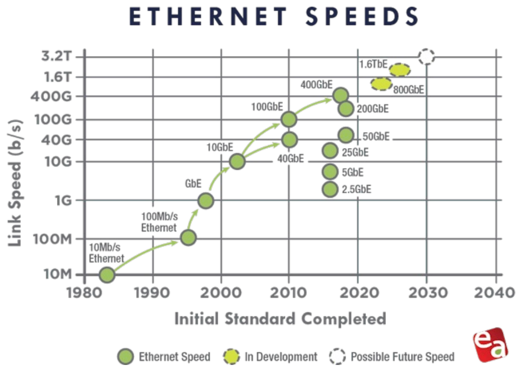
Leveraging Network Hardware in Distributed Systems Design

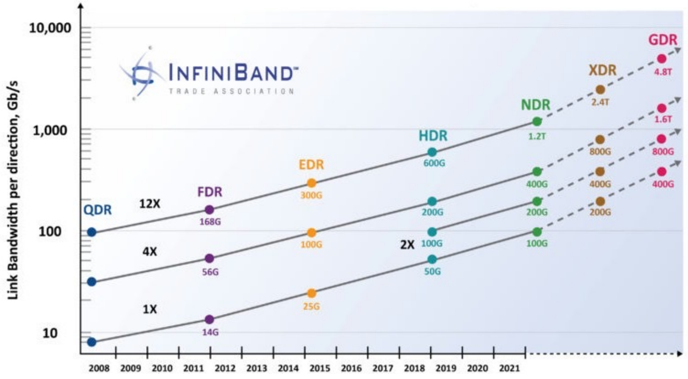
Jialin Li

NUS | Computing
National University
of Singapore

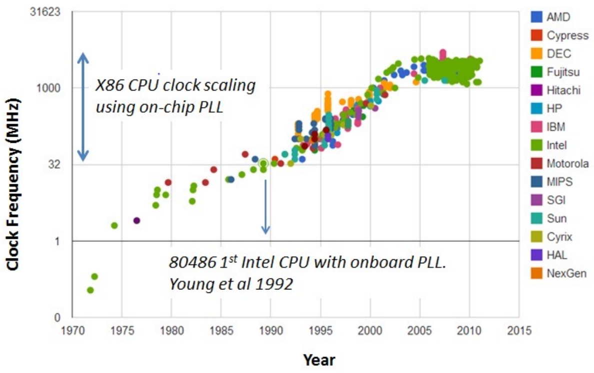
Distributed Systems Challenges in Data Centers

- Increasing network speed





- Stalling CPU speedups



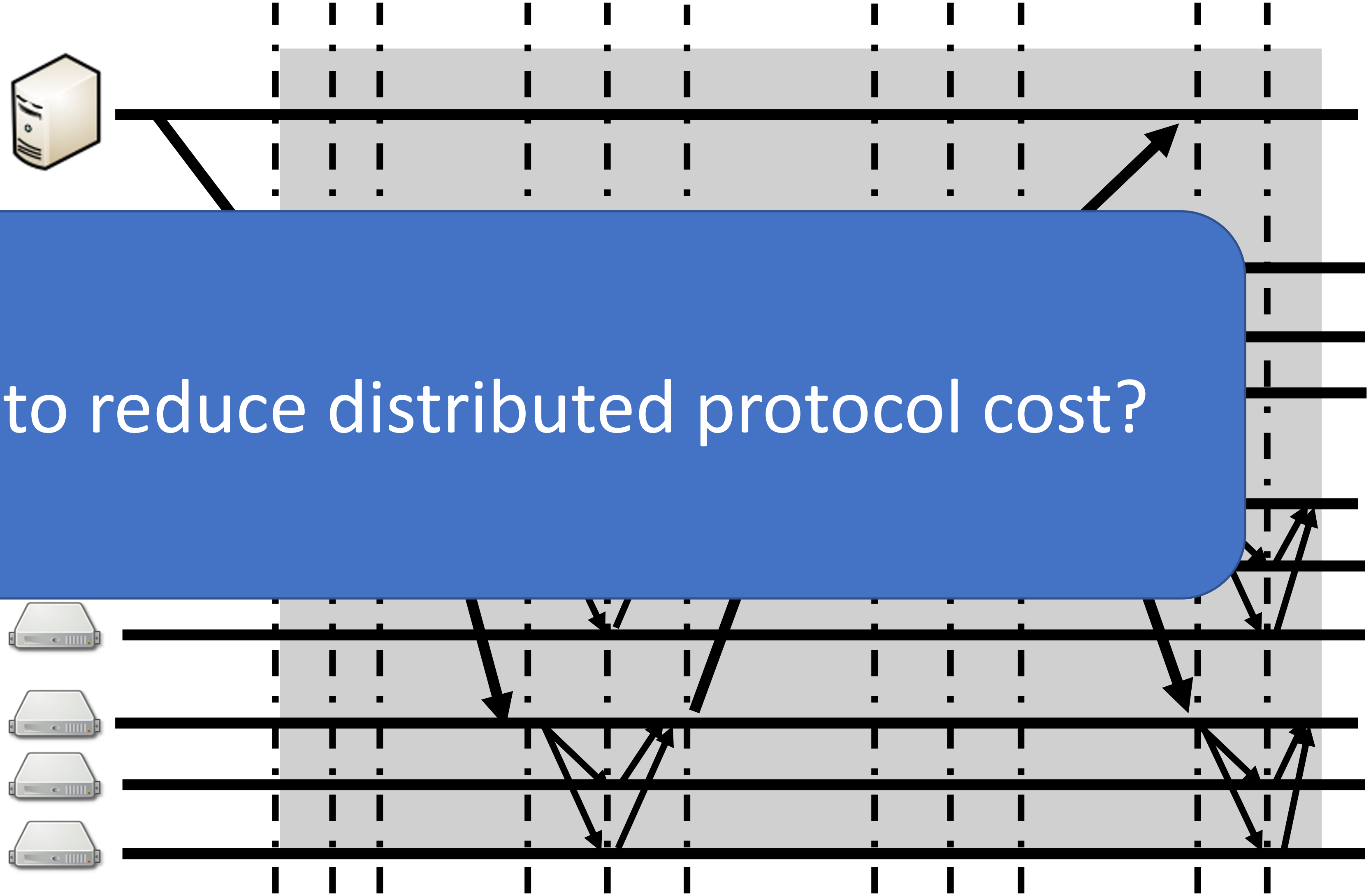
- Strict user Service-Level Agreement
  - ***us-scale*** tail latency requirement

Complex protocol communication patterns!

Fault-To

Distribu

Transact



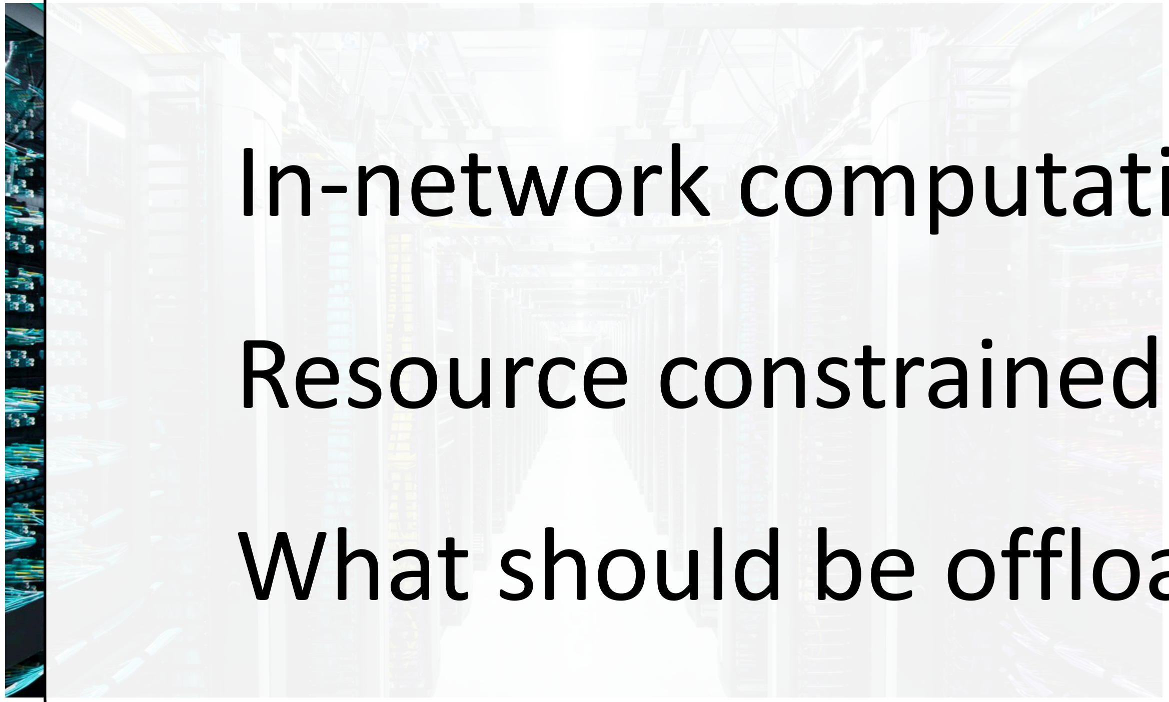
How to reduce distributed protocol cost?

Protocol offloading to the **network**?

In-network computation at line rate

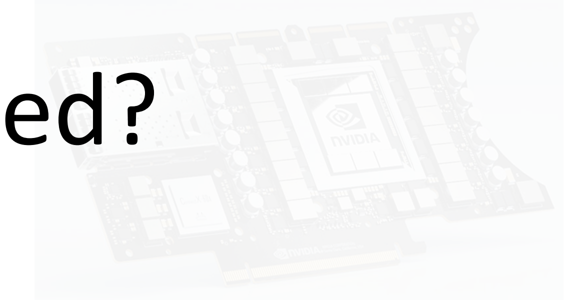
Resource constrained devices

What should be offloaded?



software defined networking

reconfigurable switches



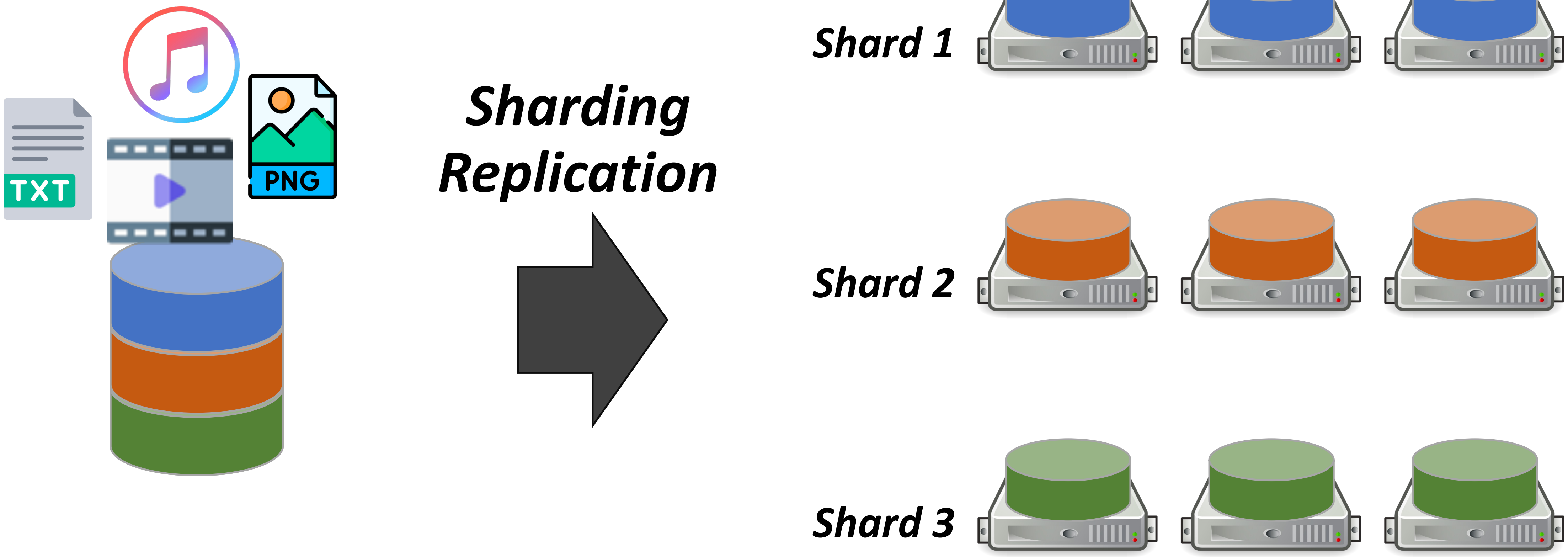
DPUs, SmartNICs

Theme: *partial protocol offloading to the network* for distributed systems

- Use programmability in the network to offload **simple protocol primitives**
- **Efficient** network implementation
- **Co-design** distributed protocols and the network
- Result: practical distributed systems with both **strong guarantees** and **high performance**

Serialization-Free Network Ordering for Strongly Consistent Distributed Applications

Scalability by *Sharding*
Fault Tolerance by *Replication*



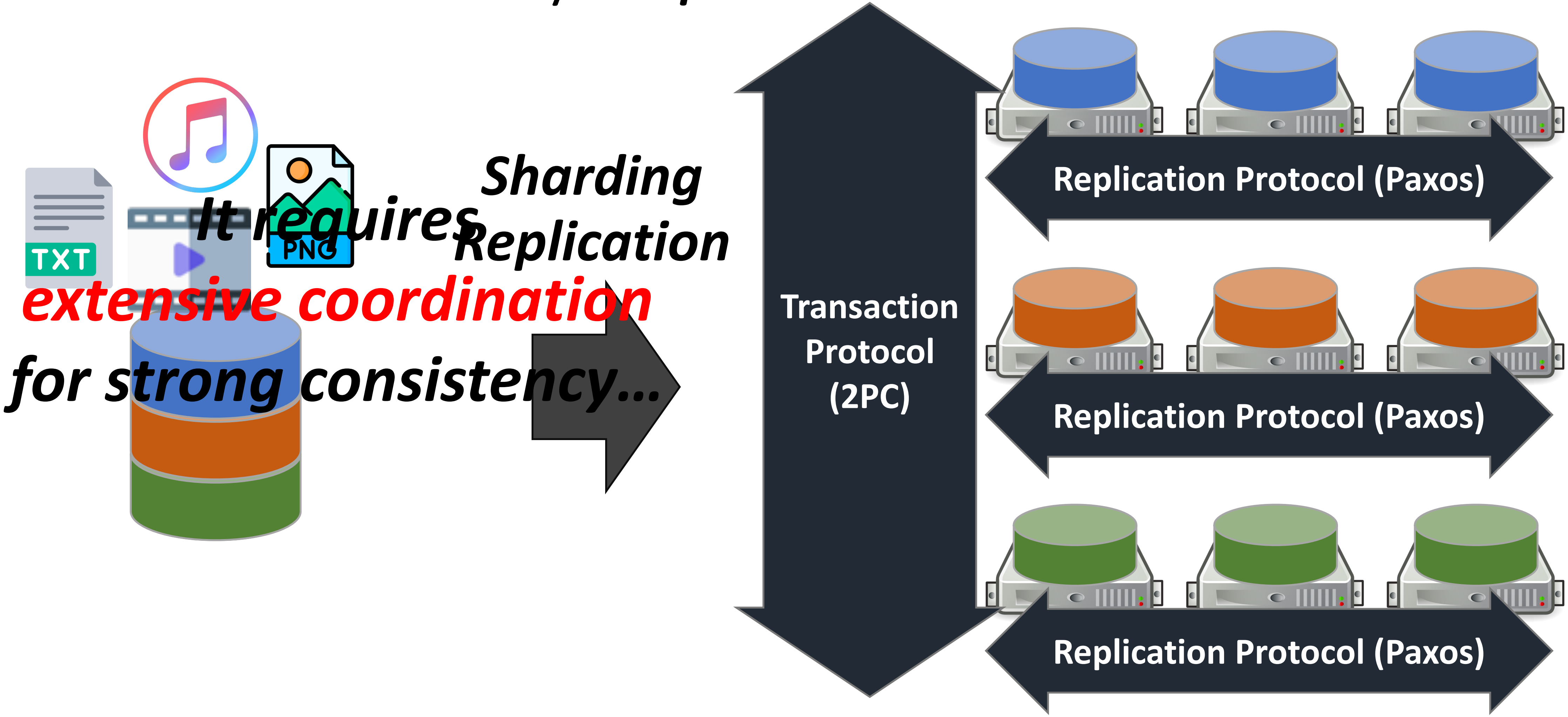
Sharding
Replication

Shard 1

Shard 2

Shard 3

Scalability by *Sharding*
Fault Tolerance by *Replication*


TXT
PNG
Sharding
Replication
Transaction Protocol (2PC)
Replication Protocol (Paxos)
Replication Protocol (Paxos)
Replication Protocol (Paxos)

It requires extensive coordination for strong consistency…

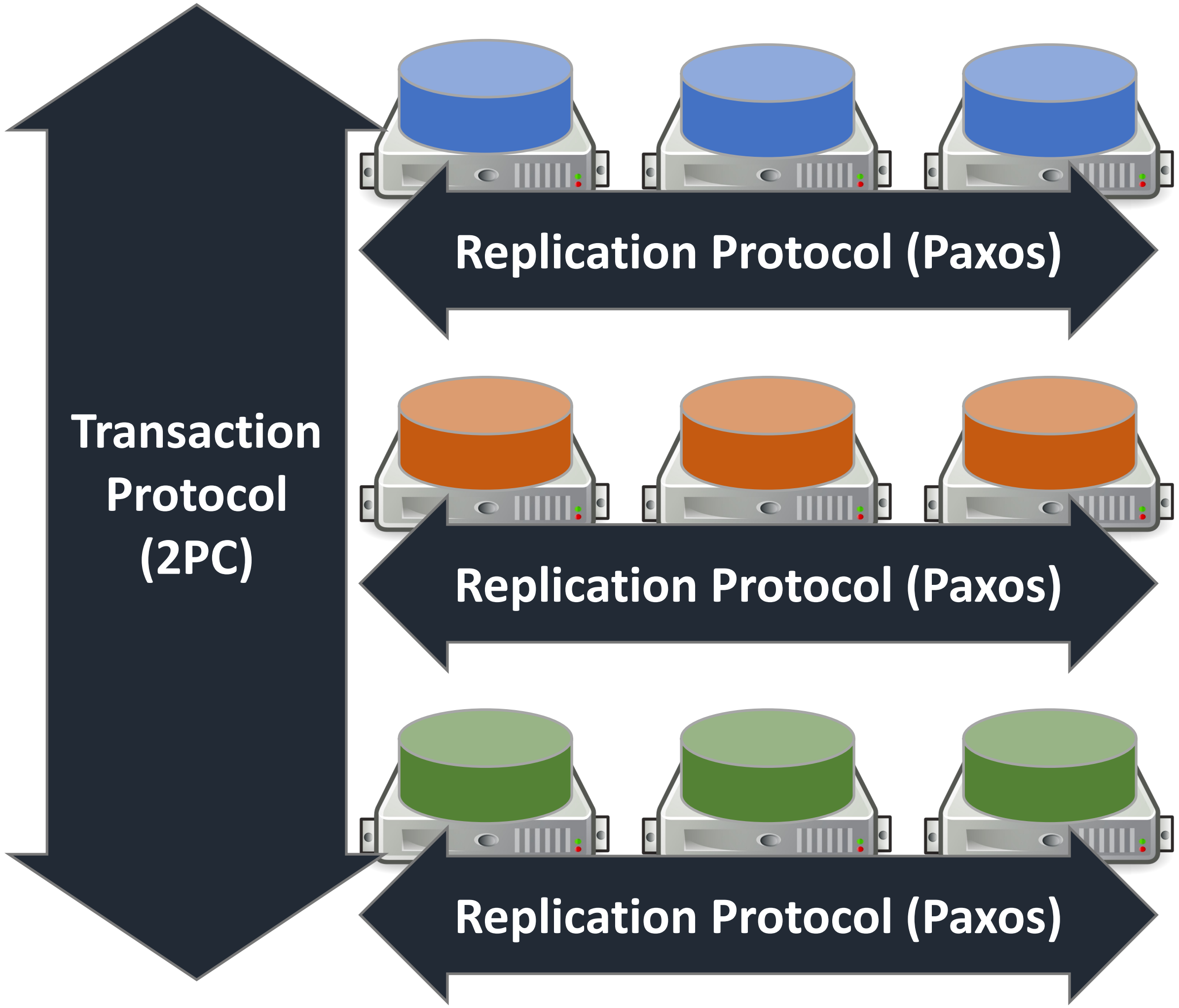
Network Ordering to Eliminate Coordination

Network

Sequencer
(e.g., Programmable Switch)

Consistent Ordering

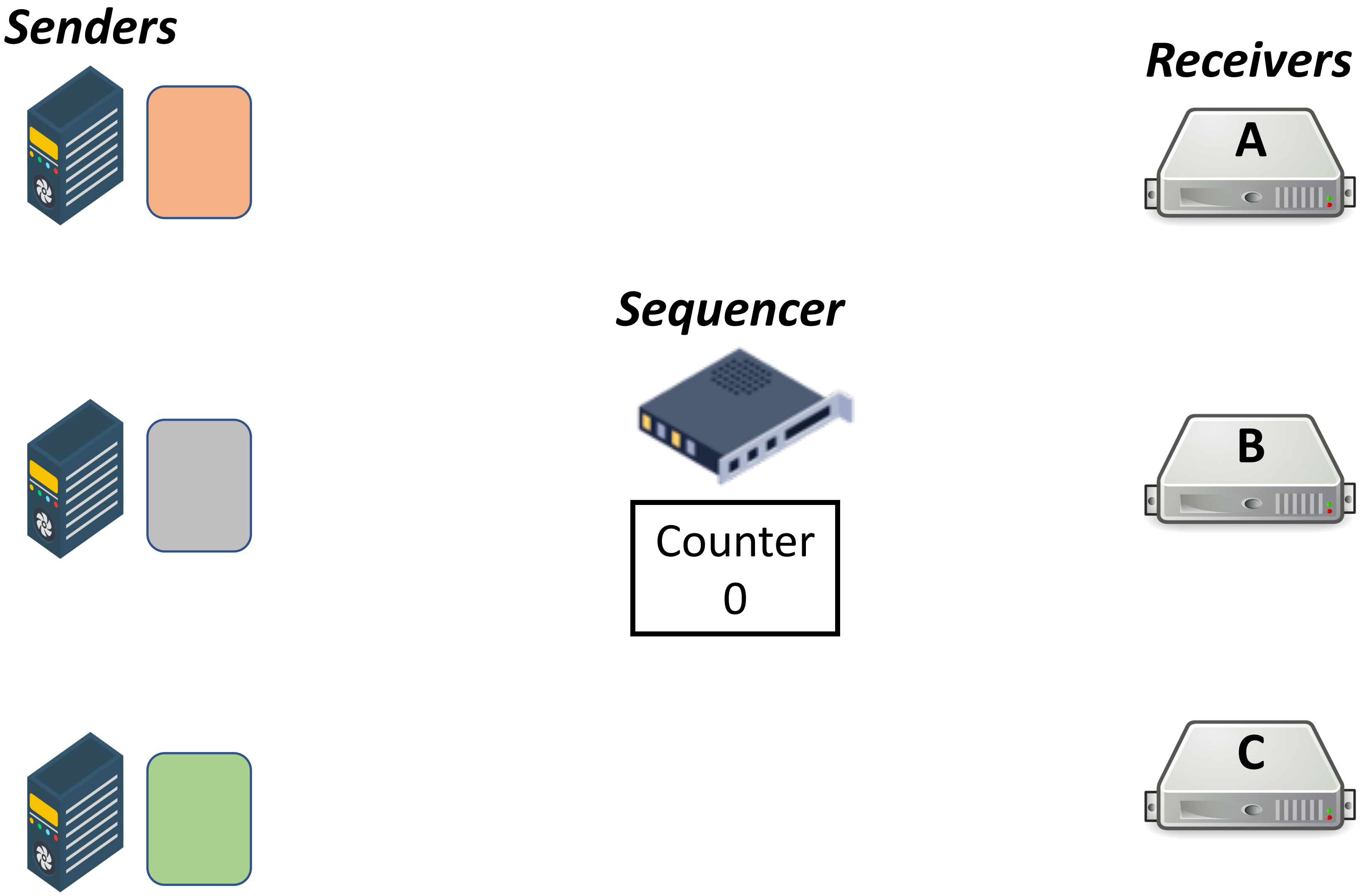
Related Works:
NOPaxos [OSDI ‘16], **Eris** [SOSP ‘17]



Transaction Protocol (2PC)
Replication Protocol (Paxos)
Replication Protocol (Paxos)
Replication Protocol (Paxos)

Network Ordering

Network Ordering



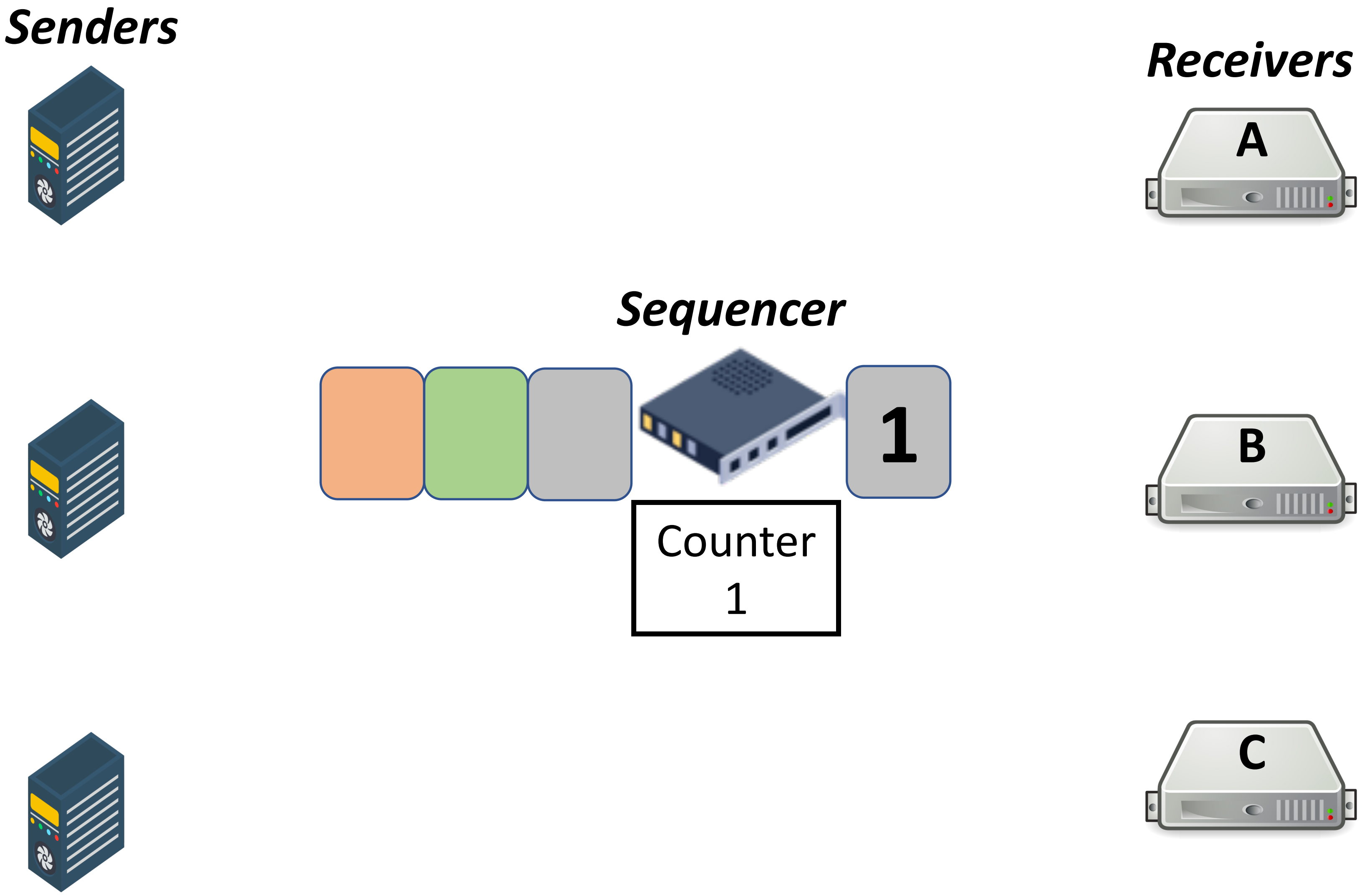
Senders

Sequencer

Counter
0

Receivers

Network Ordering



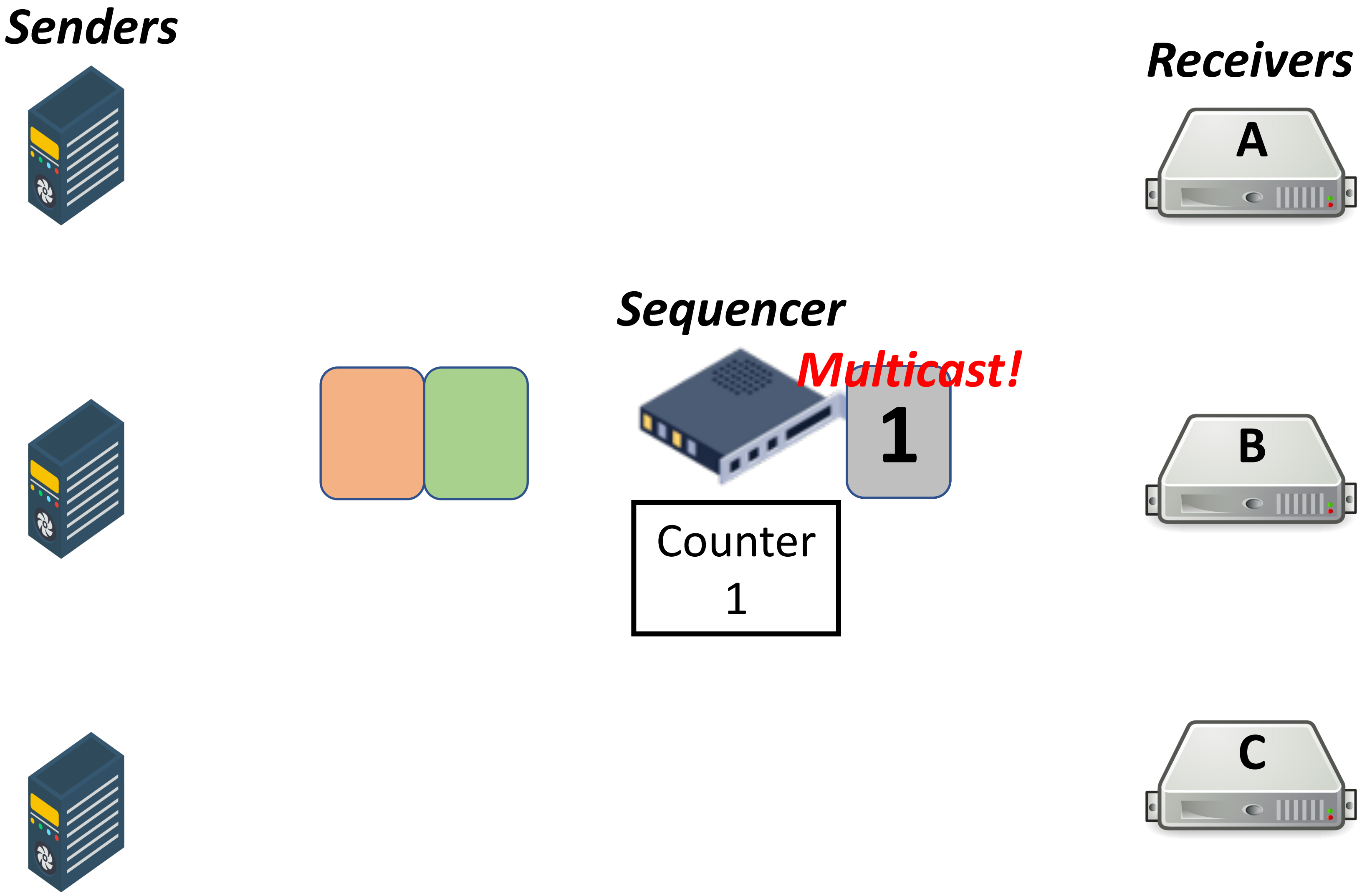
Senders

Sequencer

Counter
1

Receivers

Network Ordering



Senders

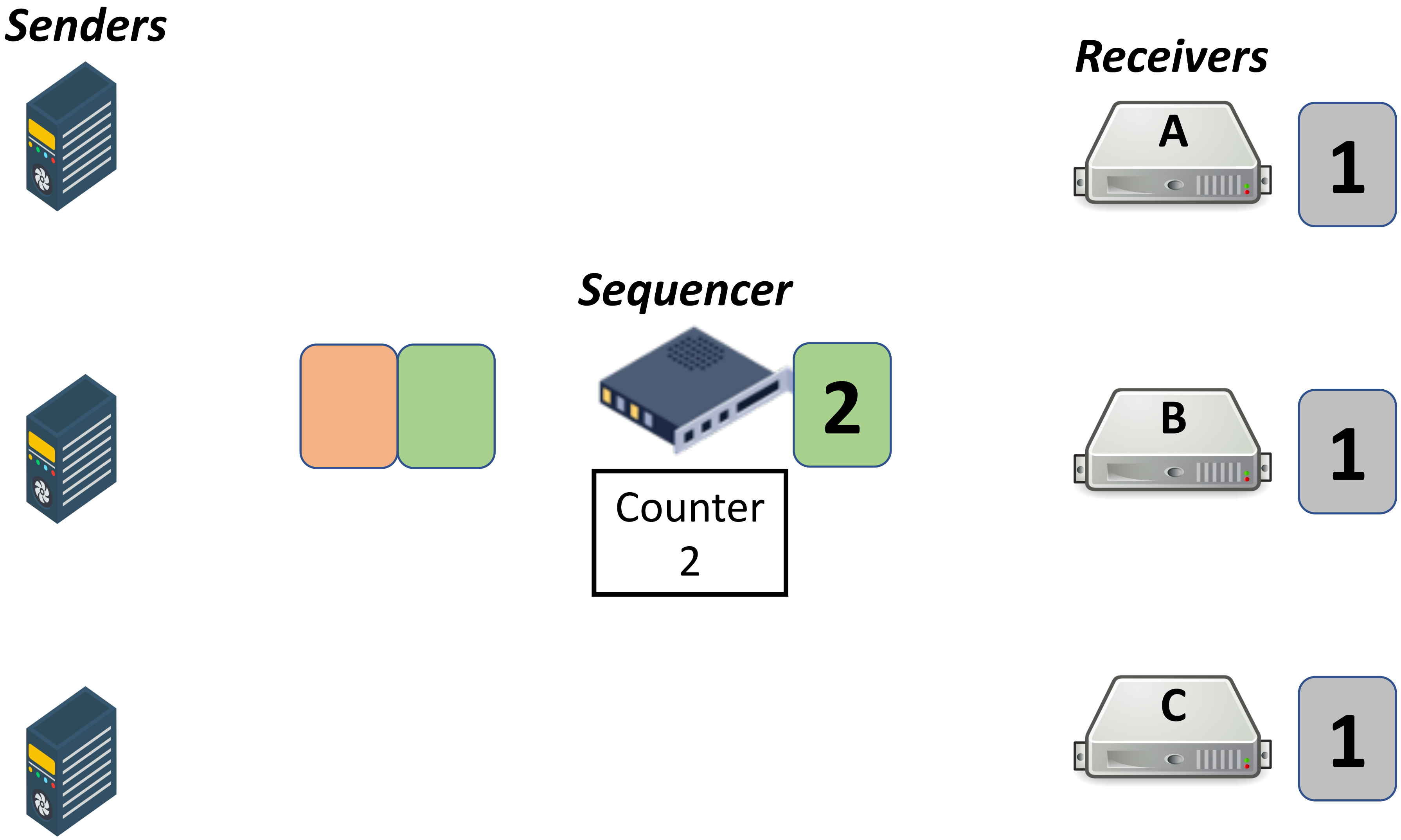
Sequencer

Multicast!

