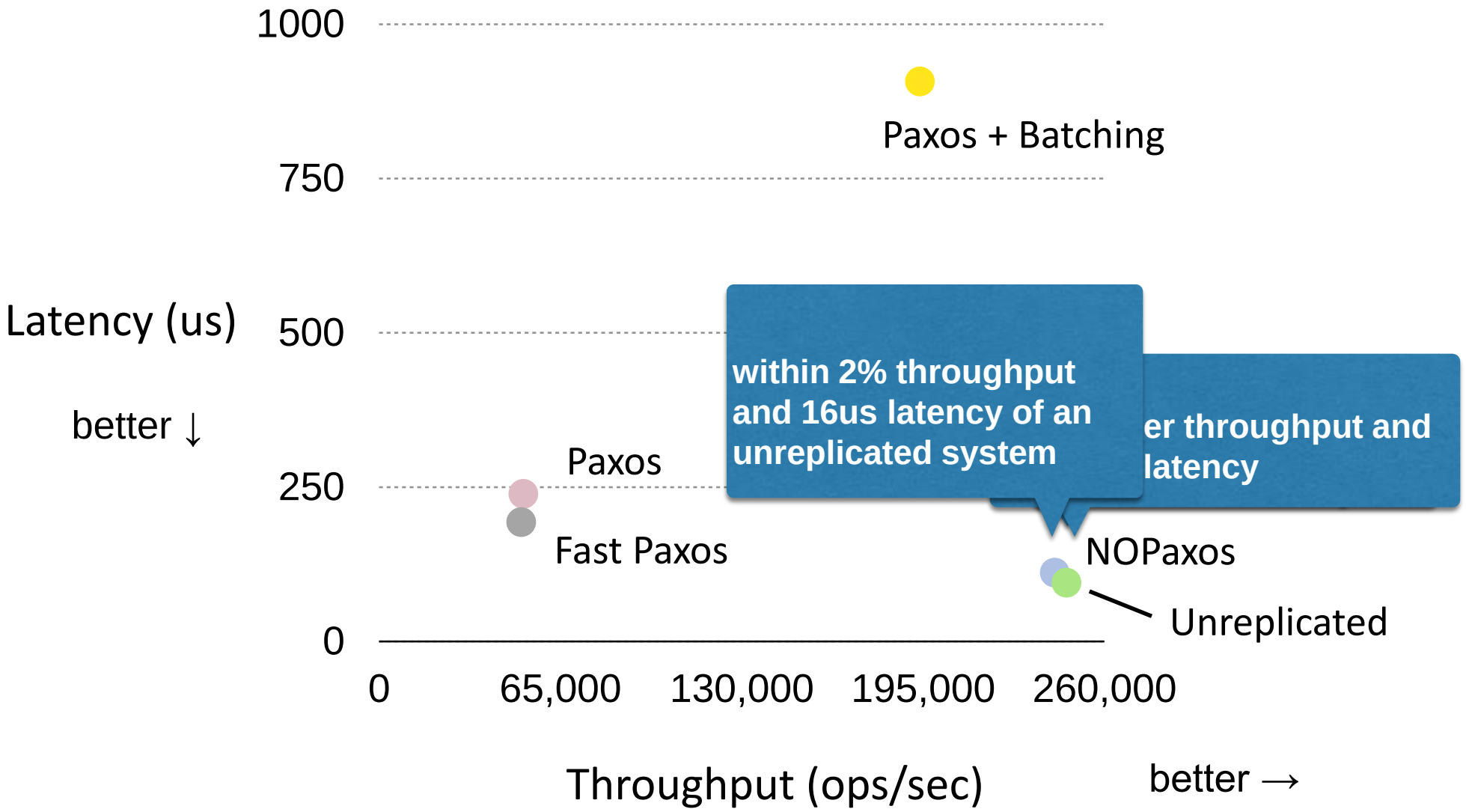
- Need to agree if the message is **received** or **permanently dropped**
- When a replica detects message drops
  - **Non-leader replicas:** recover the missing message from the leader
  - **Leader replica:** coordinates to commit a NO-OP (Paxos)
- Efficient recovery from network anomalies

