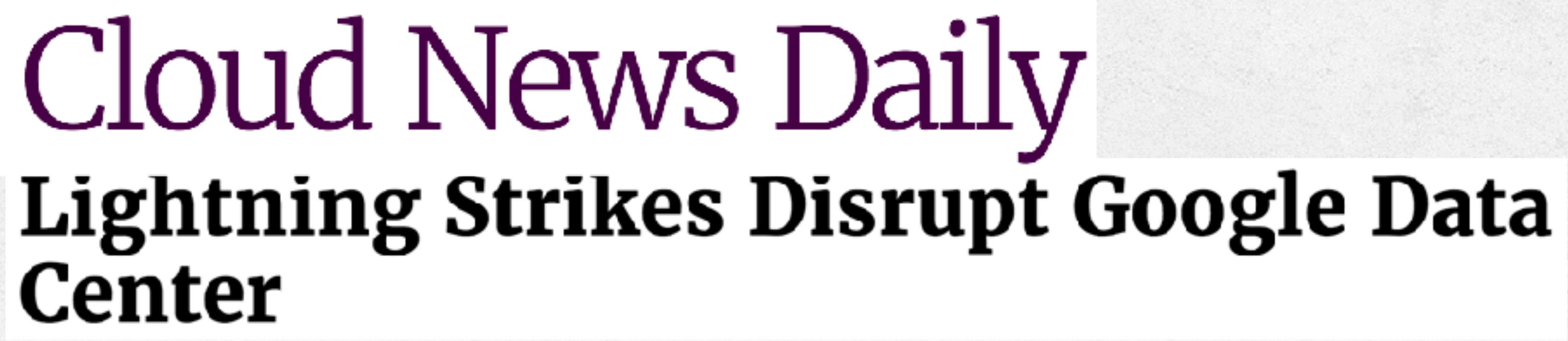
Avoiding Coordination with Network Ordering: NOPaxos and Eris

Ellis Michael

Server failures are the common case in data centers

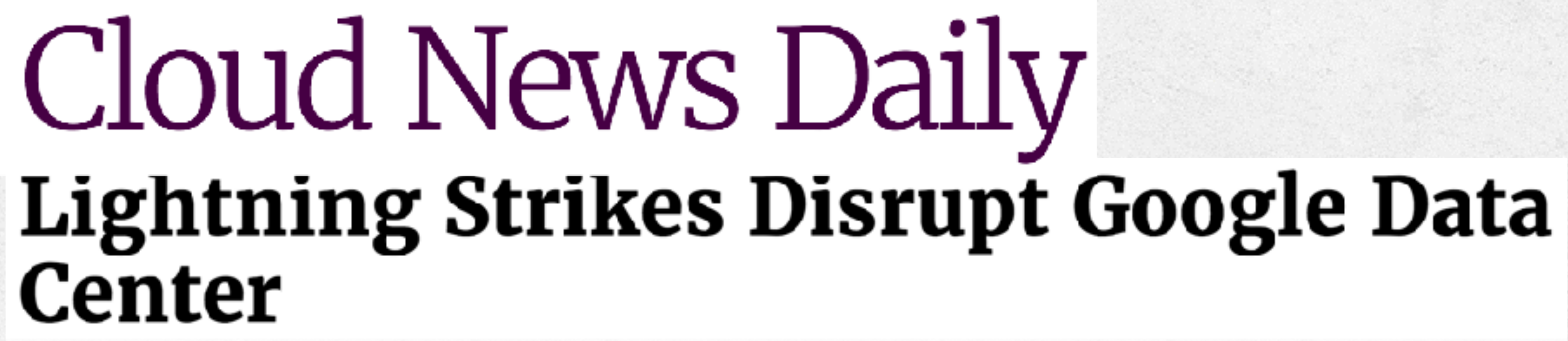
Server failures are the common case in data centers



Cloud News Daily

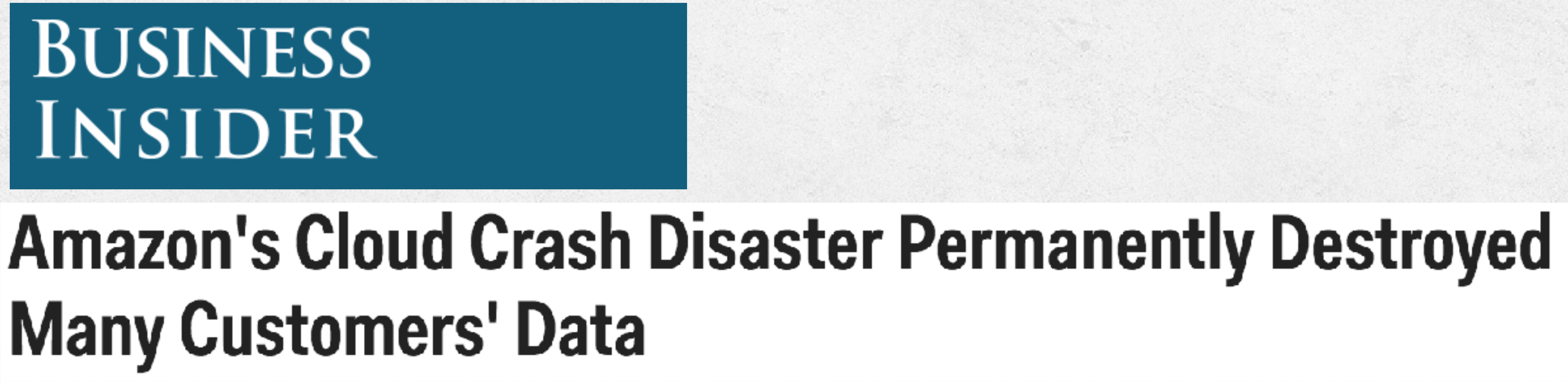
Lightning Strikes Disrupt Google Data Center

Server failures are the common case in data centers



Cloud News Daily

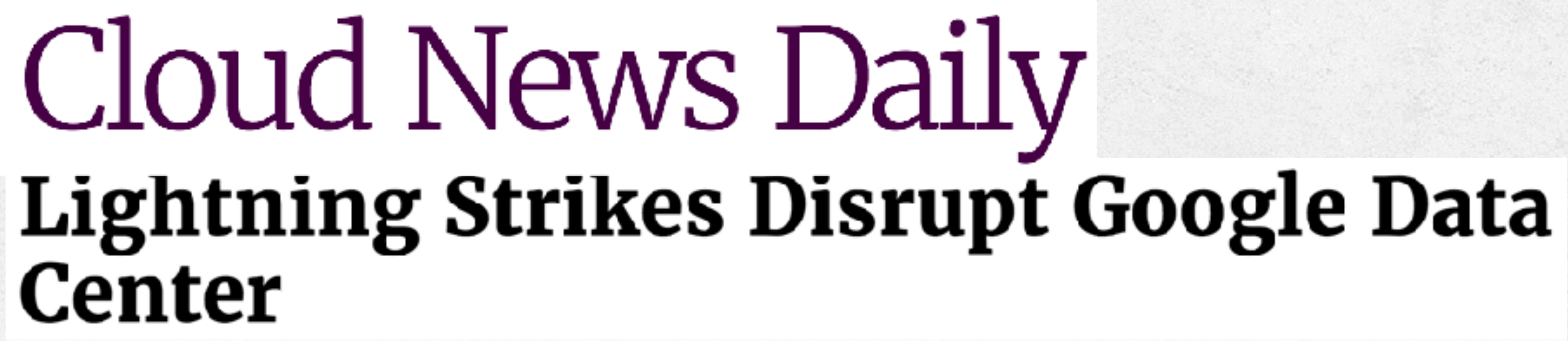
Lightning Strikes Disrupt Google Data Center



Business Insider

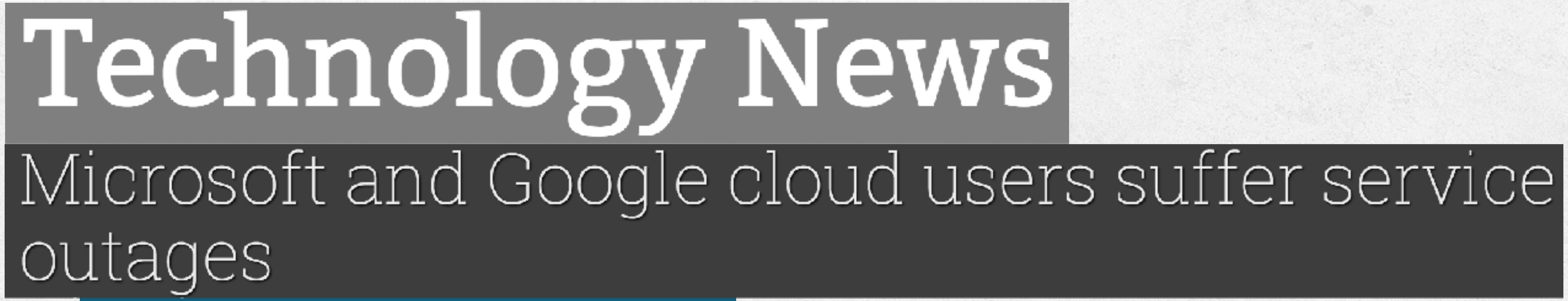
Amazon's Cloud Crash Disaster Permanently Destroyed Many Customers' Data

SERVER FAILURES ARE THE COMMON CASE IN DATA CENTERS



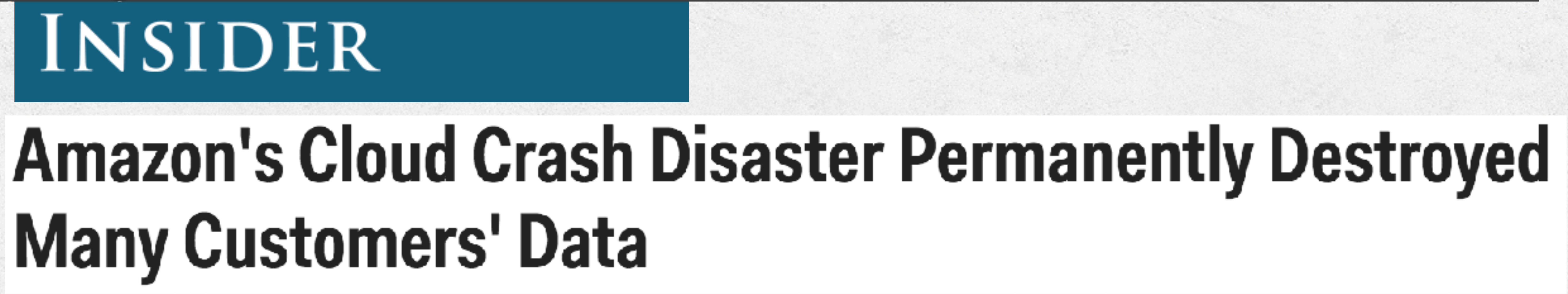
Cloud News Daily

Lightning Strikes Disrupt Google Data Center



Technology News

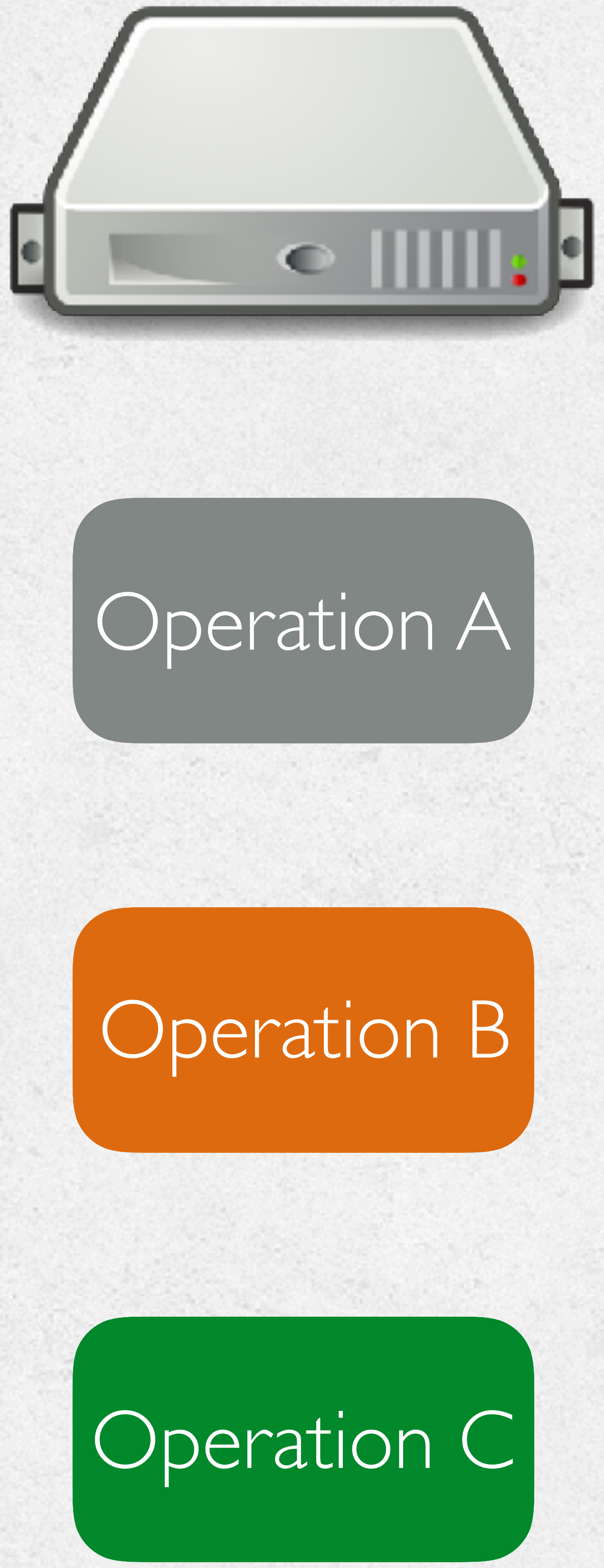
Microsoft and Google cloud users suffer service outages