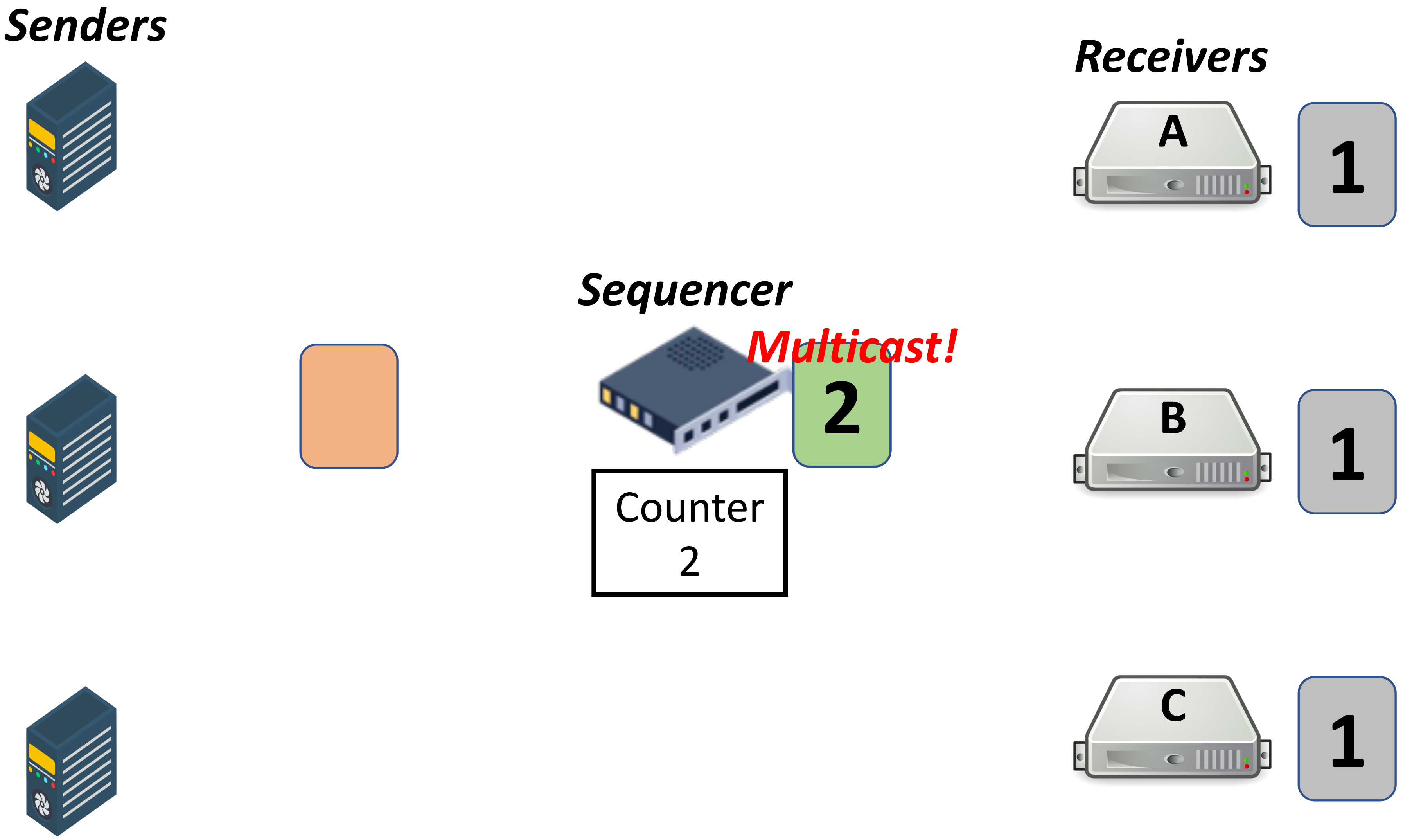
Counter
1

Receivers

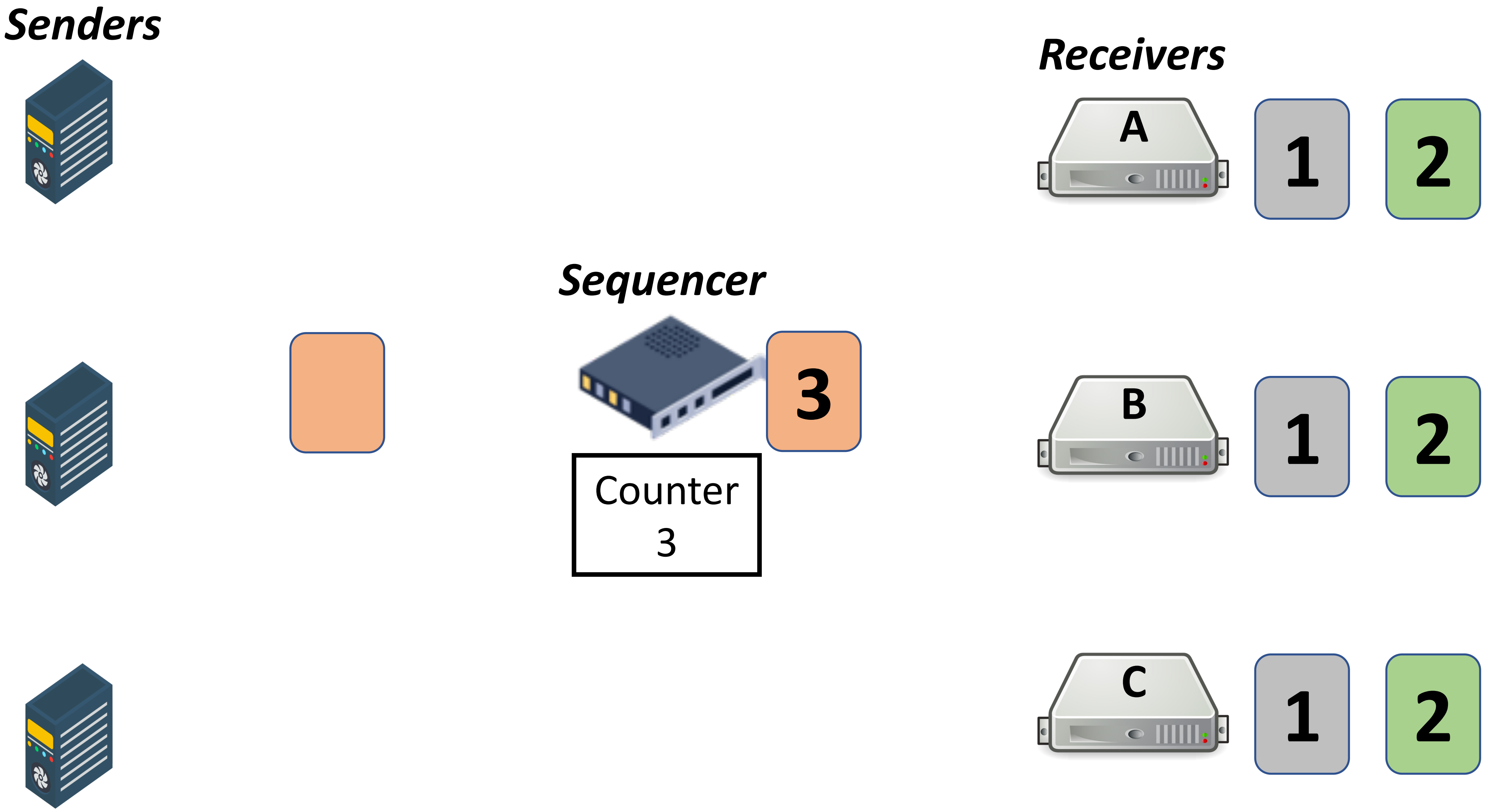
Network Ordering



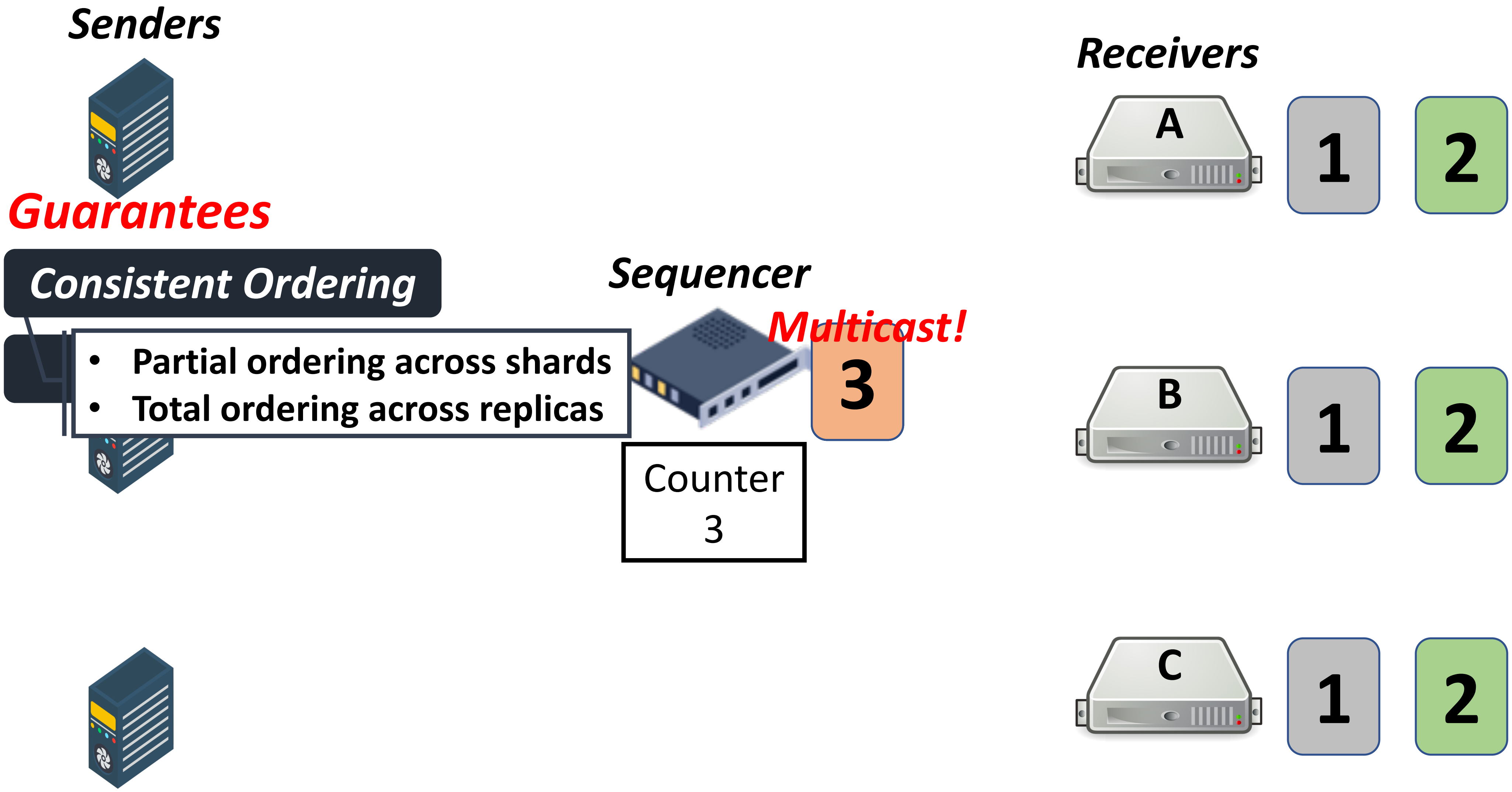
Network Ordering



Network Ordering



Network Ordering



Guarantees

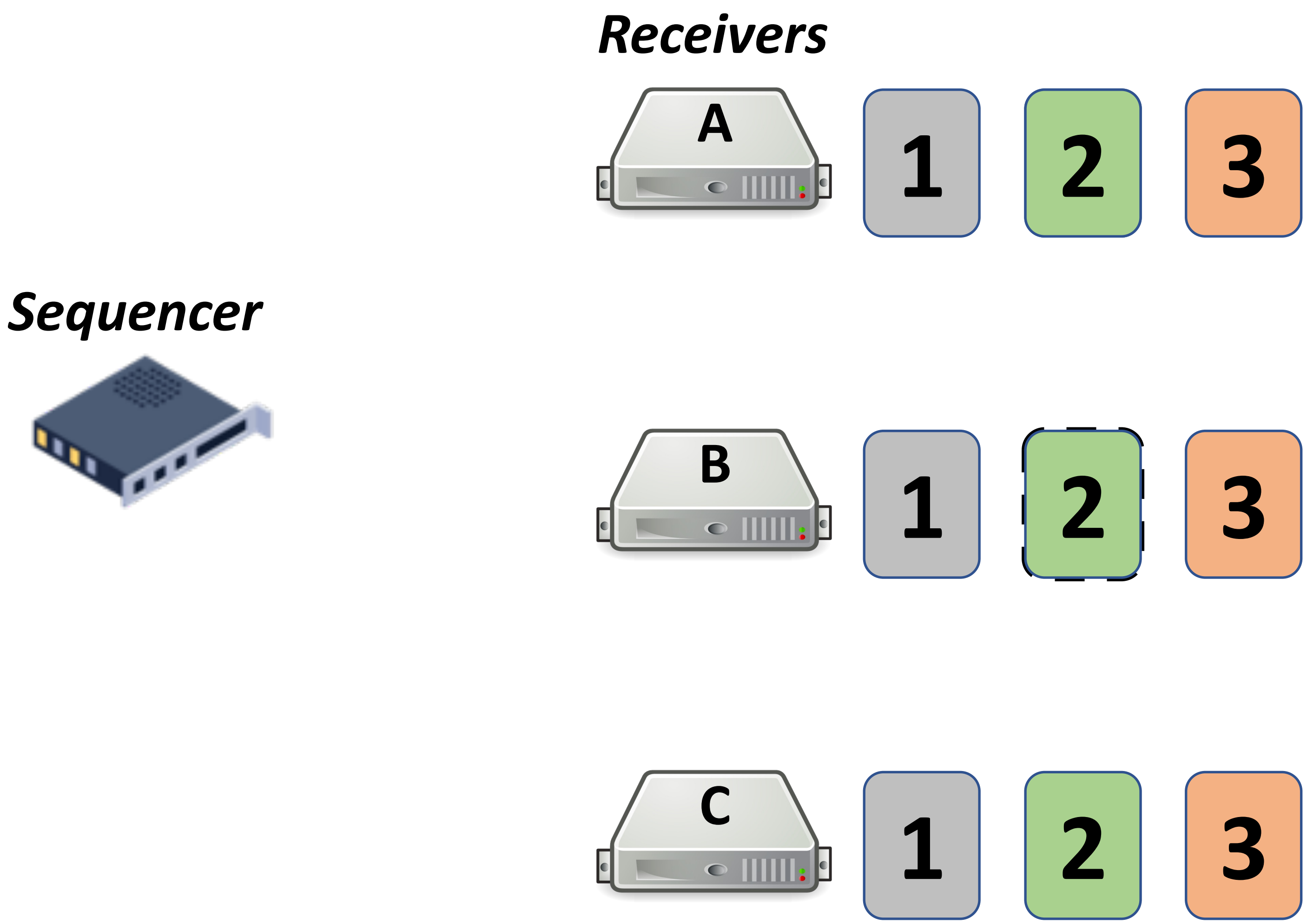
Consistent Ordering

- Partial ordering across shards
- Total ordering across replicas

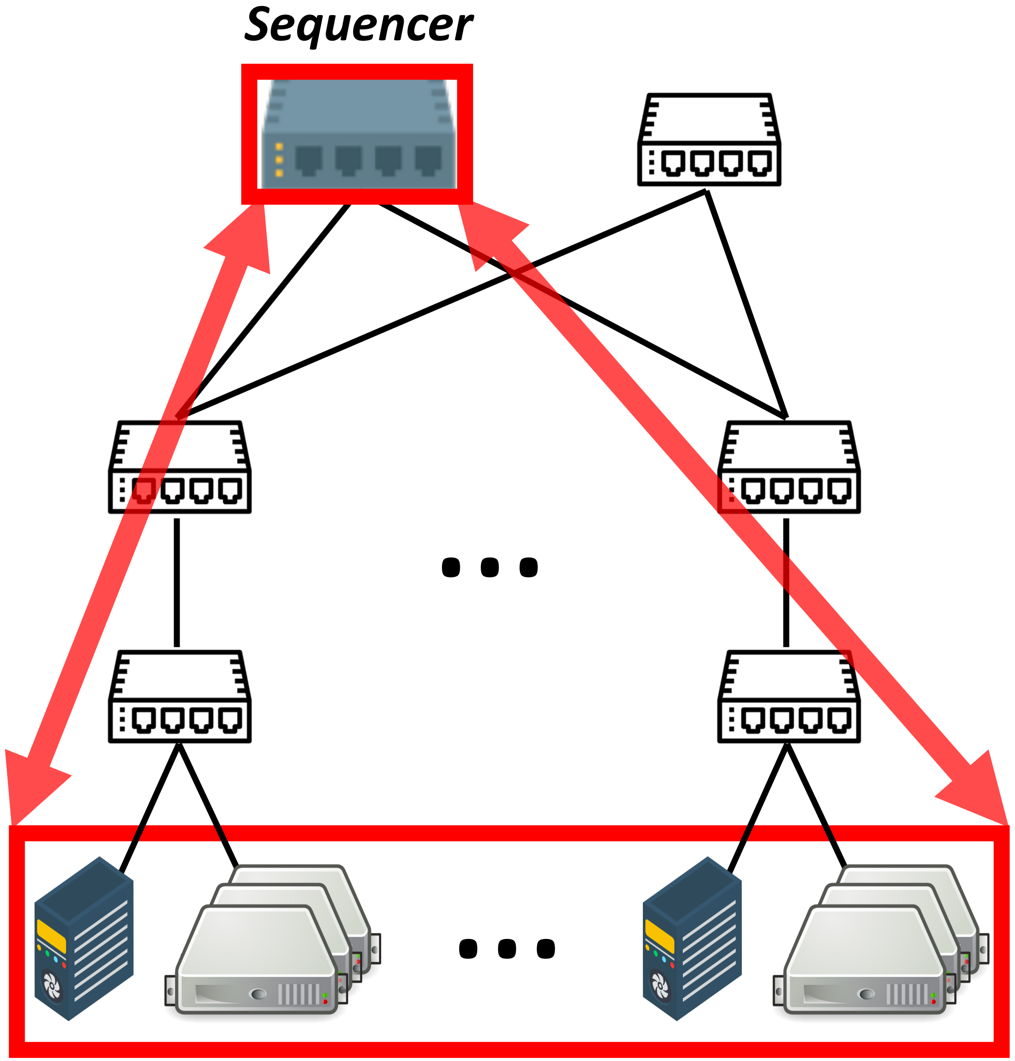
Network Ordering

Guarantees

- ***Consistent Ordering***
- ***Drop Detection***



Drawbacks due to a Single Sequencer



All request traffic

must go through the *single sequencer*

(Network Serialization)

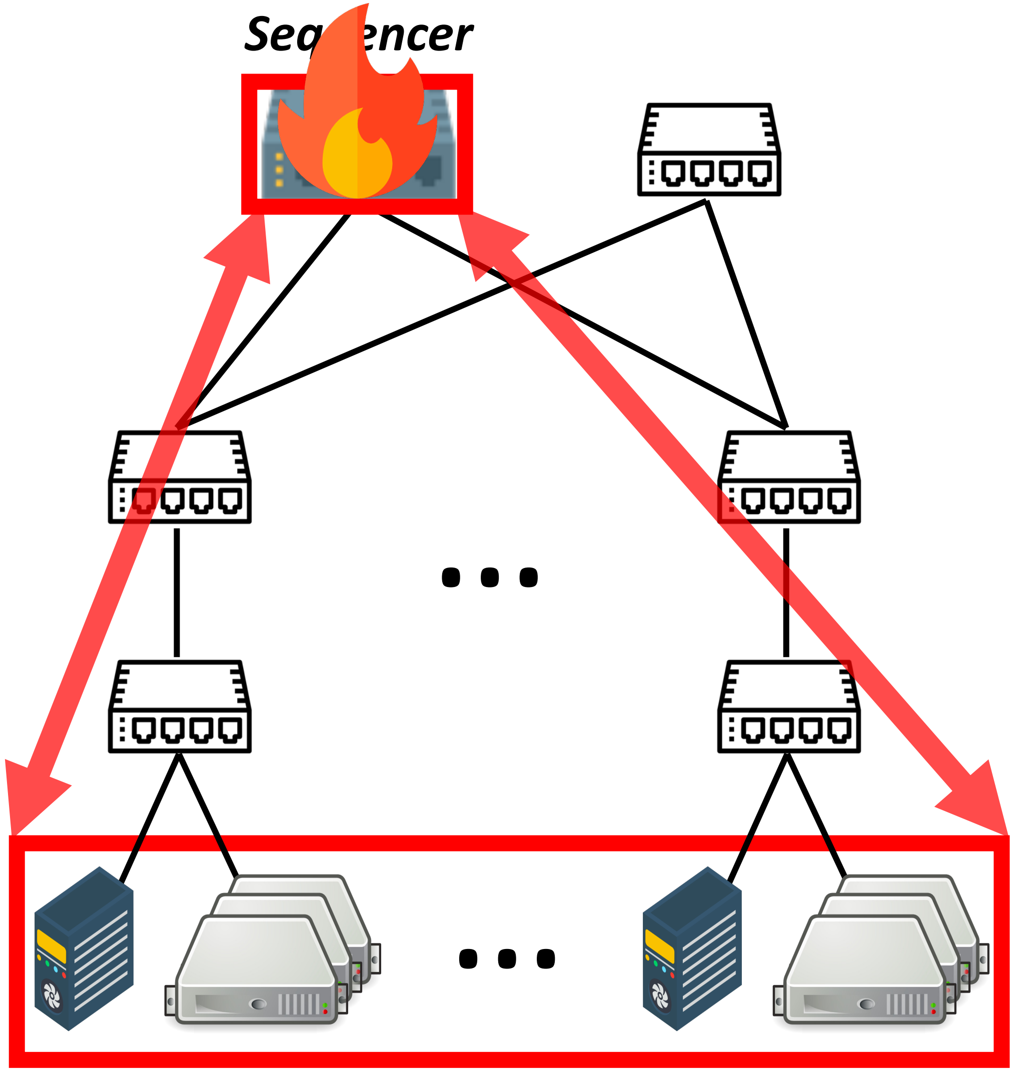
⬇

☒ ***Network load imbalance ⇒ high latency***

☒ ***Sequencer scalability bottleneck***

☒ ***Prolonged sequencer failover***

Drawbacks due to a Single Sequencer



All request traffic

must go through the *single sequencer*

(*Network Serialization*)

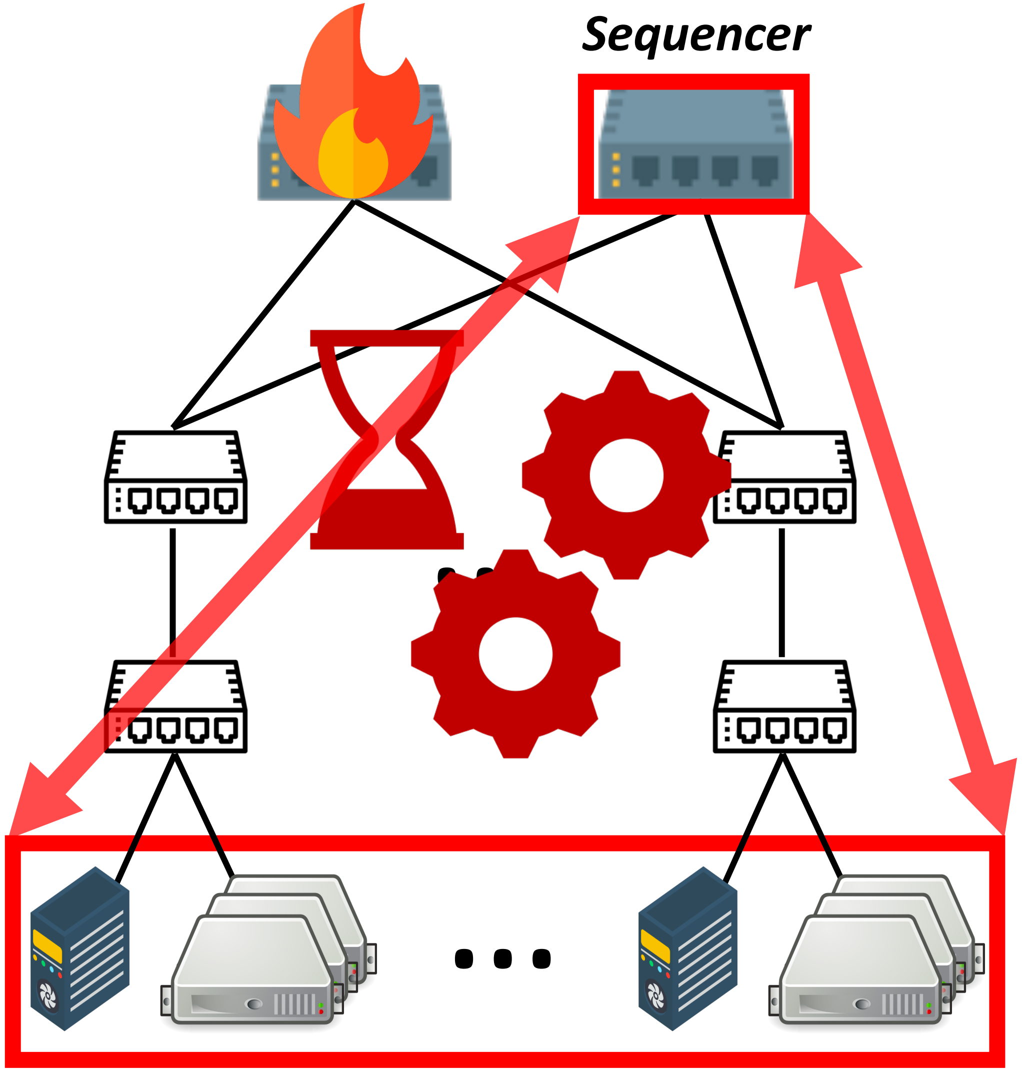
⬇

☒ ***Network load imbalance ⇒ high latency***

☒ ***Sequencer scalability bottleneck***

☒ ***Prolonged sequencer failover***

Drawbacks due to a Single Sequencer



All request traffic

must go through the *single sequencer*

(Network Serialization)

⬇

☒ ***Network load imbalance ⇒ high latency***

☒ ***Sequencer scalability bottleneck***

☒ ***Prolonged sequencer failover***

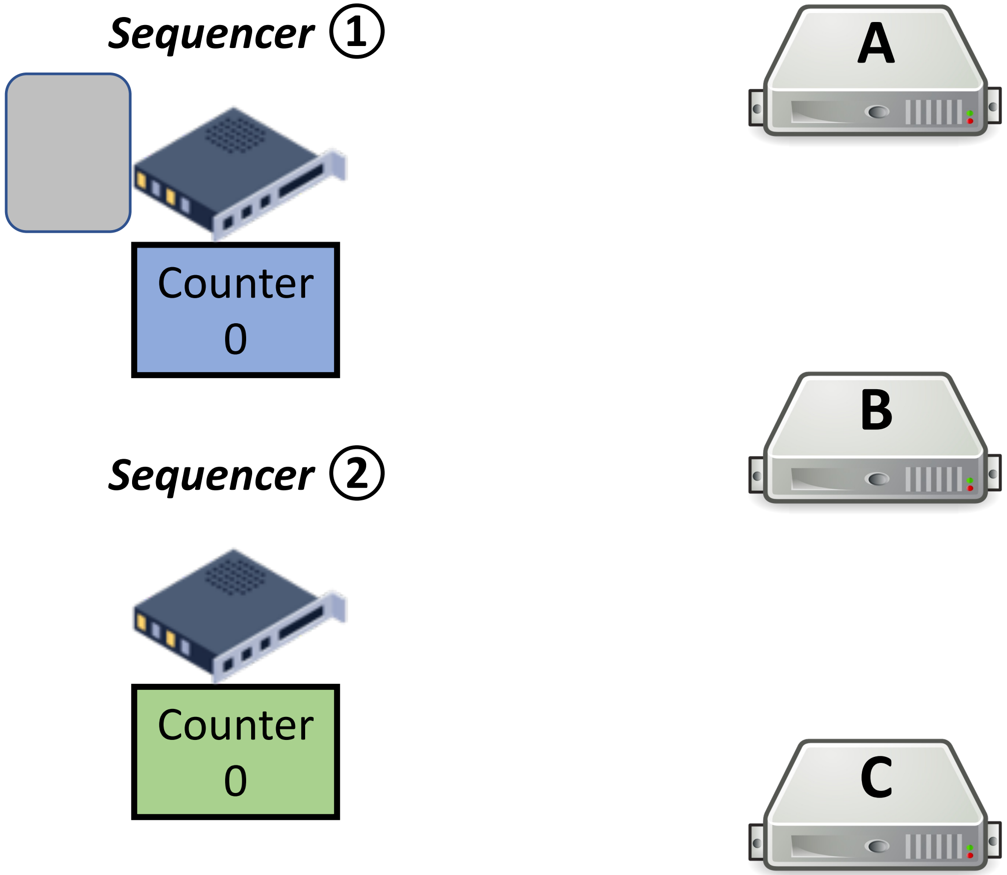
Well, let’s try multiple sequencers

Multi-Sequencer Challenge:

Guarantees

Consistent Ordering

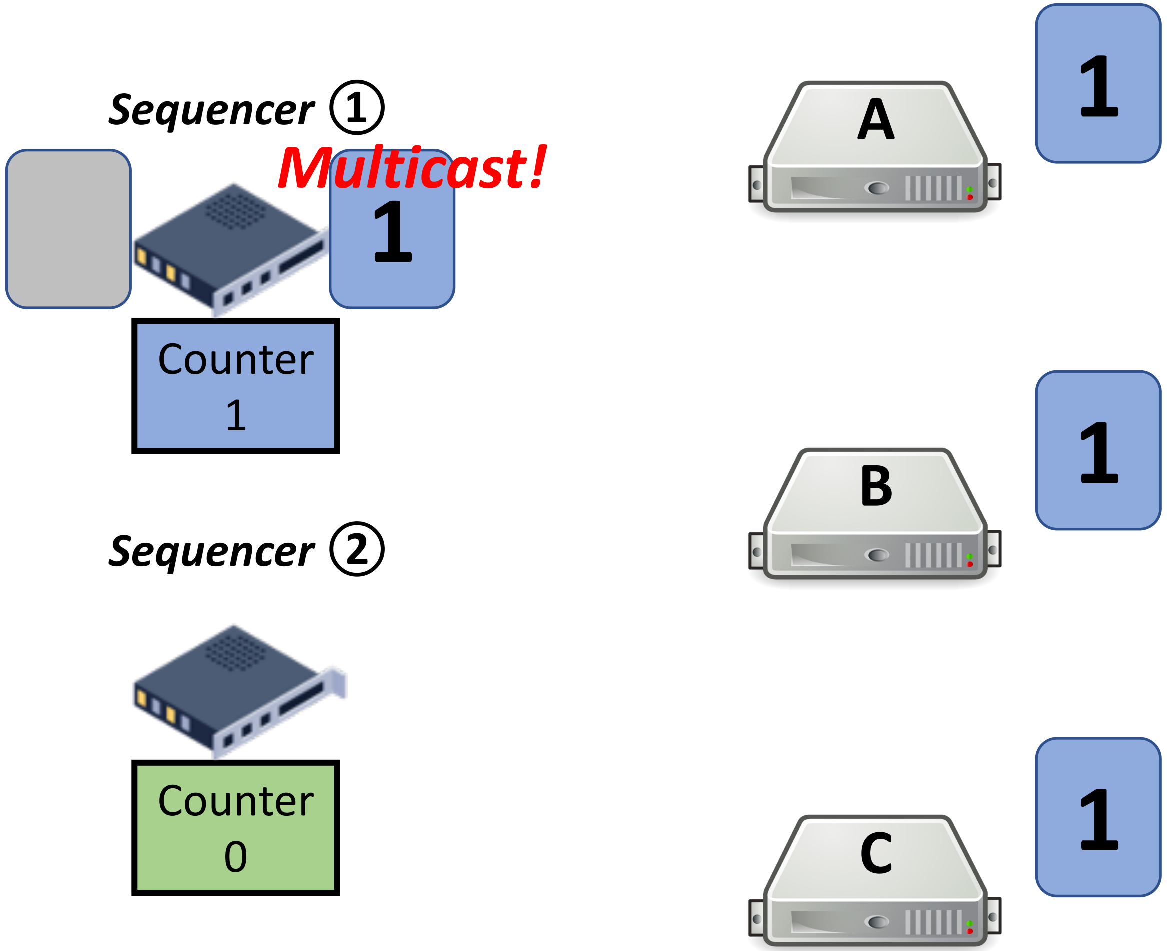
Drop Detection



Multi-Sequencer Challenge:

Guarantees

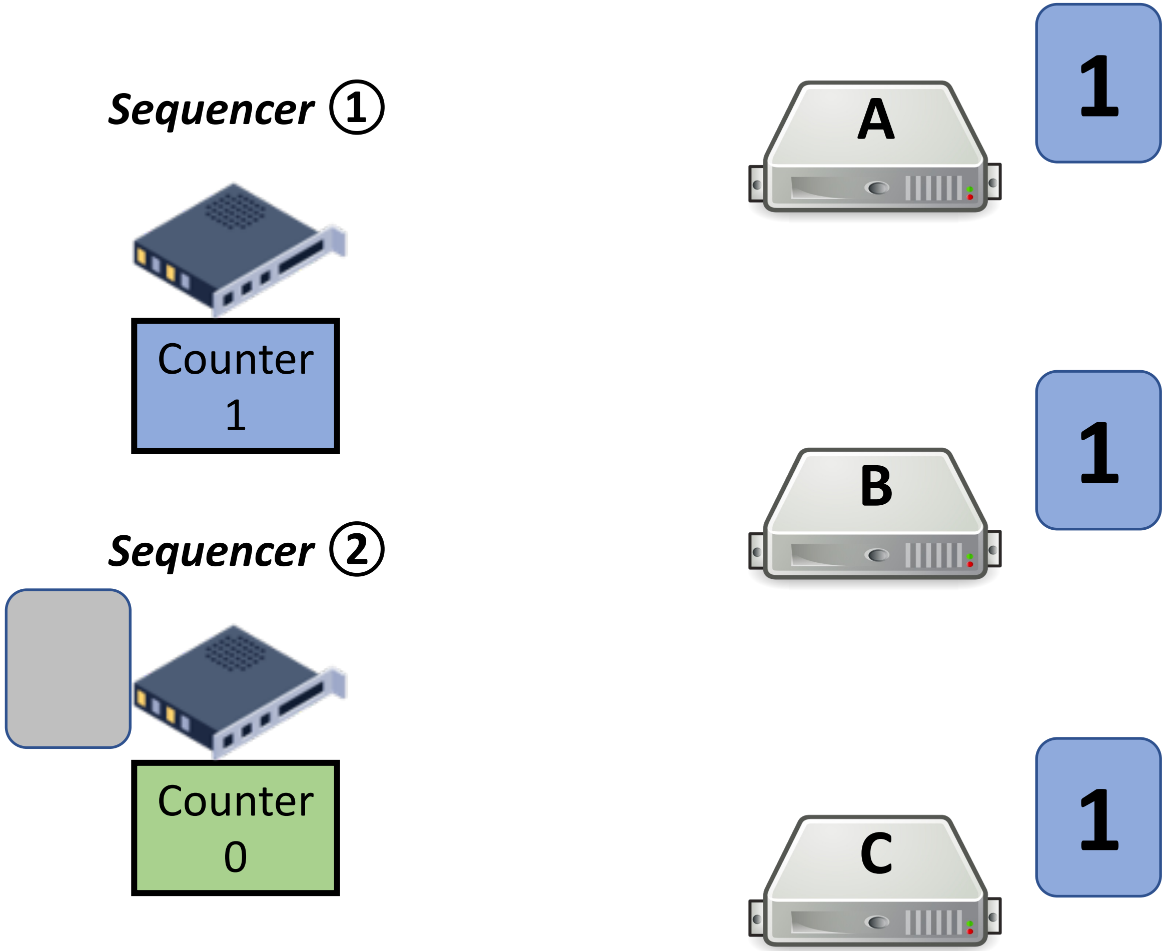
- ***Consistent Ordering***
- ***Drop Detection***



Multi-Sequencer Challenge:

Guarantees

- ***Consistent Ordering***
- ***Drop Detection***



Sequencer ①

Counter
1

Sequencer ②

Counter
0

A 1

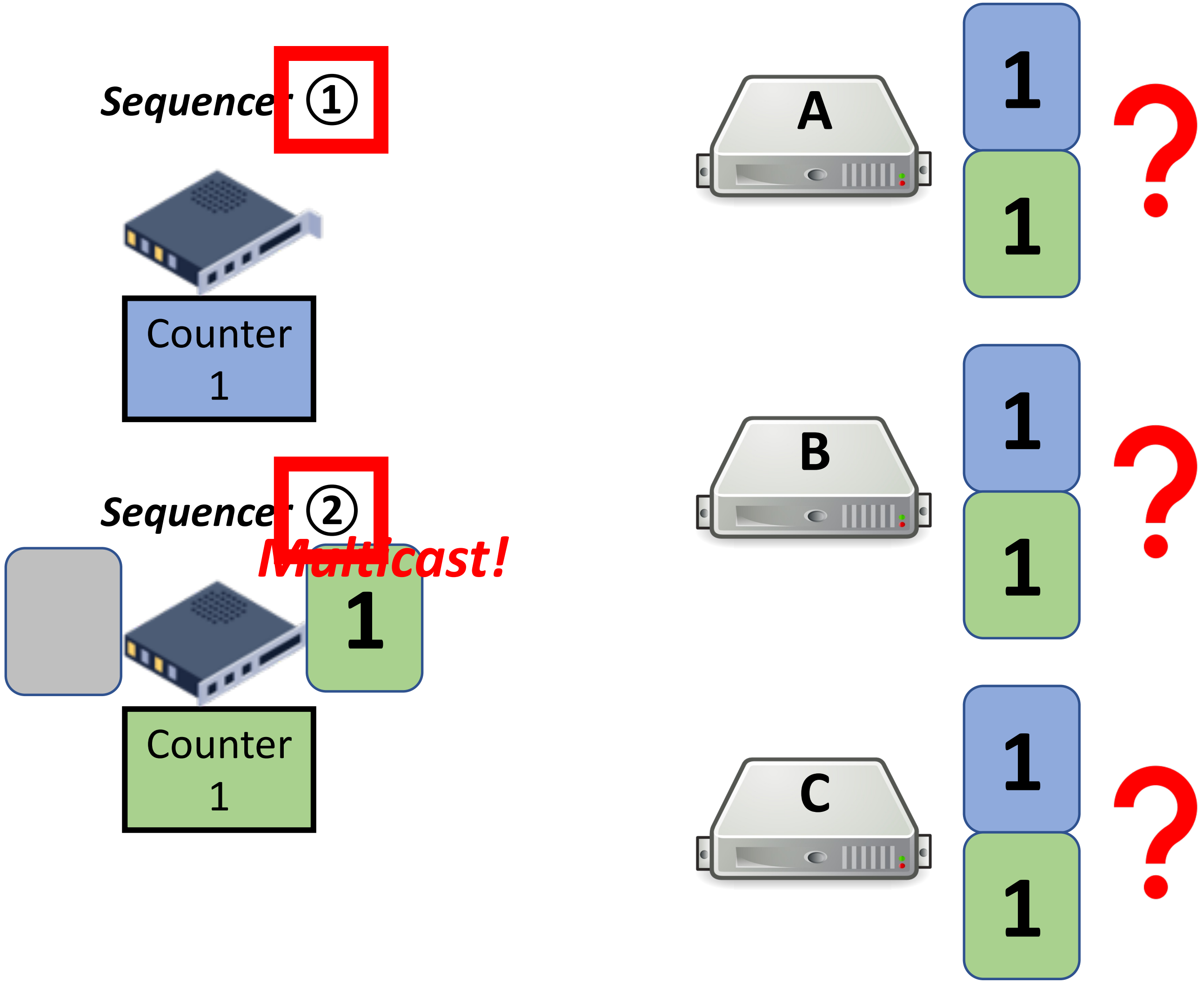
B 1

C 1

Multi-Sequencer Challenge:

Guarantees

- ~~*Consistent Ordering*~~
- *Drop Detection*



Sequencer ①

Counter
1

Sequencer ②

Multicast!