Other protocols

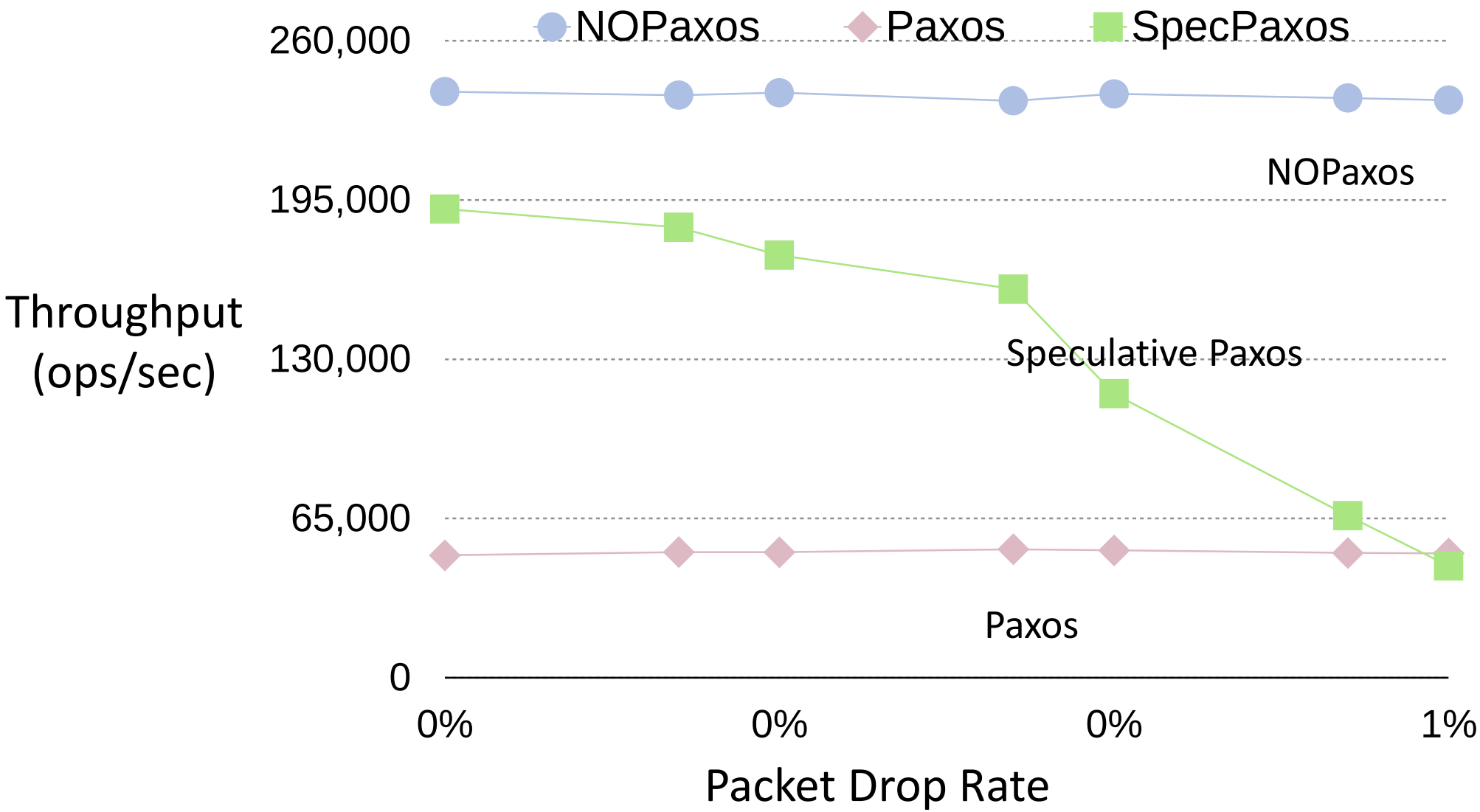
- Handling leader replica failures
- Handling sequencer switch failures
- More details in the paper!

What benefits do we get?

NOPaxos has no Paxos overhead!



NOPaxos is resilient to network anomalies



What else can do with this **Co-Design** approach?