INSIDER

Amazon's Cloud Crash Disaster Permanently Destroyed Many Customers' Data

State Machine Replication

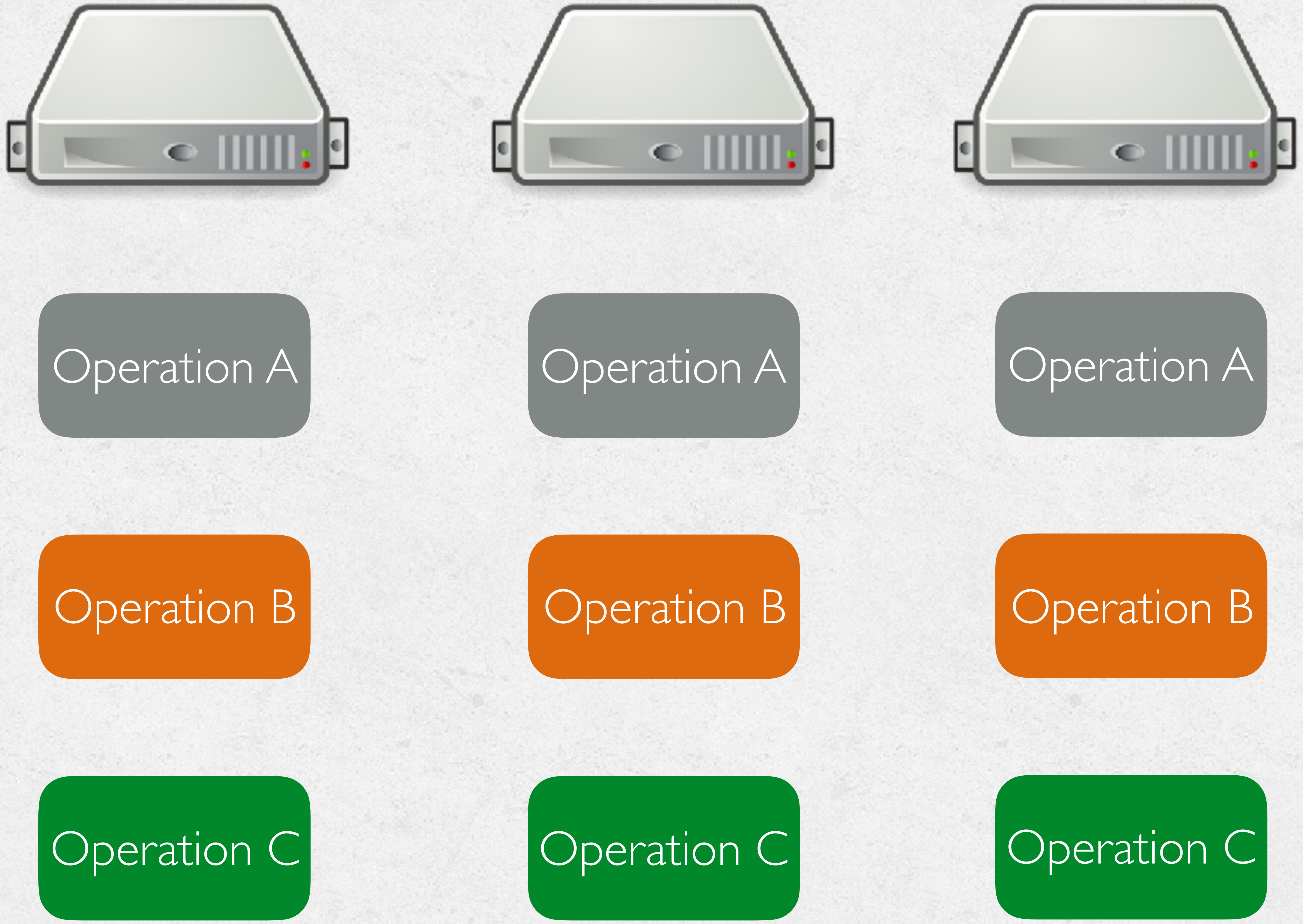


Operation A

Operation B

Operation C

State Machine Replication

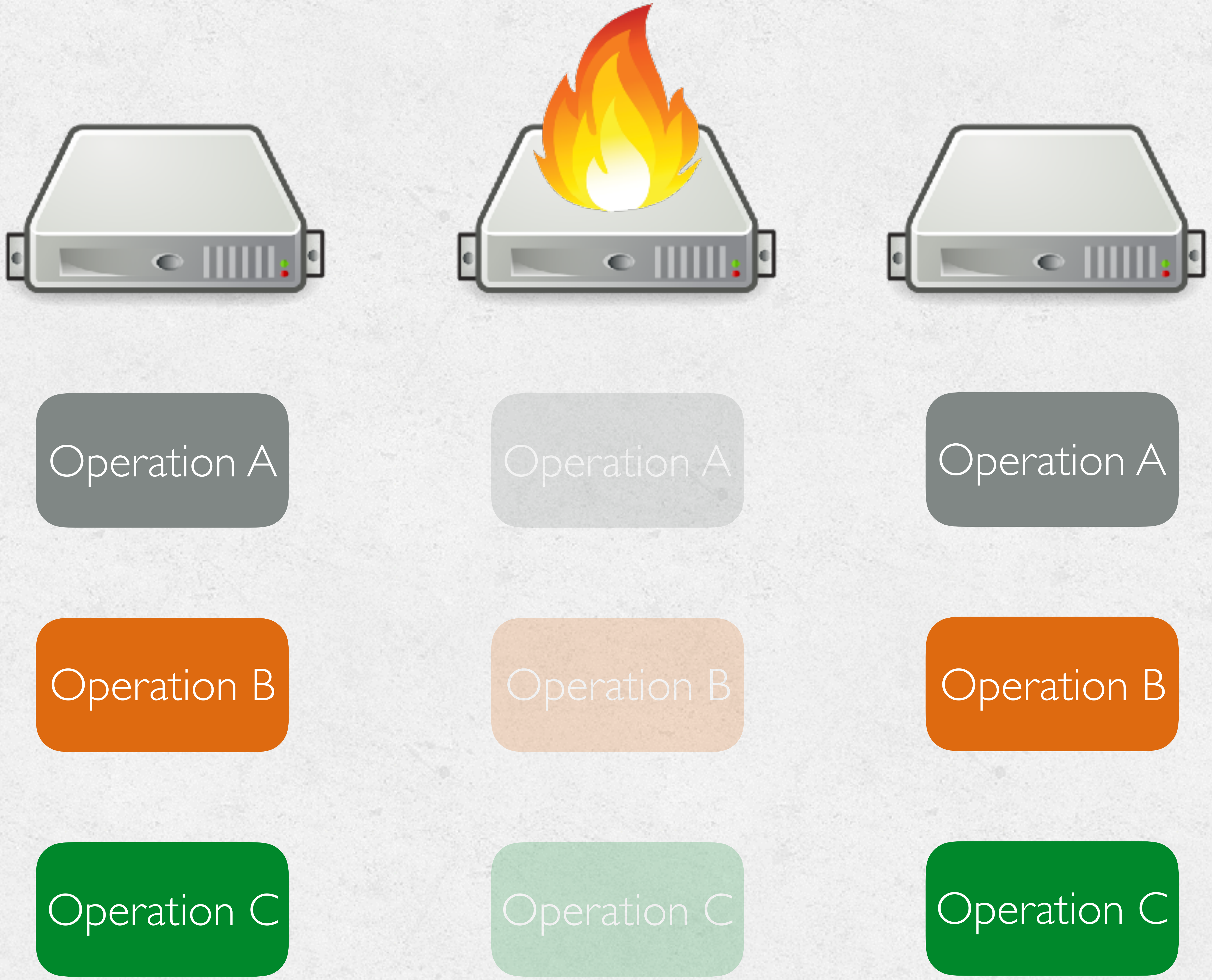


Operation A Operation A Operation A

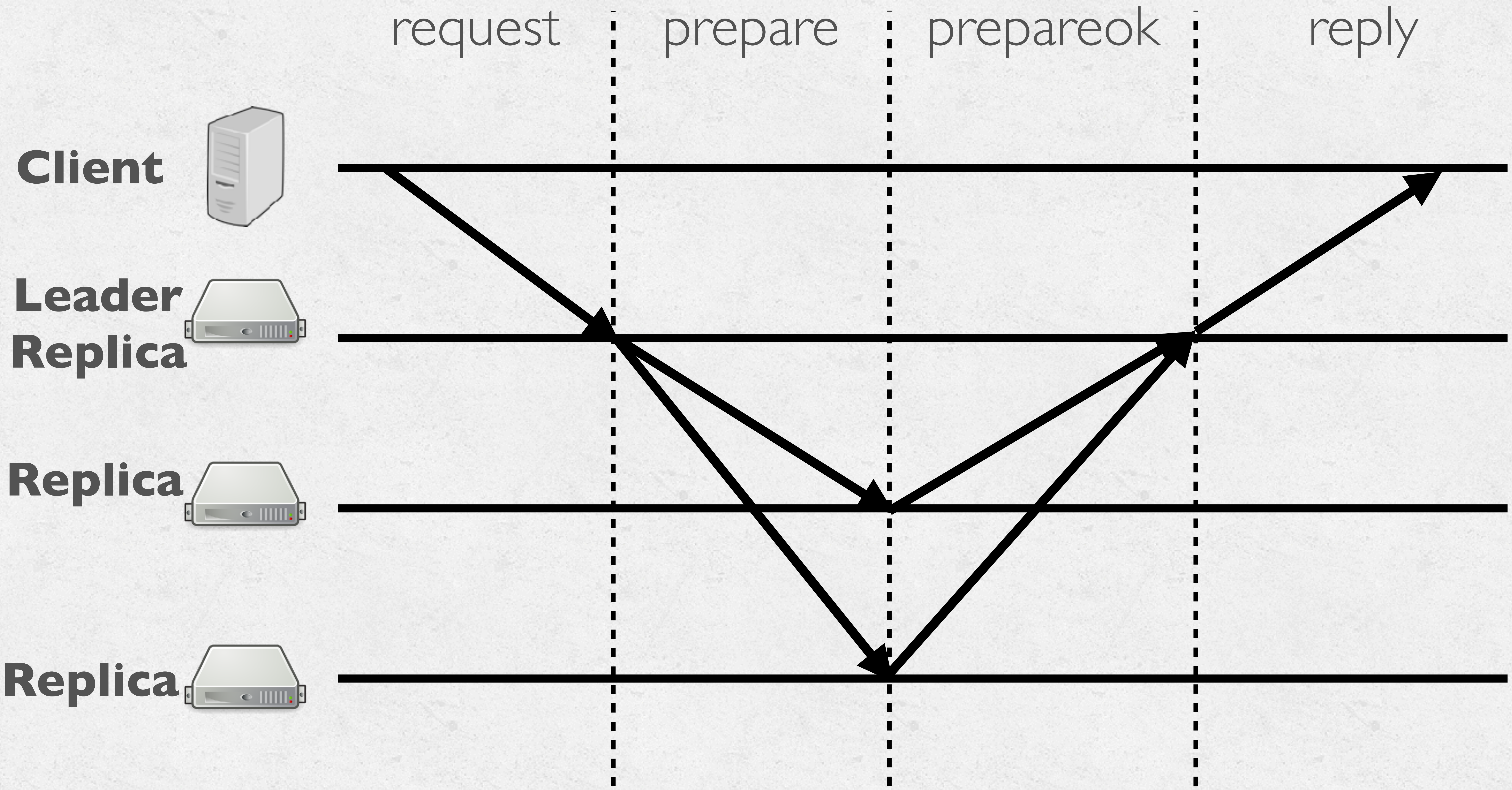
Operation B Operation B Operation B

Operation C Operation C Operation C

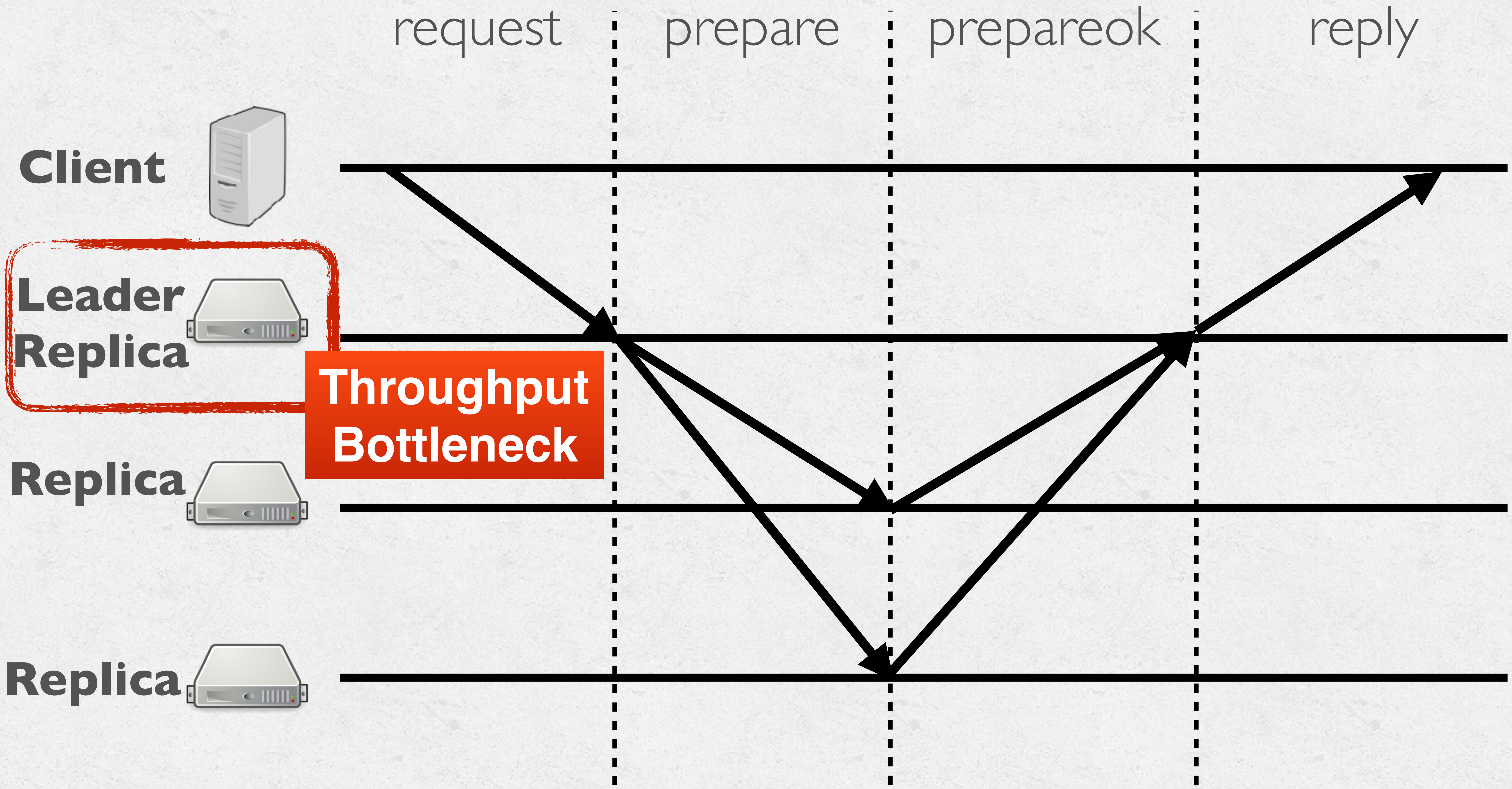