1

Counter
1

A 1 1 ?

B 1 1 ?

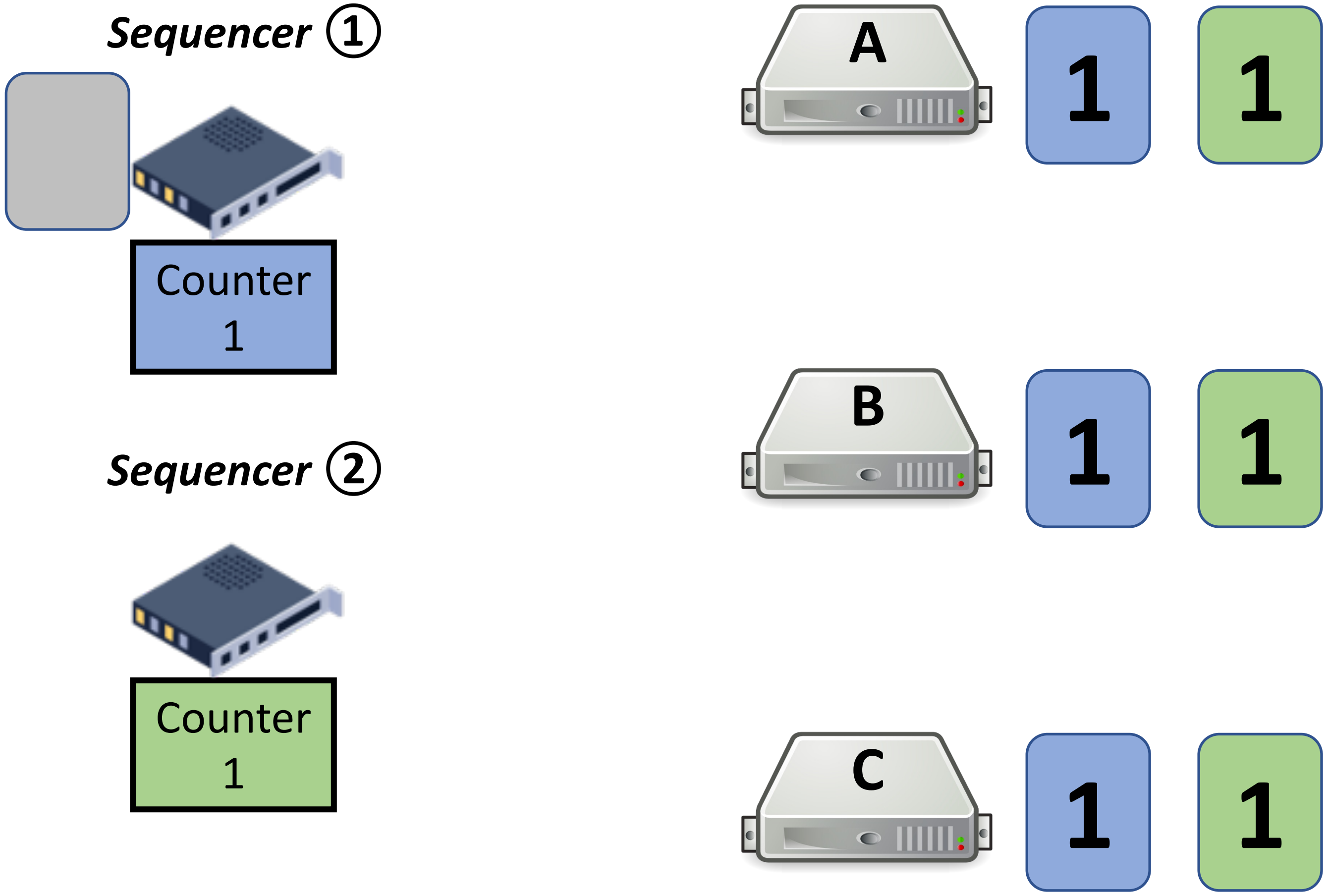
C 1 1 ?

Multi-Sequencer Challenge:

Guarantees

Consistent Ordering

Drop Detection



Sequencer ①

Counter
1

Sequencer ②

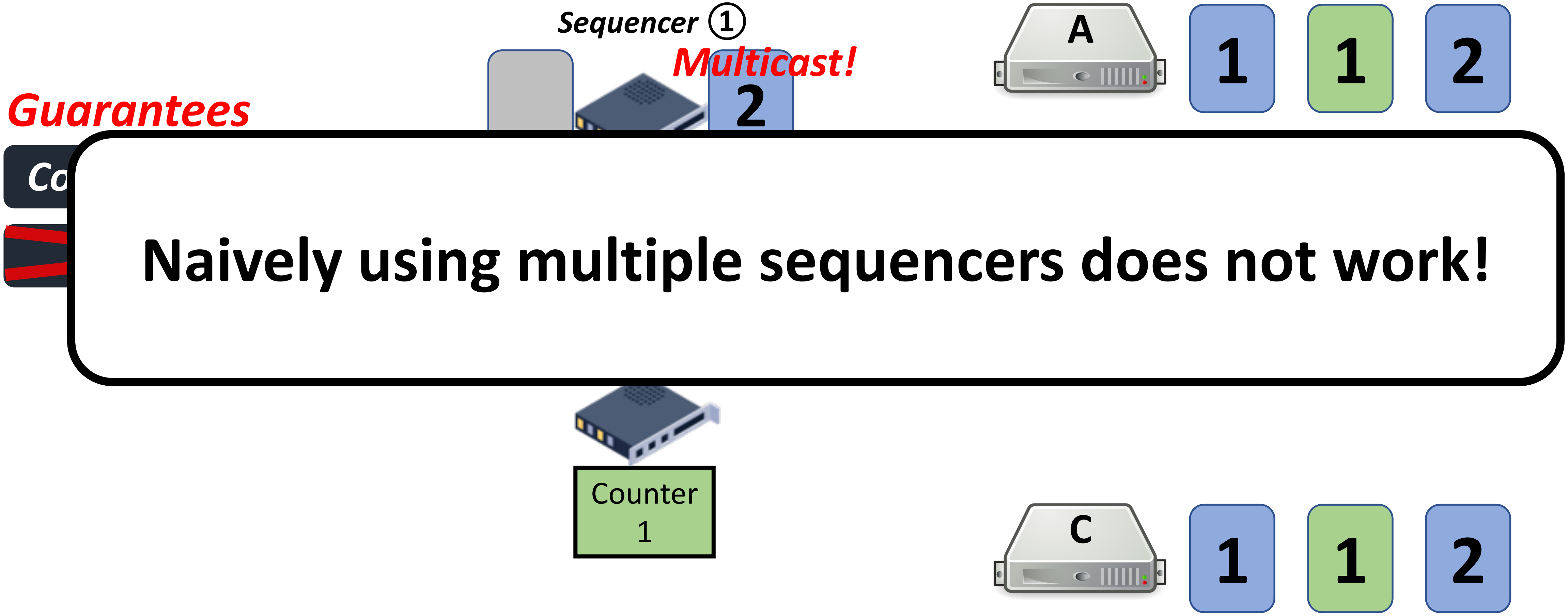
Counter
1

A 1 1

B 1 1

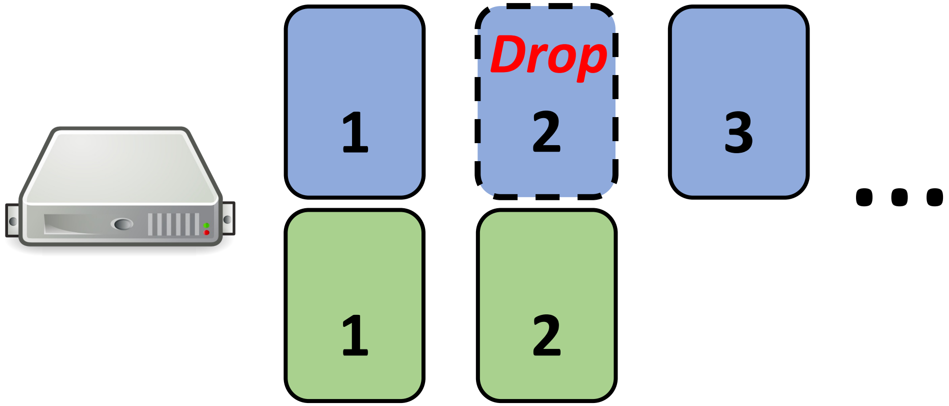
C 1 1

Multi-Sequencer Challenge:



Naively using multiple sequencers does not work!

Solution: Combine sequence number with *physical clock*

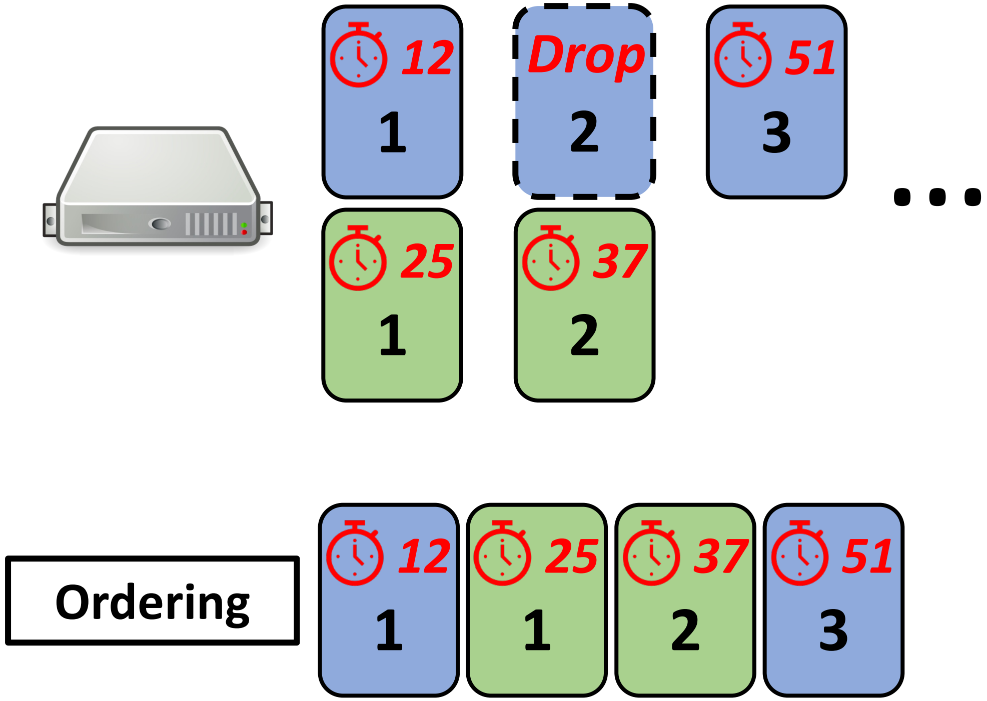


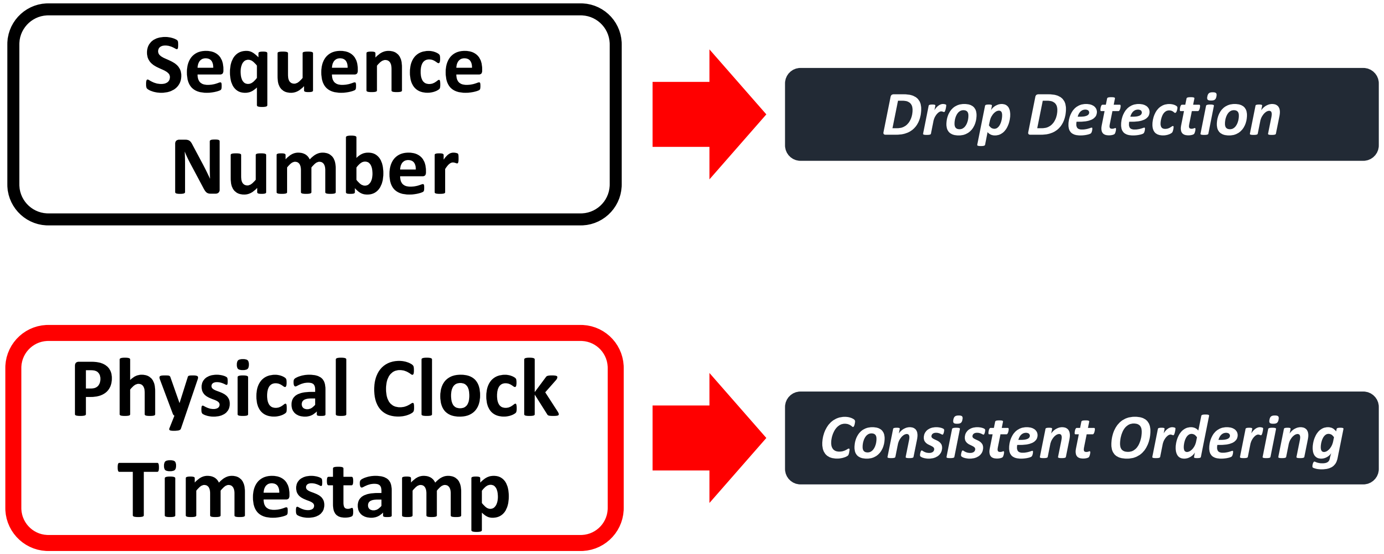
Sequence Number → ***Drop Detection***

Physical Clock Timestamp → ***Consistent Ordering***

- Consistent across receivers
- Strictly monotonically increasing
- Physical clocks can be (loosely) synchronized across sequencers

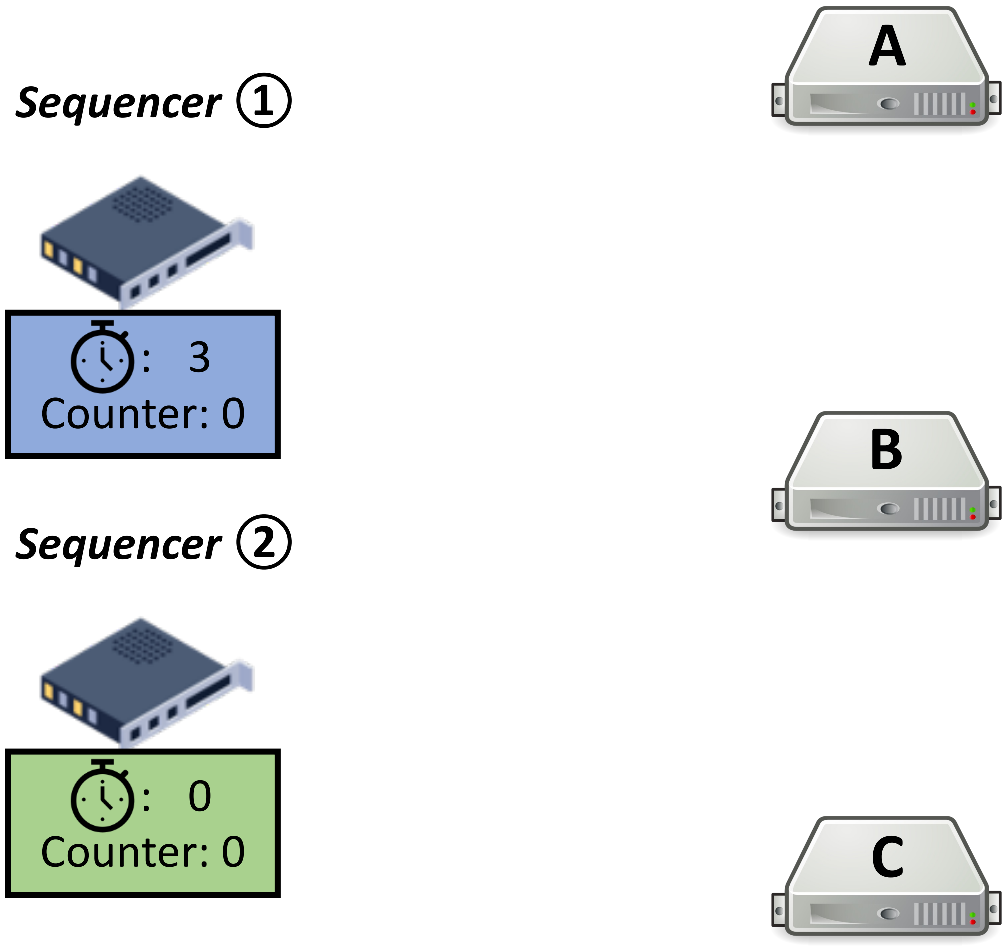
Solution: Combine sequence number with *physical clock*





- Consistent across receivers
- Strictly monotonically increasing
- Physical clocks can be (loosely) synchronized across sequencers

Hydra Network Primitive - Sequencers



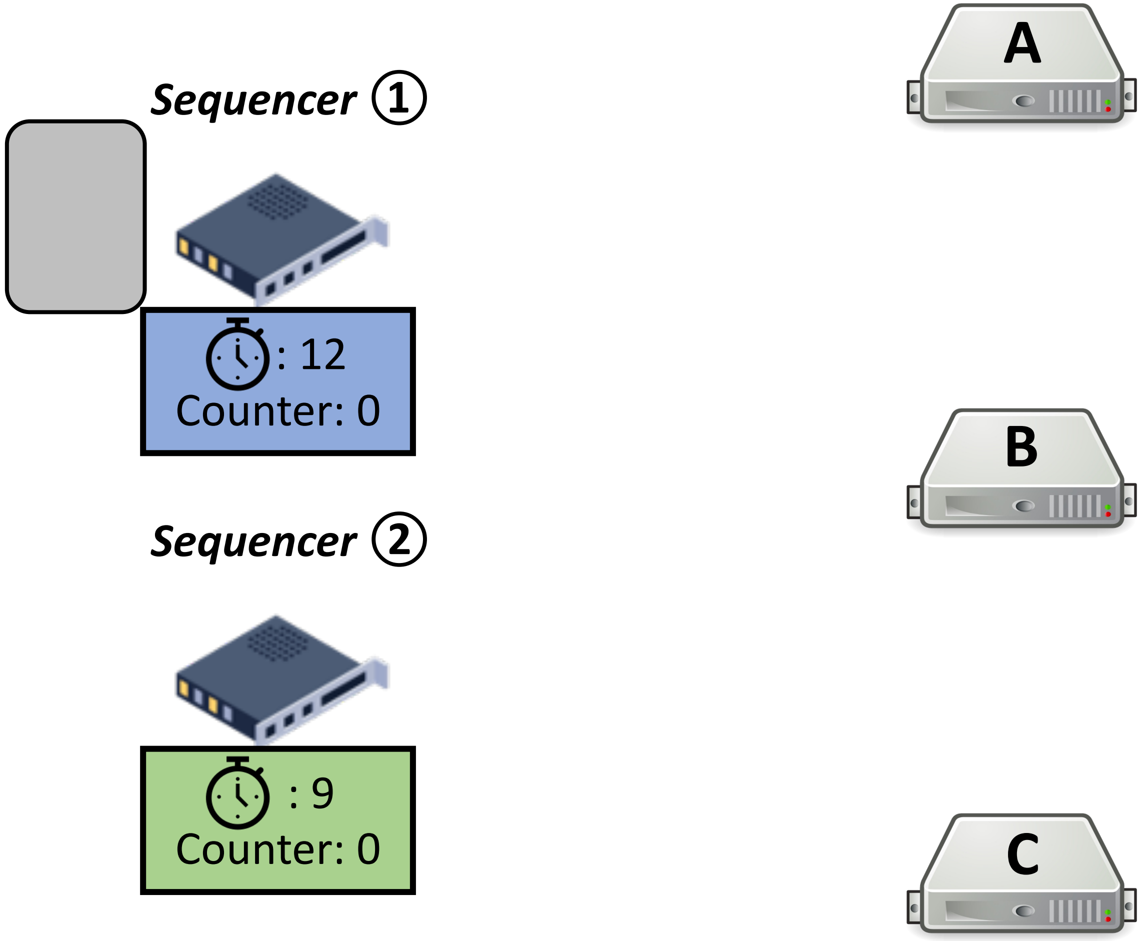
Sequencer ①

⏱: 3
Counter: 0

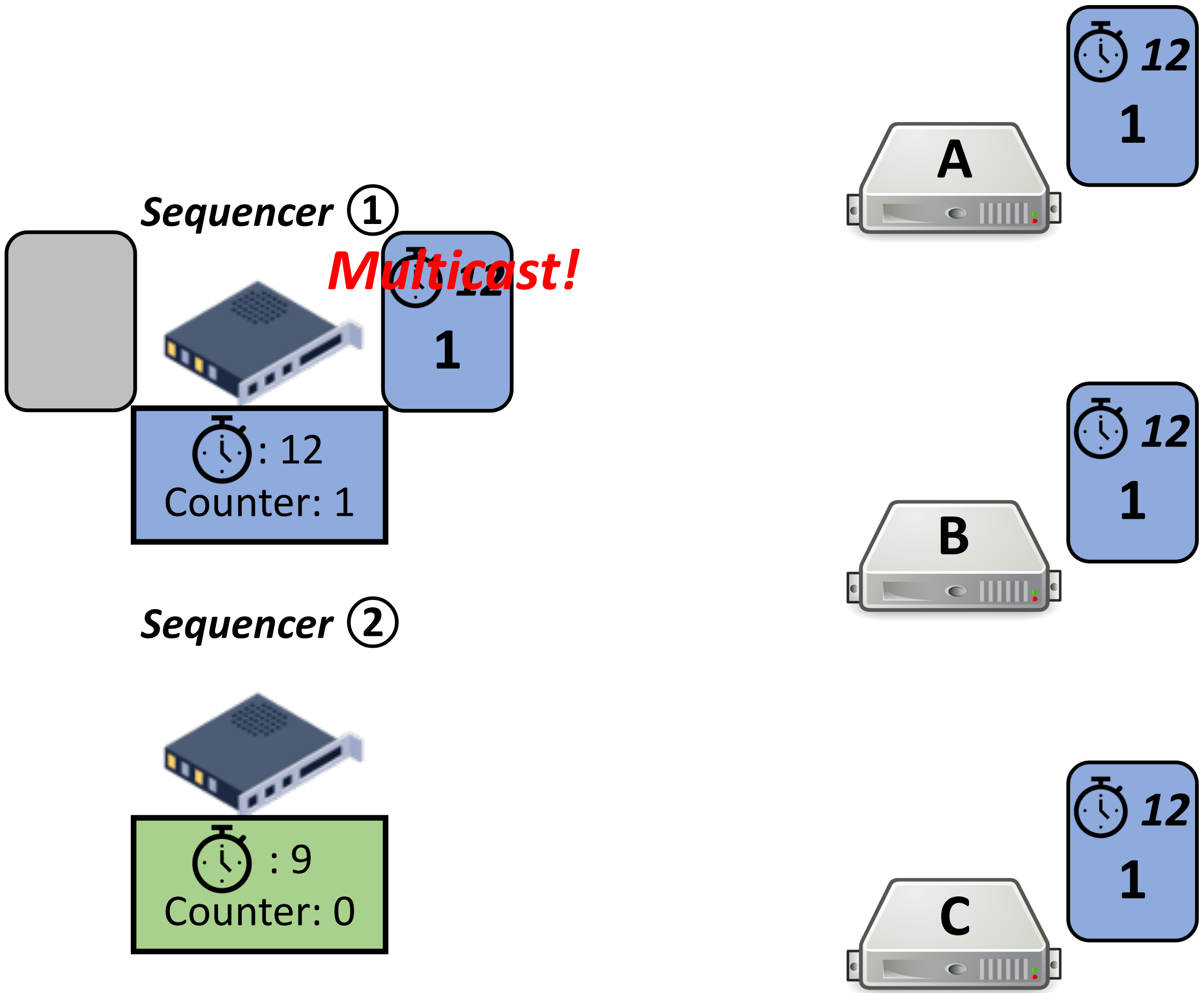
Sequencer ②

⏱: 0
Counter: 0

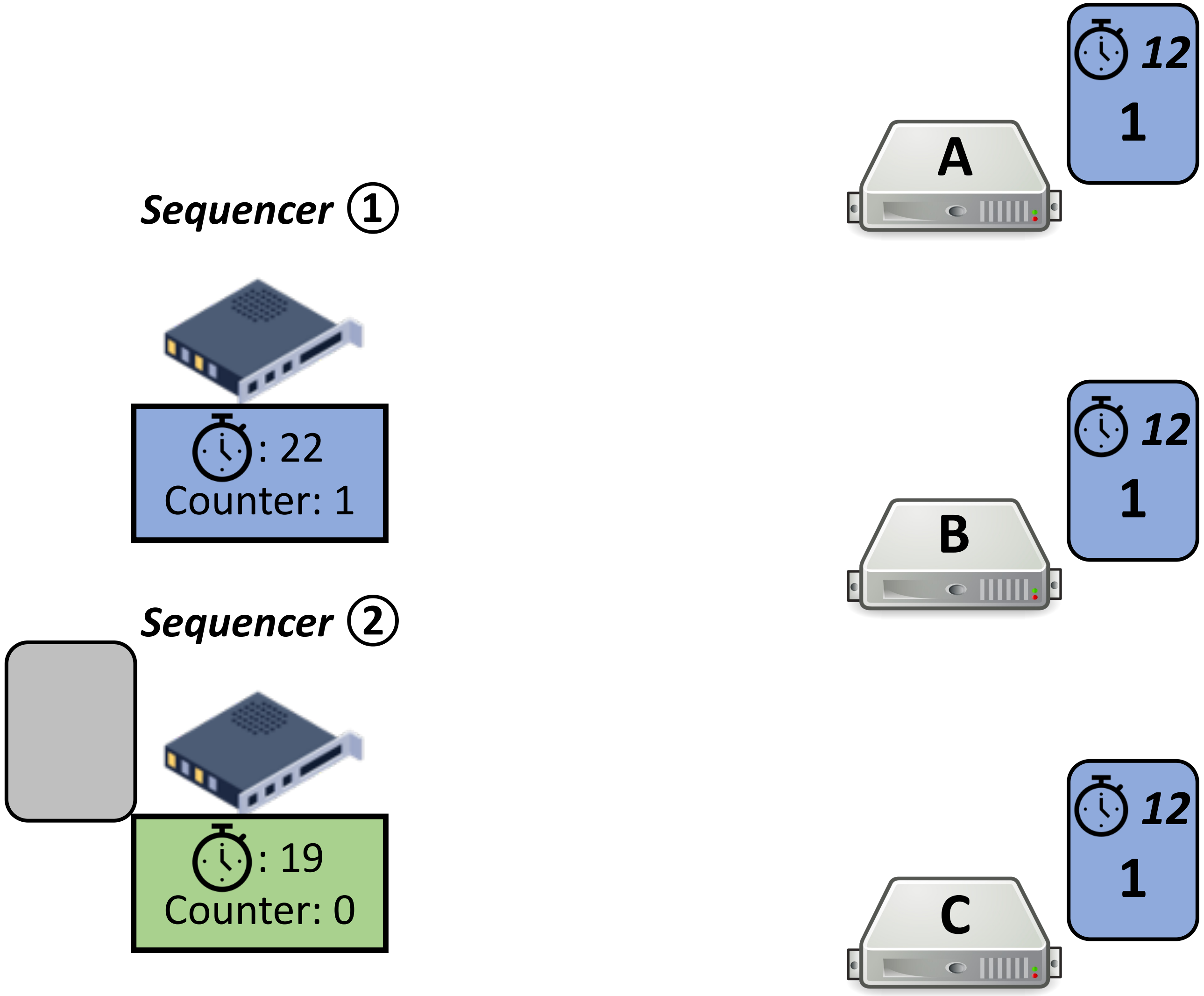
Hydra Network Primitive - Sequencers



Hydra Network Primitive - Sequencers



Hydra Network Primitive - Sequencers



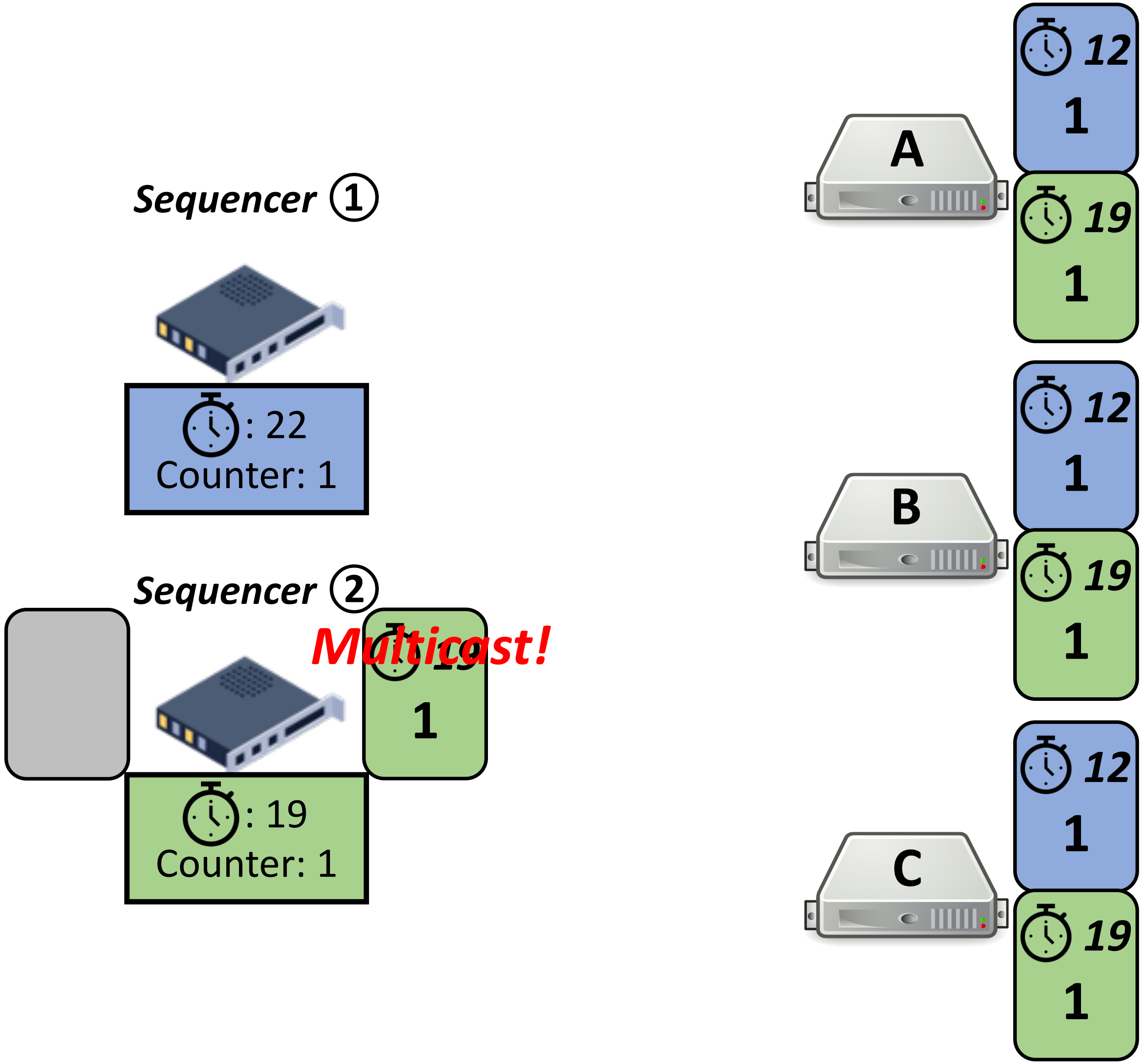
Sequencer ①

⏱: 22
Counter: 1

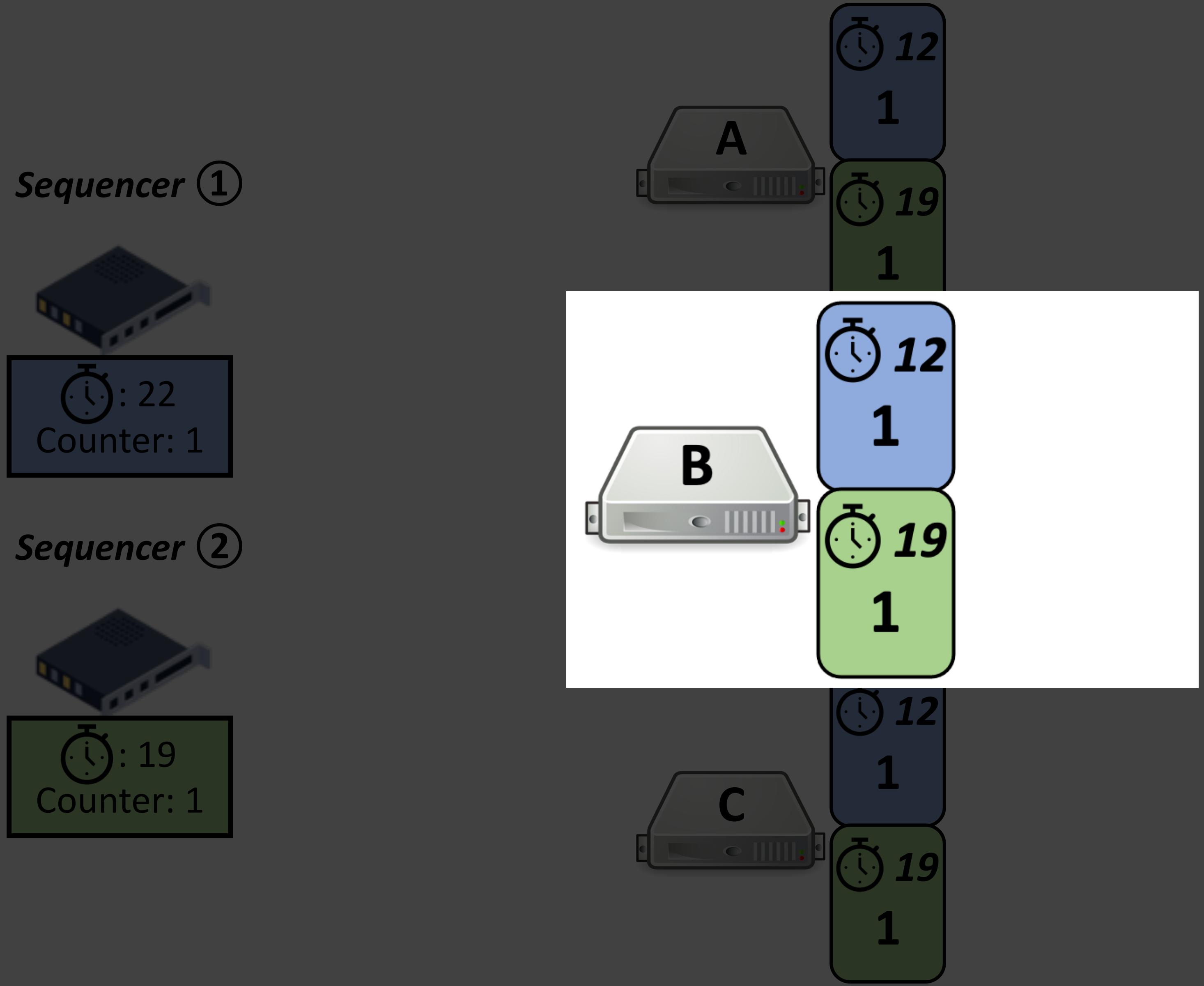
Sequencer ②

⏱: 19
Counter: 0

Hydra Network Primitive - Sequencers



Hydra Network Primitive - Sequencers



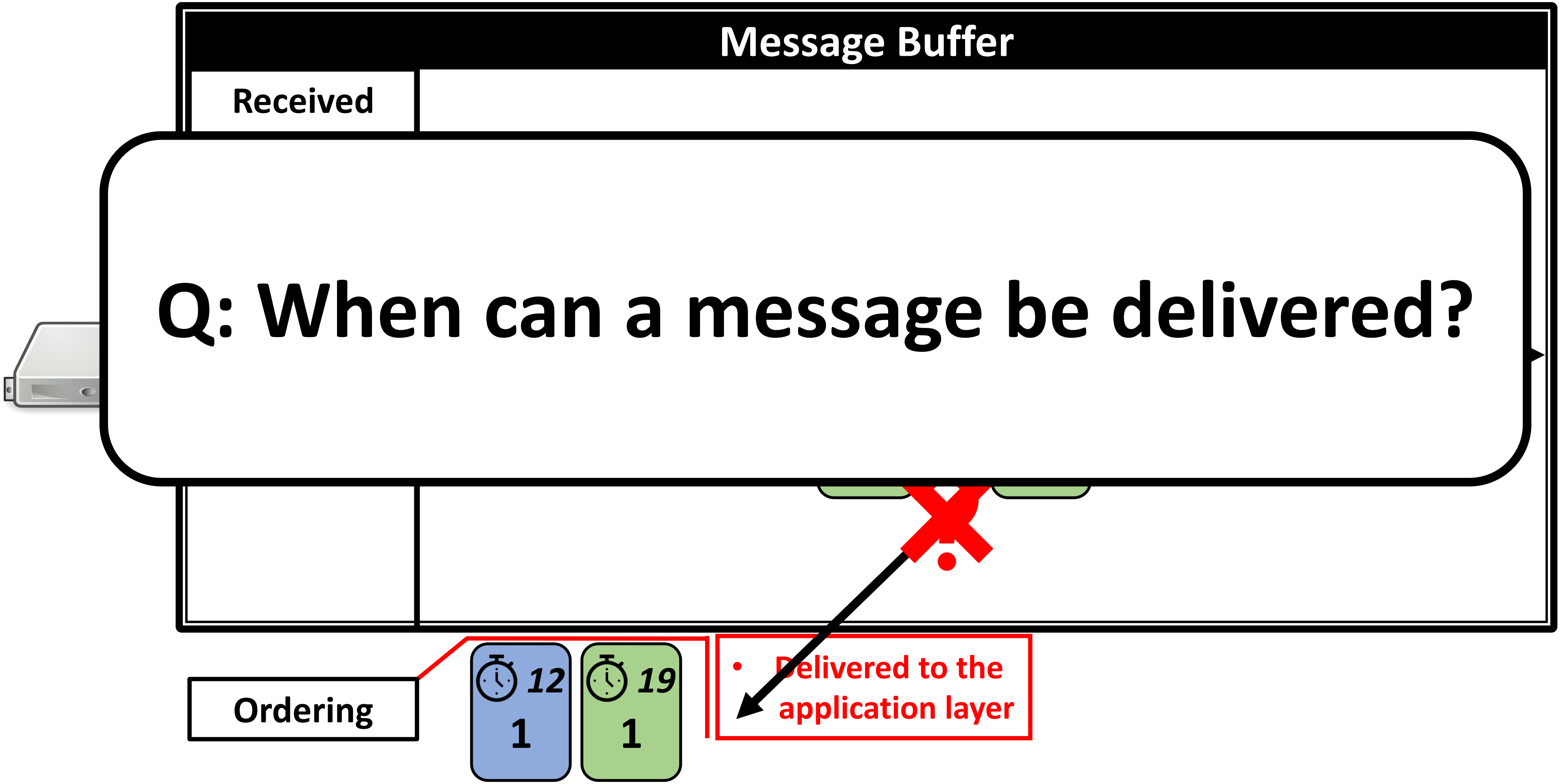
Sequencer ①

⏱: 22
Counter: 1

Sequencer ②

⏱: 19
Counter: 1

Hydra Network Primitive - Receivers



Message Buffer

Received

Q: When can a message be delivered?