Many real-workloads are **skewed** and **dynamic**

Data partitioning for **scalability**

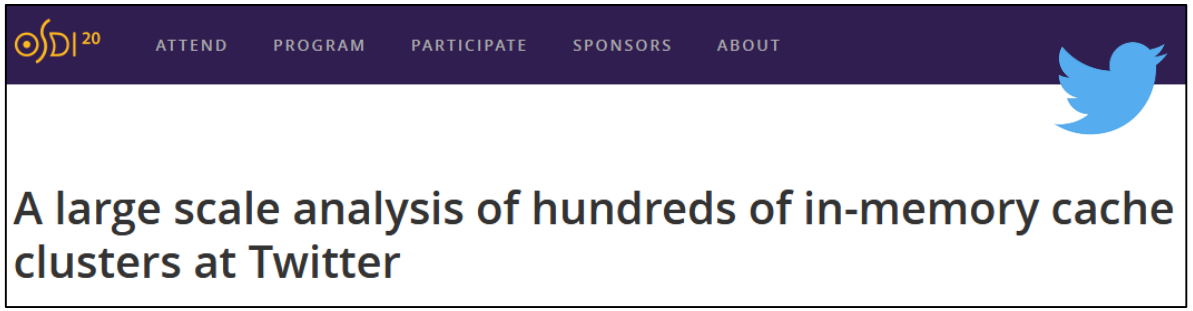
Shard Shard Shard

Skewed workloads lead to **load imbalance**

popular objects can **overload** storage servers

Skewed workloads are **diverse**

- read-heavy
- write-heavy
- read-write mixed



Workload Analysis of a Large-Scale Key-Value Store

Google

Fast key-value stores: An idea whose time has come and gone

- small objects
- large objects
- combination of both

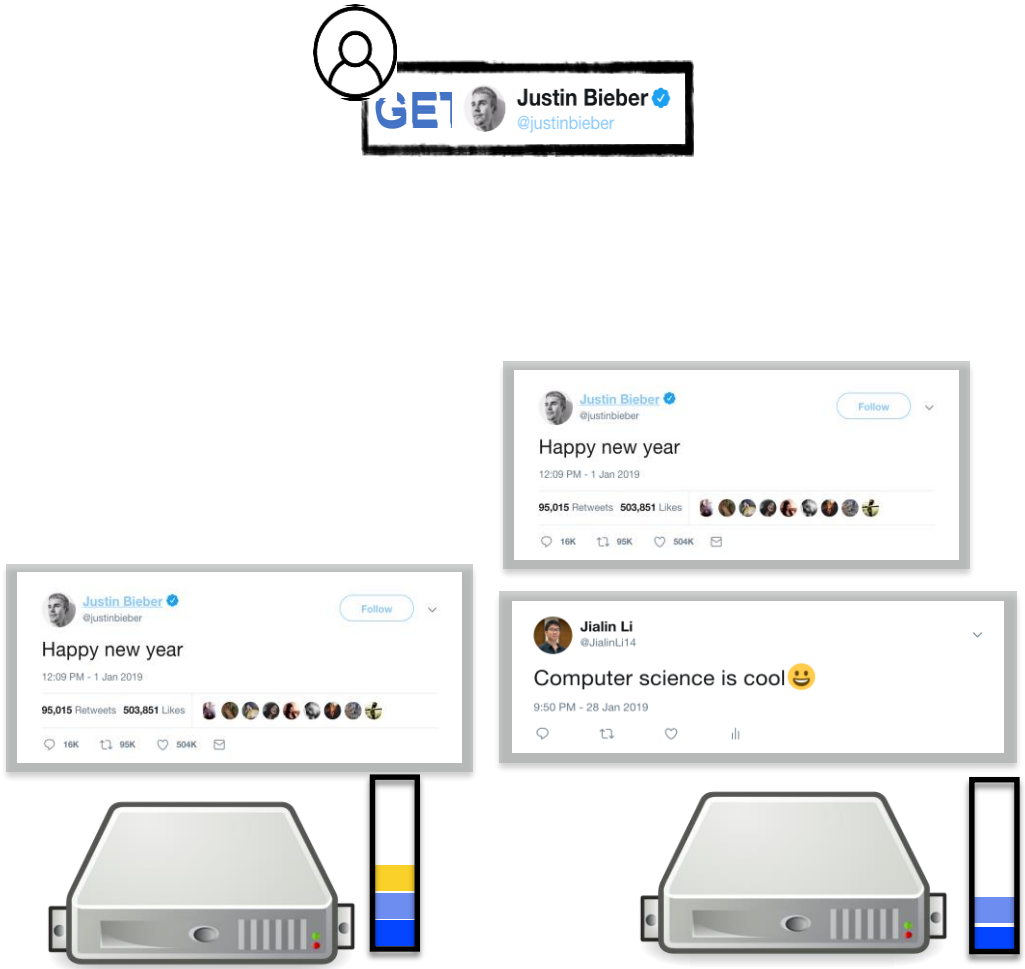
Two approaches to deal with **highly skewed workloads**

Caching

- Cache popular objects in a faster tier
- Caching tier *absorbs* traffic to popular objects
- More **uniform** load on backend storage servers

Caching Layer

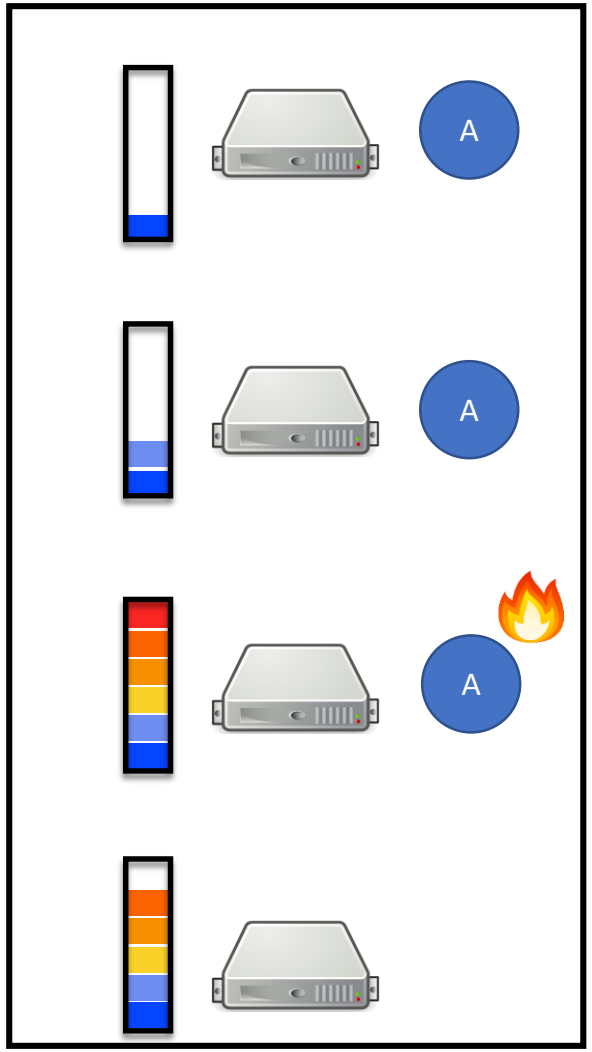
Two approaches to deal with **highly skewed workloads**



Selective Replication

- Replicate popular objects on multiple servers
- Requests to replicated objects can be forwarded to *any* replica
- **Distribute** load across servers

Selective Replication is challenging



How to route requests to the right server?