State Machine Replication



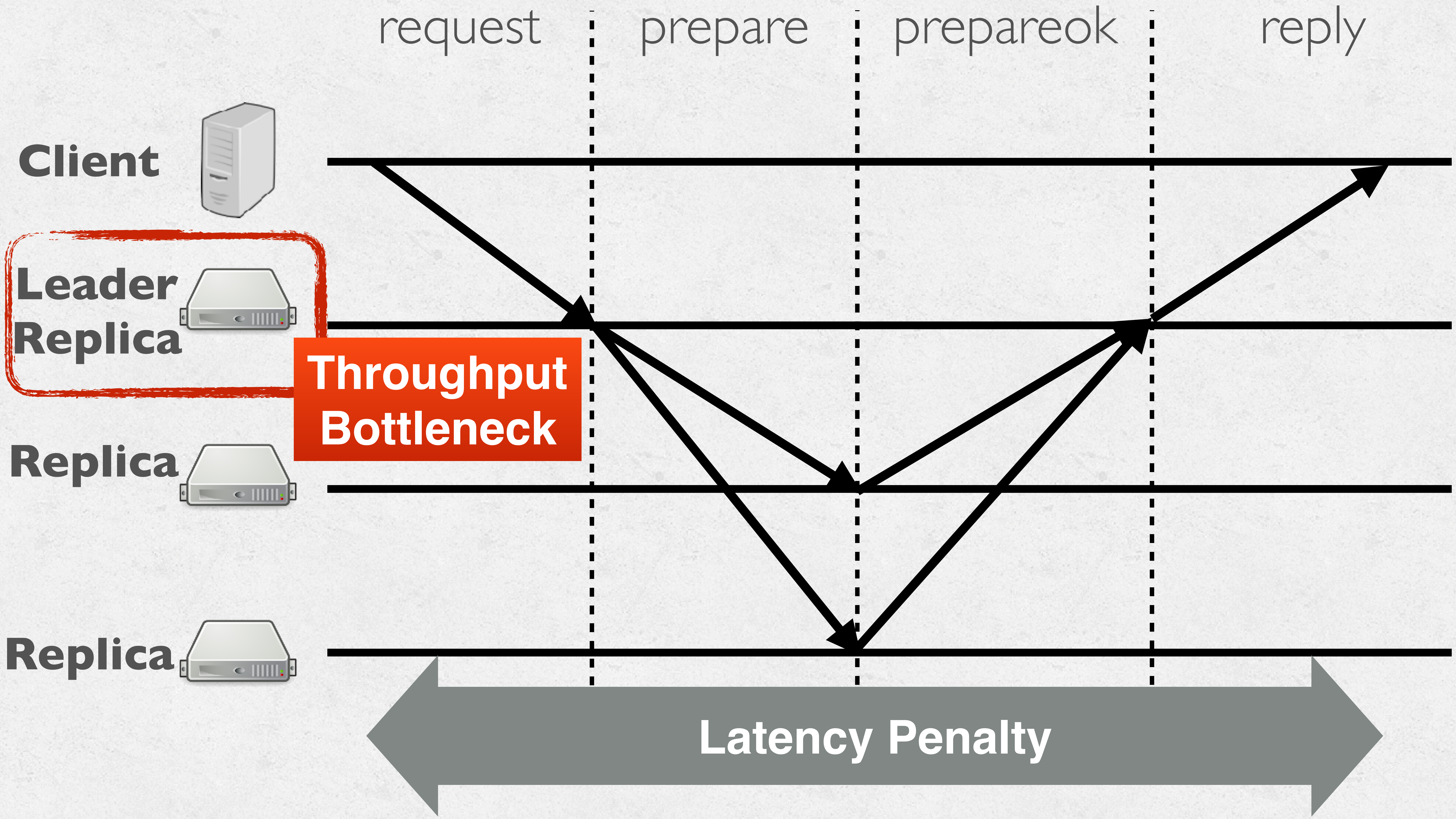
Paxos for state machine replication



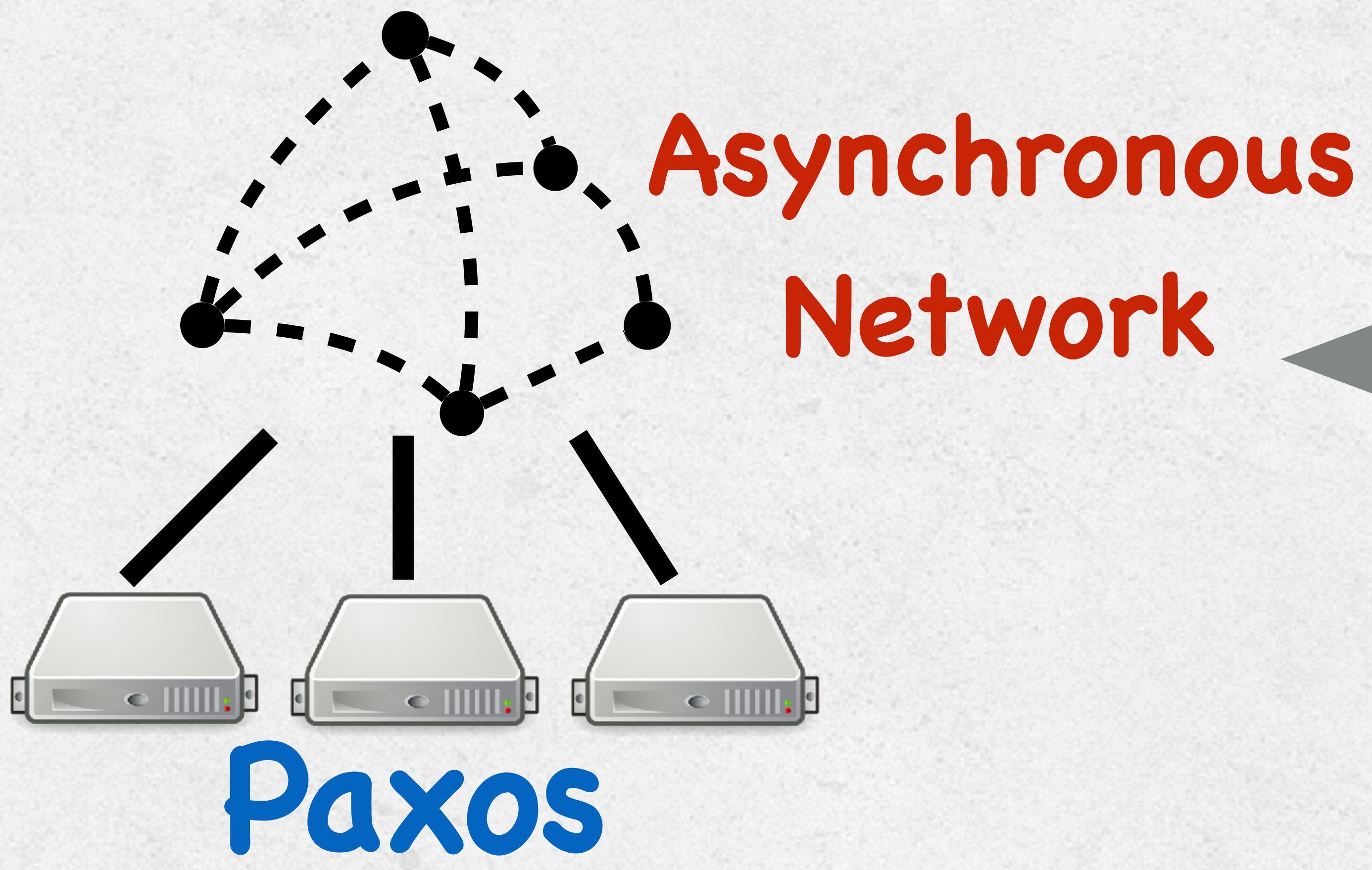
Paxos for State Machine Replication



PAXOS FOR STATE MACHINE REPLICATION



Network properties determine replication complexity

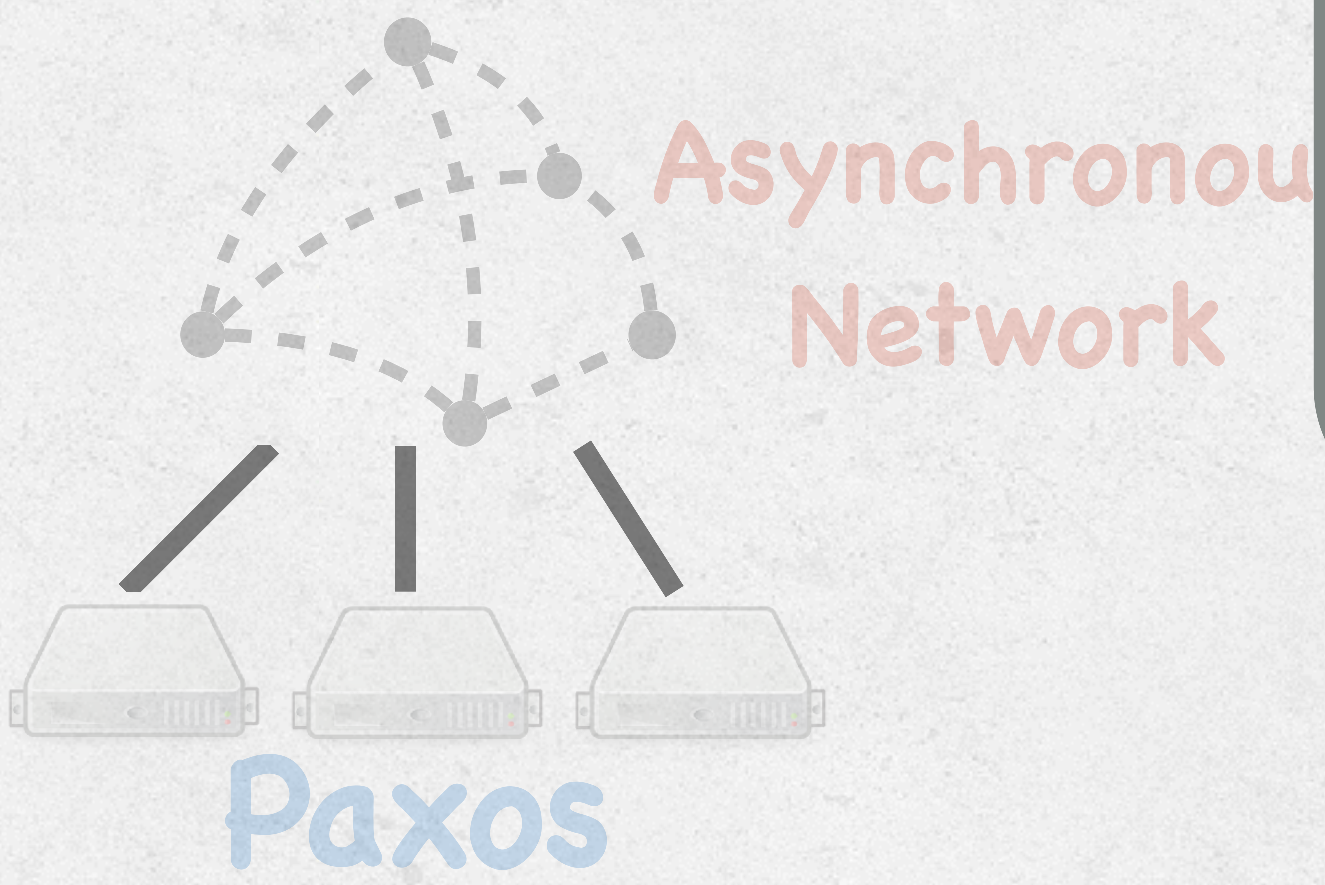


Messages may be:

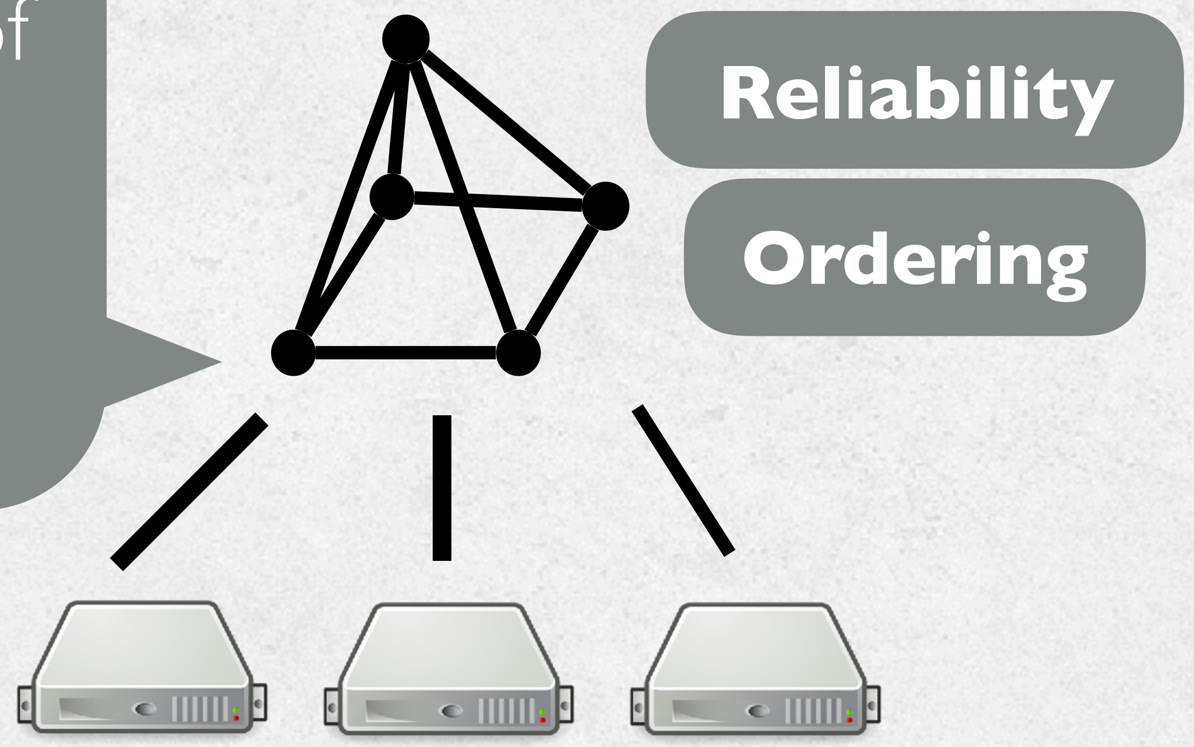
- dropped
- reordered
- delivered with arbitrary latency