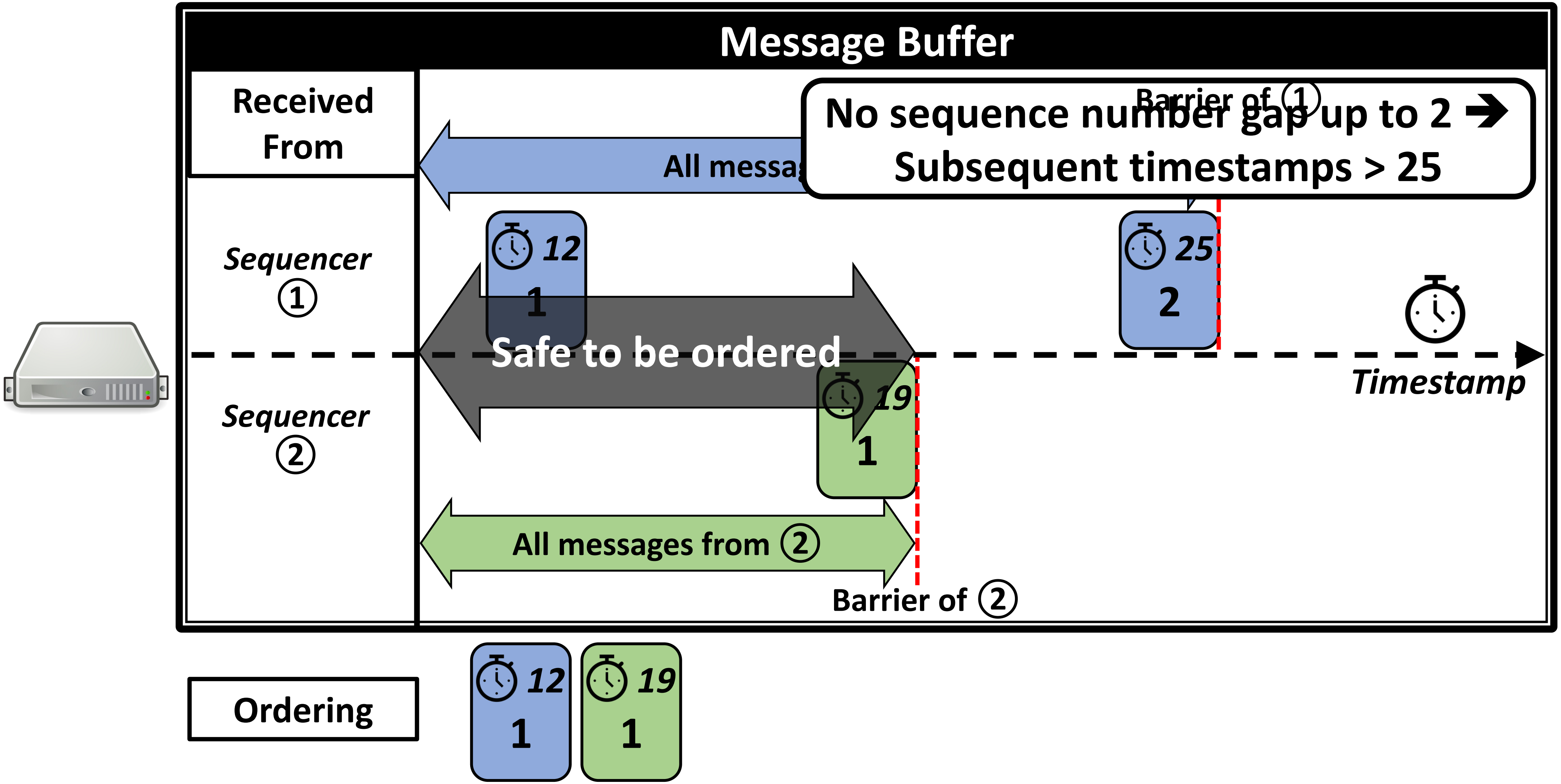
Ordering

12
1

19
1

- Delivered to the application layer

A: Once **ALL** messages with lower timestamp have been received



Message Buffer

Received From

Sequencer ①

Sequencer ②

All messag

No sequence number gap up to 2 ➔
Subsequent timestamps > 25

Barrier of ①

12 1

25 2

Safe to be ordered

19 1

Timestamp

All messages from ②

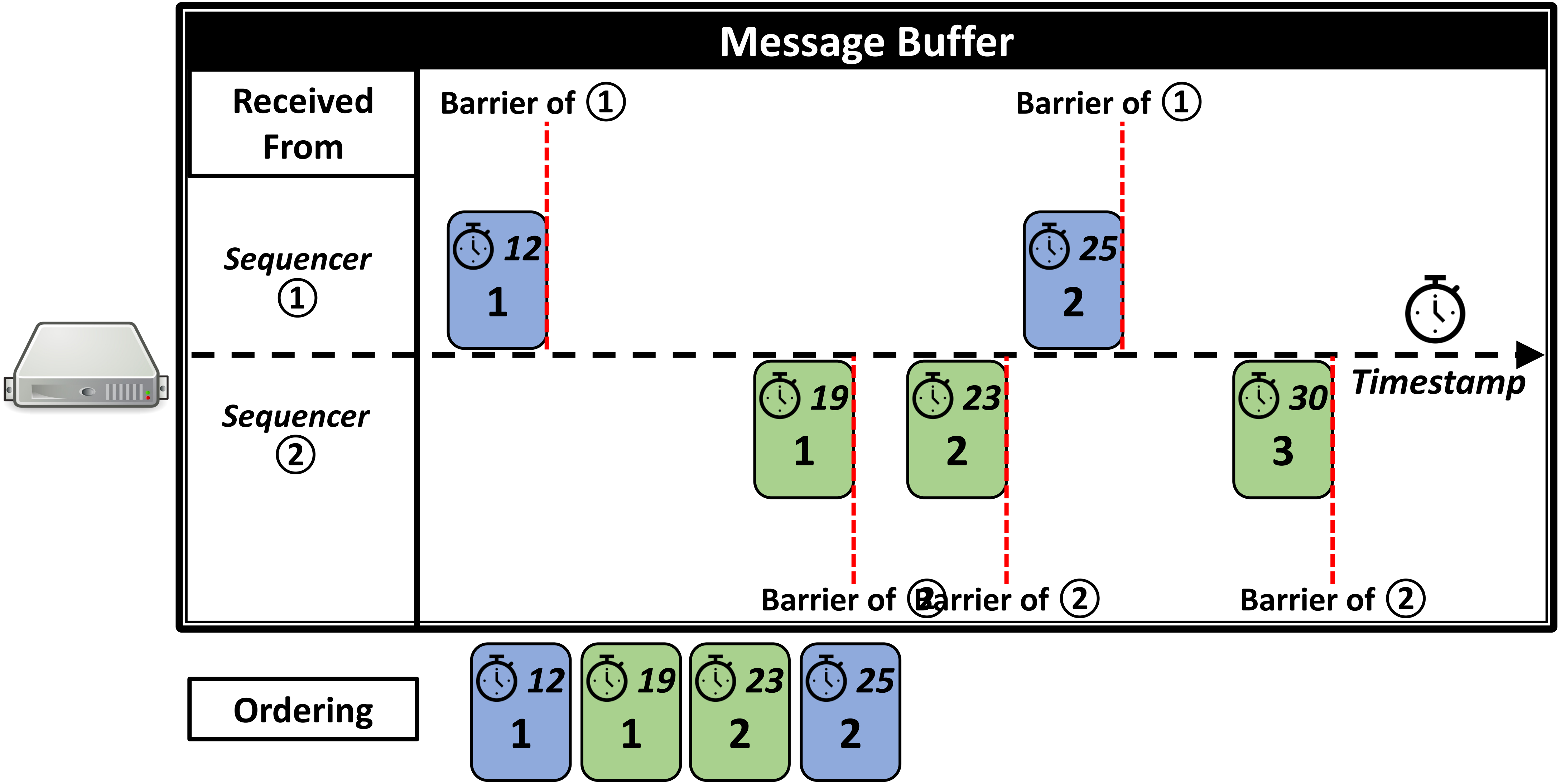
Barrier of ②

Ordering

12 1

19 1

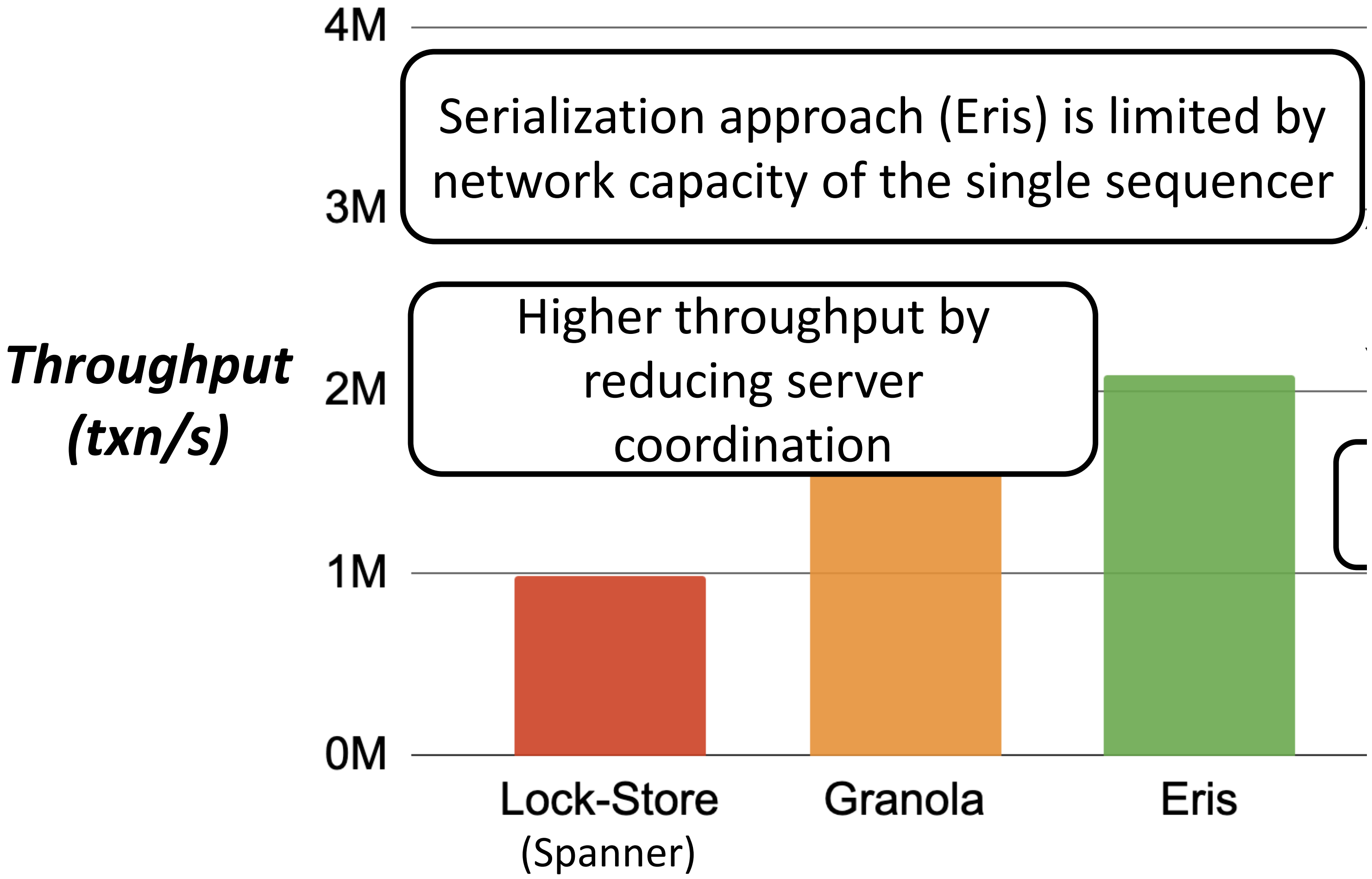
Deliver messages up to *the minimum barrier*



Other Hydra designs

- Flush messages to ensure progress
    - Receiver-side solicitation
    - In-network aggregation
- Adding/removing sequencers
- Congestion-aware routing

Scales beyond a single sequencer



Serialization approach (Eris) is limited by network capacity of the single sequencer

Higher throughput by reducing server coordination

Applying Network Programmability to BFT Protocols

BFT SMR protocols

- Systems today face sophisticated failures
  - Adversaries, malicious participants
- Byzantine fault tolerance protocols
- Permissioned deployment in data centers
  - High-performance applications
  - Low latency requirement

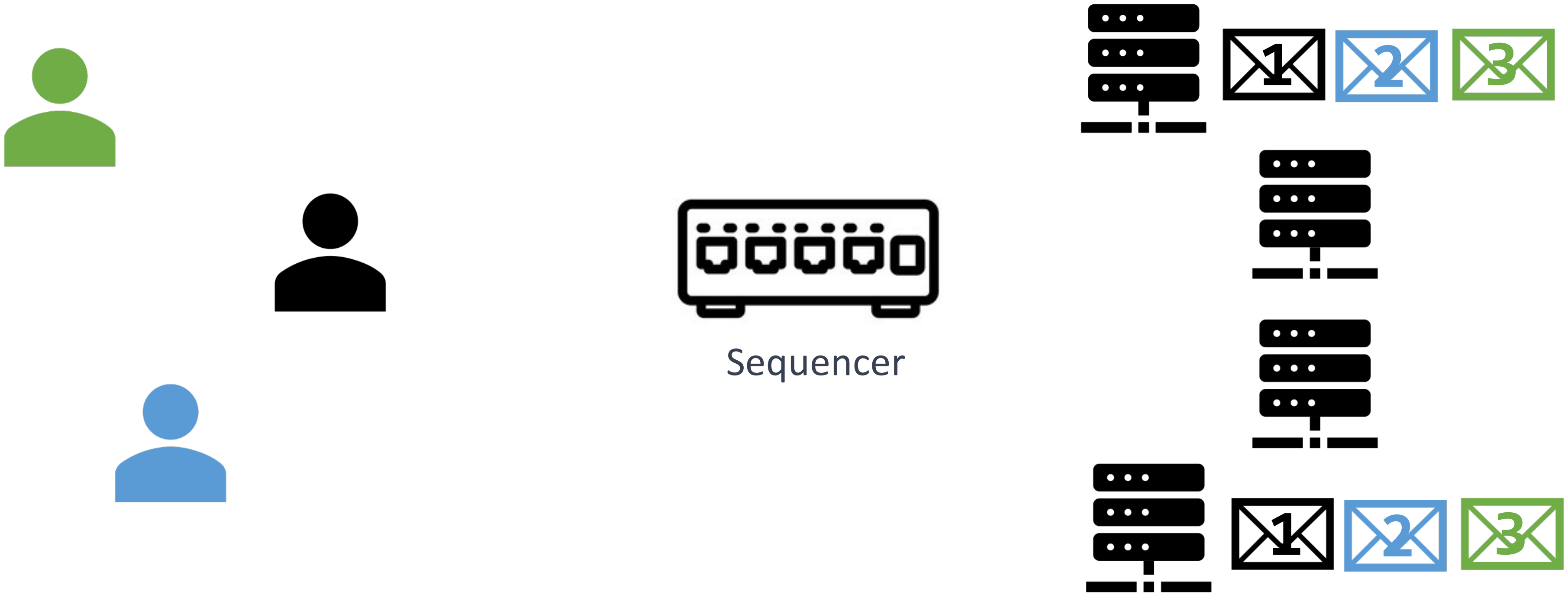
HYPERLEDGER
FABRIC

Amazon Managed
Blockchain

ORACLE
Blockchain

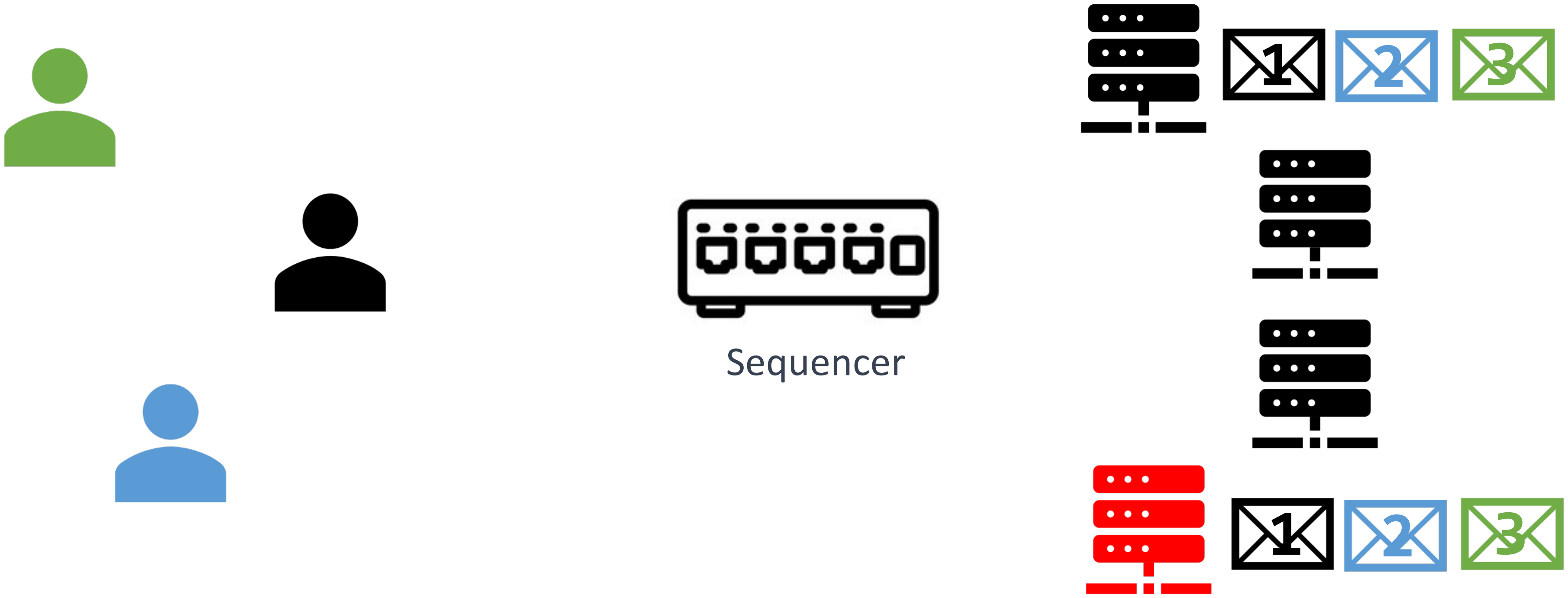
Challenges of Applying In-Network Ordering to BFT

- Adversaries can generate **conflicting** message order



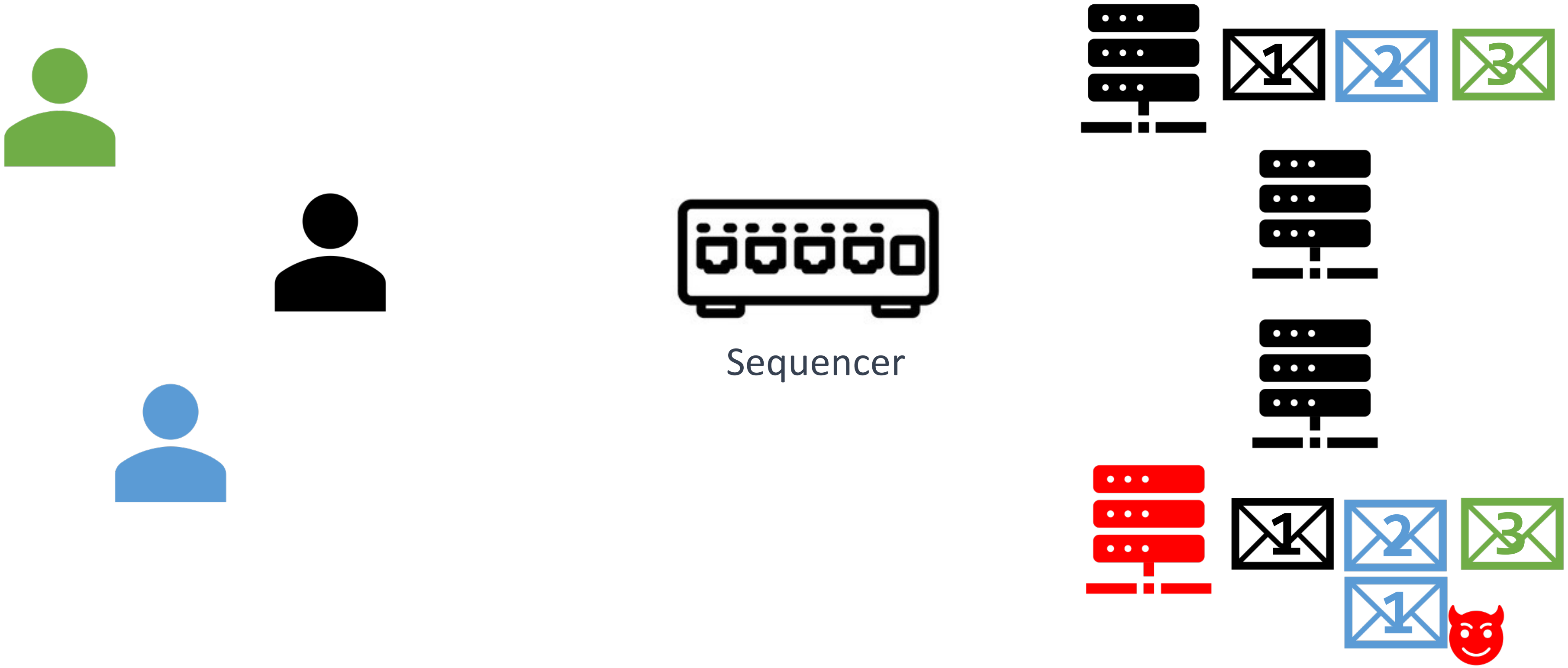
Challenges of Applying In-Network Ordering to BFT

- Adversaries can generate **conflicting** message order



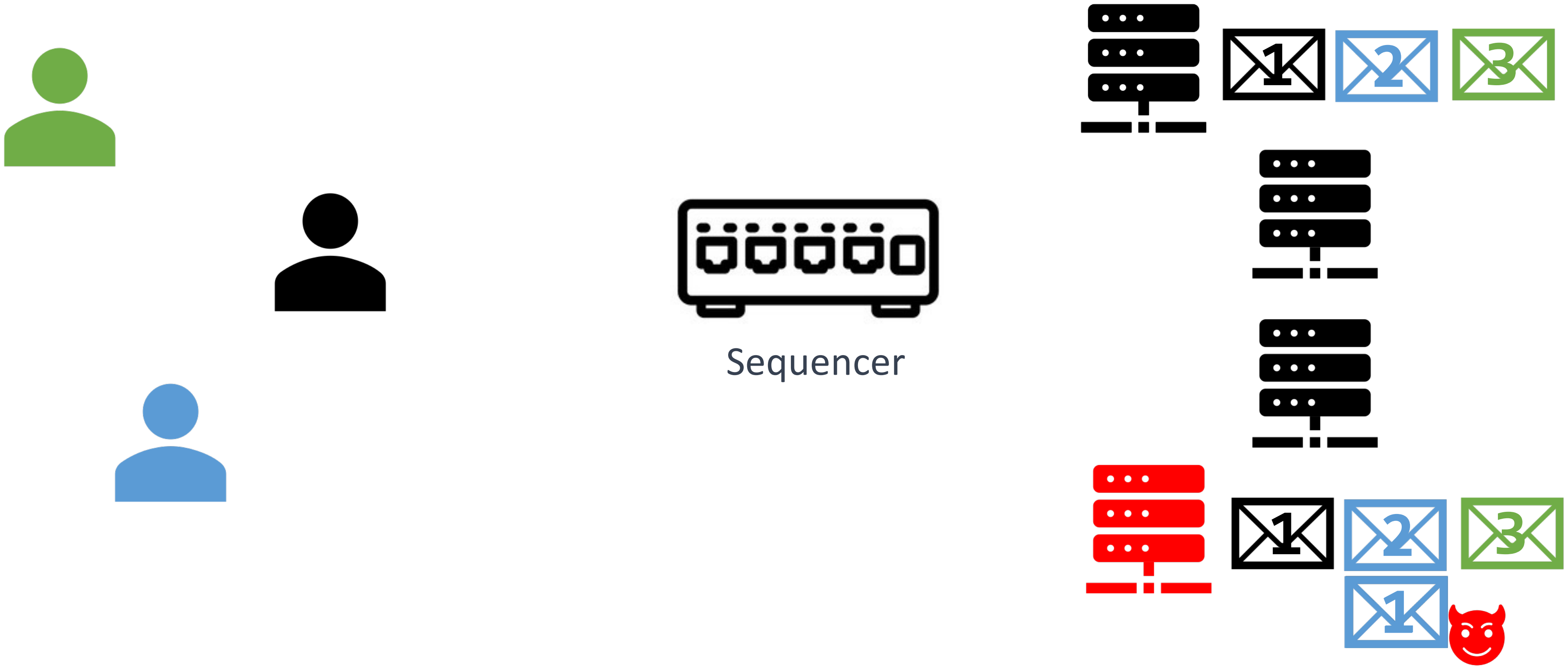
Challenges of Applying In-Network Ordering to BFT

- Adversaries can generate **conflicting** message order



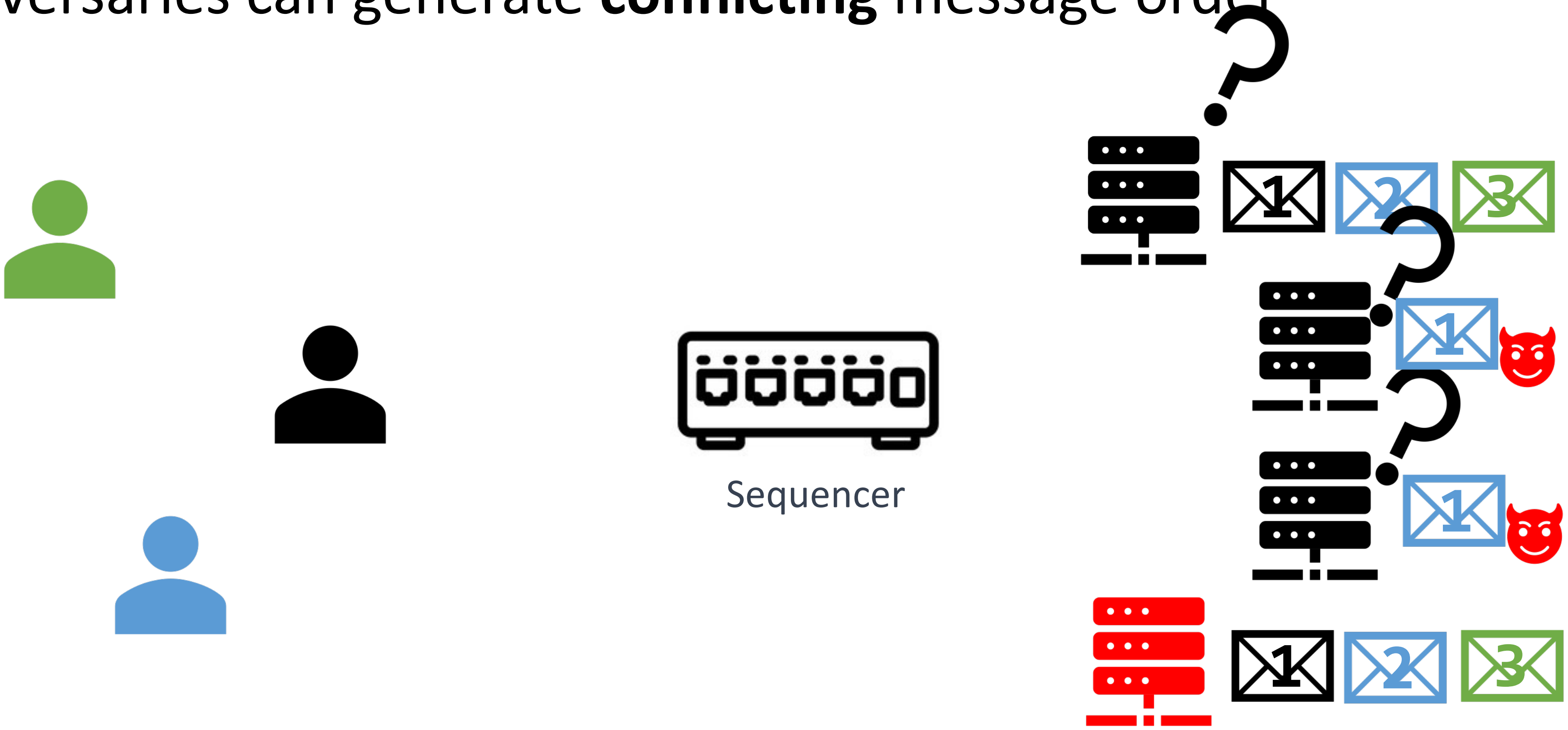
Challenges of Applying In-Network Ordering to BFT

- Adversaries can generate **conflicting** message order



Challenges of Applying In-Network Ordering to BFT

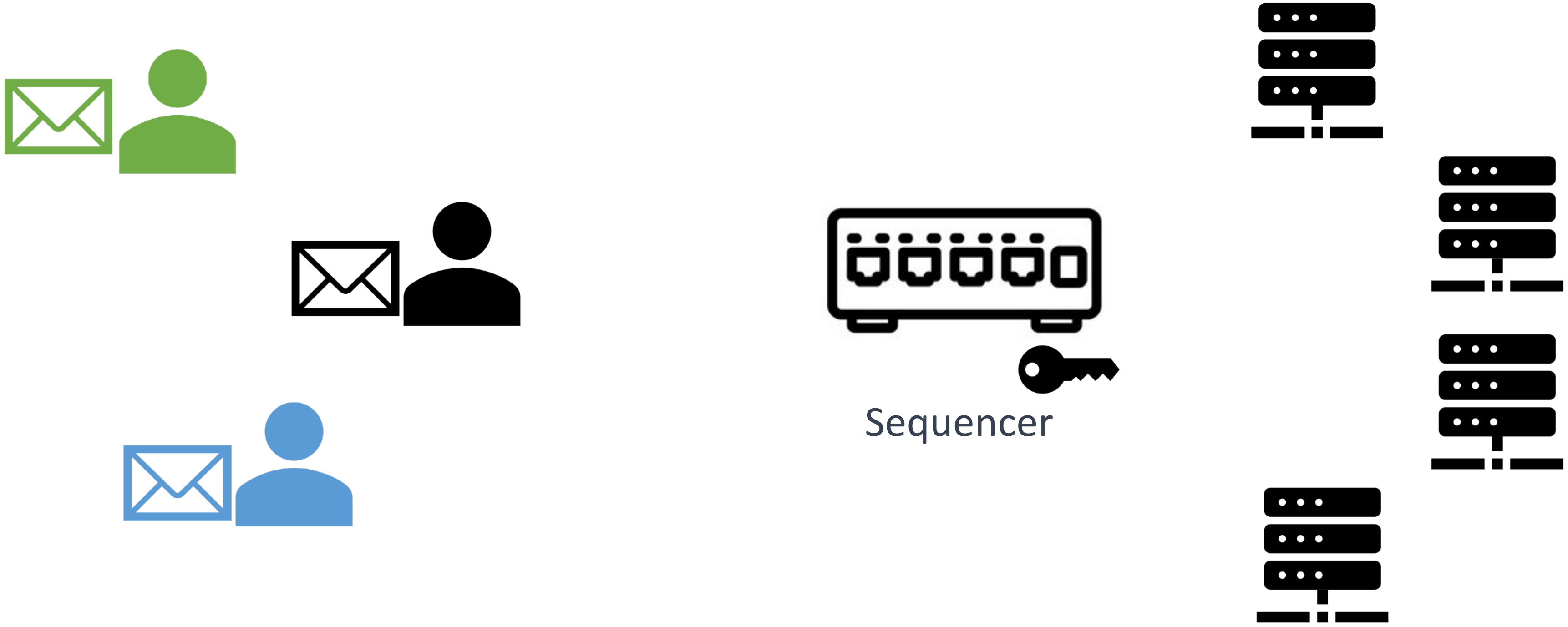
- Adversaries can generate **conflicting** message order