How to ensure consistency?

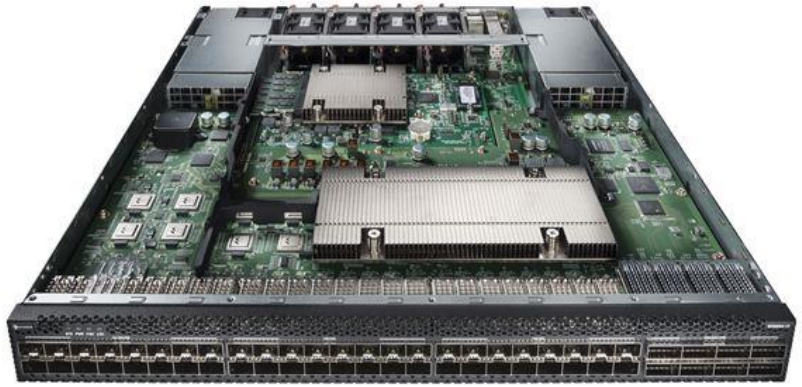
What if the network can **track** where the copies are?

Pegasus' approach

selective replication

+

in-network **coherence directory**

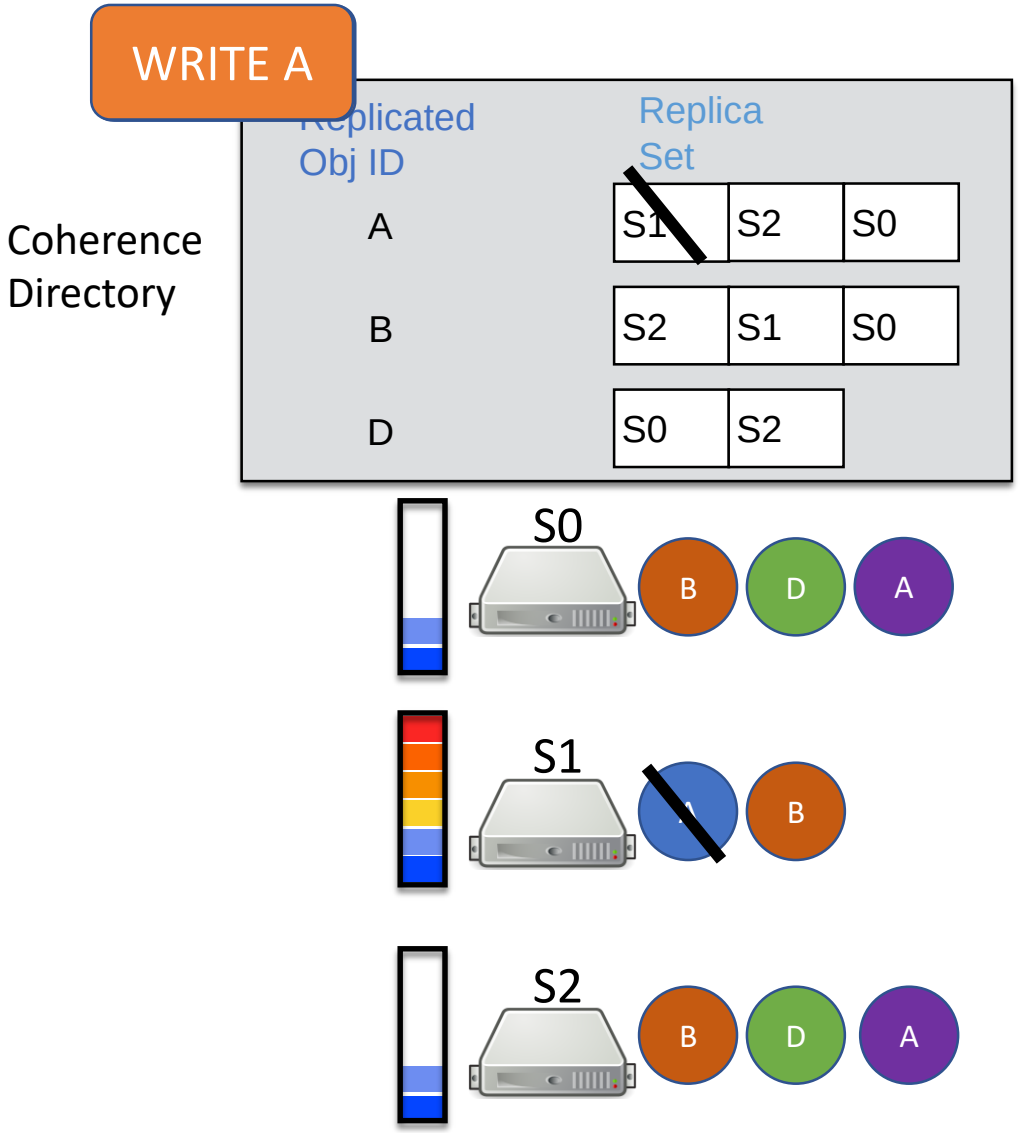


Barefoot P4 switch

Coherence directory for replicated data

- Inspired by CPU cache coherence protocols
- Centralized directory that **tracks:**
  - **Which** objects are replicated
  - **Location** of replicated objects
- **Forwards** requests to server with spare capacity
- Ensures **strong consistency**