- Paxos protocol on every operation
- High performance cost

Network properties determine replication complexity

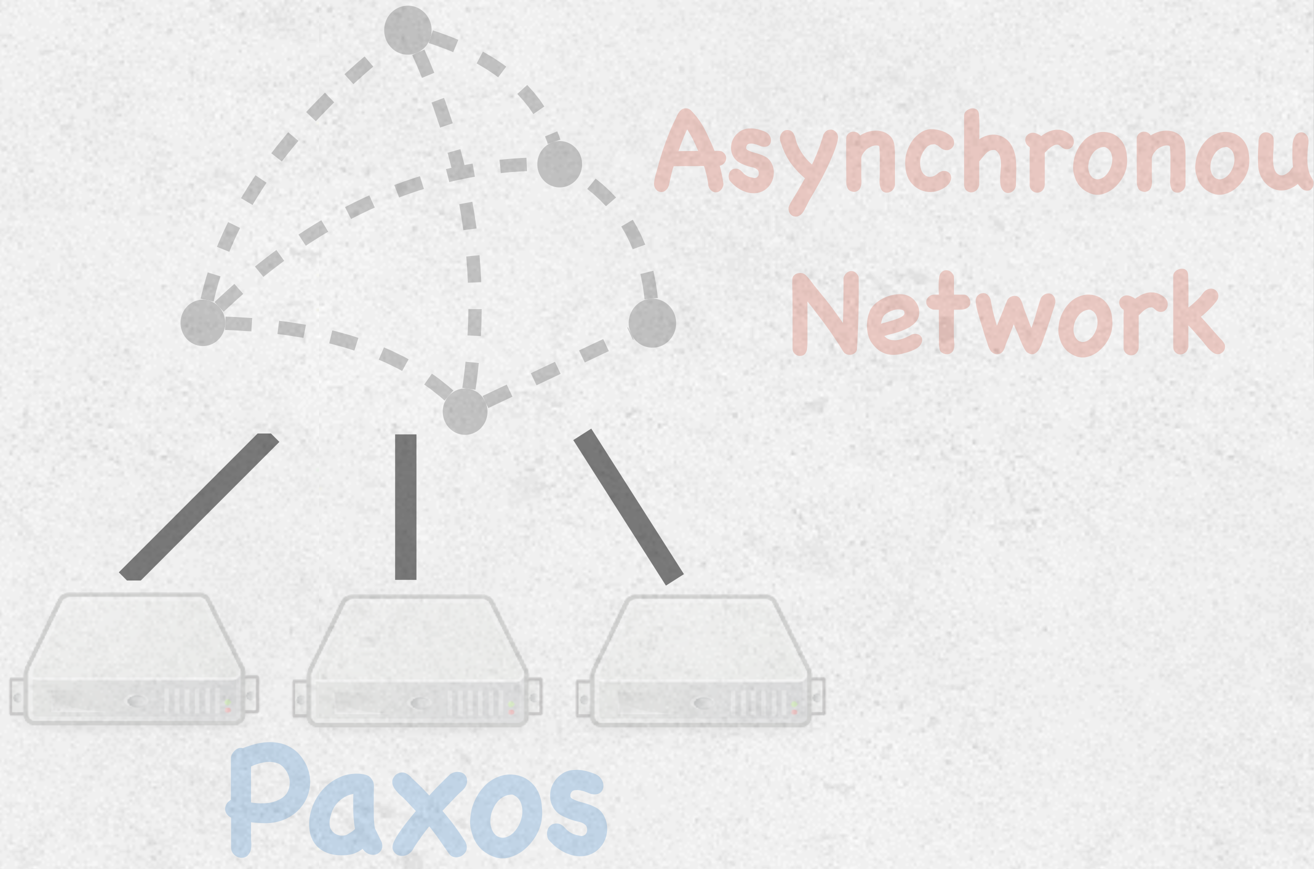


All replicas:
- receive the same set of messages
- receive them in the same order



- Paxos protocol on every operation
- High performance cost

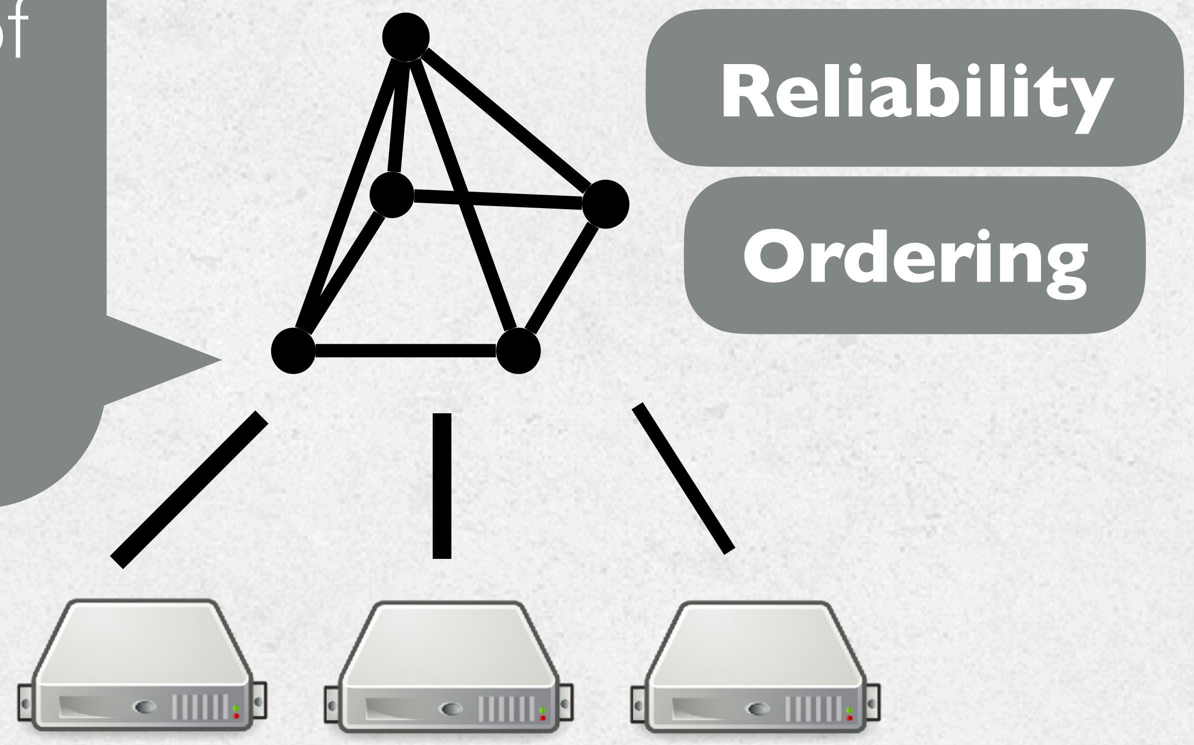
Network properties determine replication complexity



- Paxos protocol on every operation
- High performance cost

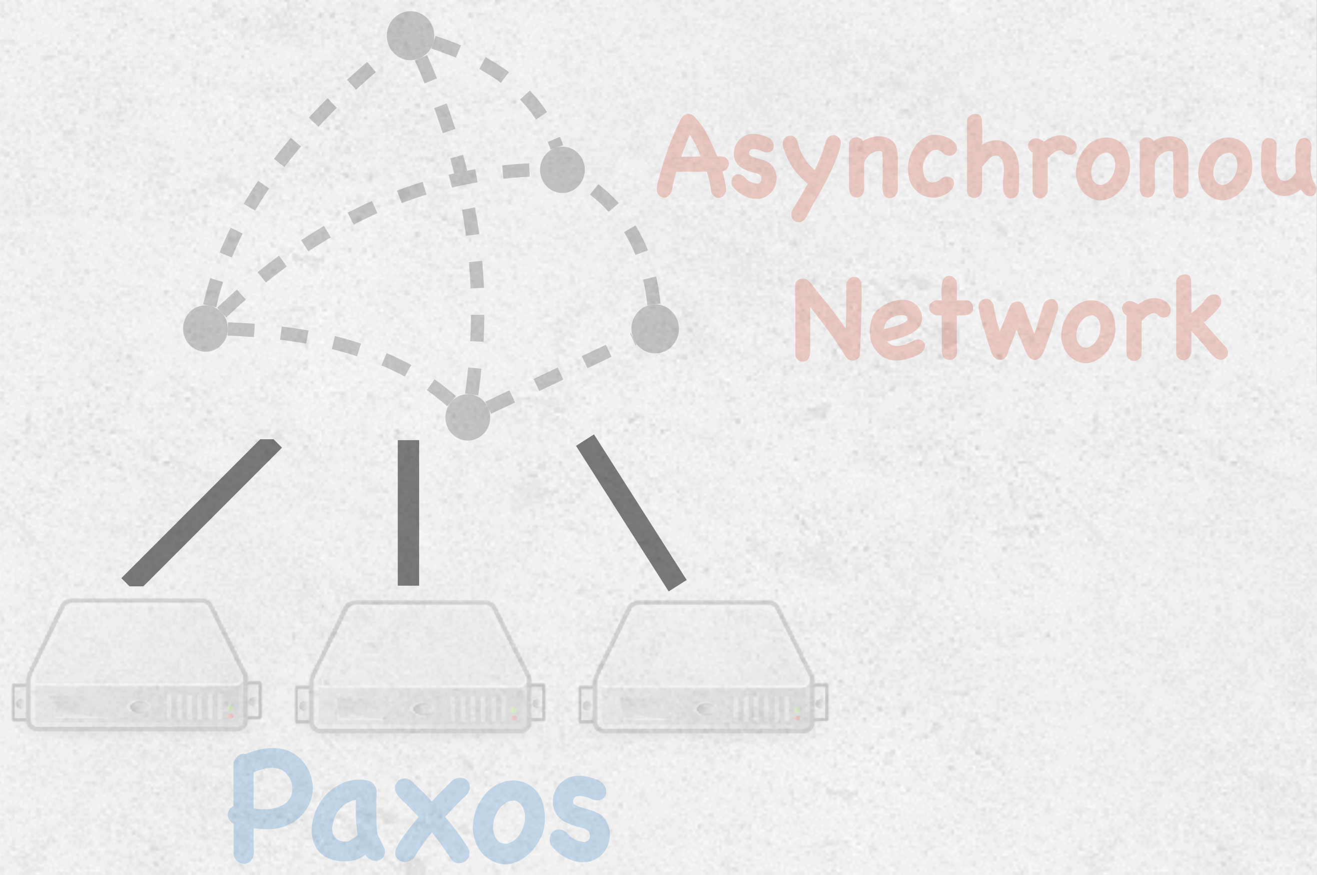
All replicas:

- receive the same set of messages
- receive them in the same order

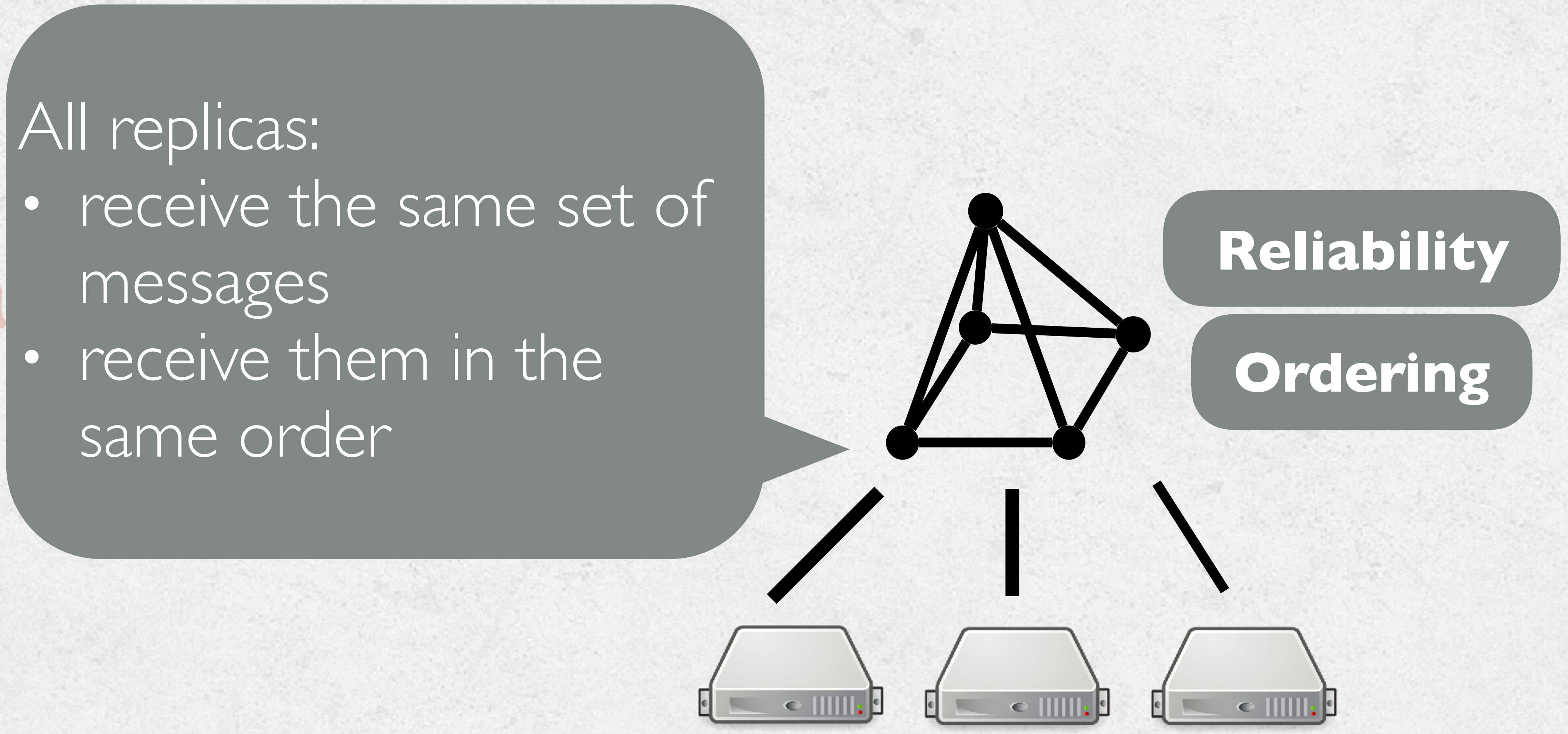


- Replication is trivial

Network properties determine replication complexity

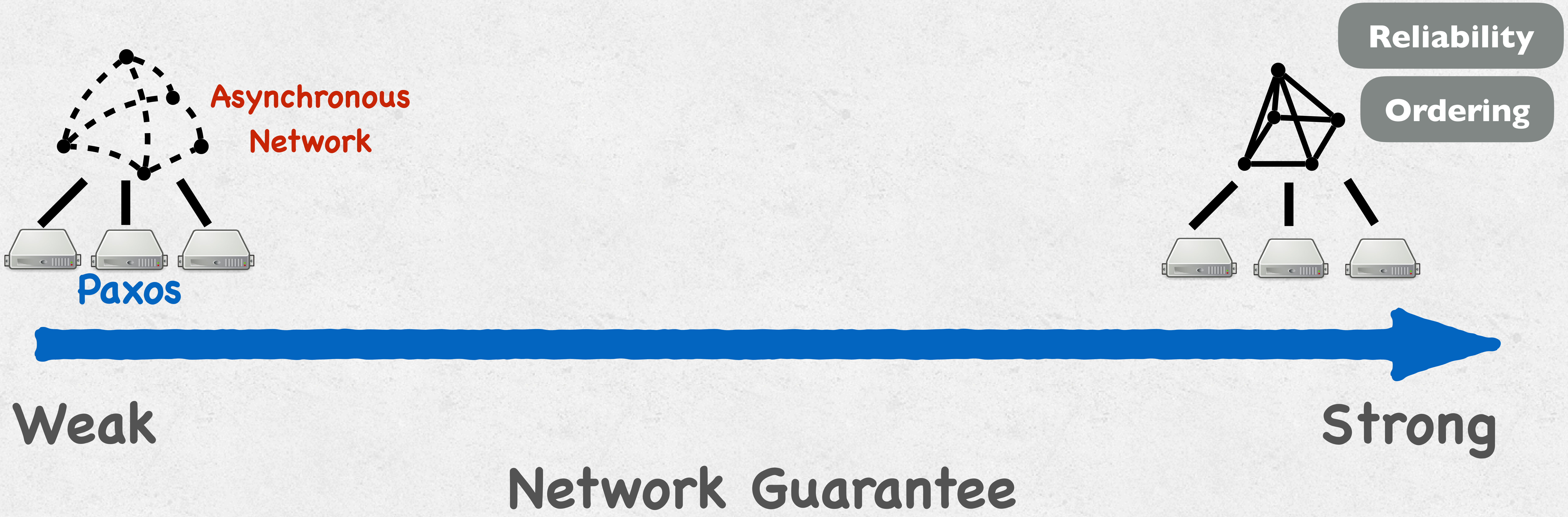


- Paxos protocol on every operation
- High performance cost


All replicas:

- receive the same set of messages
- receive them in the same order

- Replication is trivial
- Network implementation has the same complexity as Paxos



Asynchronous Network

Paxos

Reliability

Ordering

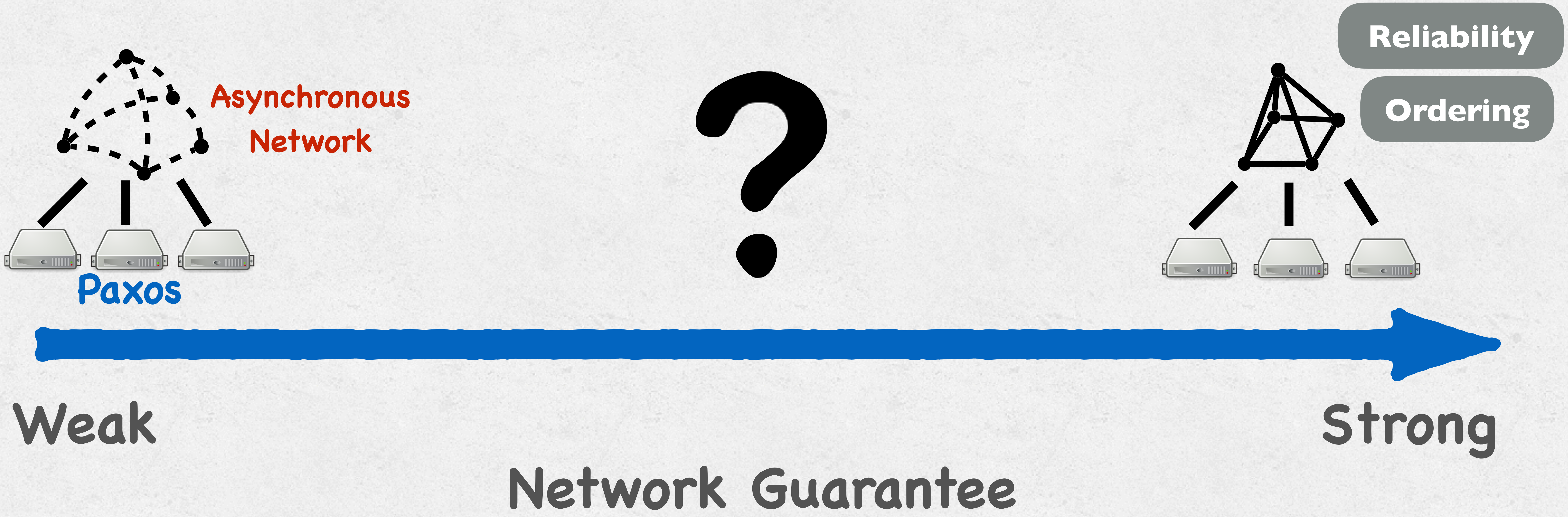
Weak

Strong

Network Guarantee

Can we build a network model that:

- provides **performance benefits**
- can be implemented more **efficiently**



SpecPaxos assumed the network was *mostly ordered*

What if it could provide an *ordering guarantee*?

Towards an ordered but unreliable network

Key Idea: Separate **ordering** from **reliable delivery** in state machine replication

Network provides **ordering**

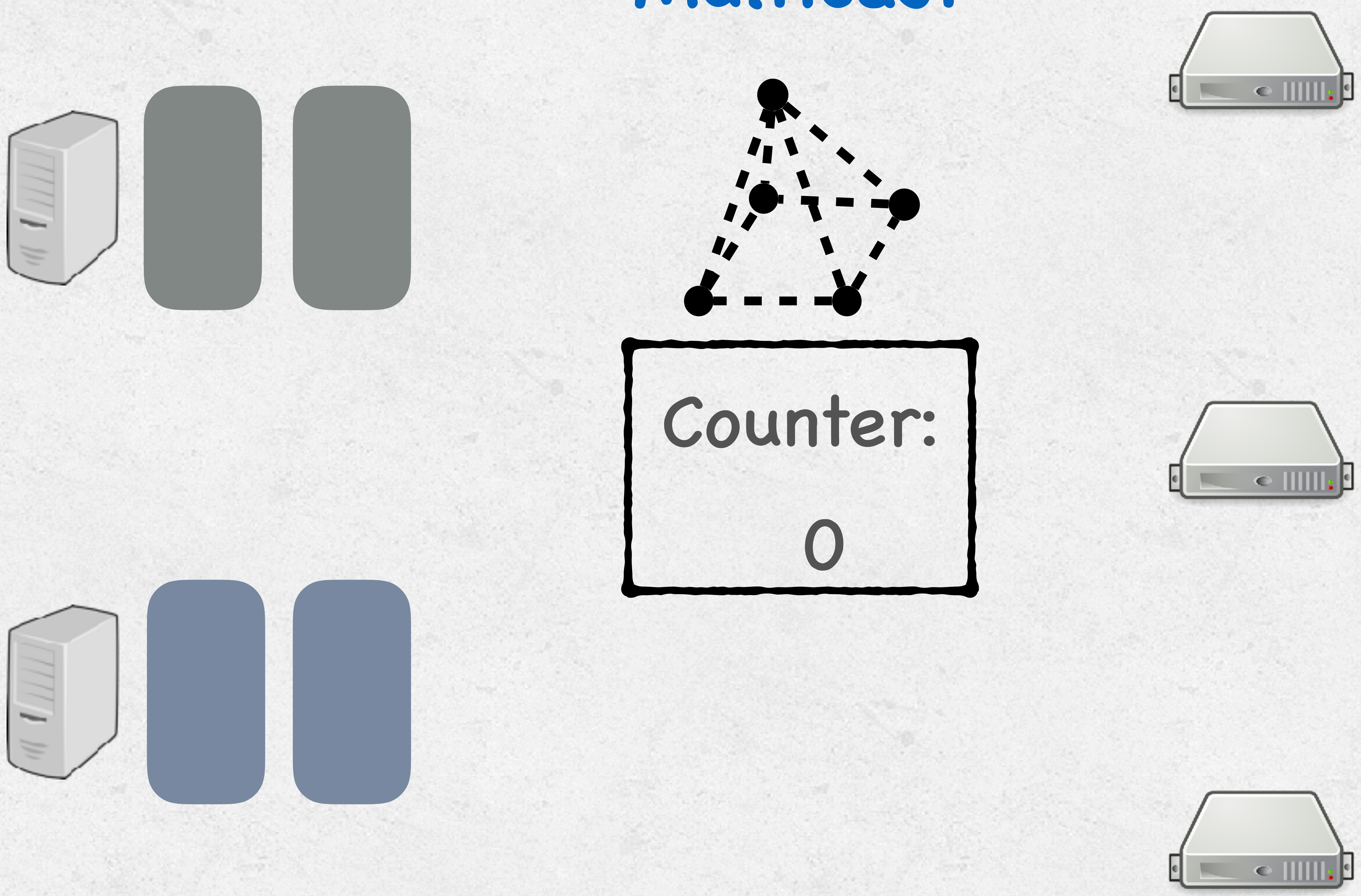
Replication protocol handles **reliability**

OUM Approach

- Designate one **sequencer** in the network
- Sequencer maintains a counter for each OUM group

1. Forward OUM messages to the sequencer
2. Sequencer increments counter and writes counter value into packet headers
3. Receivers use sequence numbers to detect **reordering** and **message drops**

Ordered Unreliable Multicast



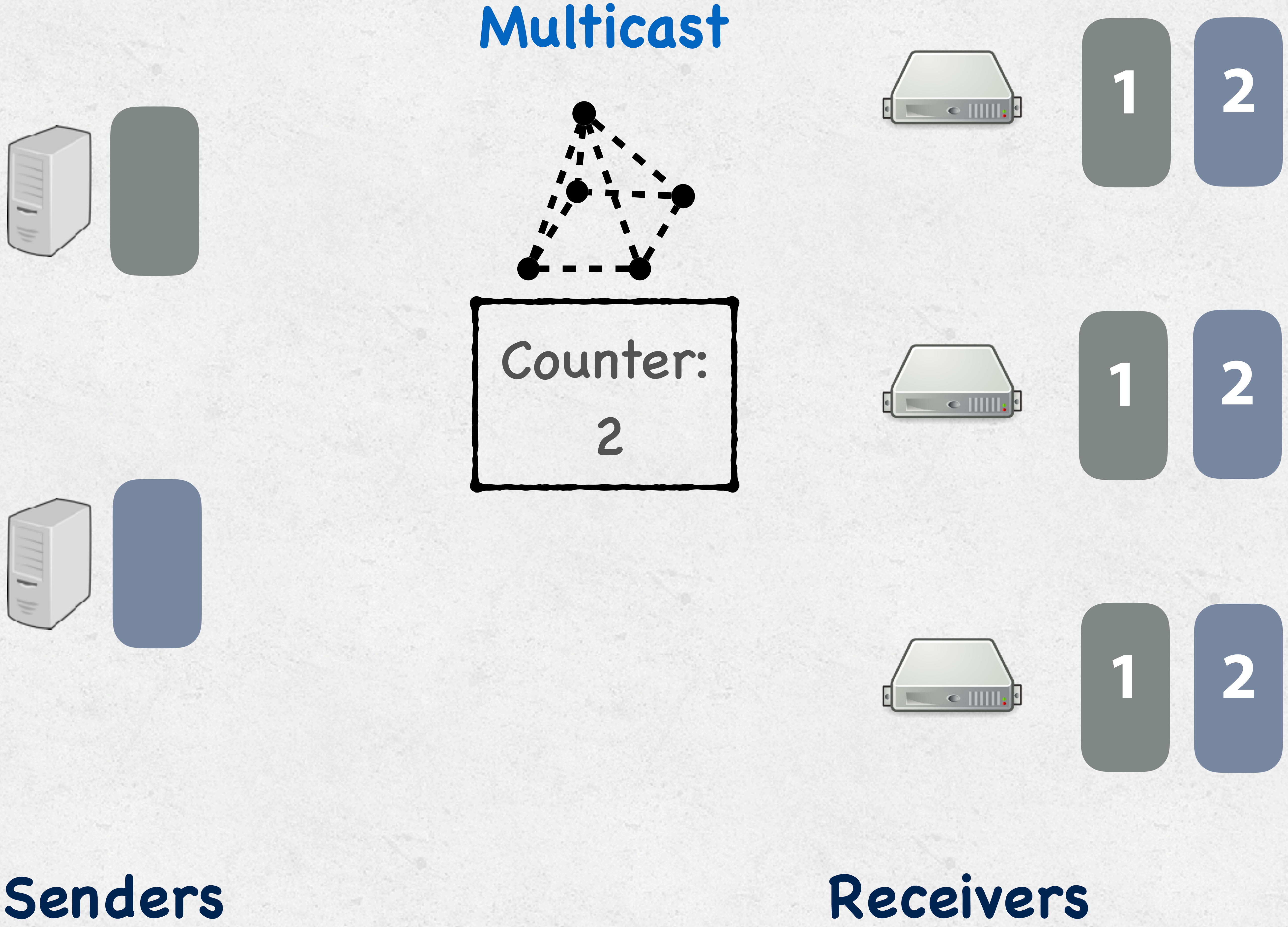
Counter:

0

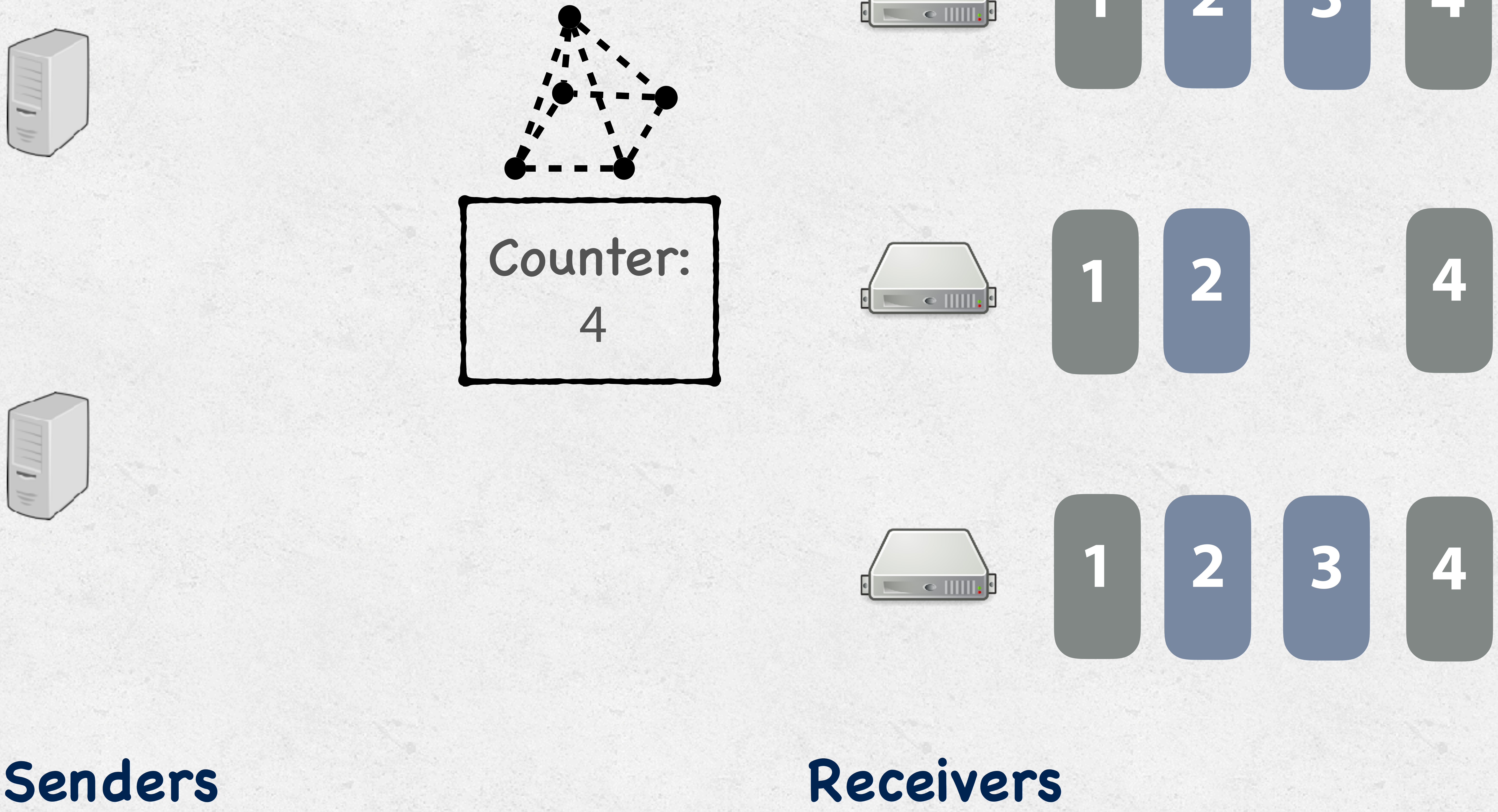
Senders

Receivers

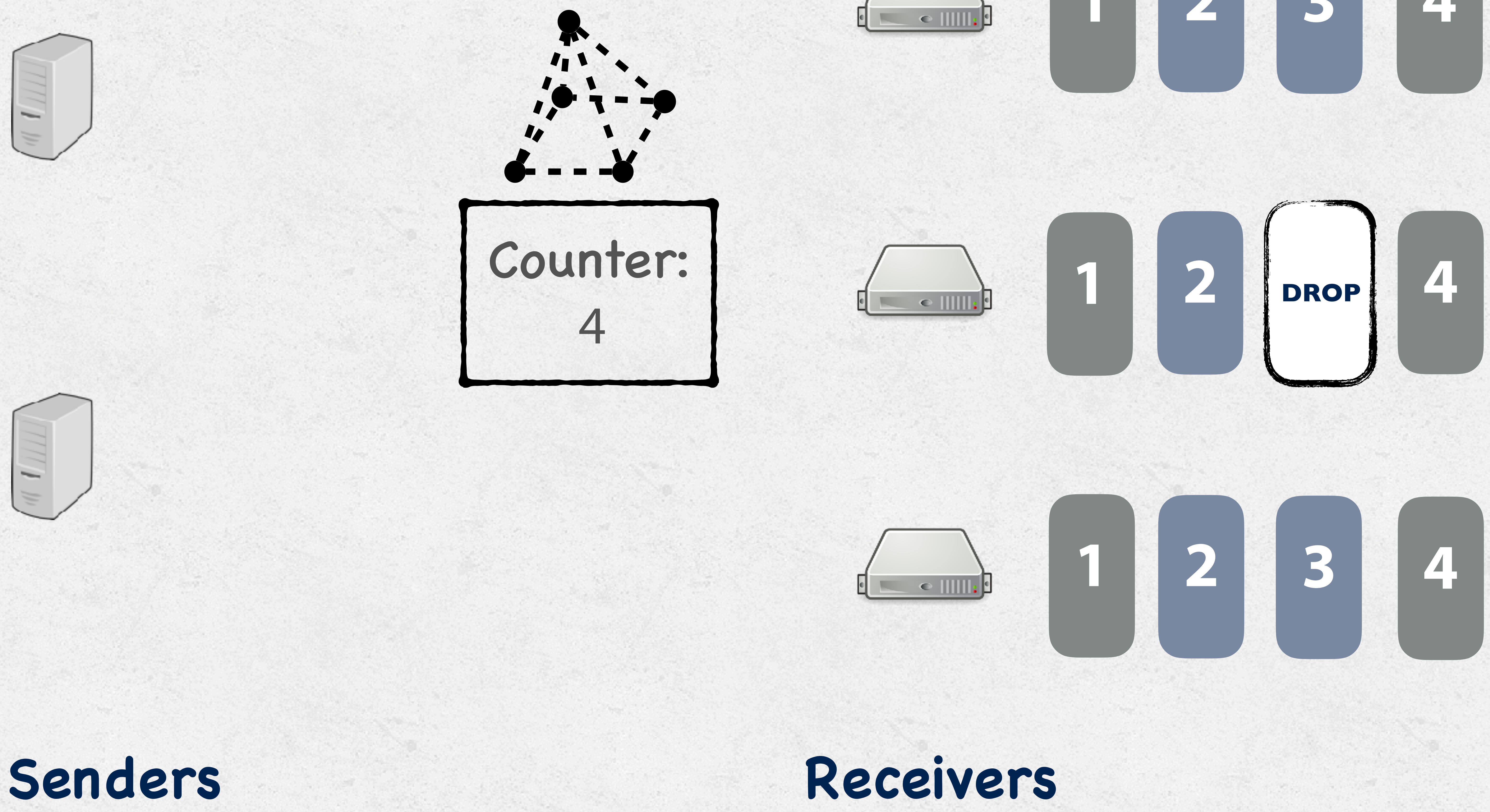
Ordered Unreliable Multicast



Ordered Unreliable Multicast

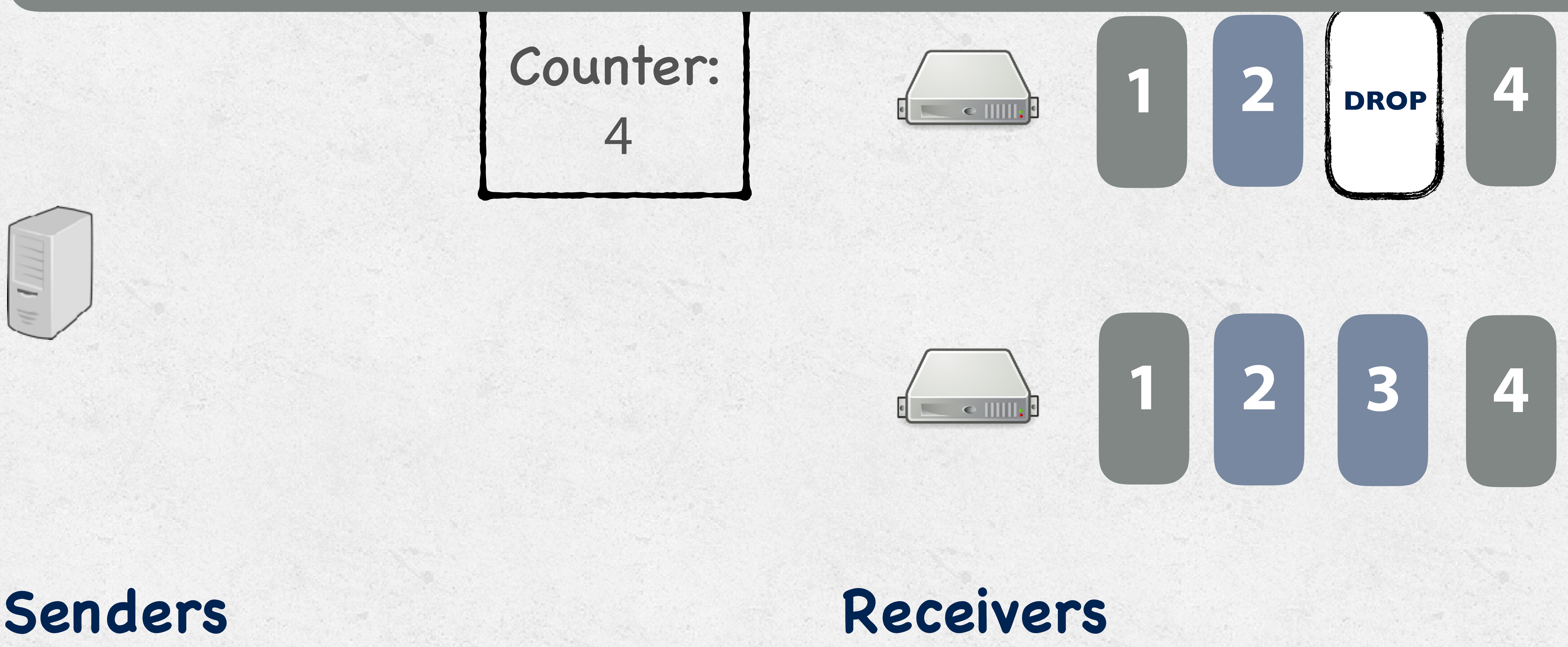


Ordered Unreliable Multicast

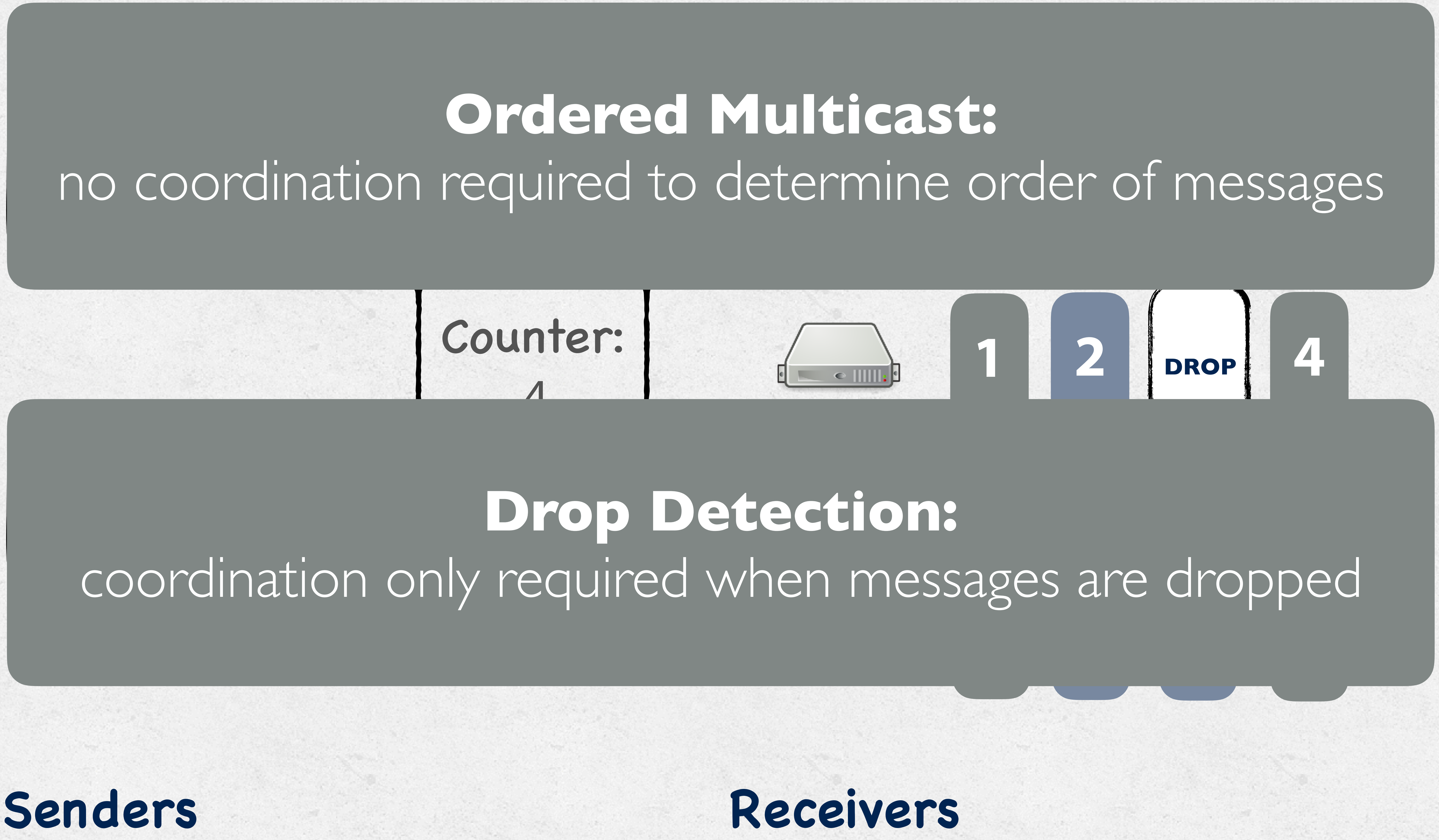


Ordered Unreliable

Ordered Multicast:
no coordination required to determine order of messages



Ordered Unreliable



Ordered Multicast:
no coordination required to determine order of messages

Drop Detection:
coordination only required when messages are dropped

Sequencer Implementations

In-switch sequencing

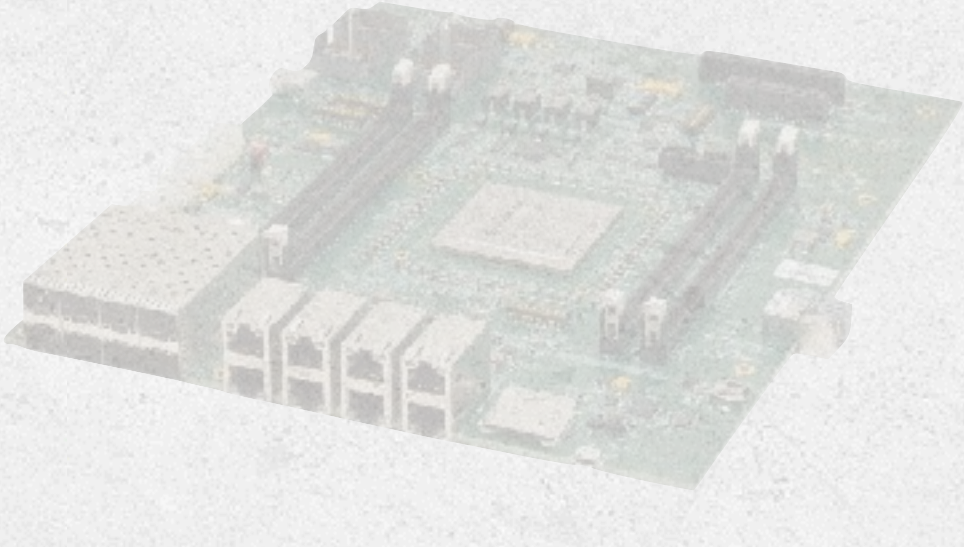
- next generation programmable switches
- implemented in P4
- nearly **zero cost**

Middlebox prototype

- Cavium Octeon network processor
- connects to root switches
- adds 8 us latency

End-host sequencing

- no specialized hardware required
- incurs higher latency penalties
- similar throughput benefits



Sequencer Implementations

In-switch sequencing

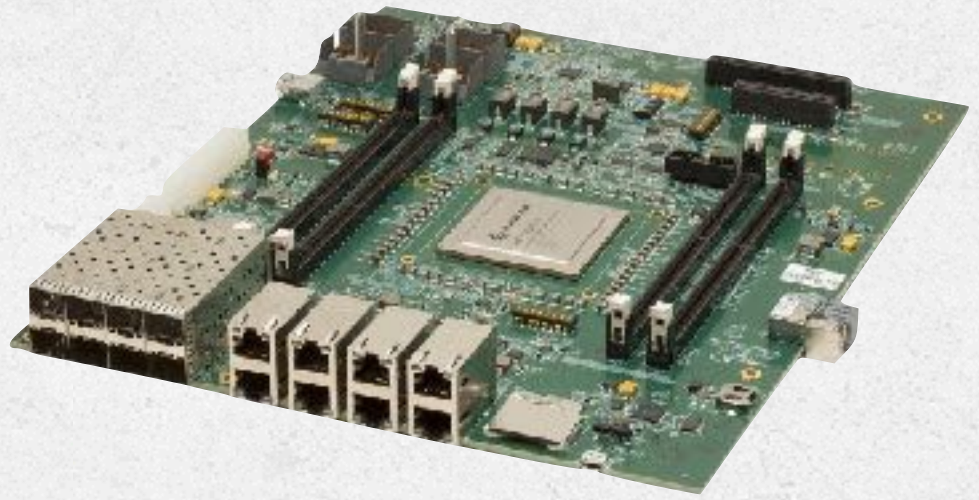
- next generation programmable switches
- implemented in P4
- nearly **zero cost**