New Approach: Network Ordering with Authentication

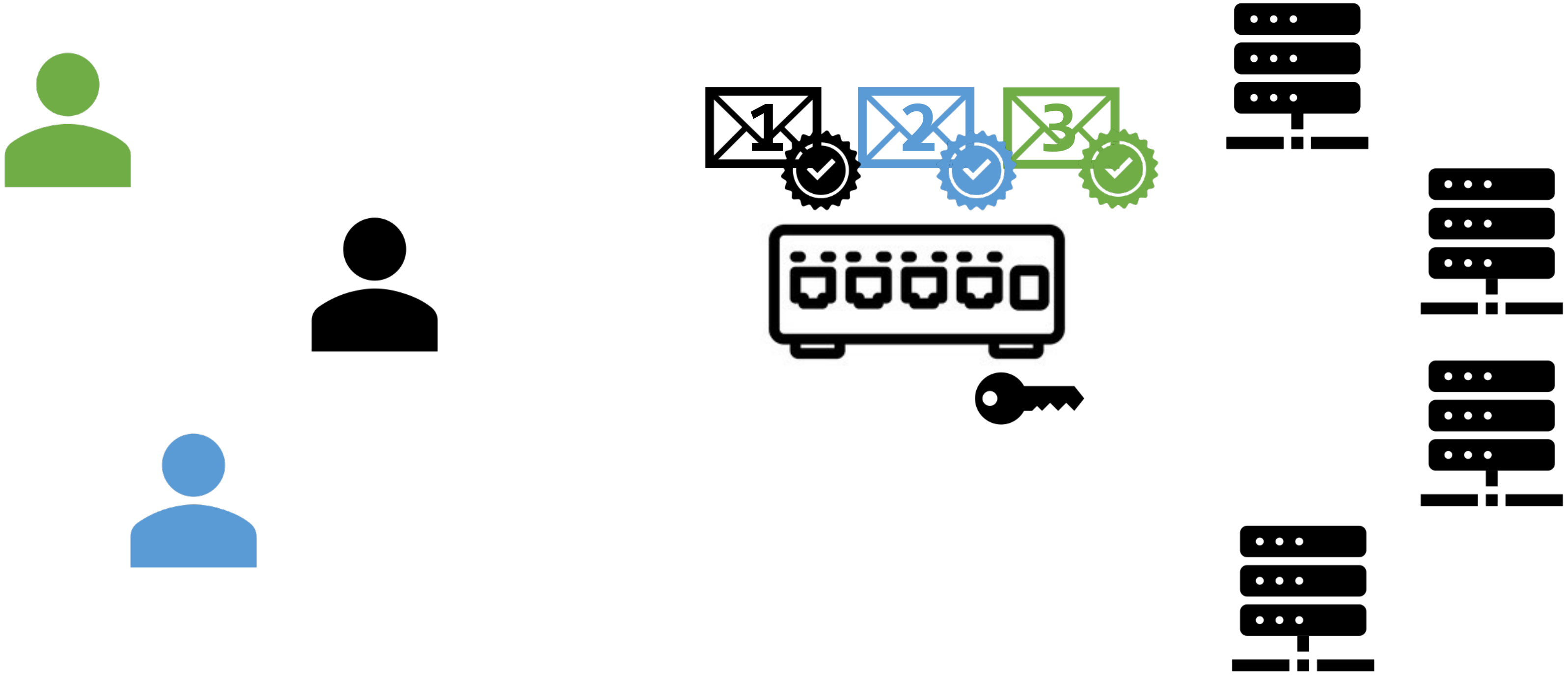
- Can replicas determine the “correct” message order?
  - Yes, if messages from network primitives can be **authenticated**
- Solution: sequencer switch **signs** messages
- Replicas independently **verify** message signatures
  - Messages are indeed generated by sequencer

Authenticated Ordered Multicast Illustration



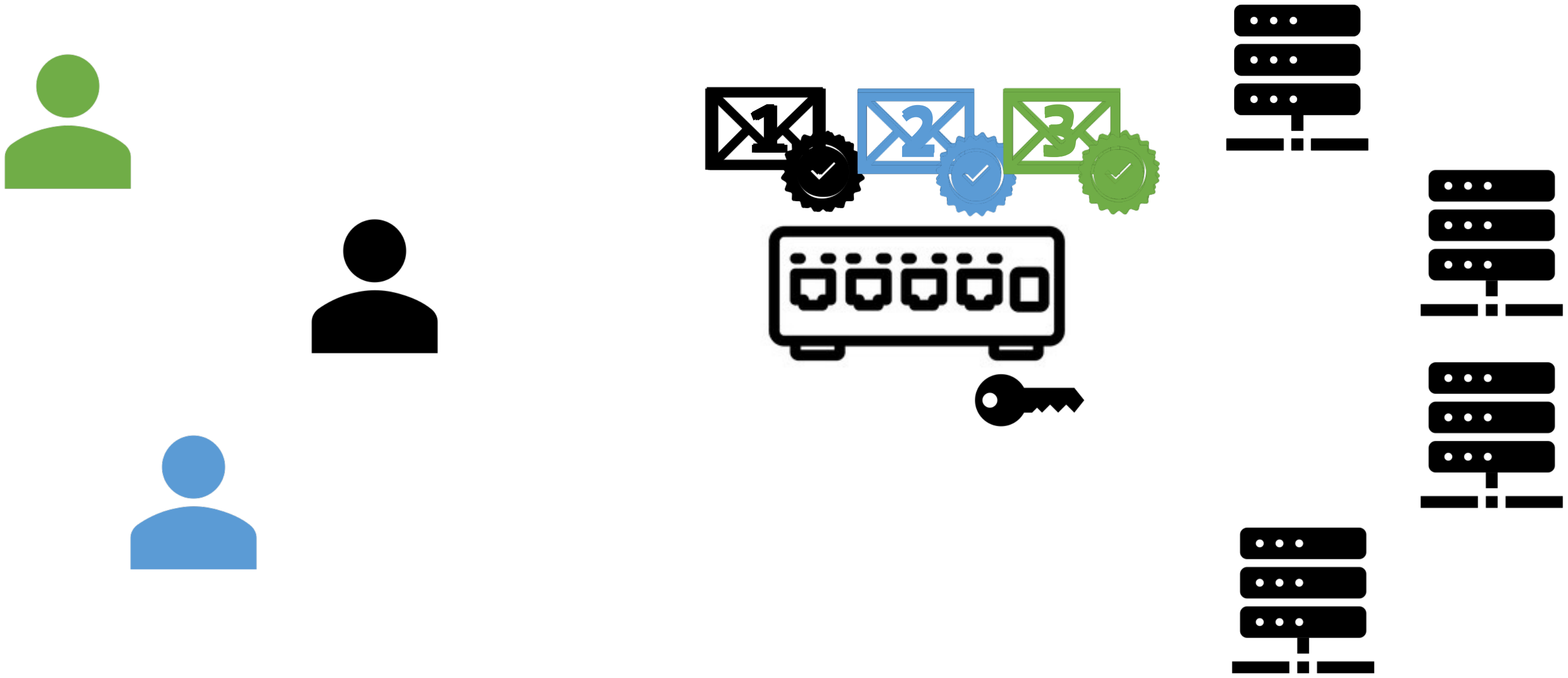
Authenticated Ordered Multicast Illustration

- Ordered: receivers receive AOM messages in the same order



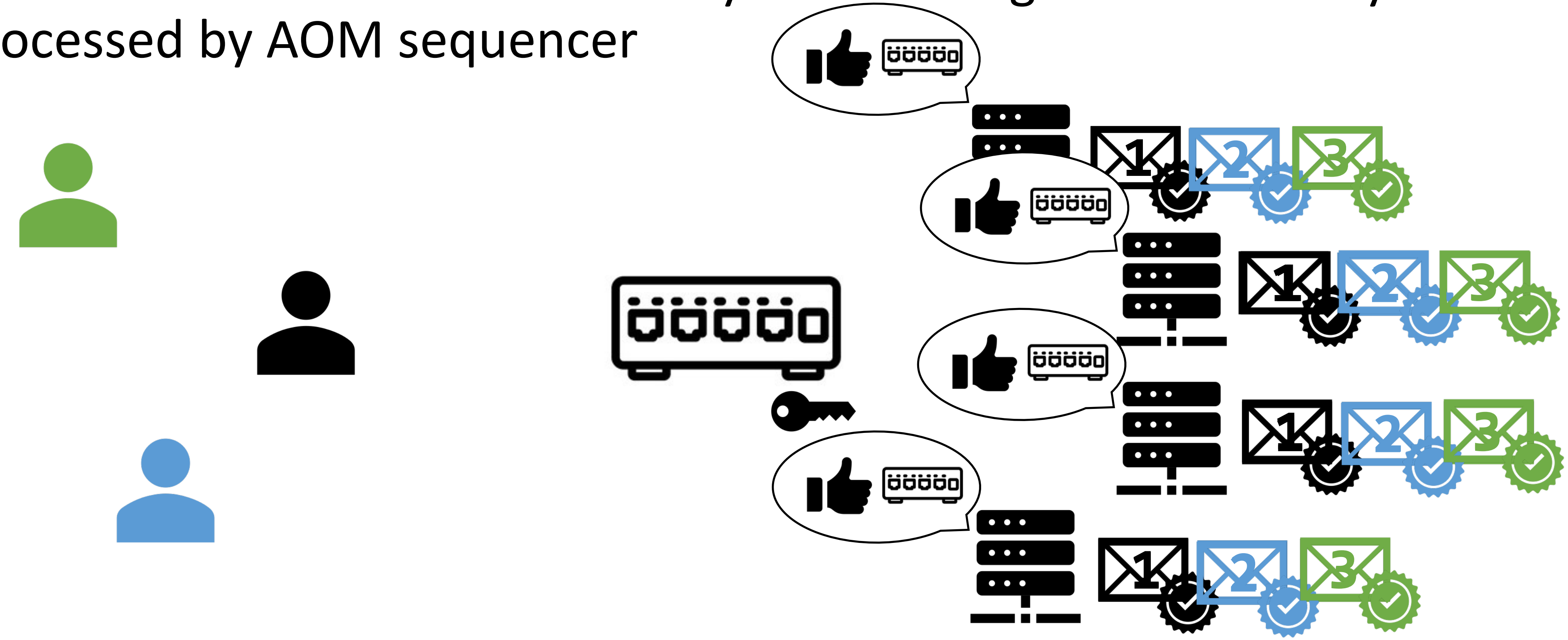
Authenticated Ordered Multicast Illustration

- Ordered: receivers receive AOM messages in the same order



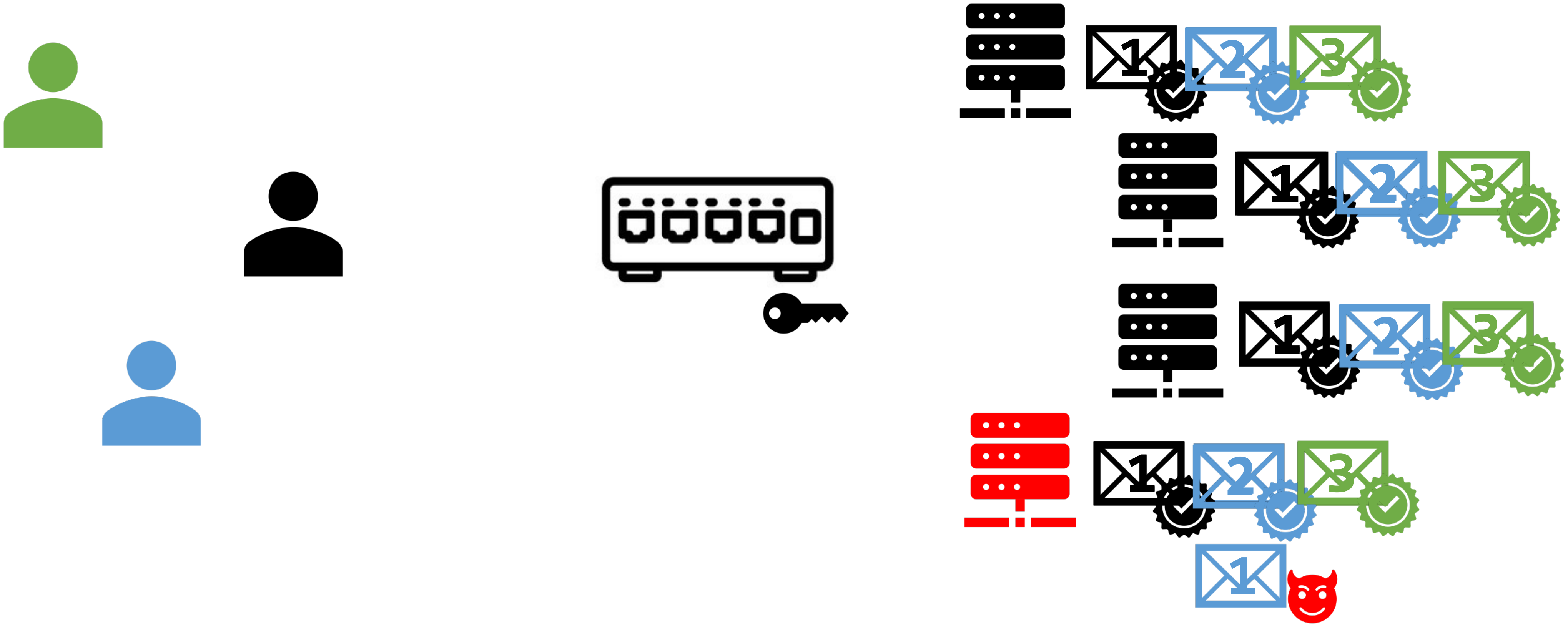
Authenticated Ordered Multicast Illustration

- Authentication: receiver can verify that messages are correctly processed by AOM sequencer



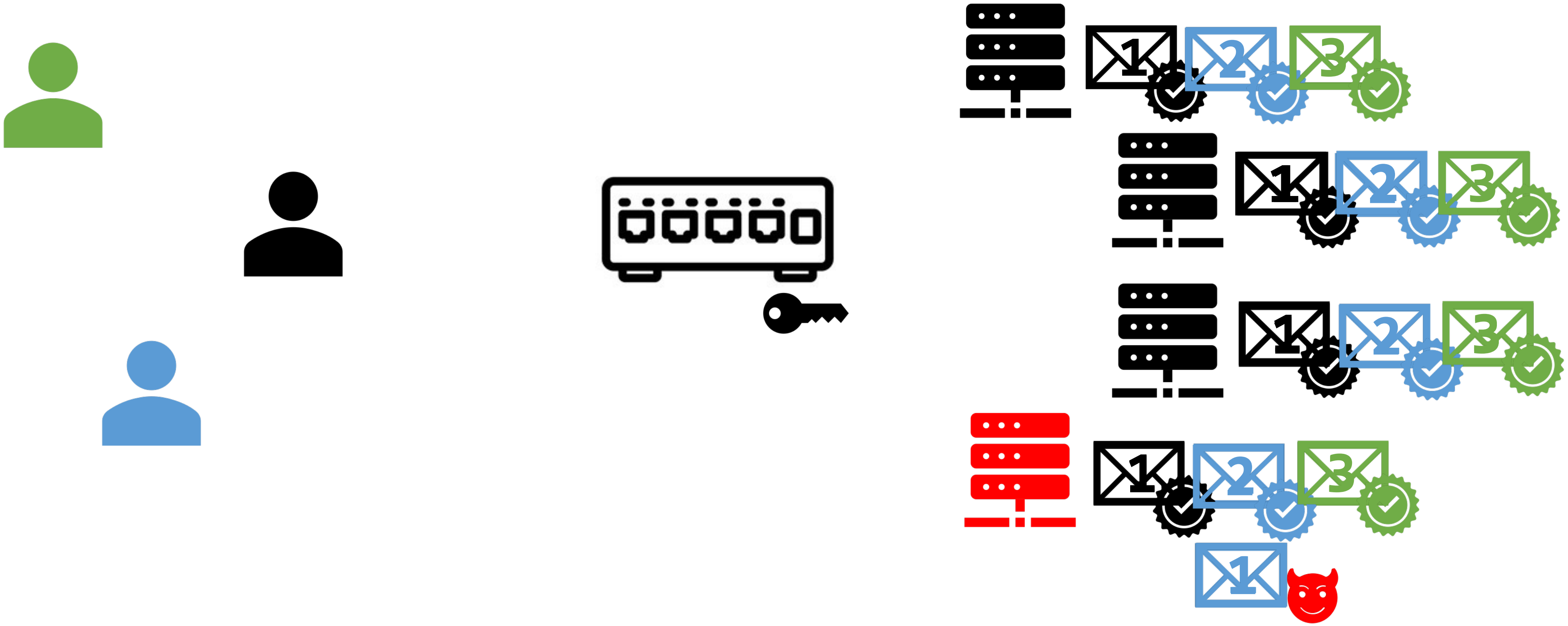
Authenticated Ordered Multicast Illustration

- Authentication: receiver can verify that messages are correctly processed by AOM sequencer



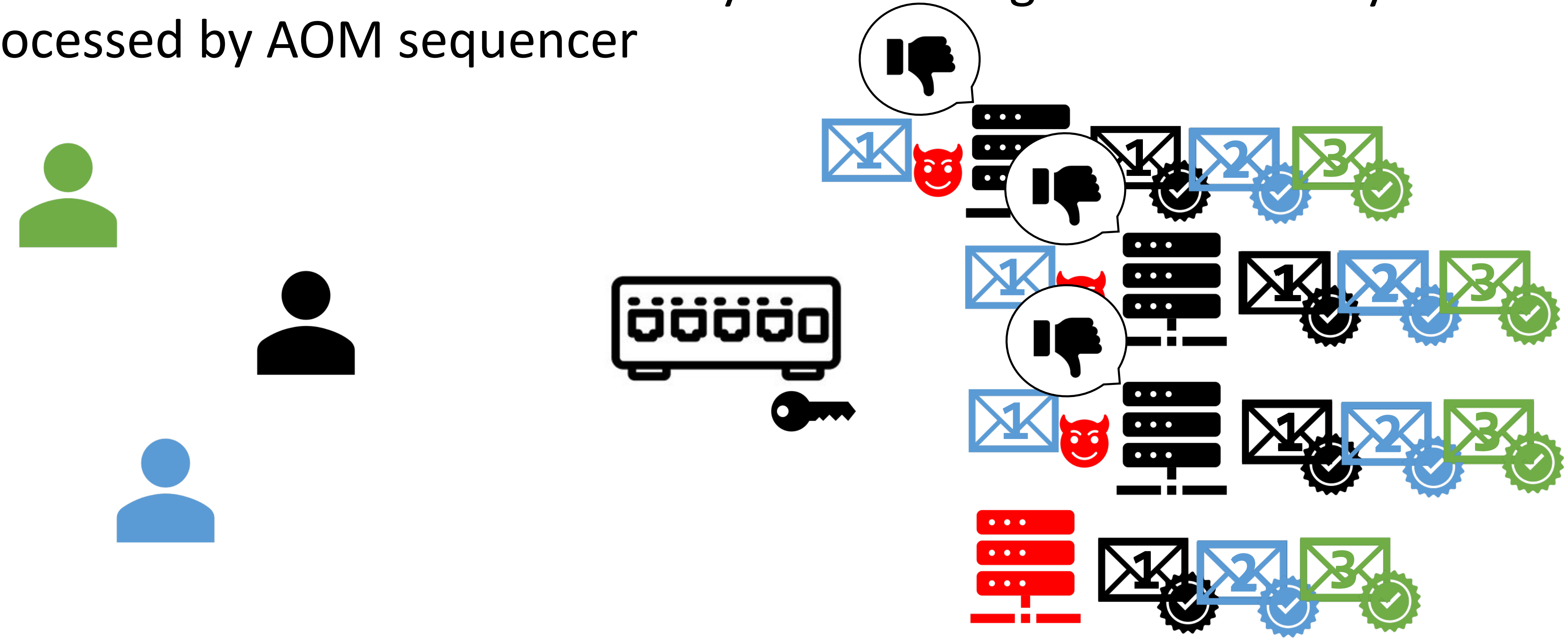
Authenticated Ordered Multicast Illustration

- Authentication: receiver can verify that messages are correctly processed by AOM sequencer



Authenticated Ordered Multicast Illustration

- Authentication: receiver can verify that messages are correctly processed by AOM sequencer



Implementing In-Network Authentication Is Challenging

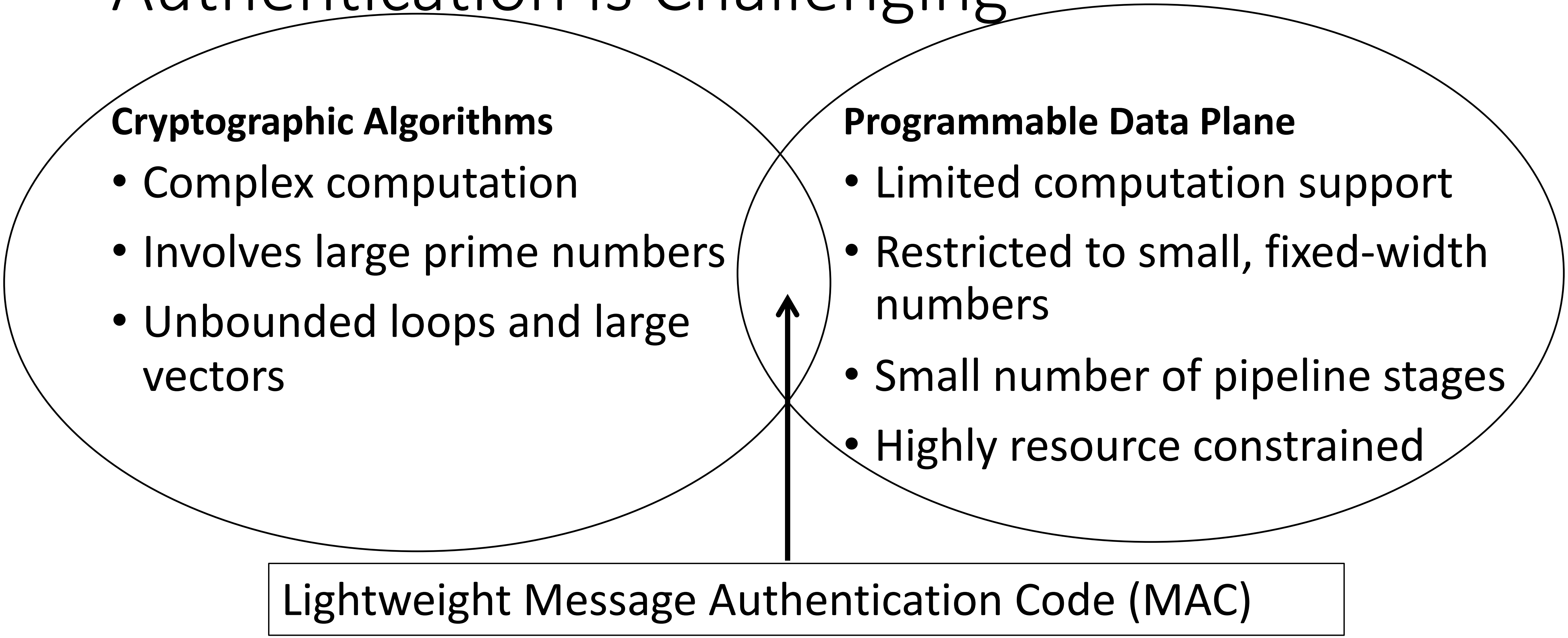
Cryptographic Algorithms

- Complex computation
- Involves large prime numbers
- Unbounded loops and large vectors

Programmable Data Plane

- Limited computation support
- Restricted to small, fixed-width numbers
- Small number of pipeline stages
- Highly resource constrained

Implementing In-Network Authentication Is Challenging



Cryptographic Algorithms

- Complex computation
- Involves large prime numbers
- Unbounded loops and large vectors

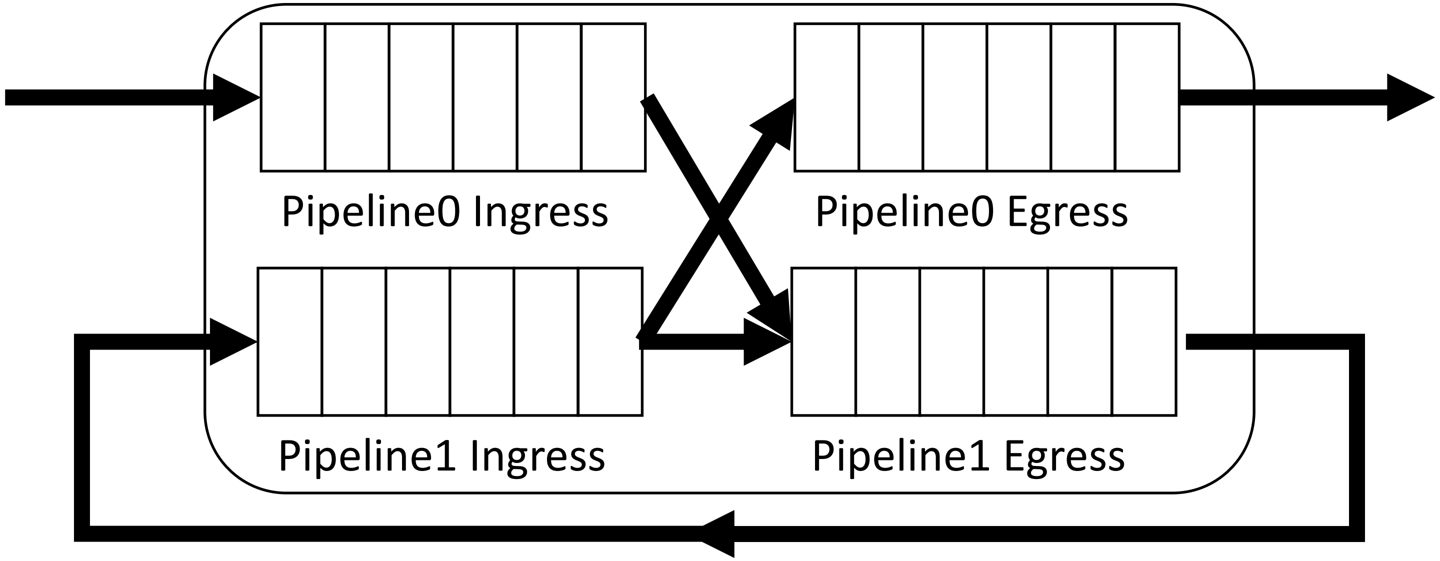
Programmable Data Plane

- Limited computation support
- Restricted to small, fixed-width numbers
- Small number of pipeline stages
- Highly resource constrained

Lightweight Message Authentication Code (MAC)

MAC on a Switch

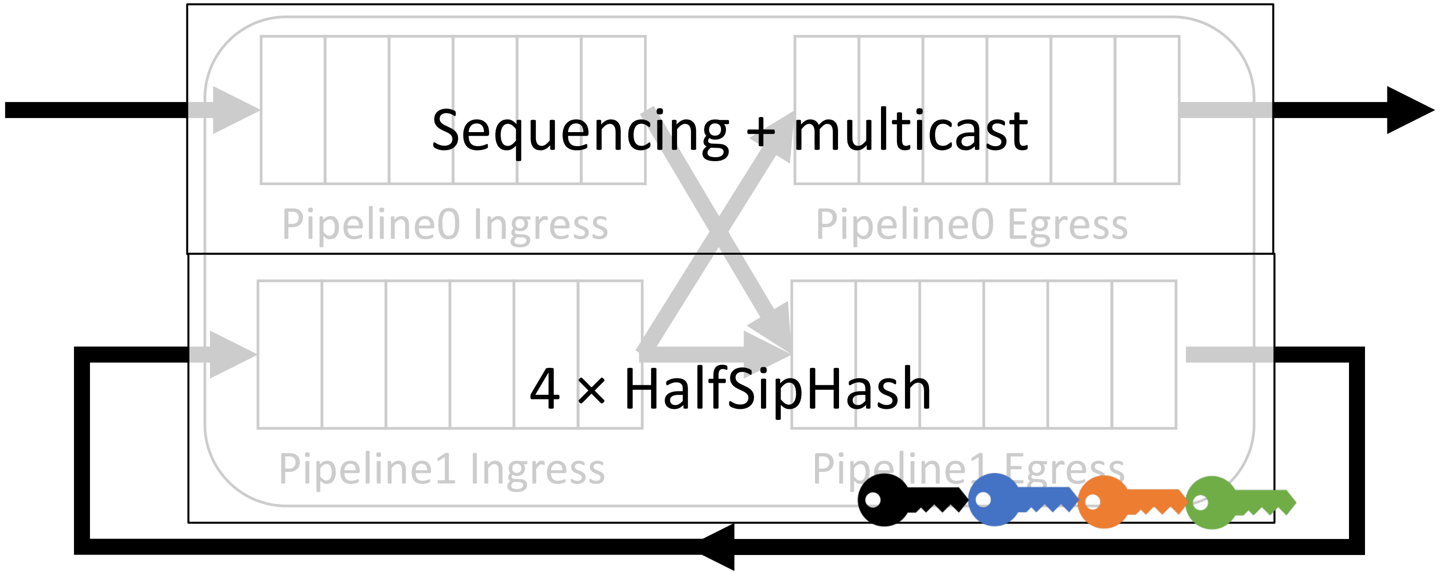
- Dedicated pipeline for computing MAC vector
- Unrolled HalfSipHash implementation
- 4 parallel MAC generation instances



Yoo et al., SPIN ’21

MAC on a Switch

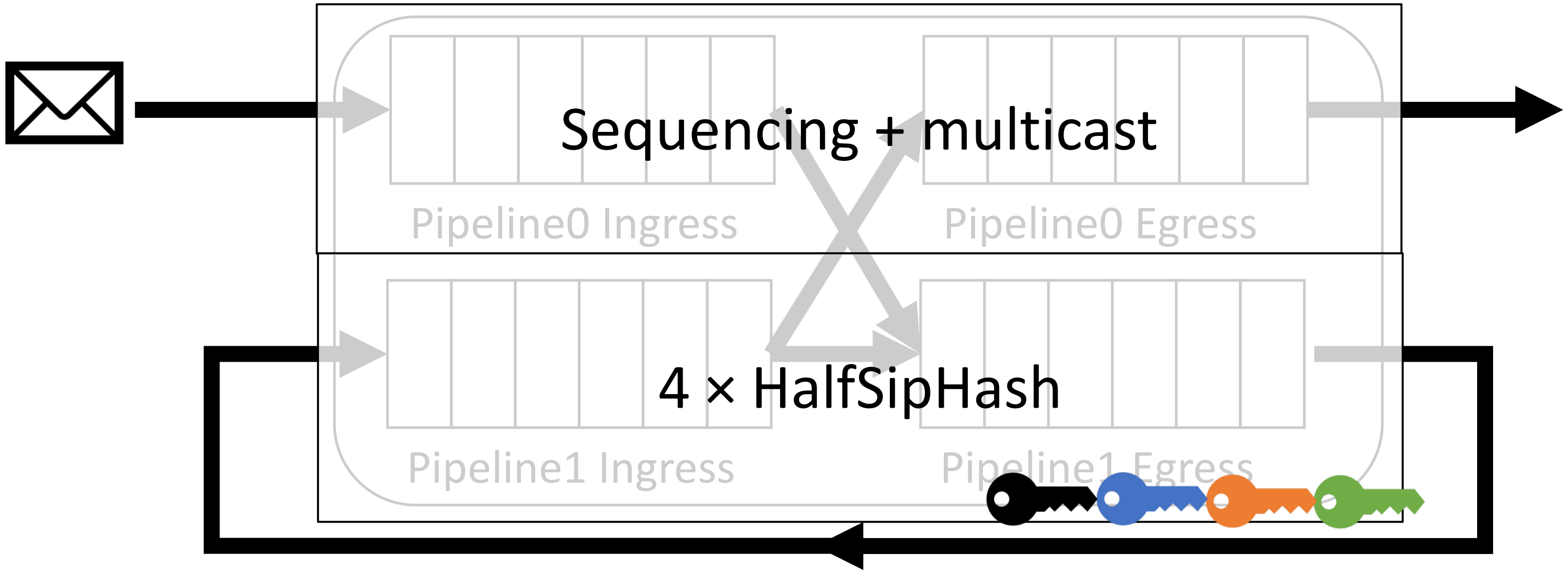
- Dedicated pipeline for computing MAC vector
- Unrolled HalfSipHash implementation
- 4 parallel MAC generation instances


Sequencing + multicast

4 × HalfSipHash

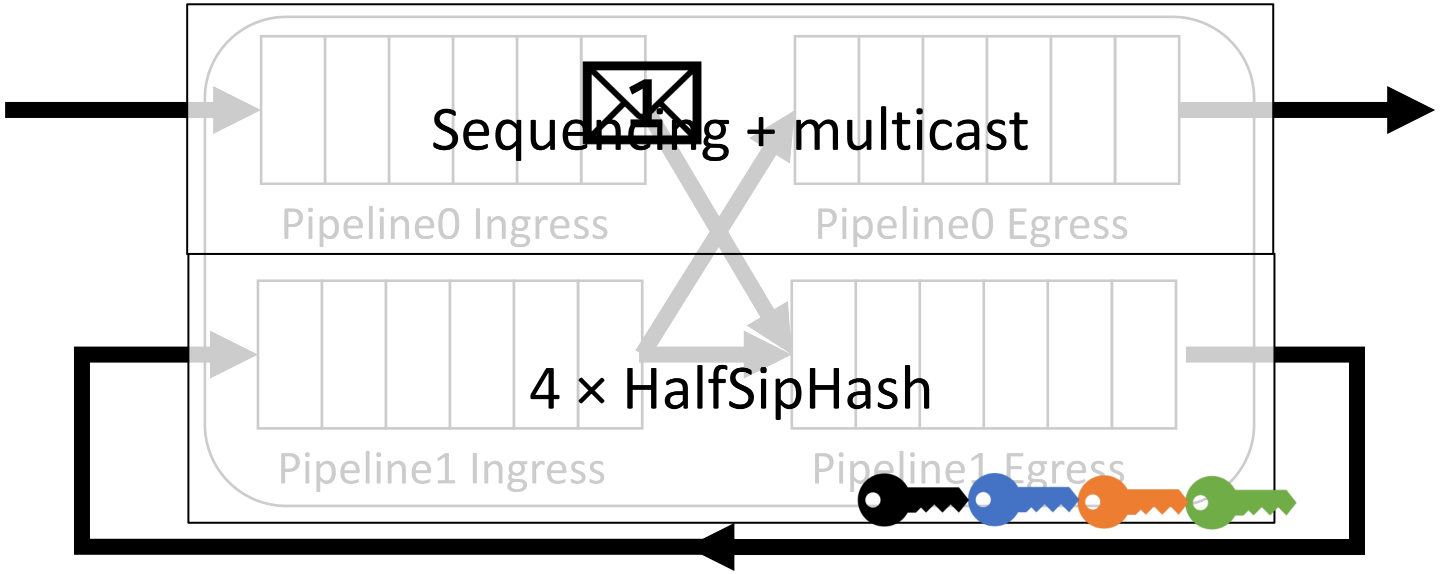
MAC on a Switch

- Dedicated pipeline for computing MAC vector
- Unrolled HalfSipHash implementation
- 4 parallel MAC generation instances



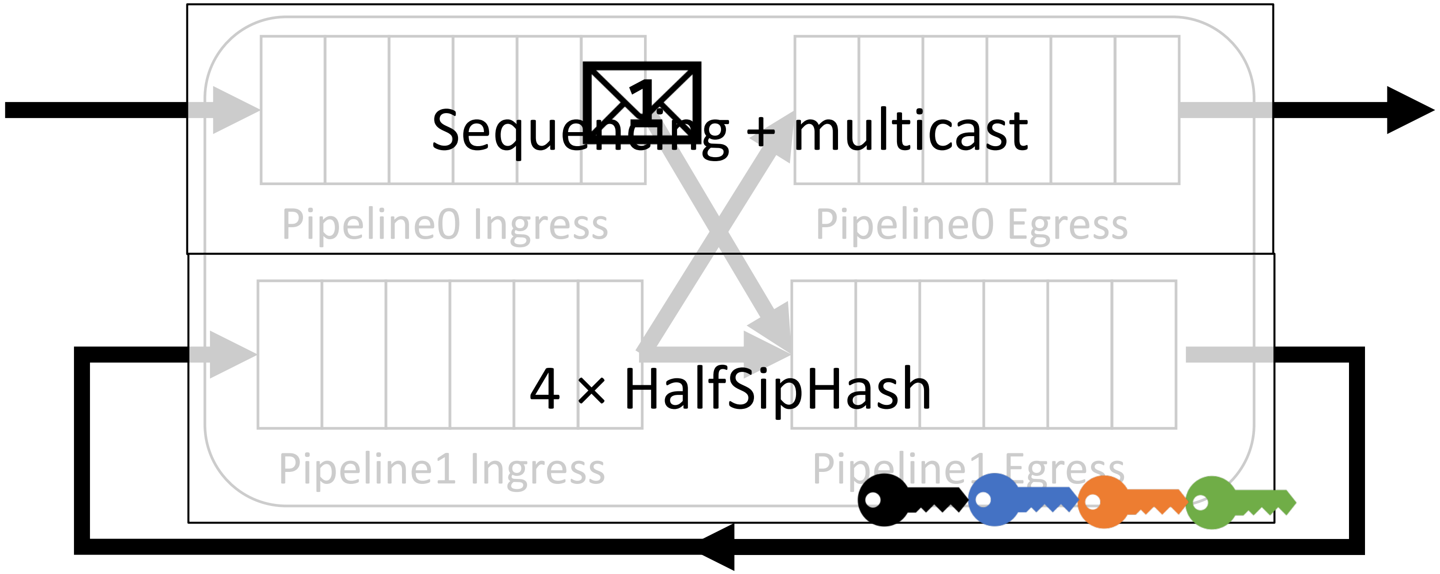
MAC on a Switch

- Dedicated pipeline for computing MAC vector
- Unrolled HalfSipHash implementation
- 4 parallel MAC generation instances