Coherence directory illustrated



Implementing coherence directory **in the network**

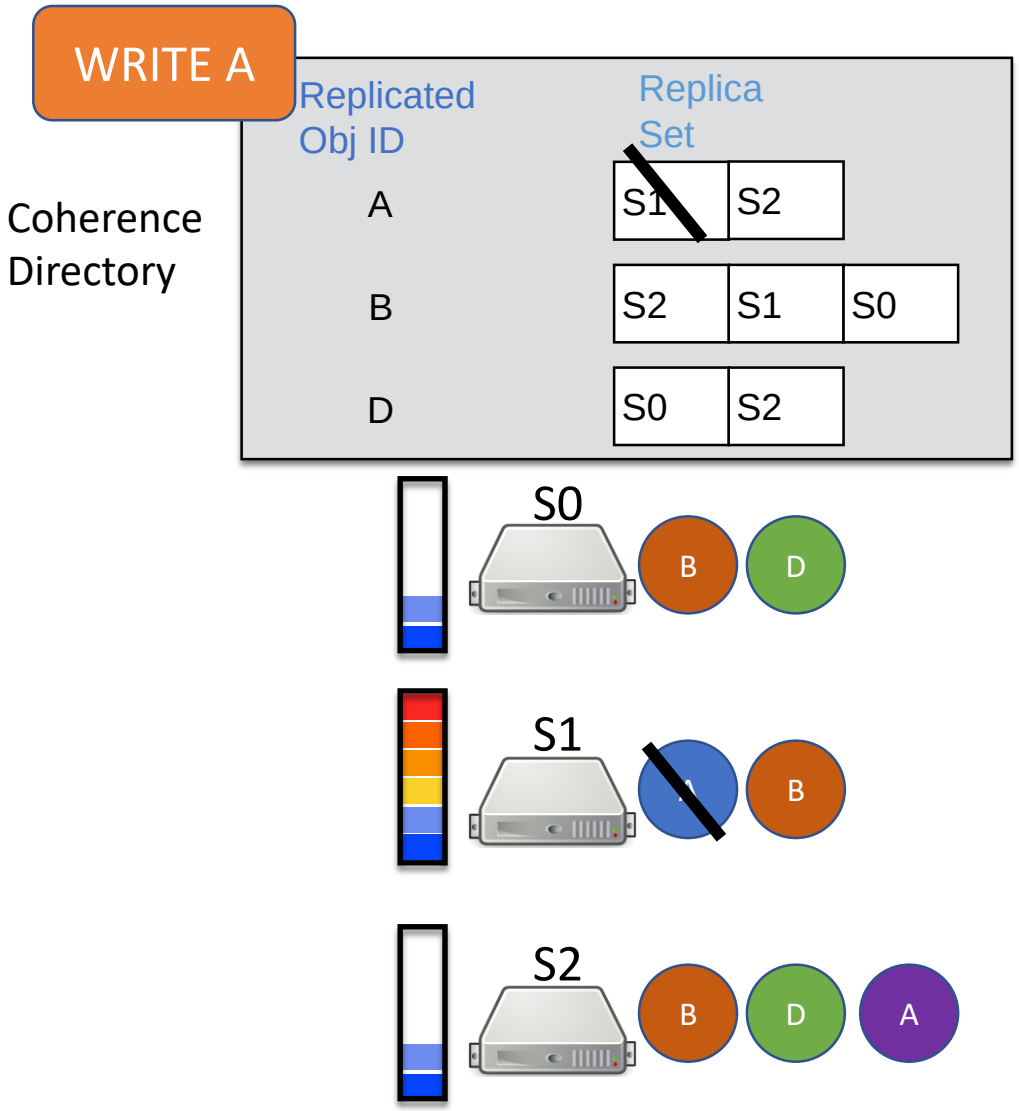
rack-scale storage system

- All requests and replies traverse the ToR switch
- ToR serves as a **central point**
- **Line-rate** packet processing
  - No throughput bottleneck
  - Zero latency overhead

How to ensure consistency?