Middlebox prototype

- Cavium Octeon network processor
- connects to root switches
- adds 8 us latency

End-host sequencing

- no specialized hardware required
- incurs higher latency penalties
- similar throughput benefits



Sequencer Implementations

In-switch sequencing

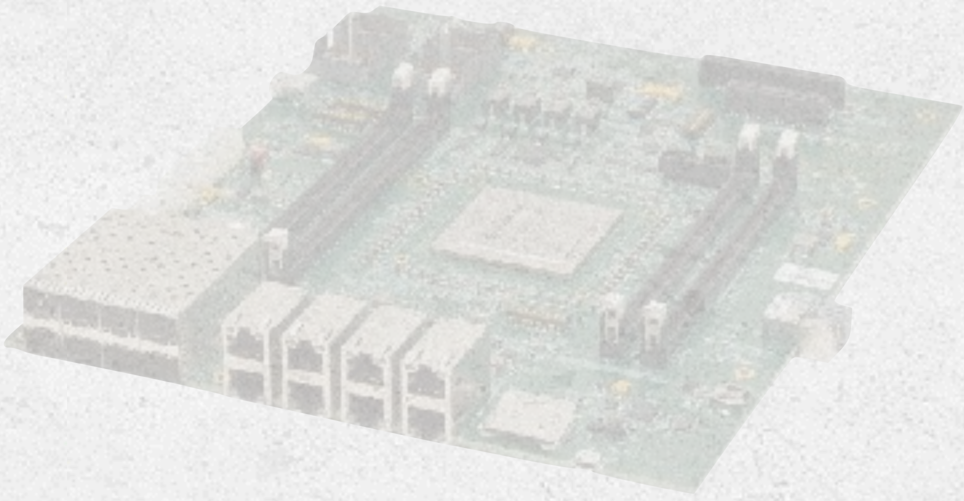
- next generation programmable switches
- implemented in P4
- nearly **zero cost**

Middlebox prototype

- Cavium Octeon network processor
- connects to root switches
- adds 8 us latency

End-host sequencing

- no specialized hardware required
- incurs higher latency penalties
- similar throughput benefits



NOPaxos Overview

- Built on top of the guarantees of OUM
- Client requests are **totally ordered** but can be dropped
- **No coordination** in the common case
- Replicas run agreement on drop detection
- View change protocol for leader or sequencer failure

Normal Operation

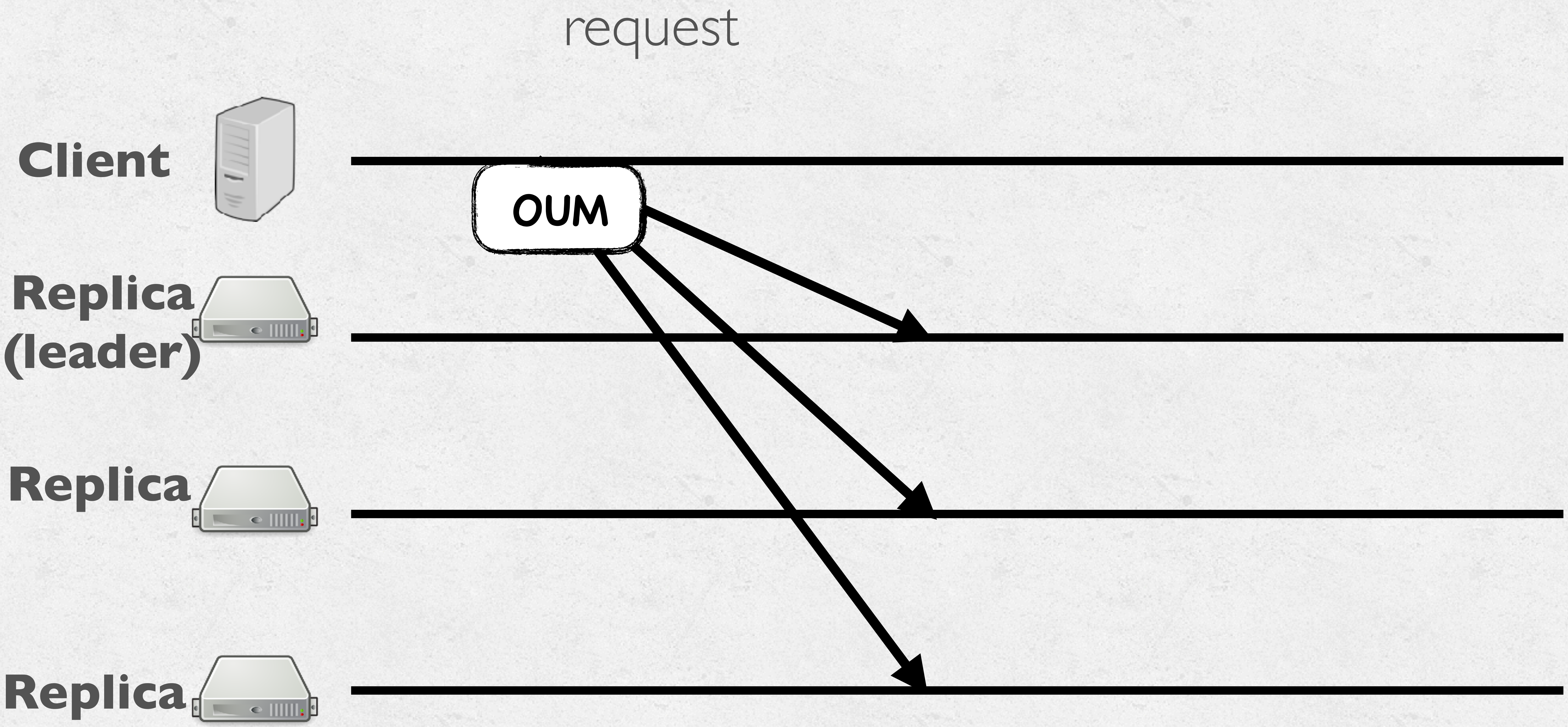
Client

Replica (leader)