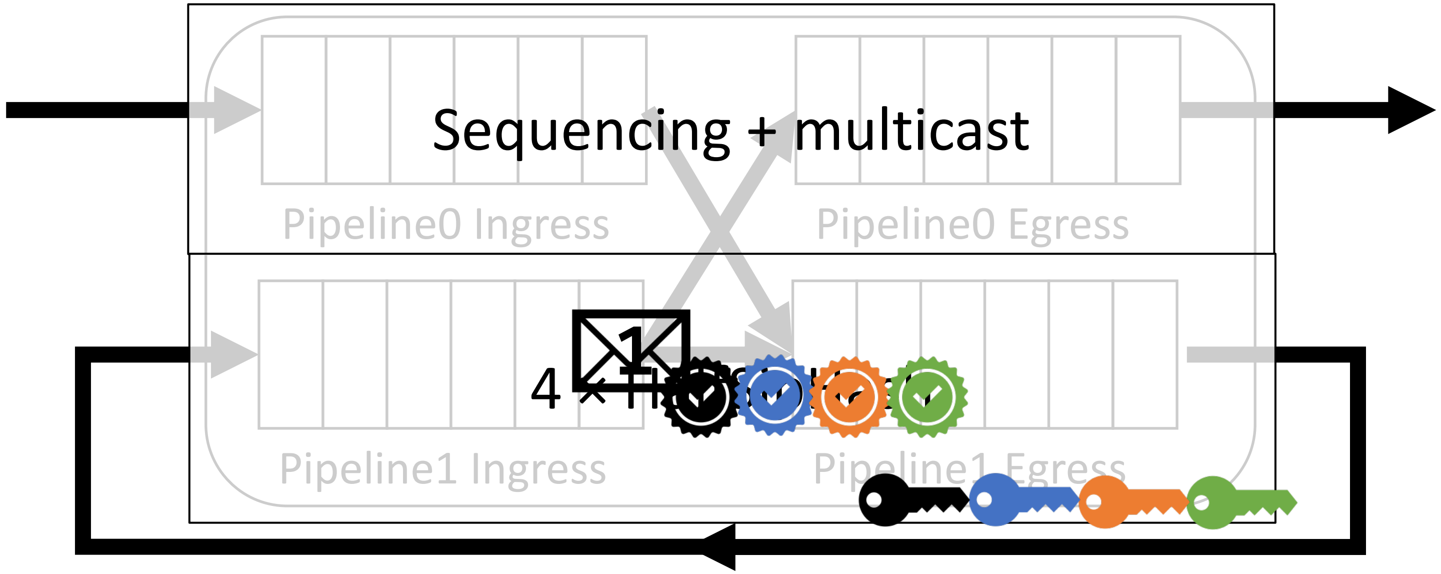
MAC on a Switch

- Dedicated pipeline for computing MAC vector
- Unrolled HalfSipHash implementation
- 4 parallel MAC generation instances



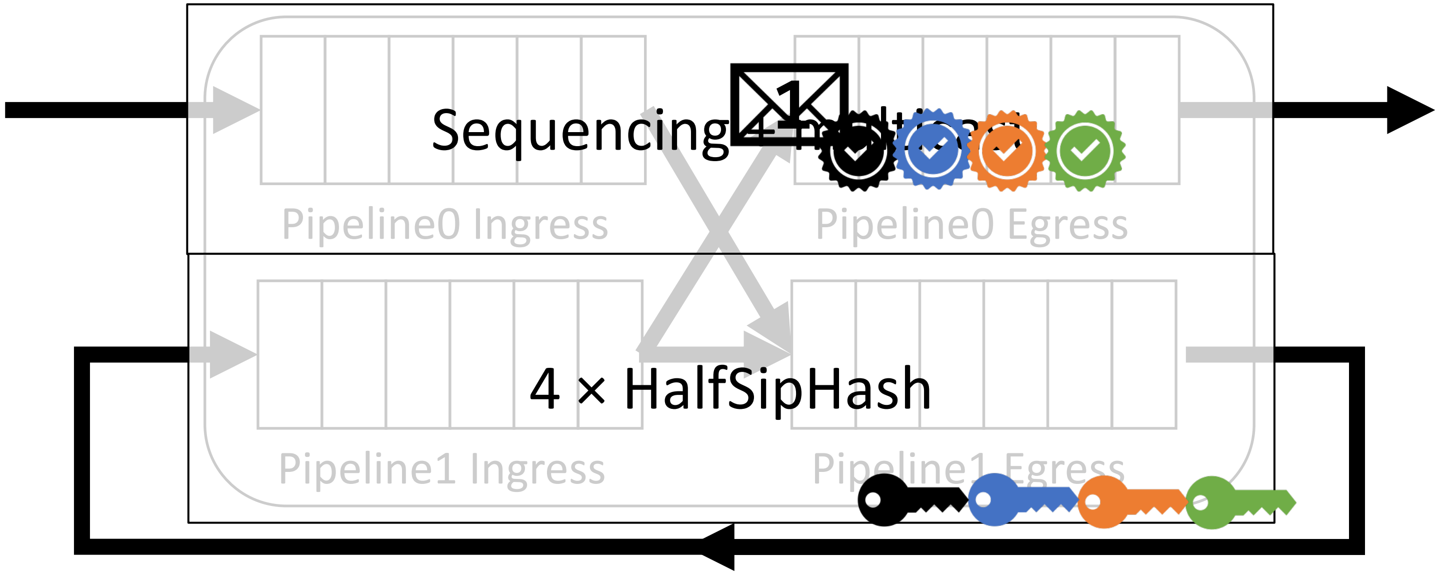
MAC on a Switch

- Dedicated pipeline for computing MAC vector
- Unrolled HalfSipHash implementation
- 4 parallel MAC generation instances



MAC on a Switch

- Dedicated pipeline for computing MAC vector
- Unrolled HalfSipHash implementation
- 4 parallel MAC generation instances

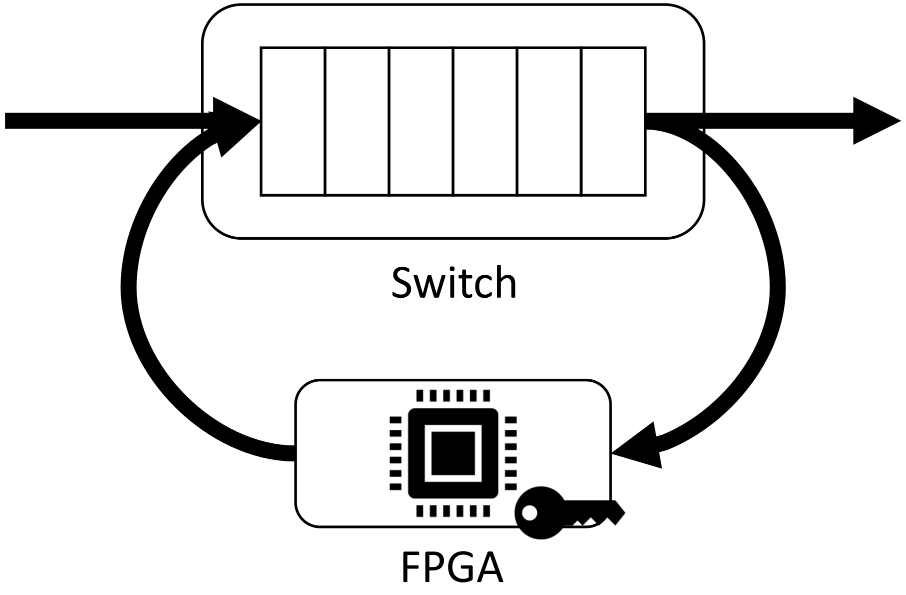


Another Authentication Implementation: Public-Key Cryptography

- Problem with MAC vector: poor scalability
  - One MAC per receiver
- Public-key cryptography offers “*infinite*” scalability
  - Too complex to implement directly in the switch data plane
- New switch architecture that integrates a **cryptographic coprocessor**
- FPGA-based implementation in our prototype

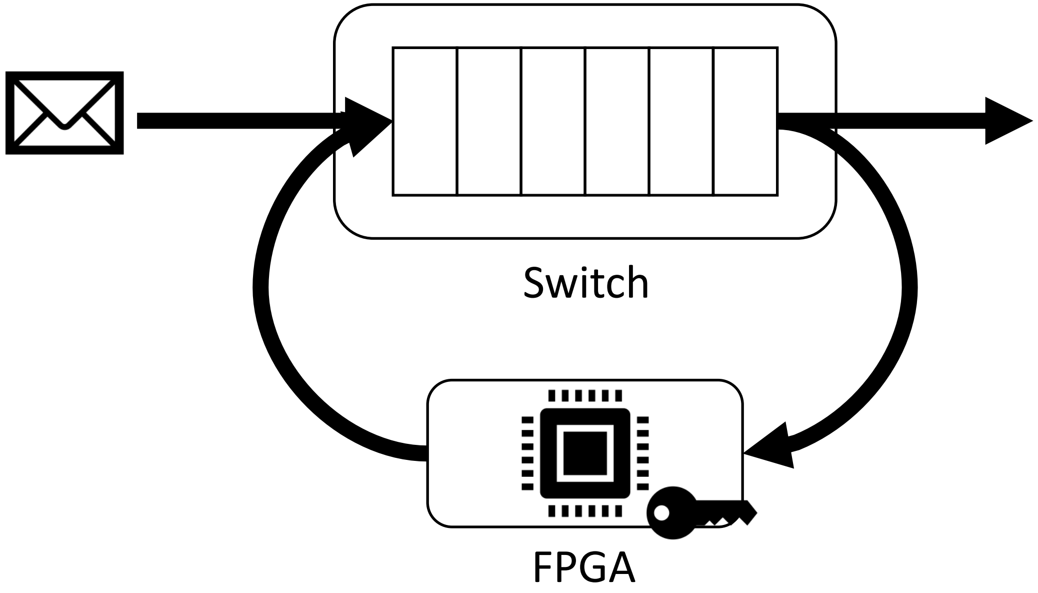
FPGA-based Public-Key Cryptography Prototype

- SECP256K1 signature implemented using FPGA



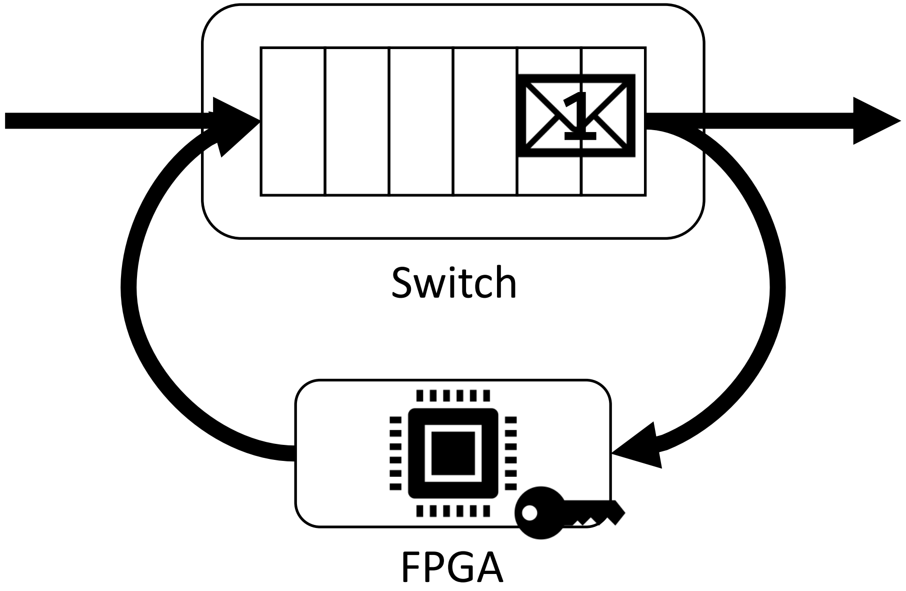
FPGA-based Public-Key Cryptography Prototype

- SECP256K1 signature implemented using FPGA



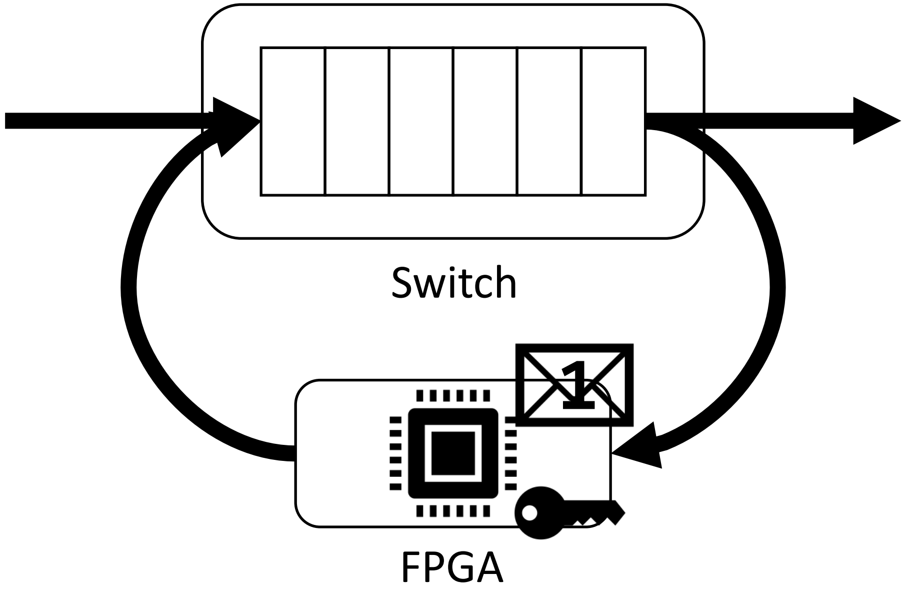
FPGA-based Public-Key Cryptography Prototype

- SECP256K1 signature implemented using FPGA



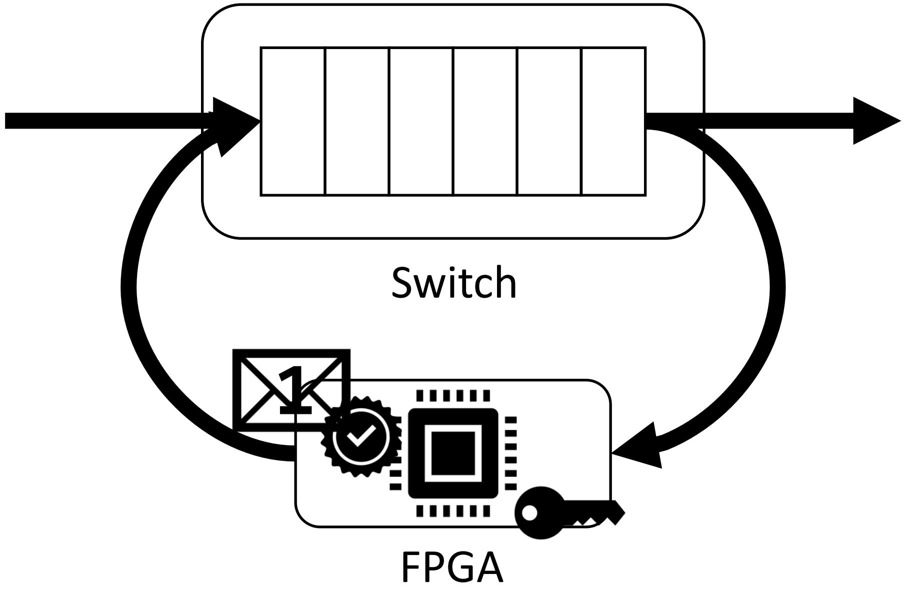
FPGA-based Public-Key Cryptography Prototype

- SECP256K1 signature implemented using FPGA



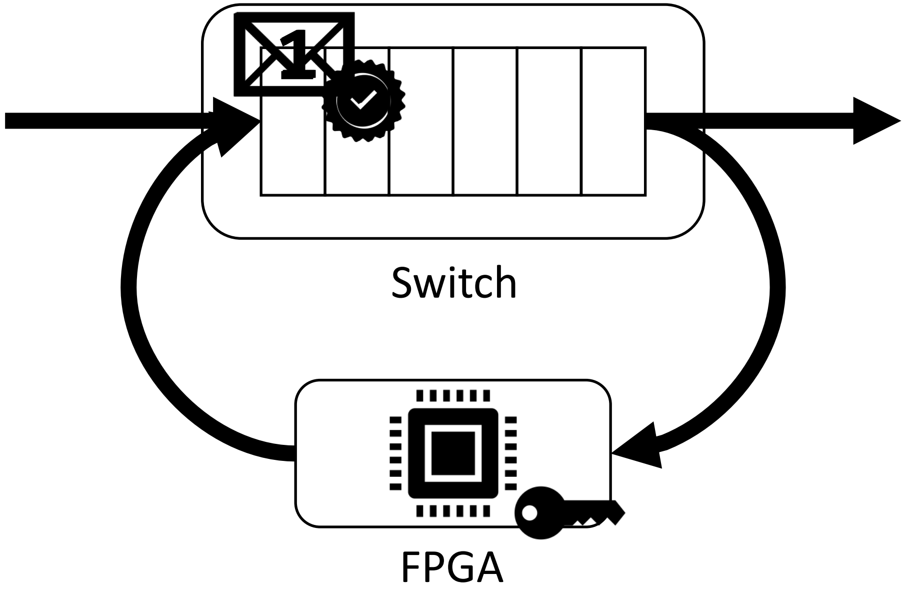
FPGA-based Public-Key Cryptography Prototype

- SECP256K1 signature implemented using FPGA



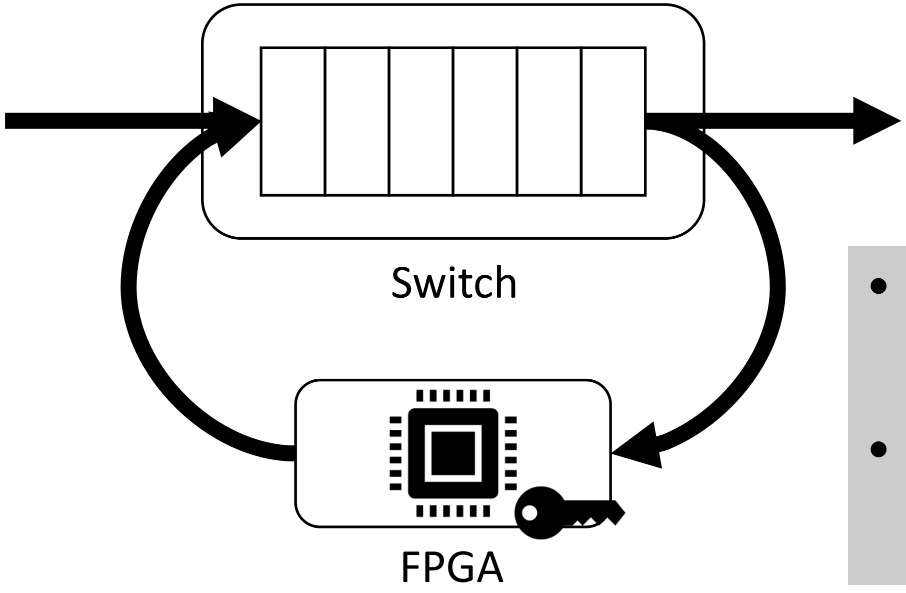
FPGA-based Public-Key Cryptography Prototype

- SECP256K1 signature implemented using FPGA



FPGA-based Public-Key Cryptography Prototype

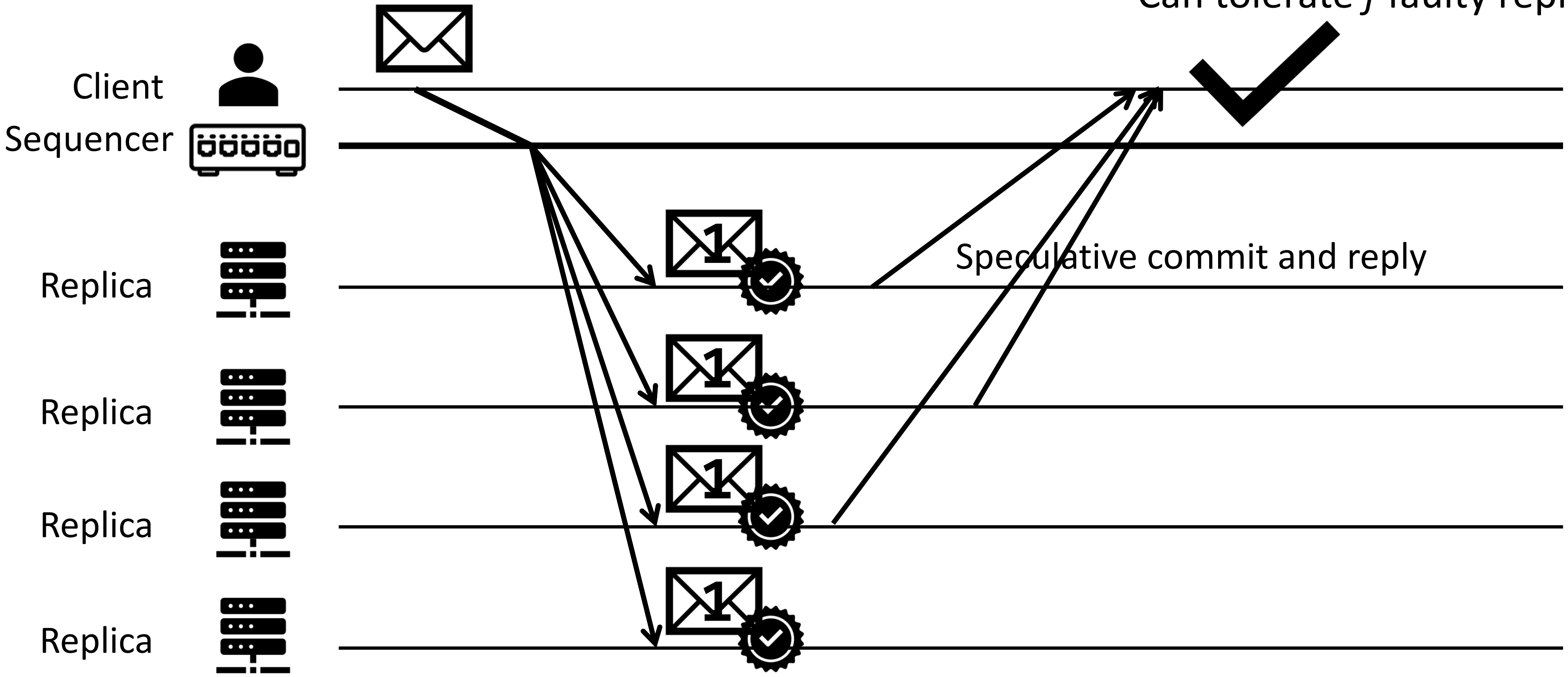
- SECP256K1 signature implemented using FPGA



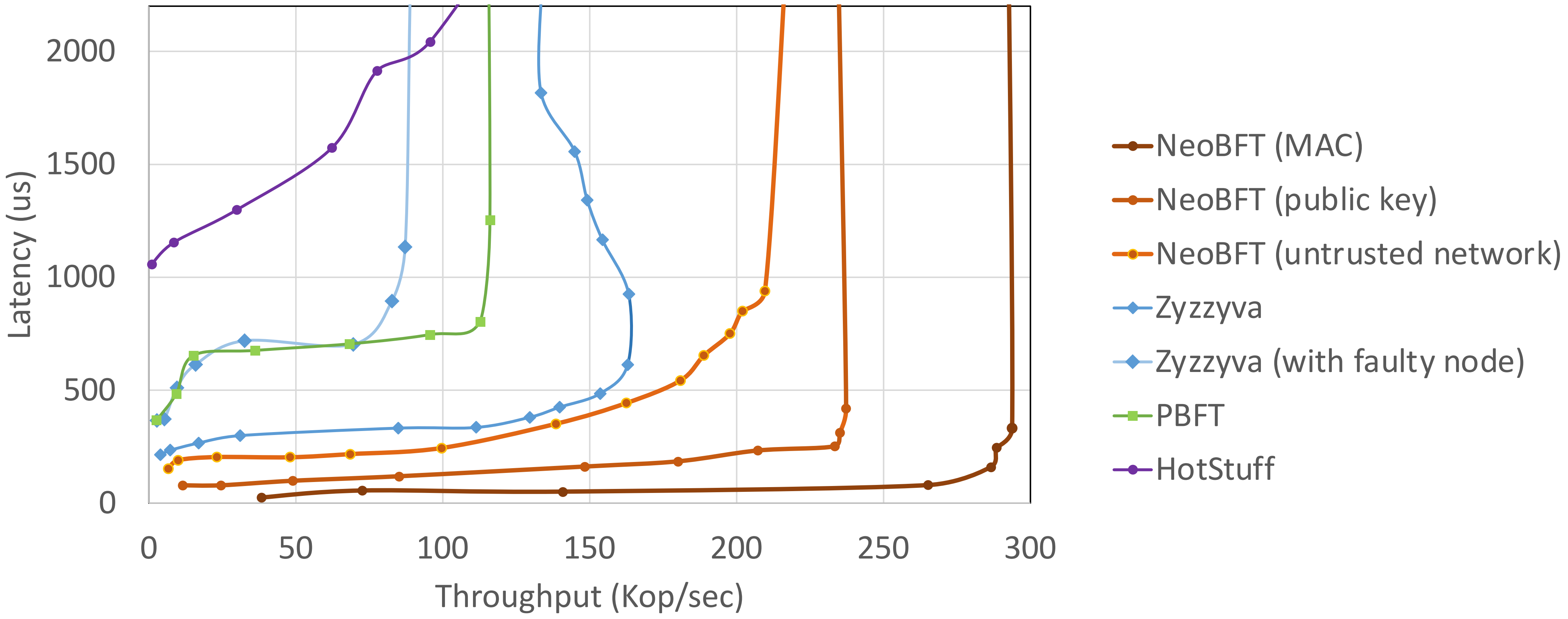
- Message Authentication Code
  - Up to 76.24Mpps
- Public-Key Cryptography
  - 1.11Mpps

NeoBFT: Normal Operation

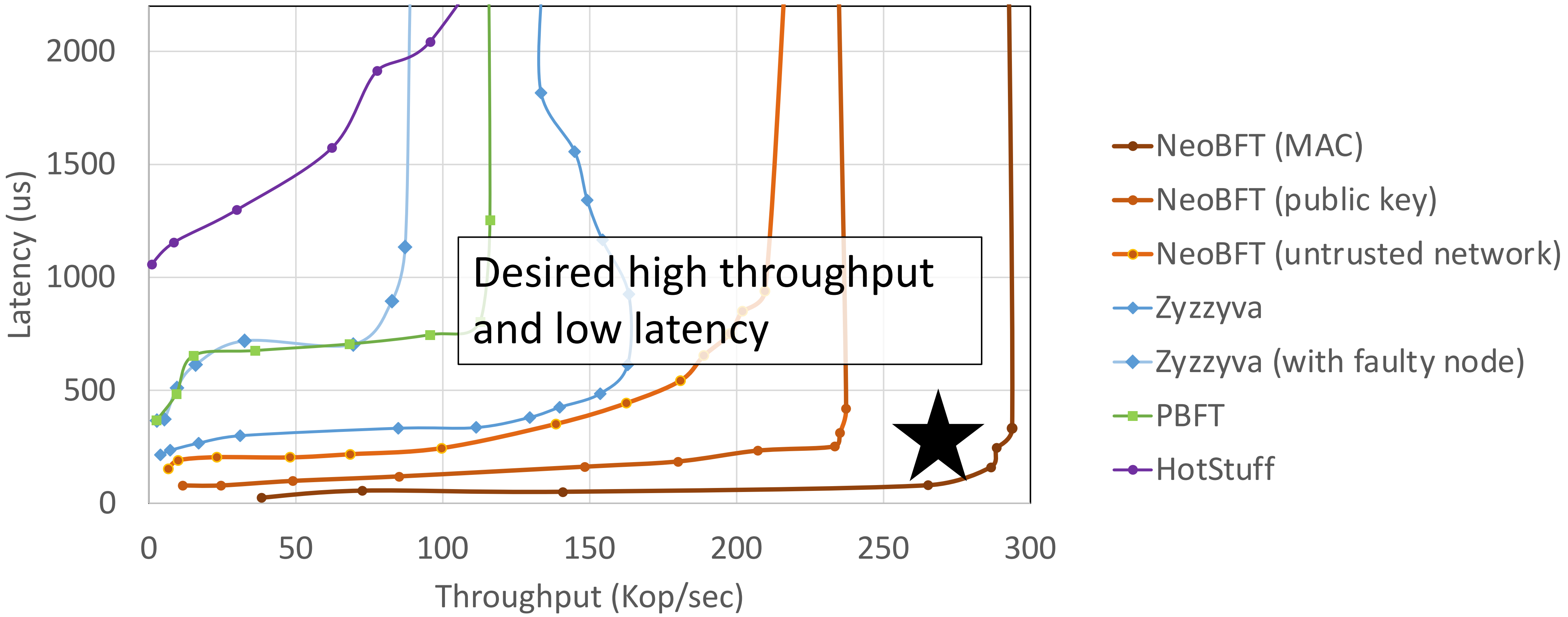
- Client commit in 1 RTT
- No coordination among replicas
- Can tolerate *f* faulty replicas



NeoBFT Achieves both **Lower Latency** and **Higher Throughput**

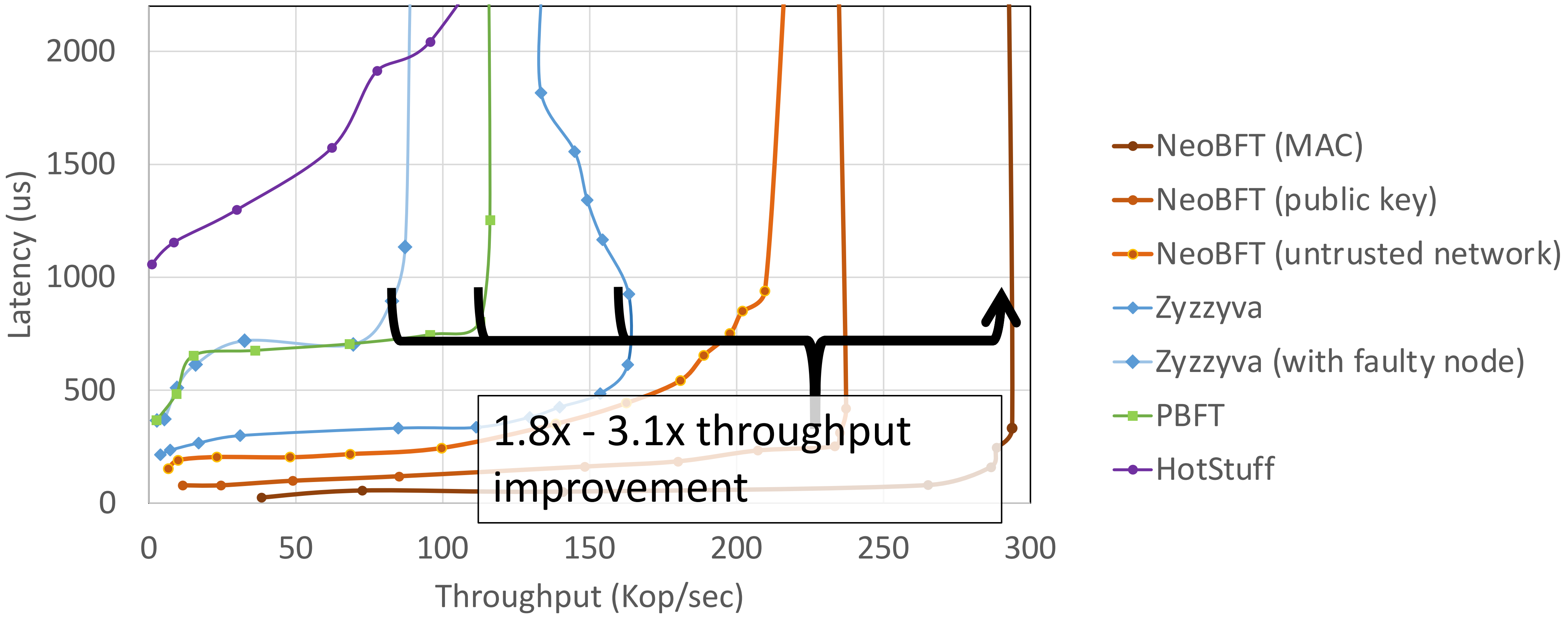


NeoBFT Achieves both **Lower Latency** and **Higher Throughput**



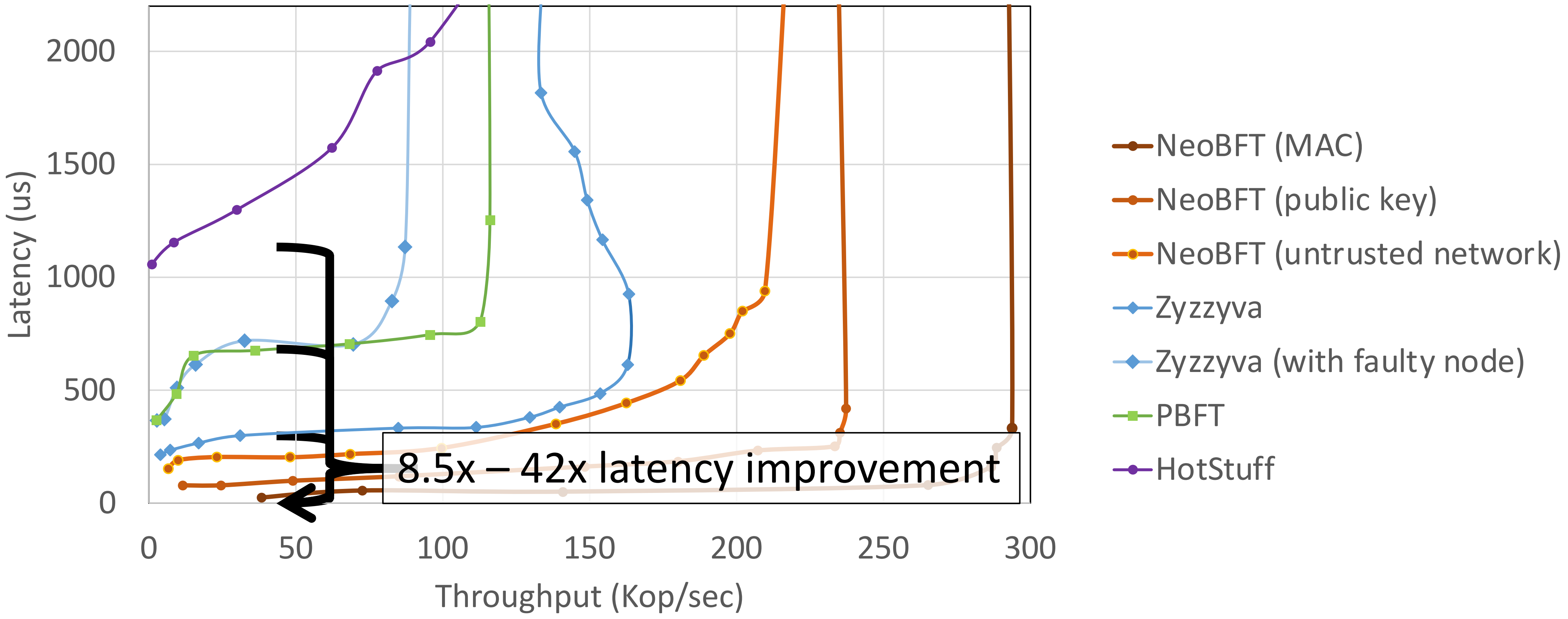
Desired high throughput and low latency

NeoBFT Achieves both **Lower Latency** and **Higher Throughput**



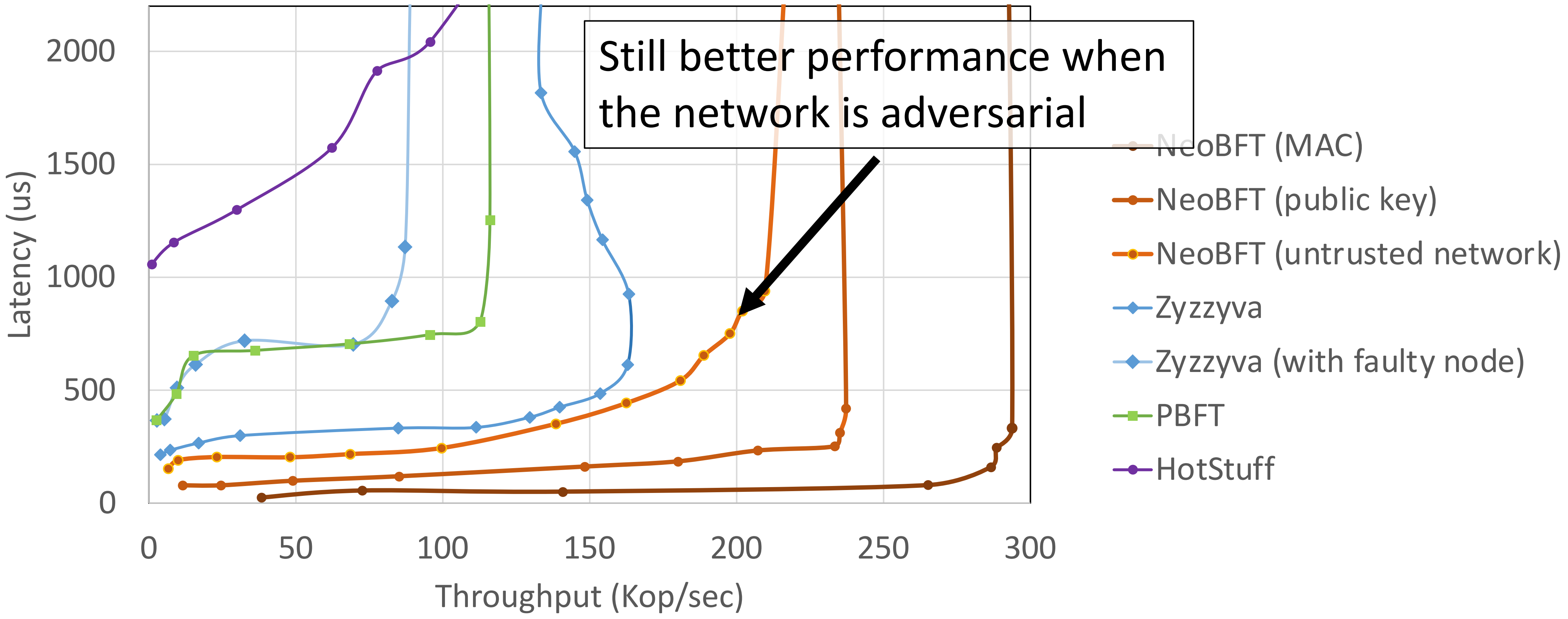
1.8x - 3.1x throughput improvement

NeoBFT Achieves both **Lower Latency** and **Higher Throughput**



8.5x – 42x latency improvement

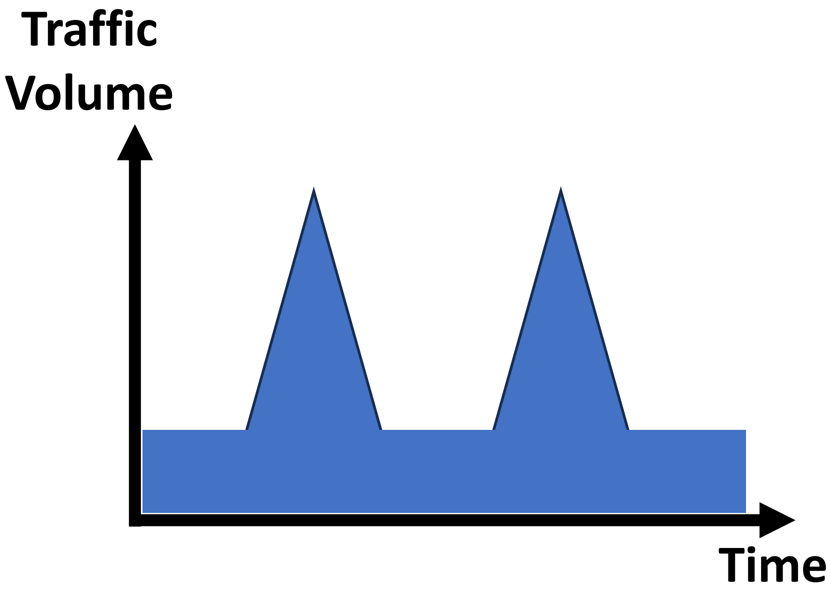
NeoBFT Achieves both **Lower Latency** and **Higher Throughput**