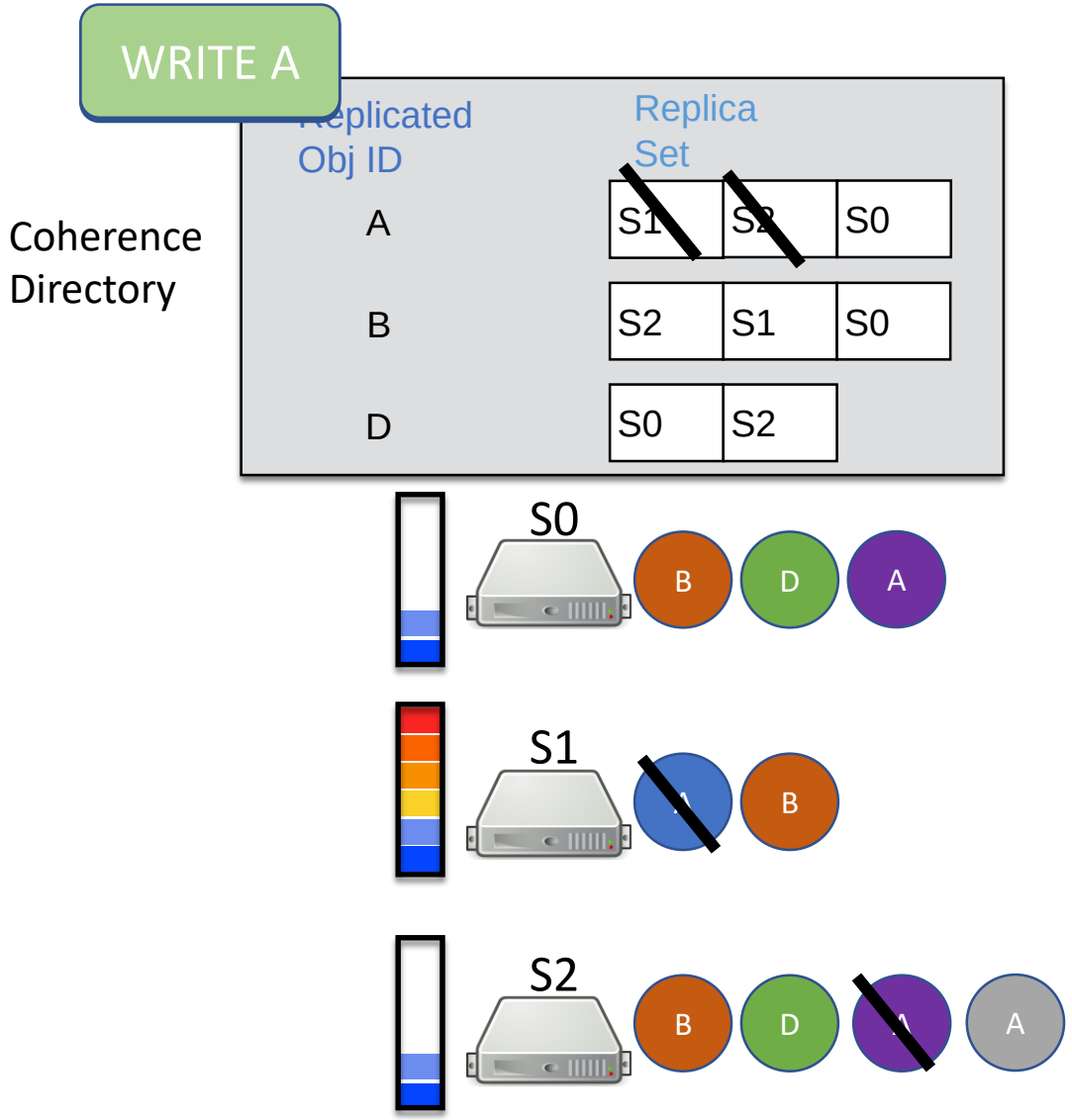
- Switch processes **all** requests
- Switch directory tracks which servers have the **latest** copy
- When do you update the directory?
  - Receive request?
  - Receive reply?

When do you update the directory?



How to deal with **asynchrony** in the system?

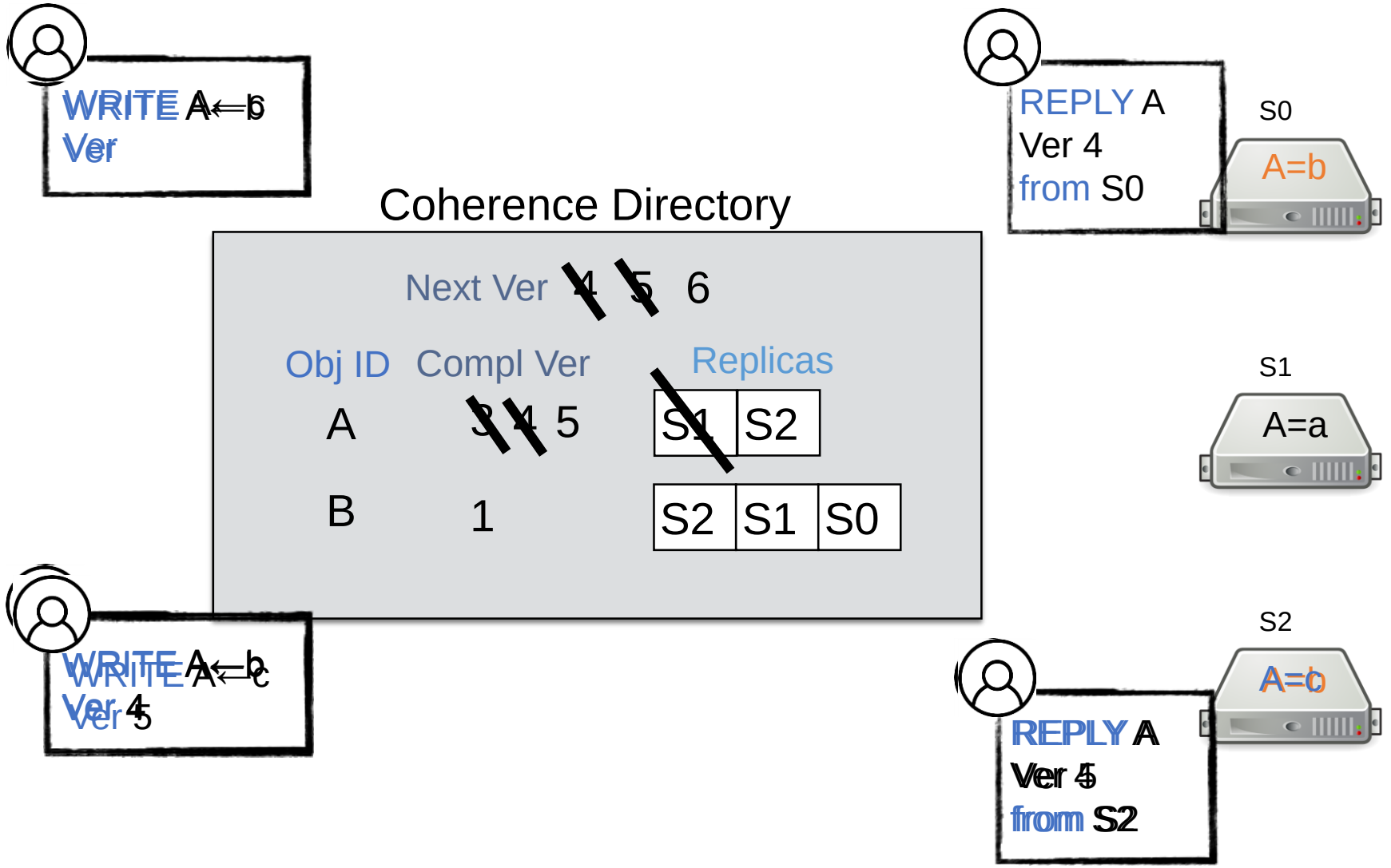
Issues with Asynchrony



How to deal with **asynchrony**?

Version-based
coherence protocol!