Still better performance when the network is adversarial

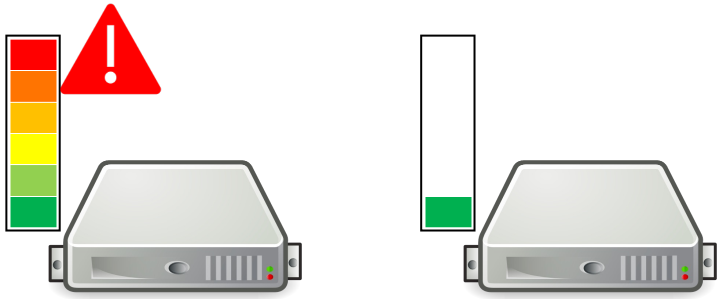
Leveraging Network Programmability for Load Balancing

Challenge: Unpredictable Server Overloads



Bursty Traffic

Unpredictability



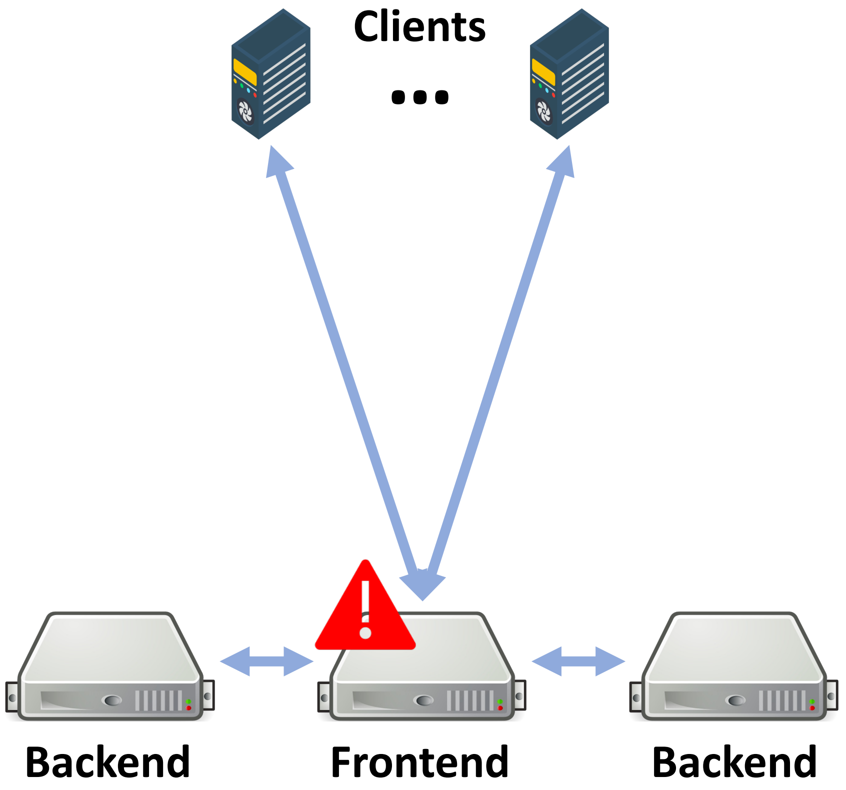
Load Imbalance

Existing Load Balancing Approaches

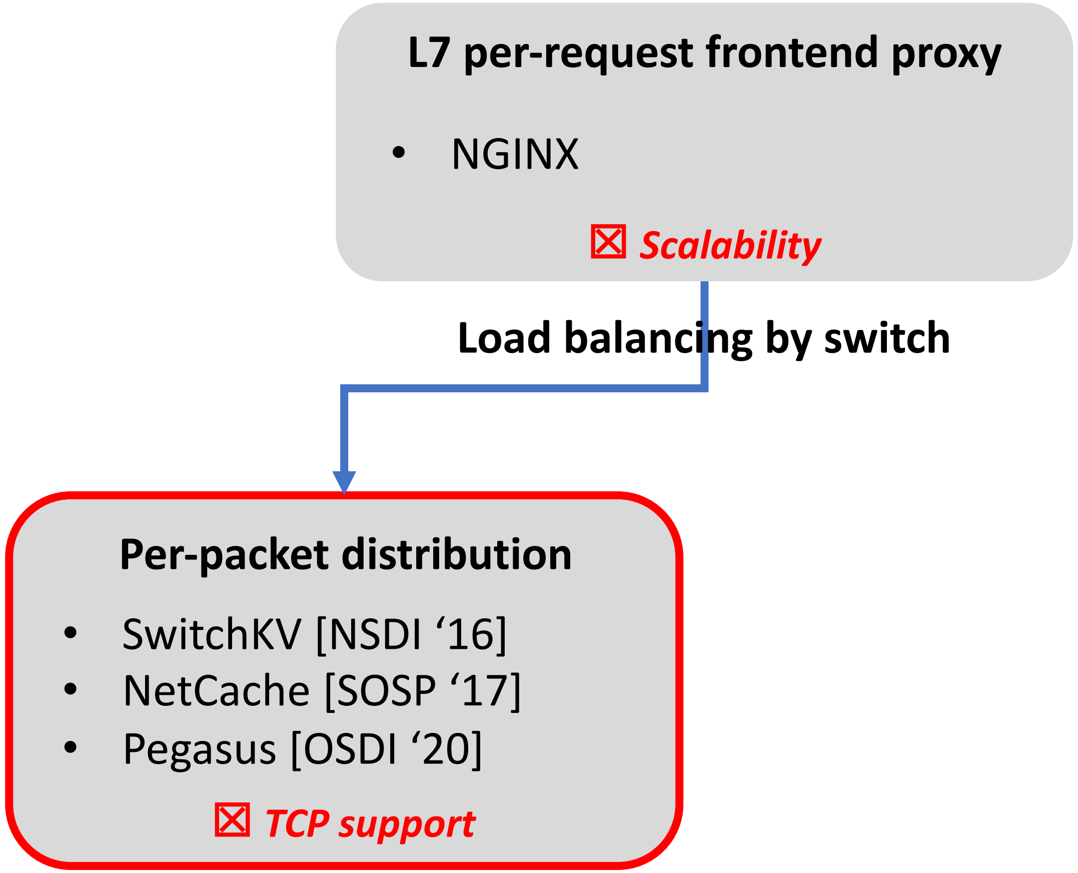
L7 per-request frontend proxy

- NGINX

☒ ***Scalability***



Existing Load Balancing Approaches



L7 per-request frontend proxy

- NGINX

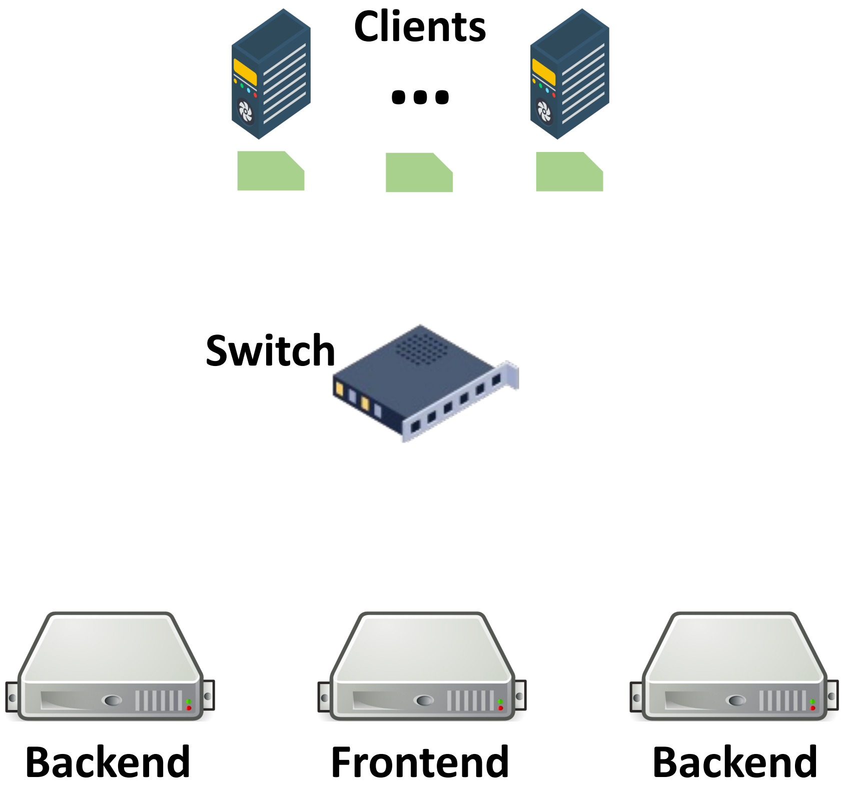
☒ ***Scalability***

Load balancing by switch

Per-packet distribution

- SwitchKV [NSDI ‘16]
- NetCache [SOSP ‘17]
- Pegasus [OSDI ‘20]

☒ ***TCP support***



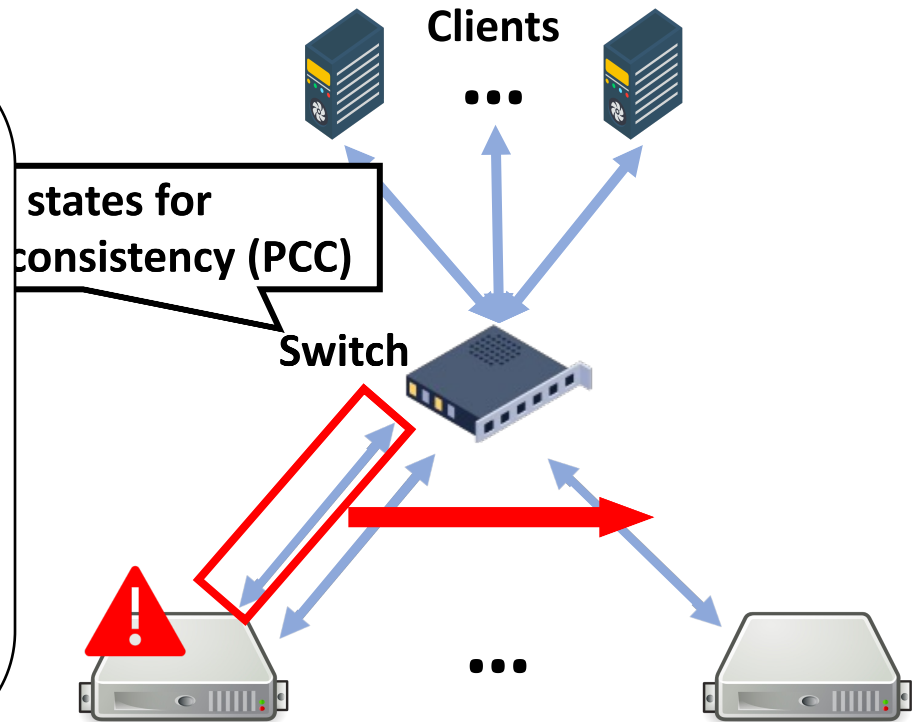
Existing Load Balancing Approaches

L7 per-request frontend proxy

What if servers can migrate live TCP connections?

☒ *TCP support*

☒ *Balancing skewed workloads*



Existing Load Balancing Approaches

L7 per-request frontend proxy

What if servers can migrate live TCP connections?

➔ Strong load balancing systems with

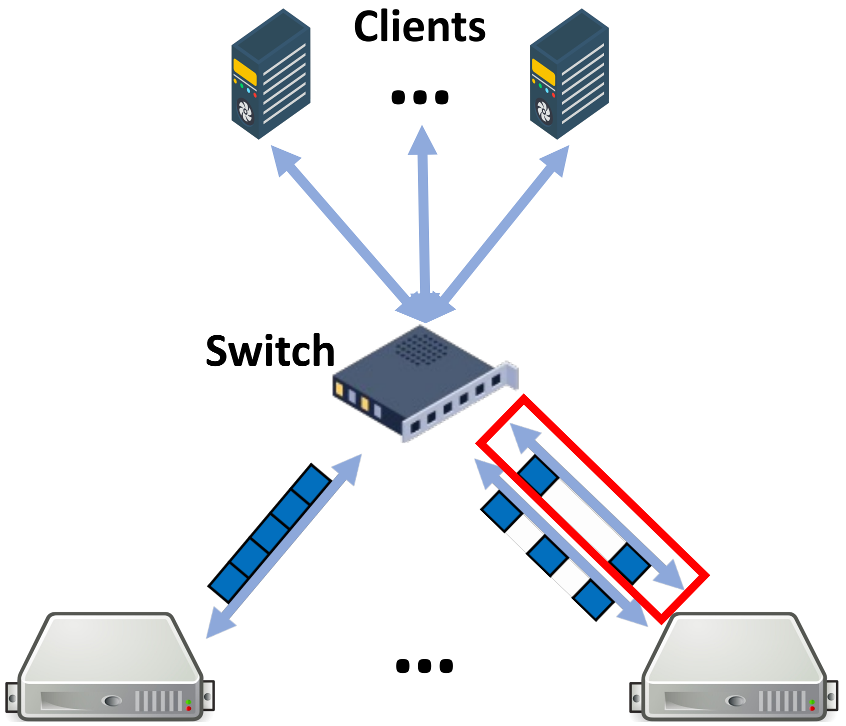
✅ *Scalability*

✅ *TCP support*

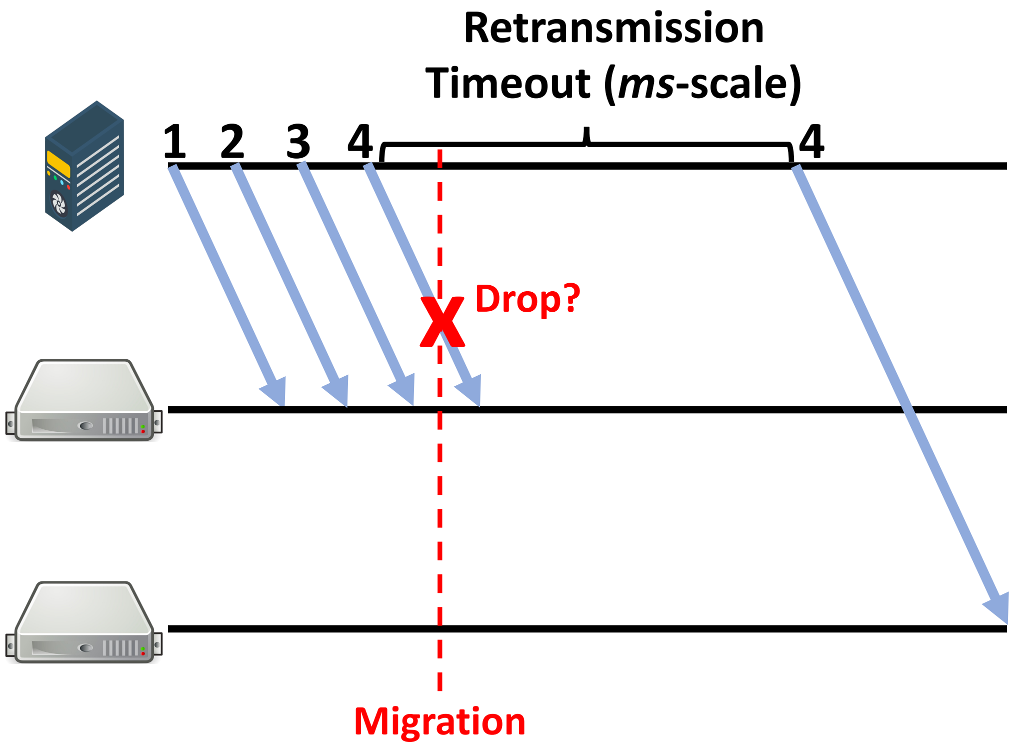
✅ *Balancing skewed workloads*

☒ *TCP support*

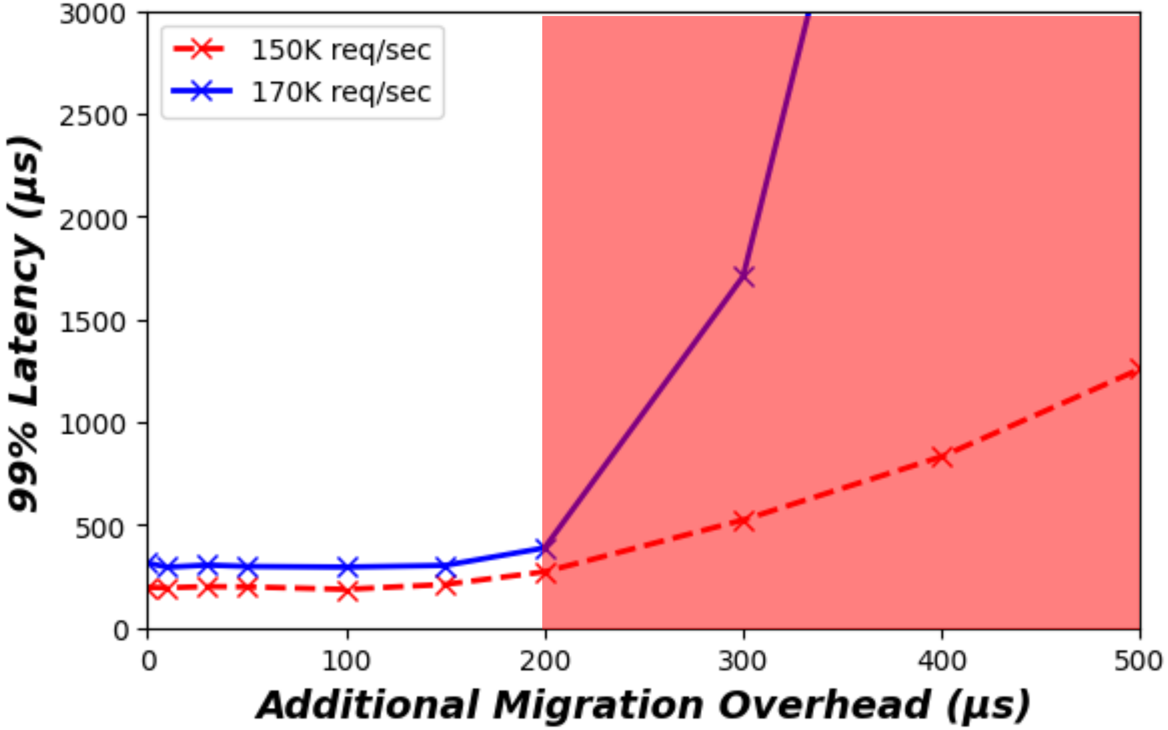
☒ *Balancing skewed workloads*



Disruptive or slow migration can make things even worse



Disruptive (Dropping) Migration

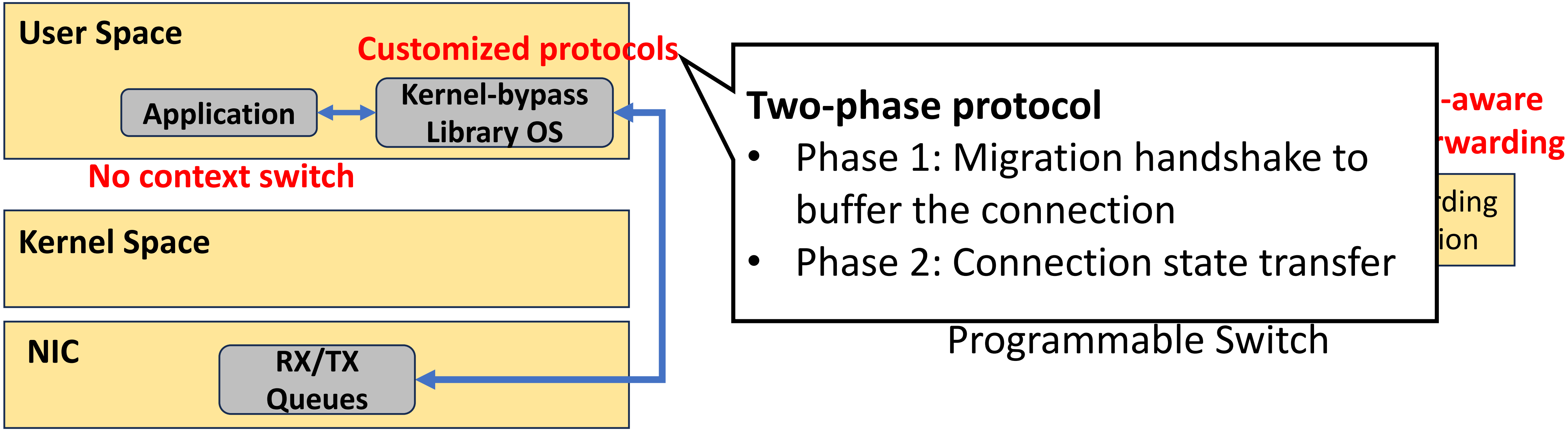


Slow Migration

Capybara

Design goal: μs-scale-fast and <u>client-transparent</u> TCP migration

"without disconnection or packet drops"



Two-phase protocol

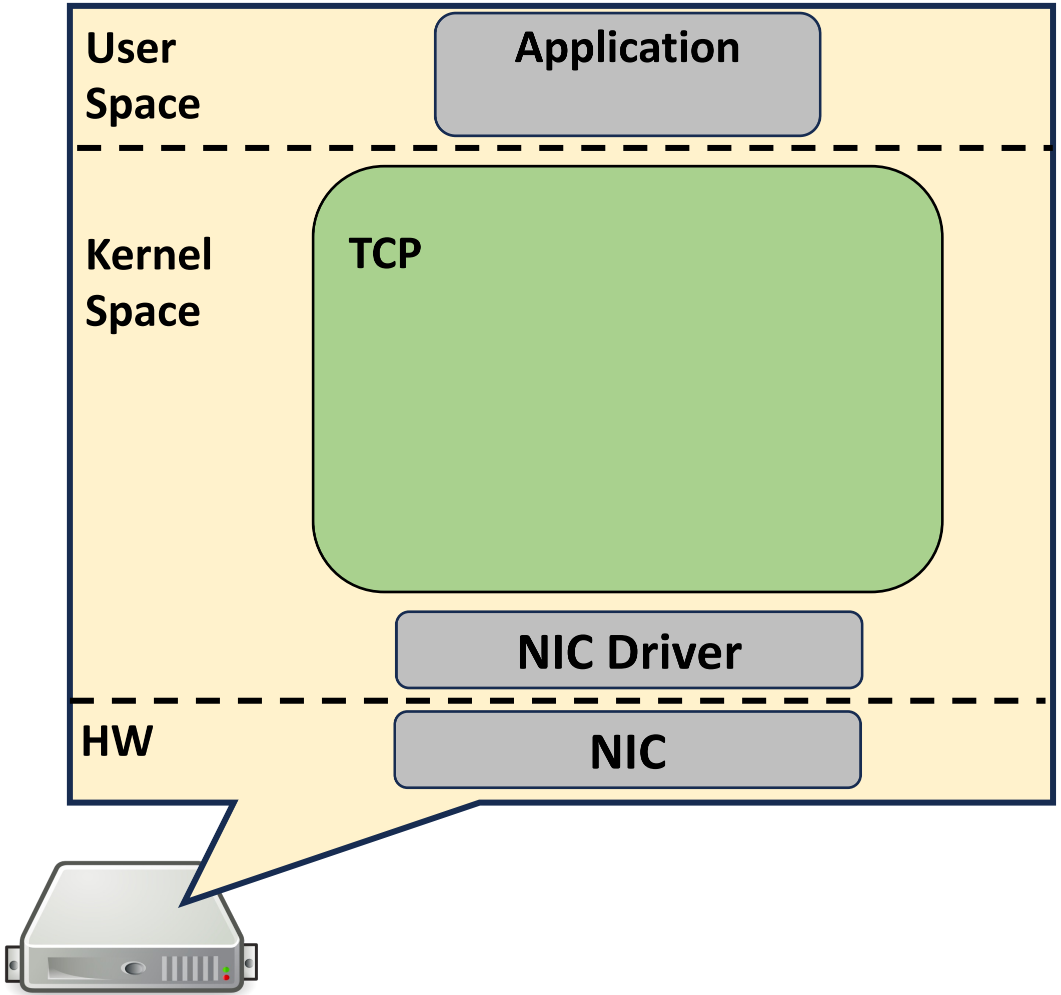
- Phase 1: Migration handshake to buffer the connection
- Phase 2: Connection state transfer

Kernel-bypassing OS

Programmable Switch

✓ **Transparently migrate a live TCP connection within 15 μs**

Server Architecture



Standard (Linux-based) Server

Server Architecture

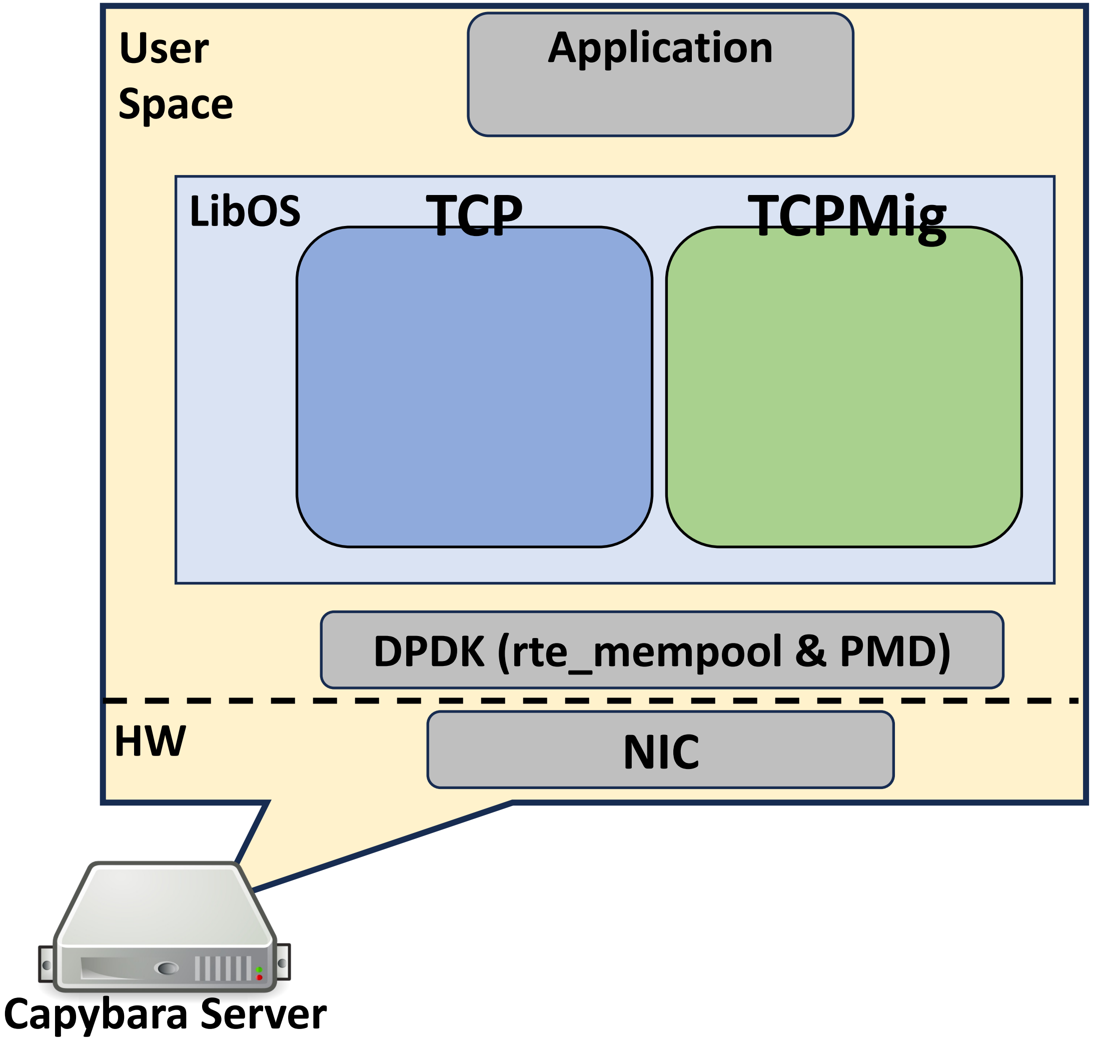
Implements TCP migration protocol in Demikernel LibOS [SOSP '21]

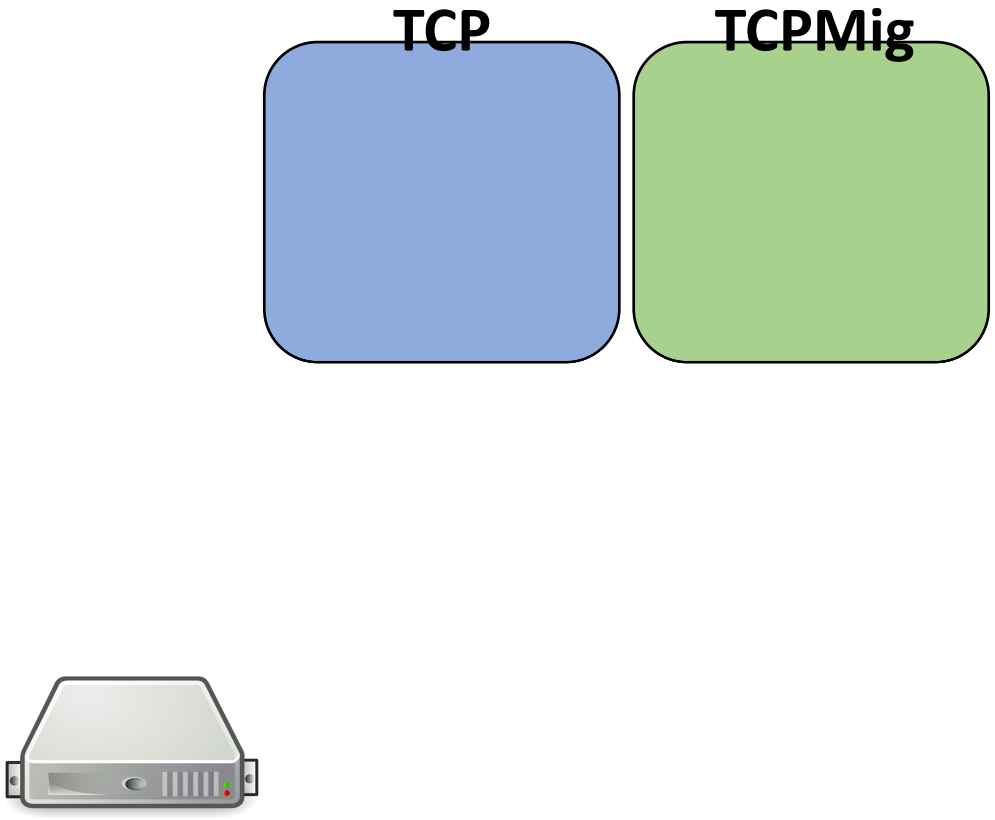
TCP

- Maintains TCP stats (e.g., per-connection request rate)
- TCP state management

TCPMig (Layer 4)

- Migration protocol implementation
- Process migration msgs (PREPARE_MIG, etc.)
- Transient packet buffering



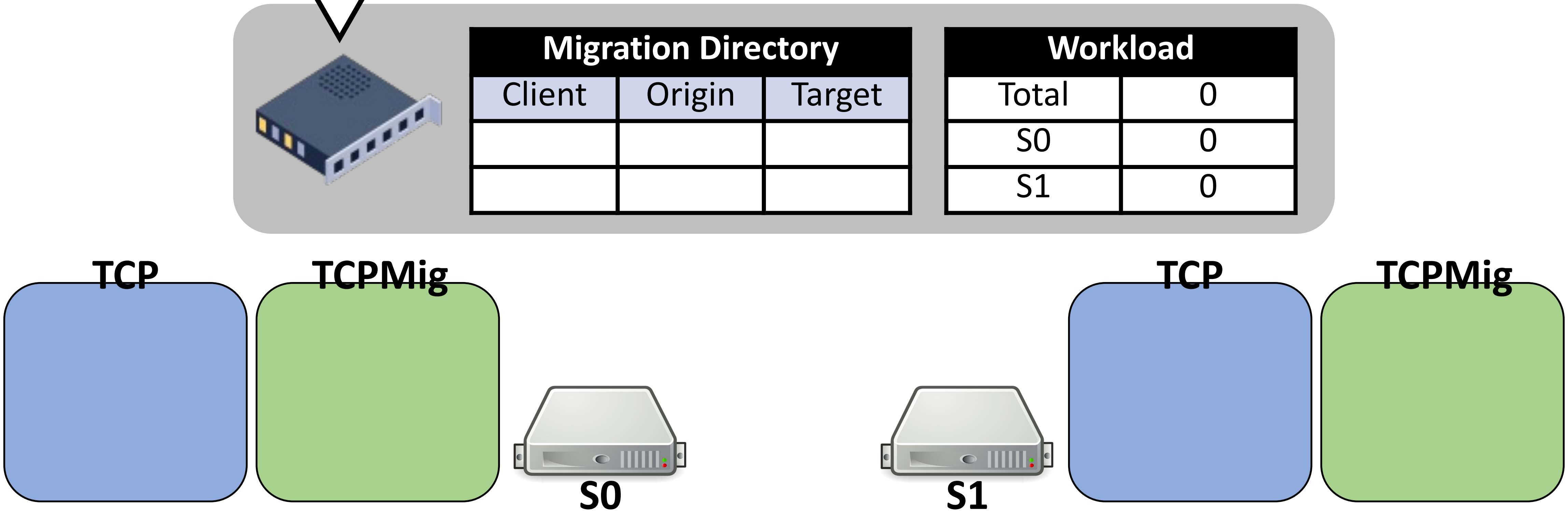


TCP

TCPMig

Switch Architecture

1. **Migration-aware packet forwarding**

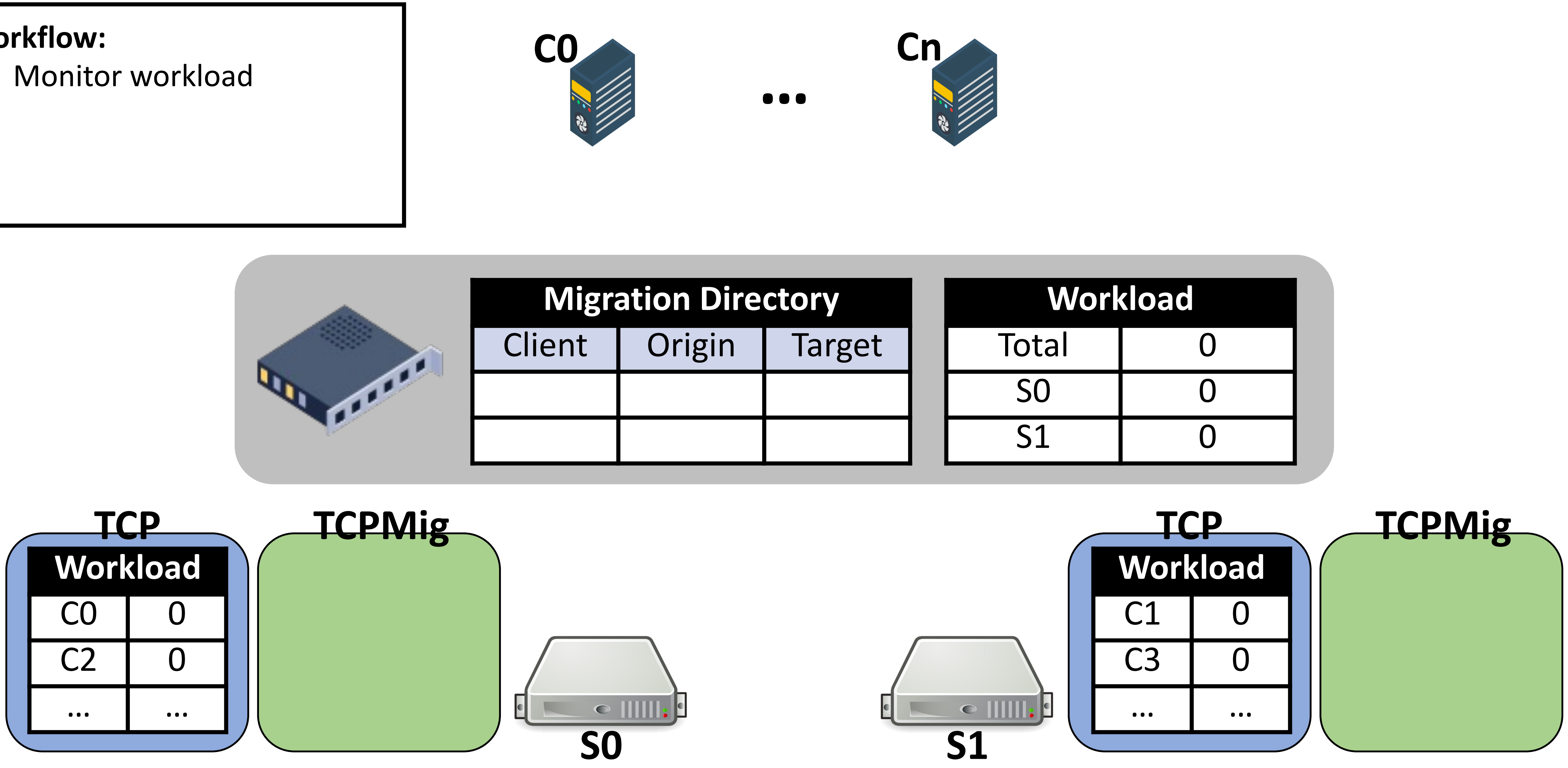


Migration Directory		
Client	Origin	Target

Workload	
Total	0
S0	0
S1	0

Workflow:

1. Monitor workload



Migration Directory		
Client	Origin	Target

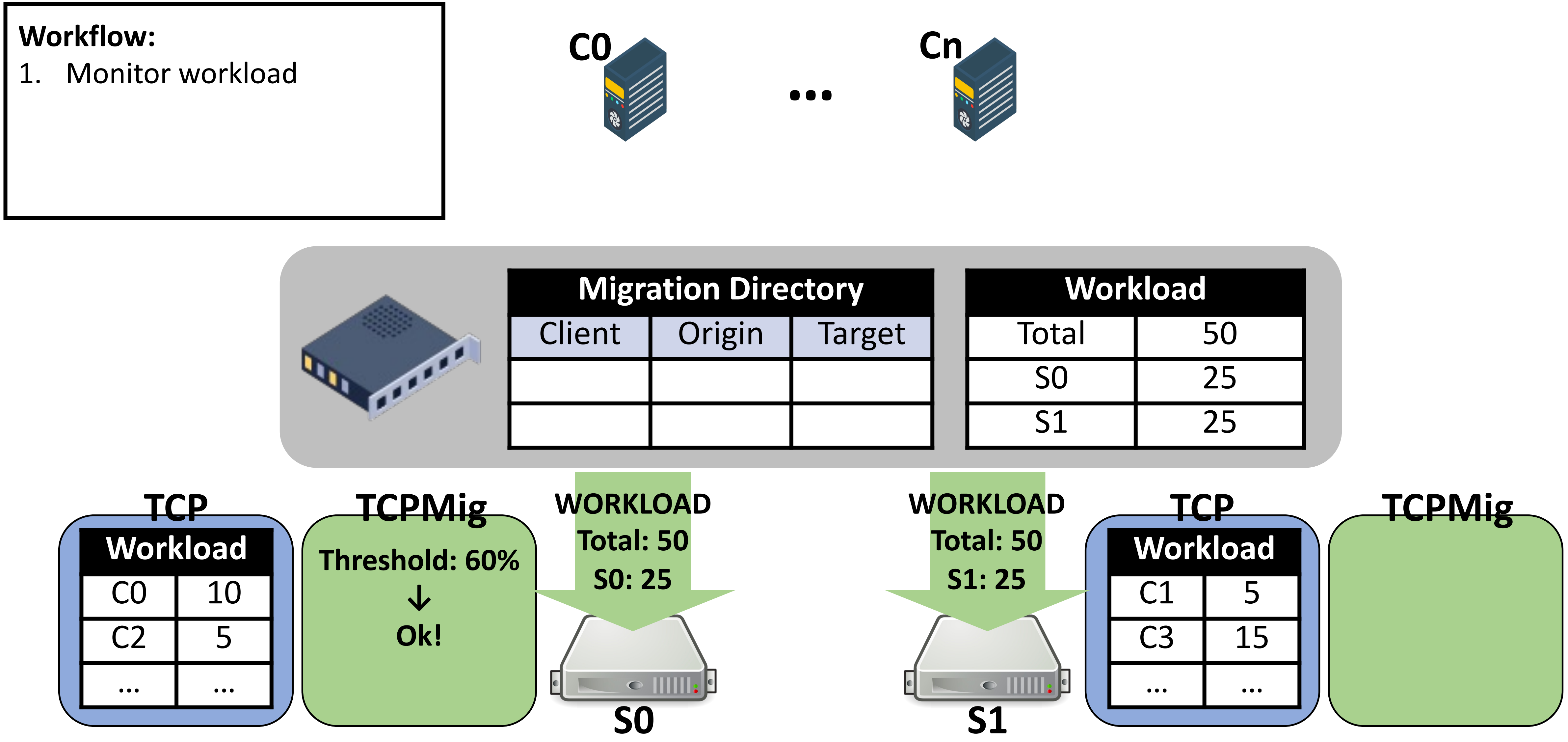
Workload	
Total	0
S0	0
S1	0

Workload	
C0	0
C2	0
…	…

Workload	
C1	0
C3	0
…	…

Workflow:

1. Monitor workload



Migration Directory

Client	Origin	Target

Workload

Total	50
S0	25
S1	25

Workload

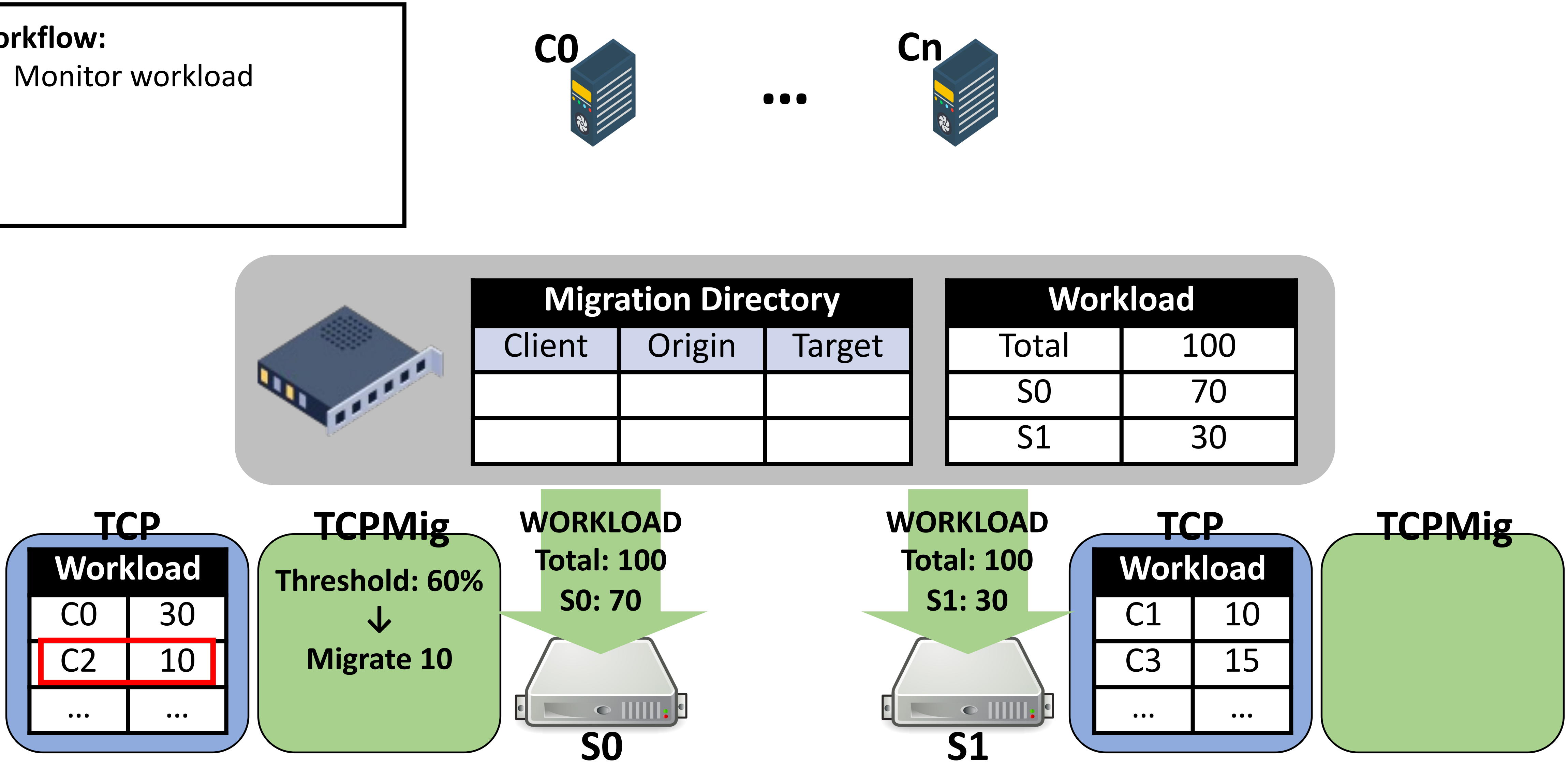
C0	10
C2	5
…	…

Workload

C1	5
C3	15
…	…

Workflow:

1. Monitor workload



Migration Directory		
Client	Origin	Target

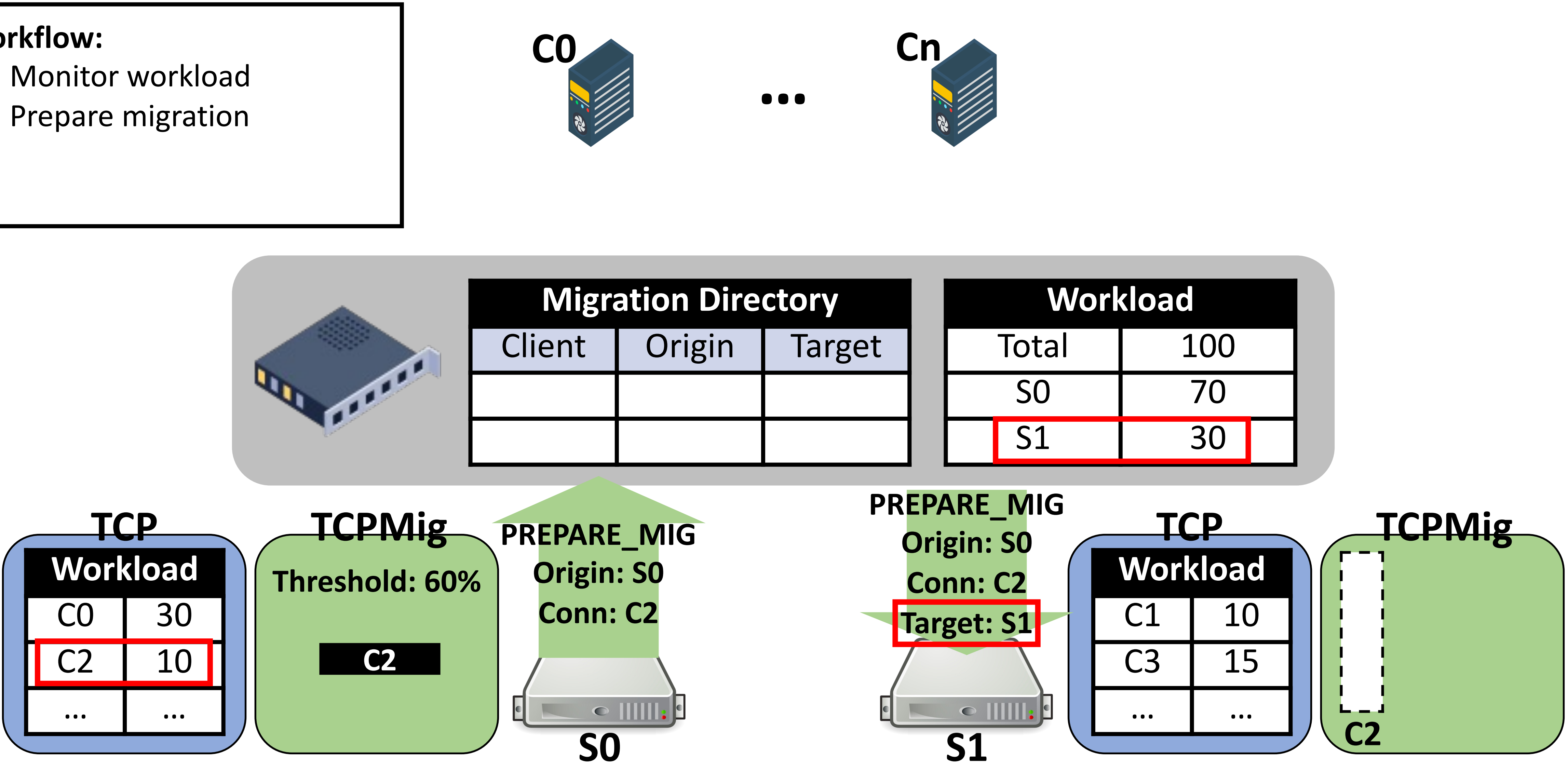
Workload	
Total	100
S0	70
S1	30

Workload	
C0	30
C2	10
…	…

Workload	
C1	10
C3	15
…	…

Workflow:

1. Monitor workload
2. Prepare migration



Migration Directory		
Client	Origin	Target

Workload	
Total	100
S0	70
S1	30

TCP

Workload	
C0	30
C2	10
…	…

TCPMig

Threshold: 60%

PREPARE_MIG
Origin: S0
Conn: C2

PREPARE_MIG
Origin: S0
Conn: C2
Target: S1

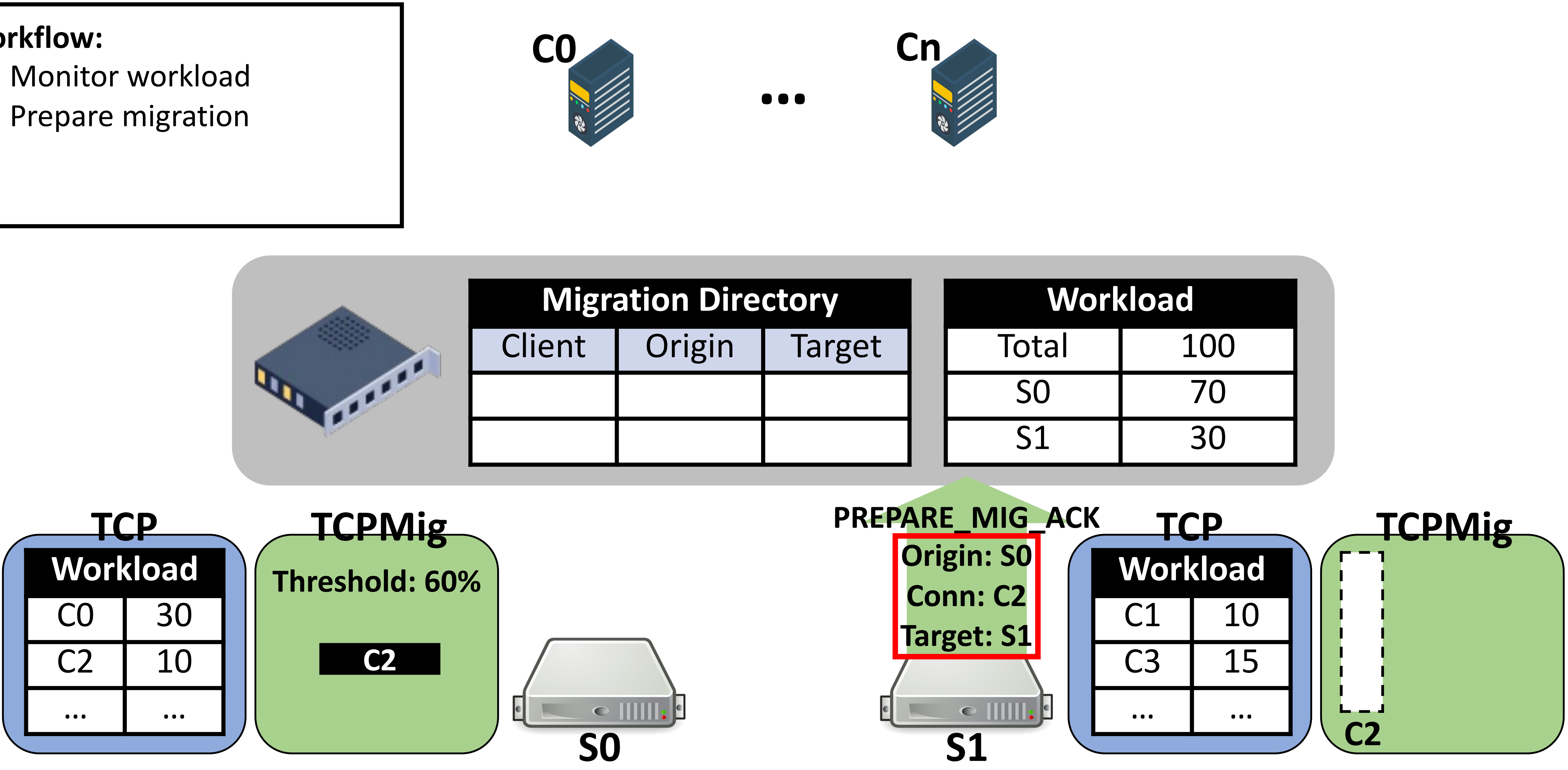
TCP

Workload	
C1	10
C3	15
…	…

TCPMig

Workflow:

1. Monitor workload
2. Prepare migration



Migration Directory		
Client	Origin	Target

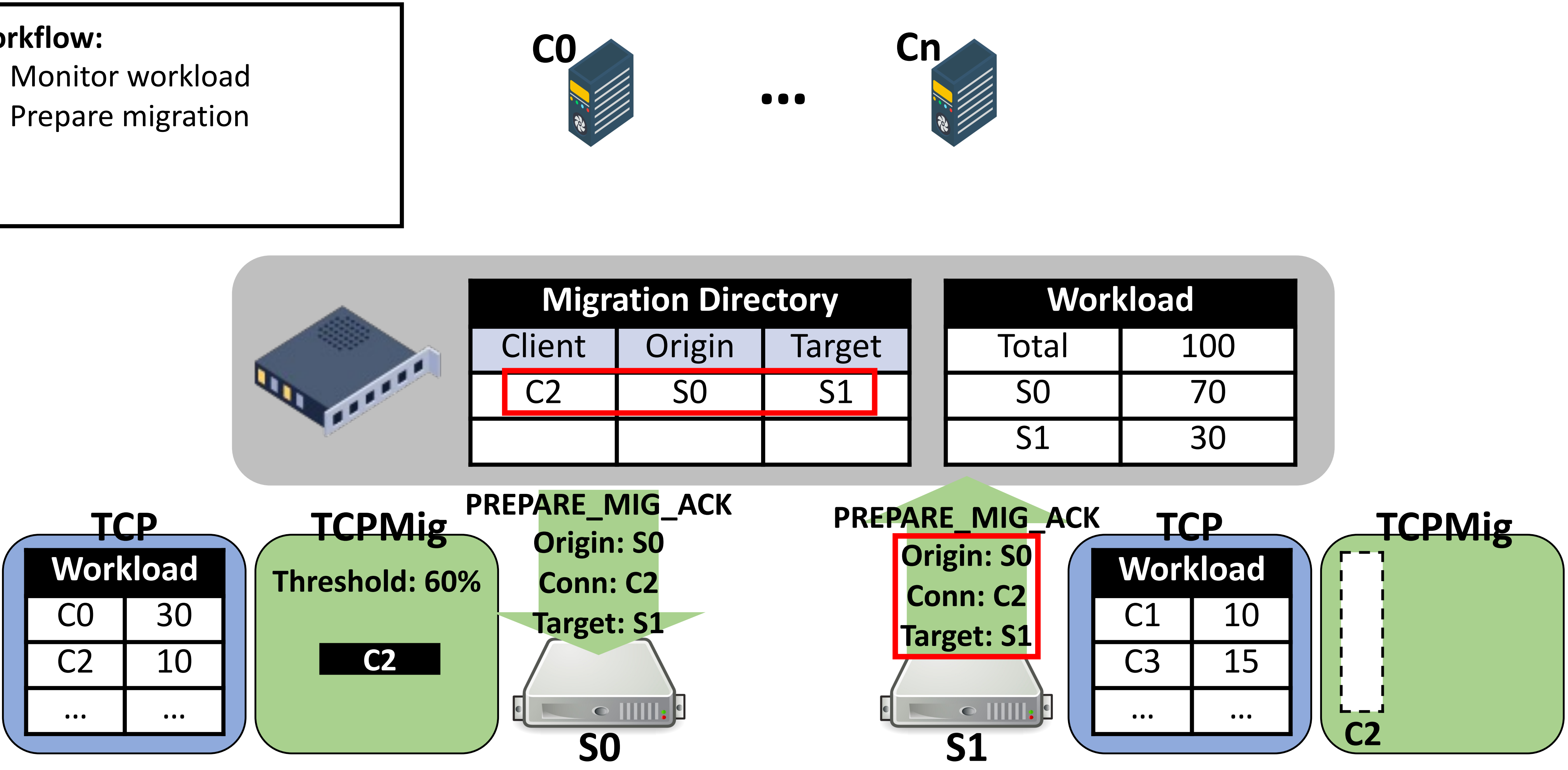
Workload	
Total	100
S0	70
S1	30

Workload	
C0	30
C2	10
…	…

Workload	
C1	10
C3	15
…	…

Workflow:

1. Monitor workload
2. Prepare migration



Migration Directory		
Client	Origin	Target
C2	S0	S1

Workload	
Total	100
S0	70
S1	30

PREPARE_MIG_ACK
Origin: S0
Conn: C2
Target: S1

Workload	
C0	30
C2	10
…	…

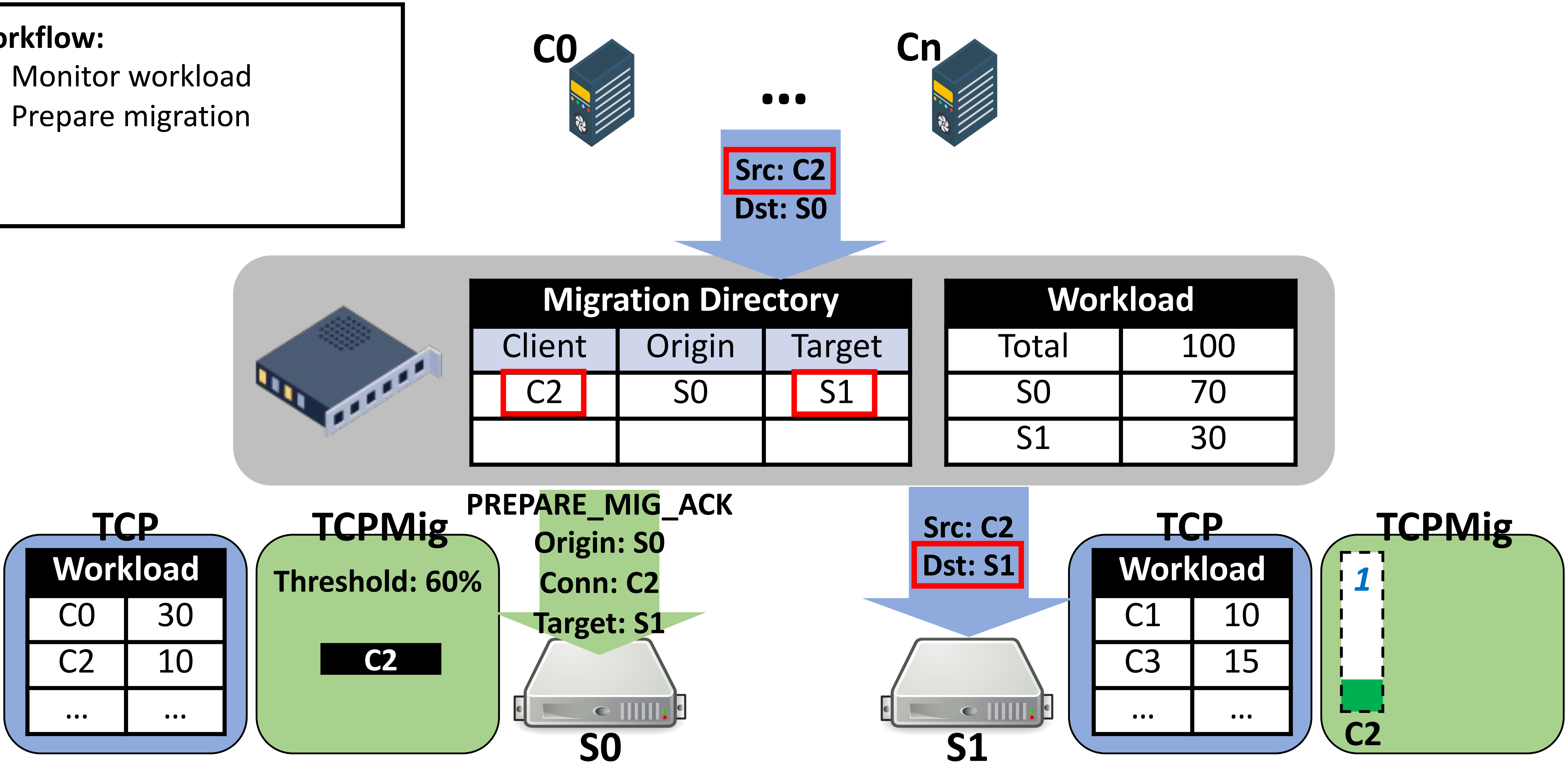
Threshold: 60%

C2

Workload	
C1	10
C3	15
…	…

Workflow:

1. Monitor workload
2. Prepare migration



Migration Directory		
Client	Origin	Target
C2	S0	S1

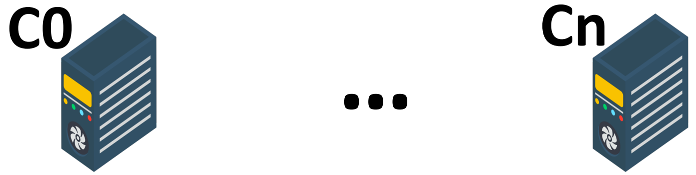
Workload	
Total	100
S0	70
S1	30

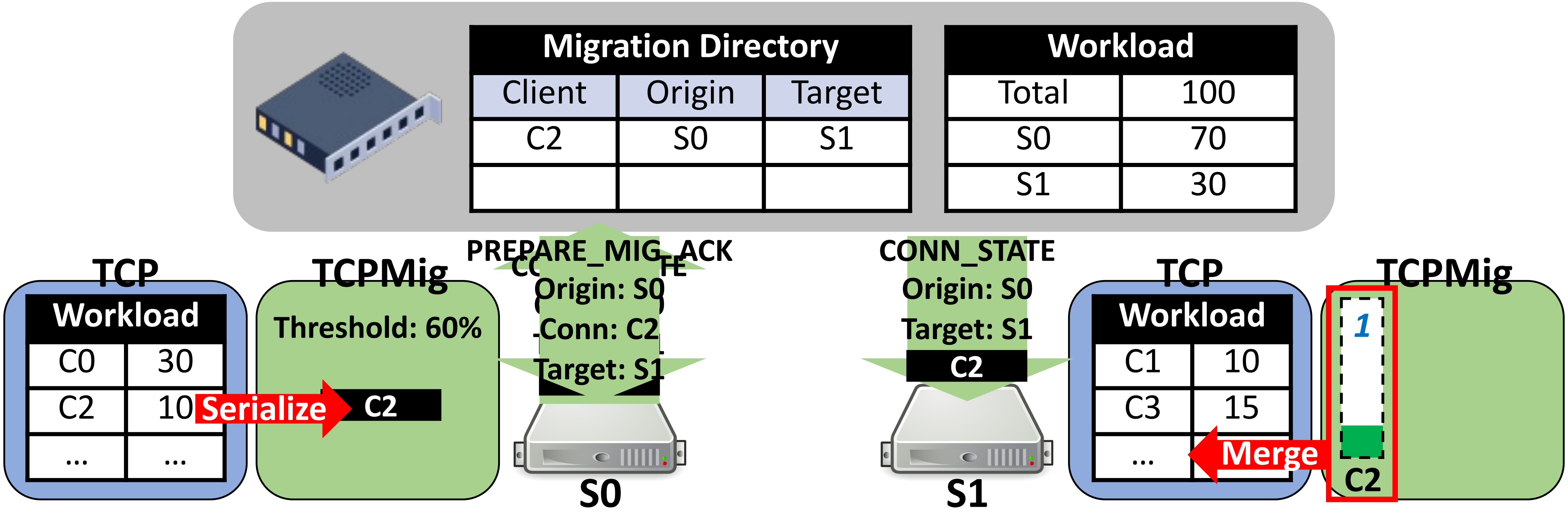
Workload	
C0	30
C2	10
…	…

Workload	
C1	10
C3	15
…	…

Workflow:

1. Monitor workload
2. Prepare migration
3. State transfer





Migration Directory		
Client	Origin	Target
C2	S0	S1

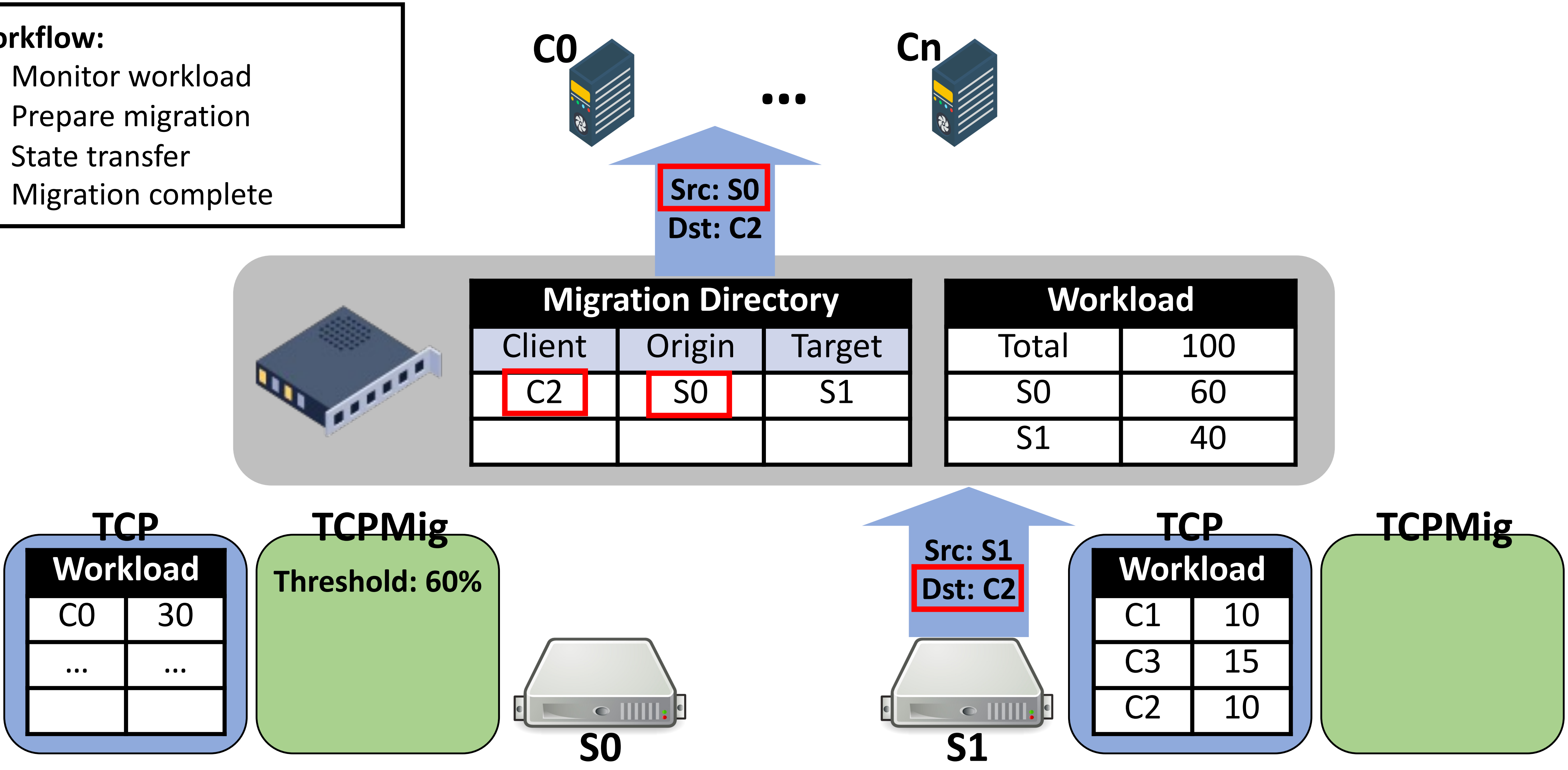
Workload	
Total	100
S0	70
S1	30

Workload	
C0	30
C2	10
…	…

Workload	
C1	10
C3	15
…	

Workflow:

1. Monitor workload
2. Prepare migration
3. State transfer
4. Migration complete



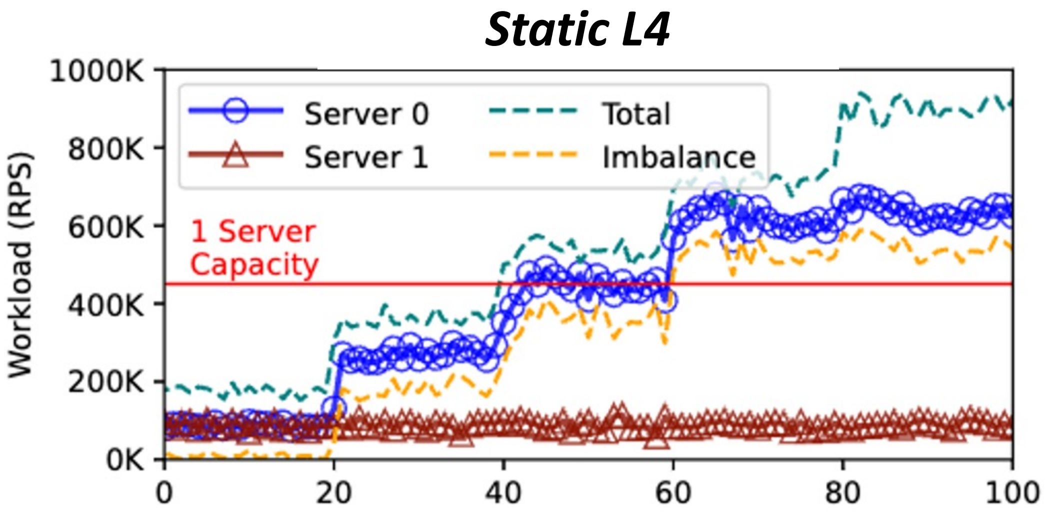
Migration Directory		
Client	Origin	Target
C2	S0	S1

Workload	
Total	100
S0	60
S1	40

Workload	
C0	30
…	…

Workload	
C1	10
C3	15
C2	10

Load Balancing Benefit of Capybara



Conclusion

- New approach to designing distributed systems
  - Leverage programmability in data center networks for partial protocol offloading
- Co-designed distributed systems with both **strong guarantees** and **high performance**
  - Serialization-free network ordering for strongly consistent dapps
  - Authenticated network ordering for BFT protocols
  - us-scale live TCP migration for load balancing