Pegasus' coherence protocol in action

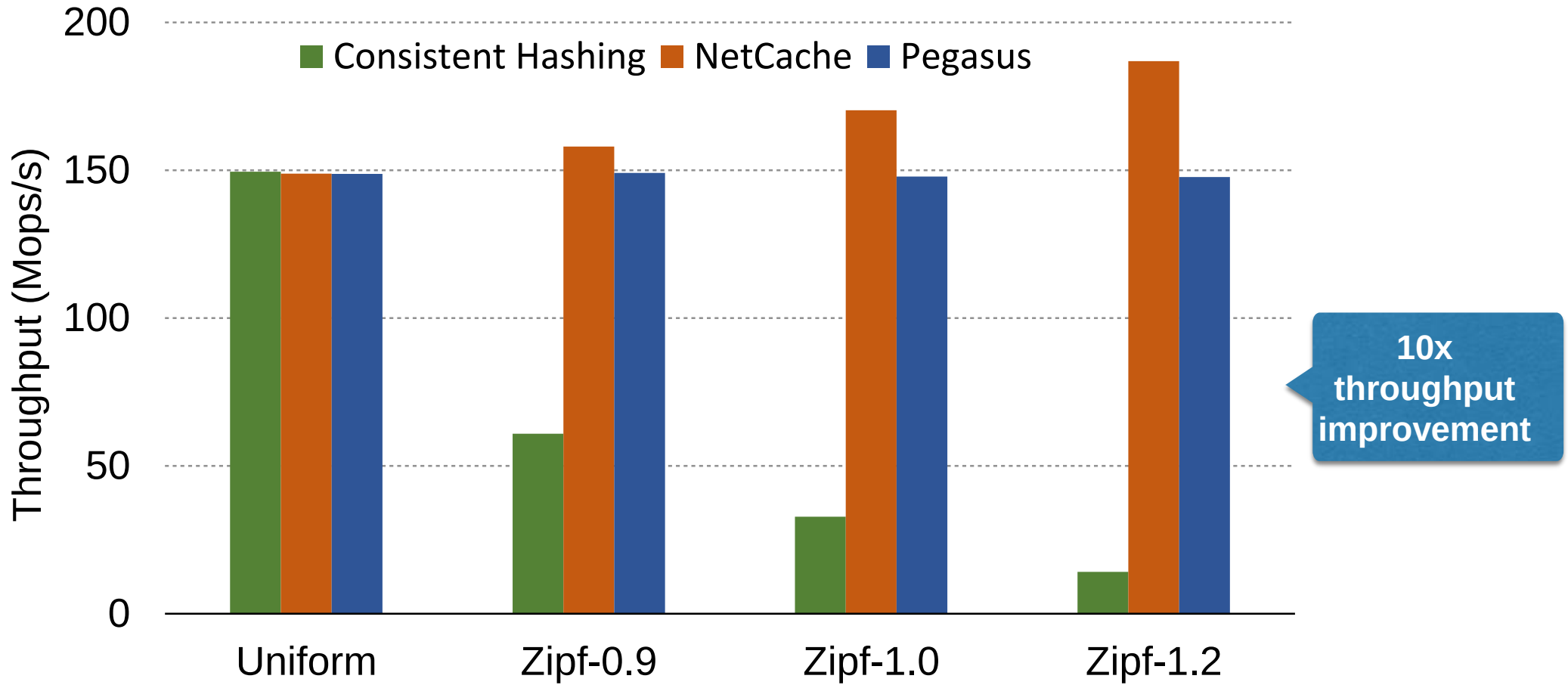


Other protocol details

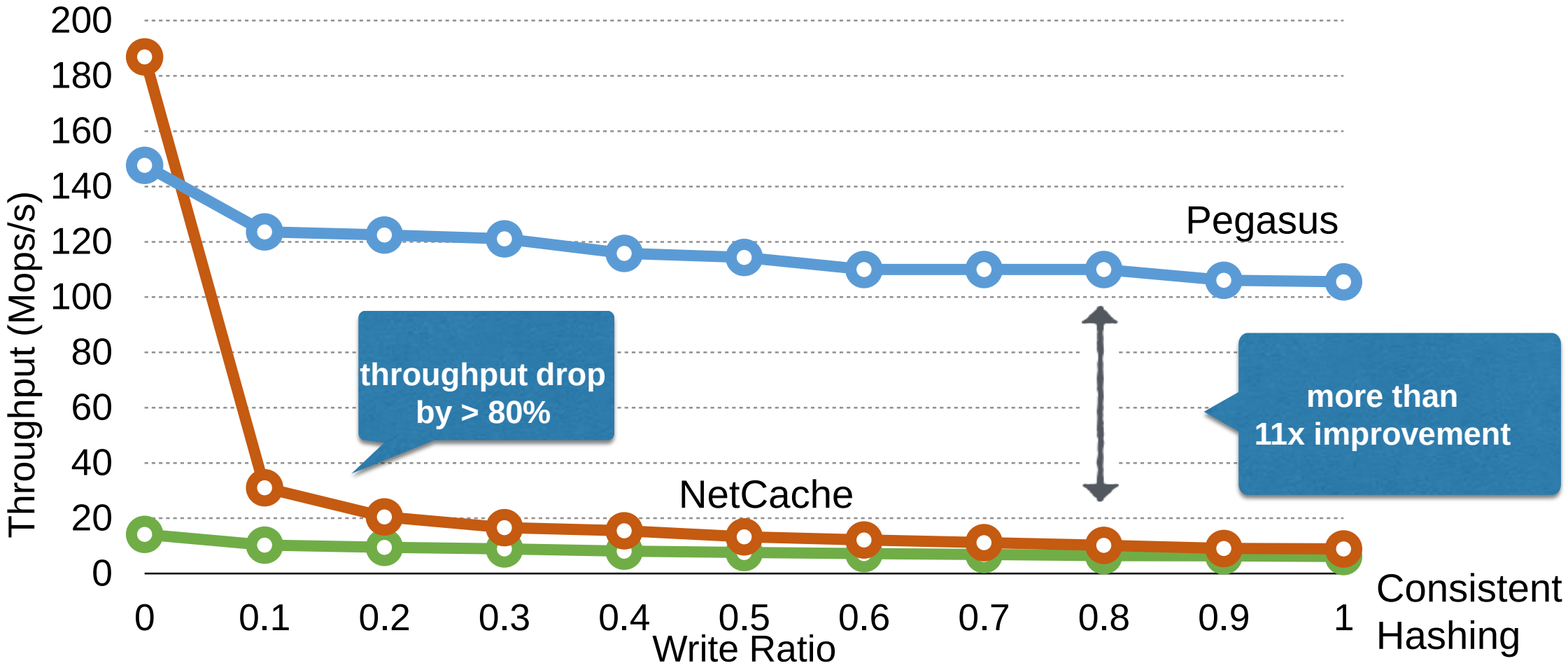
- Adding and removing replicated objects
  - Pegasus controller monitors object access frequencies
  - Updates coherence directory with most popular objects
- Server selection policy
  - Random
  - Weighted round-robin
- Handling server and rack failure
  - Multi-rack deployment
  - Each rack runs a Pegasus instance
  - Chain replication across racks

What benefits do we get?

Pegasus is effective under **highly skewed** workloads



Pegasus equally effective under **different** read/write ratios



Conclusion

- Co-designing distributed systems with the network
  - Identify the right abstractions to be implemented **in the network**
  - Design protocols to take advantages of network primitives and provide **strong guarantees**
- Distributed systems with both **strong guarantees** and **high performance**
  - State machine replication: Network-Ordered Paxos
  - Distributed storage load balancing: Pegasus