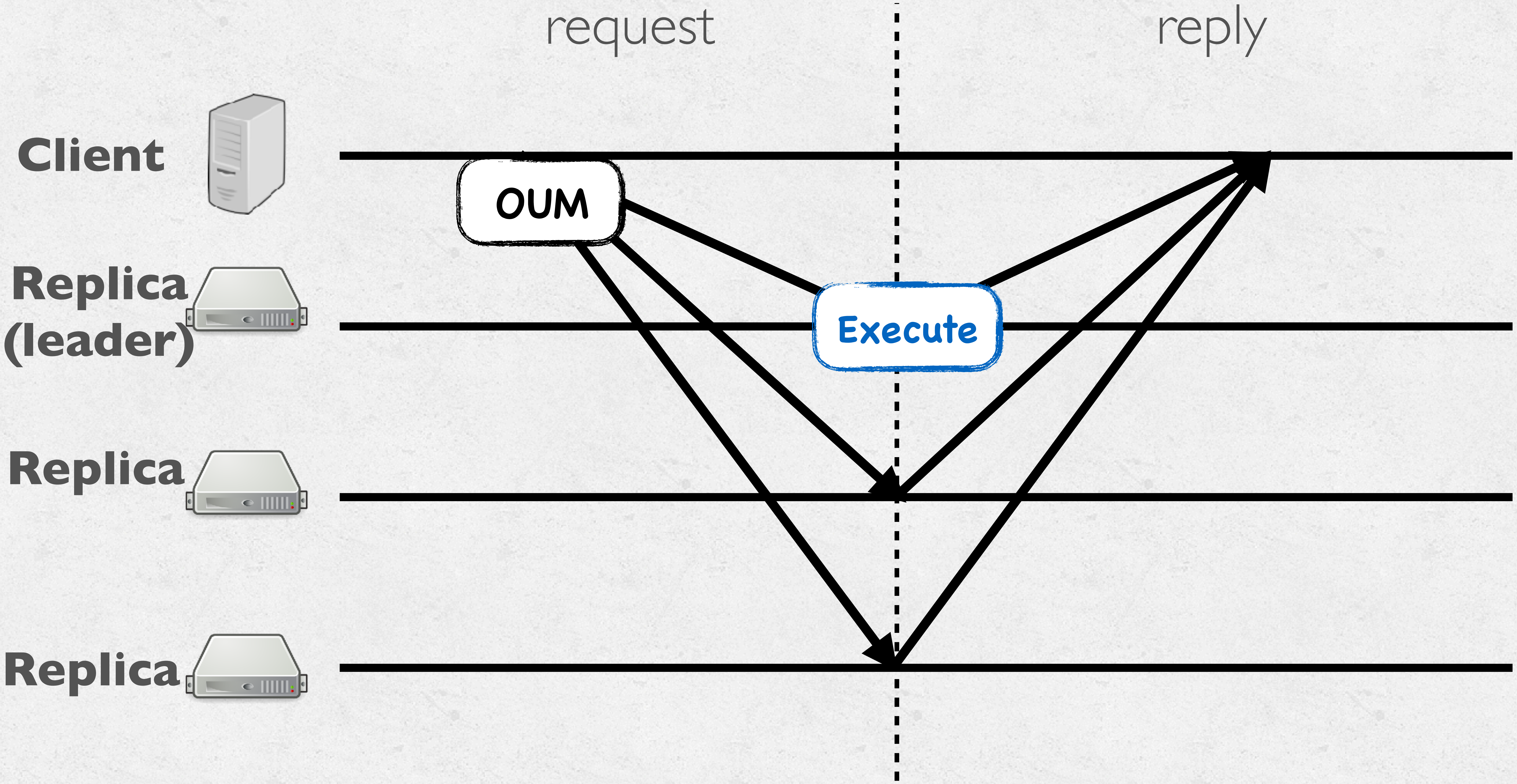
Replica

Replica

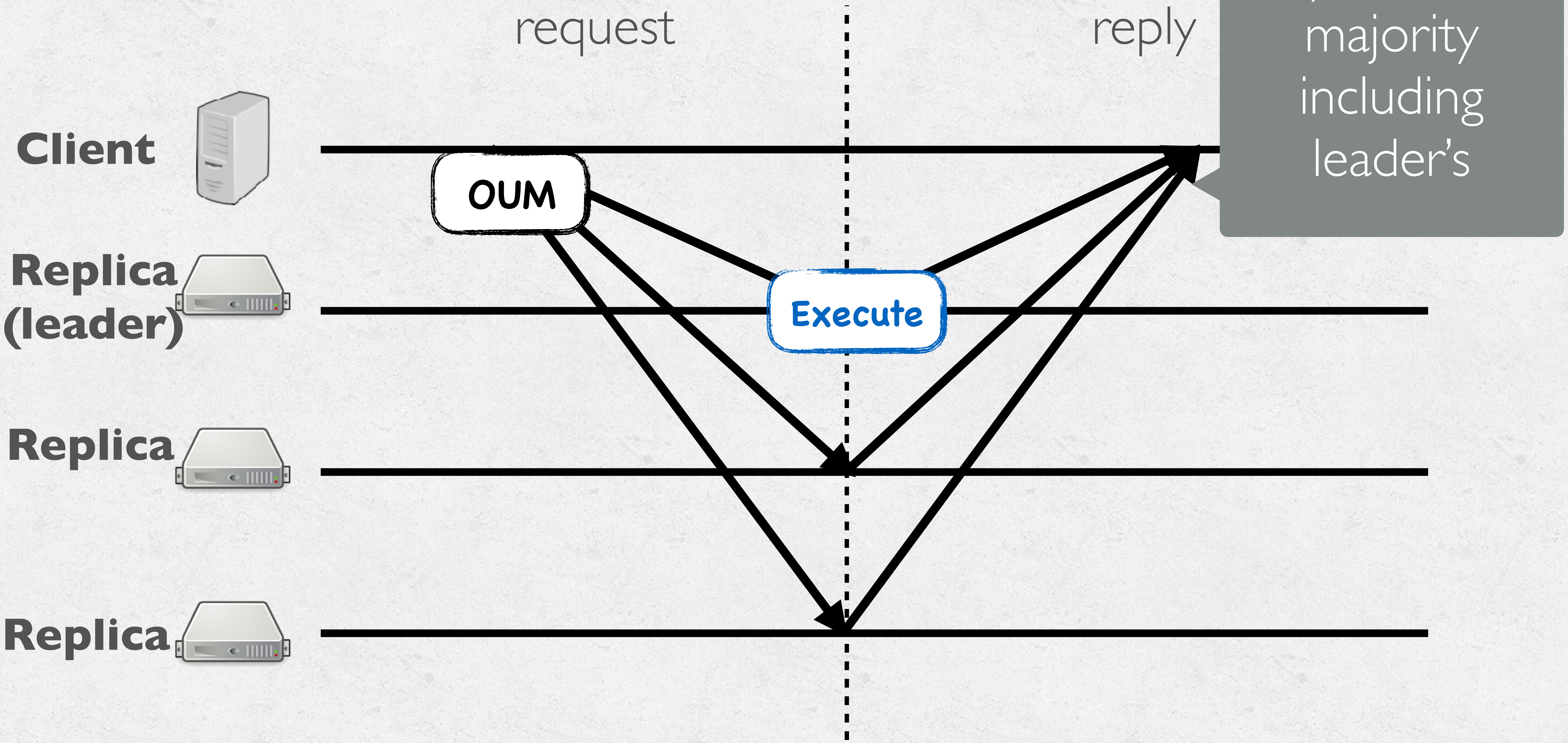
Normal Operation



Normal Operation

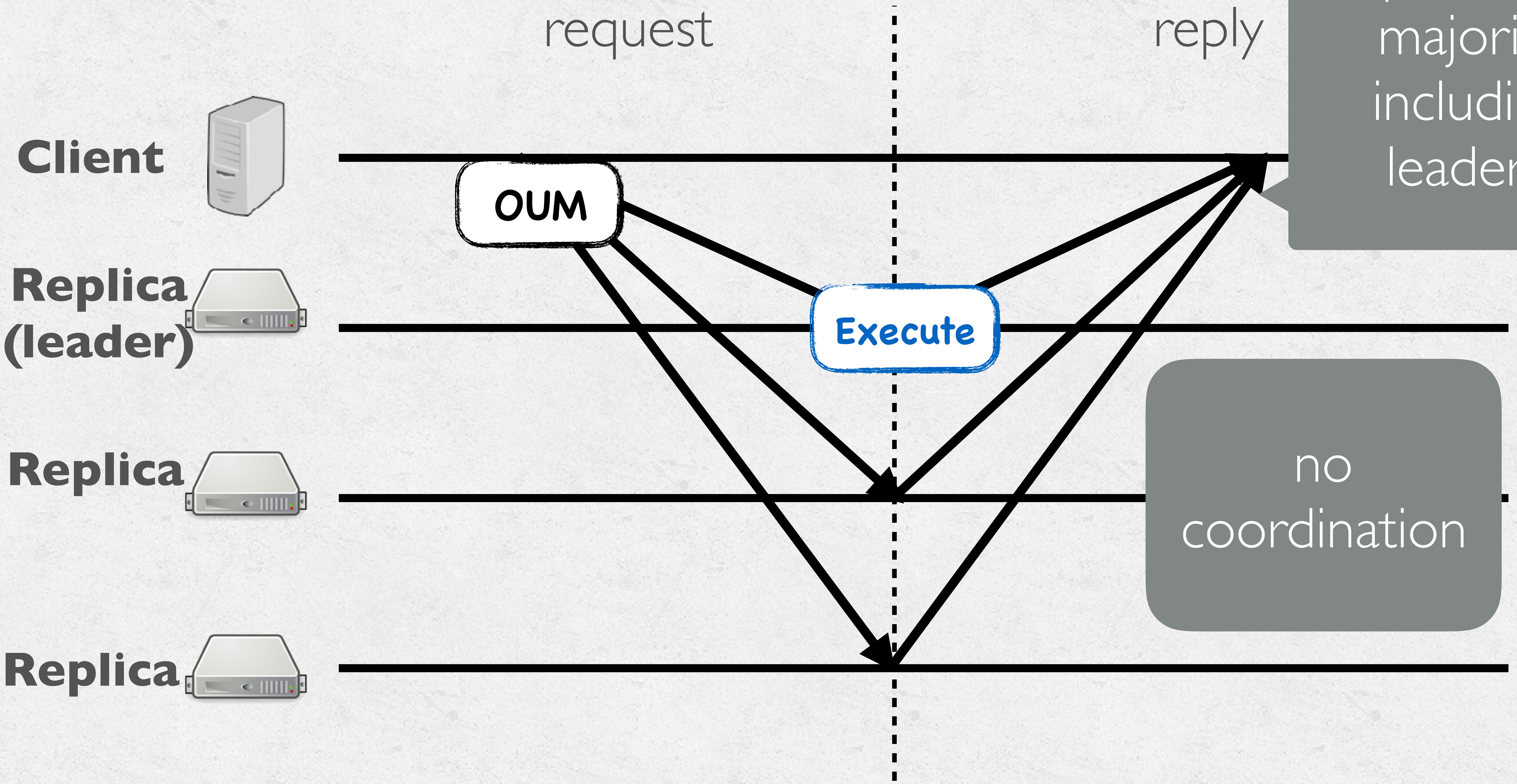


Normal Operation



waits for replies from majority including leader’s

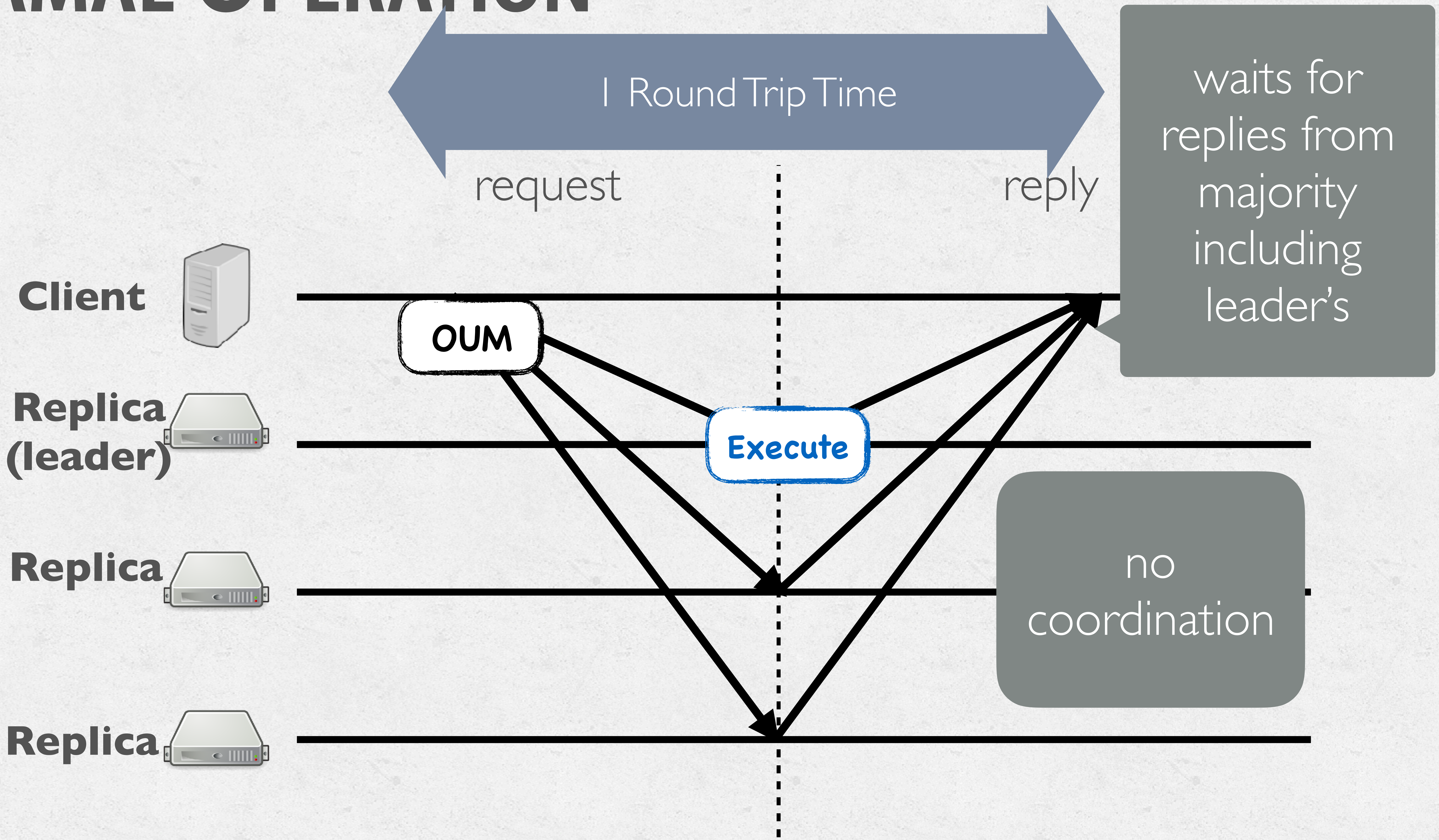
Normal Operation



waits for replies from majority including leader's

no coordination

Normal Operation



waits for replies from majority including leader’s

no coordination

Gap Agreement

Replicas detect 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

Why Do Followers Not Execute?

- Request logs in NOPaxos are **non-authoritative**. The followers might not be involved in the quorum to commit a no-op. The leader might get replaced.

- Followers simply log operations. Operations are permanently committed with periodic **synchronization**.

- If a leader gets replaced and discovers that some of its commands weren't actually committed, it can roll-back or get a state transfer.

View Change

- Handles leader or sequencer failure
- Ensures that all replicas are in a **consistent state** and agree on all of the commands and no-ops committed in the previous view.
- Runs a view change protocol similar to VR
- *view-number* is a tuple of *<leader-number, session-number>*

NOPAXOS ACHIEVES BETTER THROUGHPUT AND LATENCY

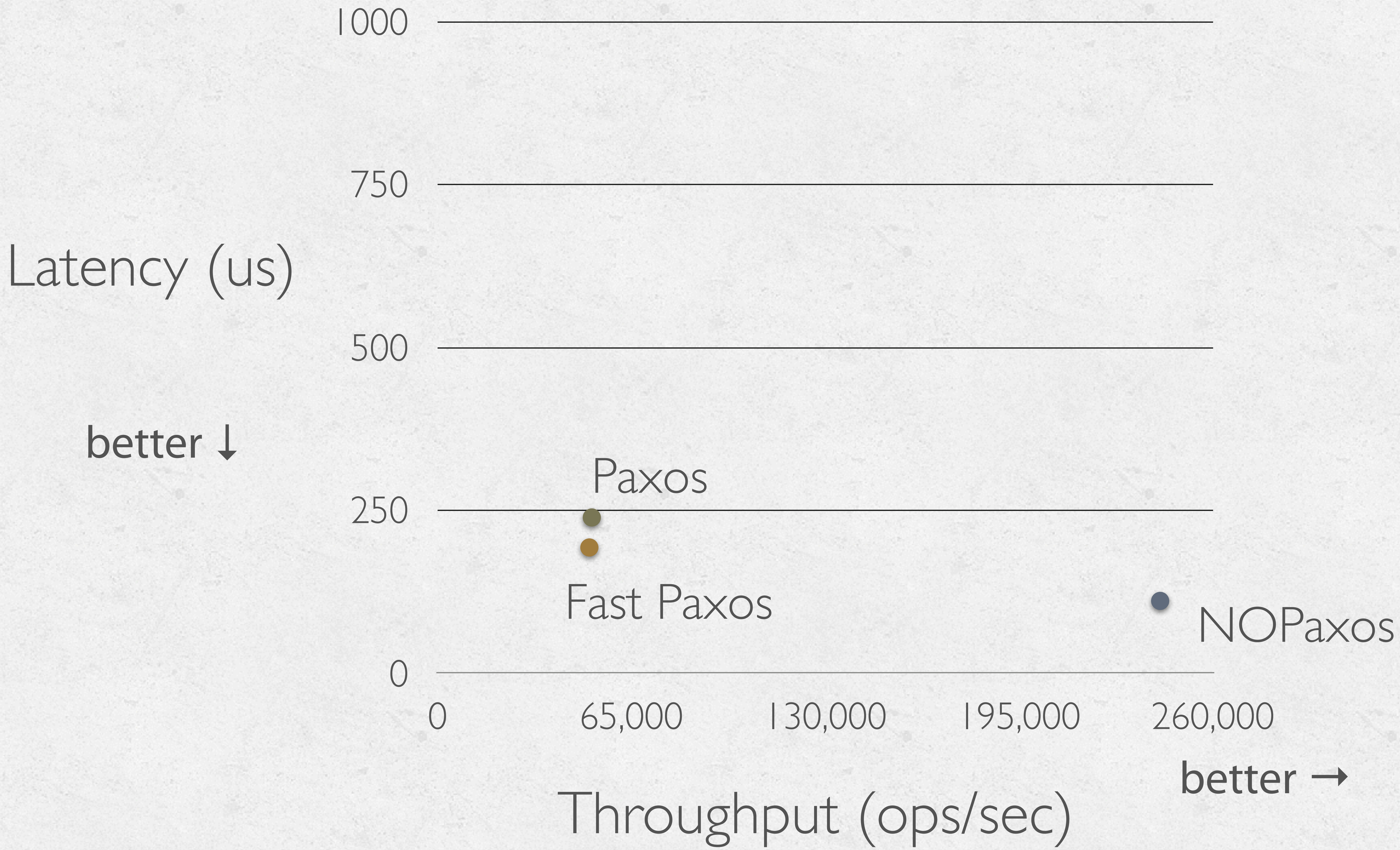
Latency (us)

better ↓

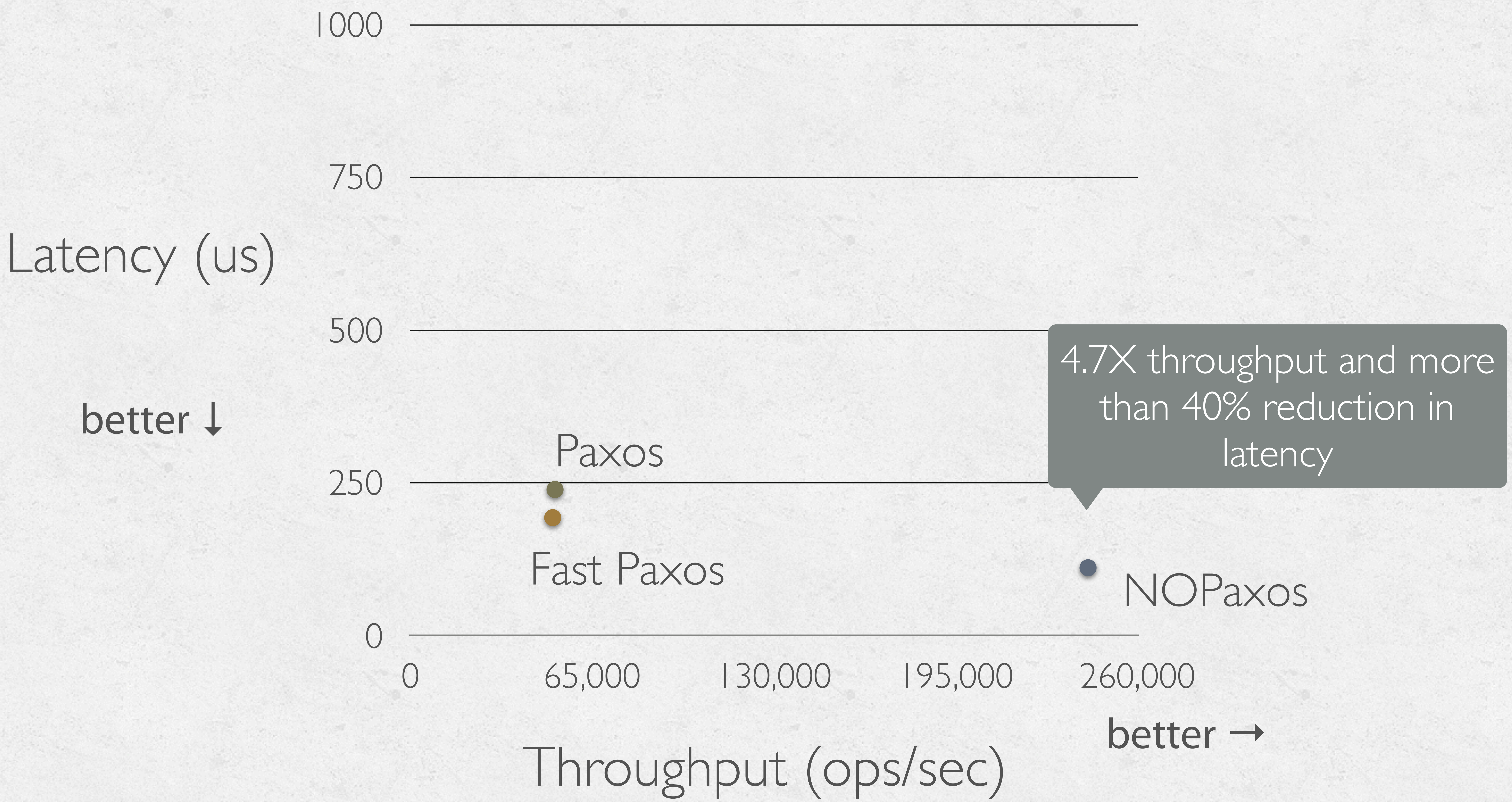
better →

Throughput (ops/sec)

NOPaxos achieves better throughput and latency

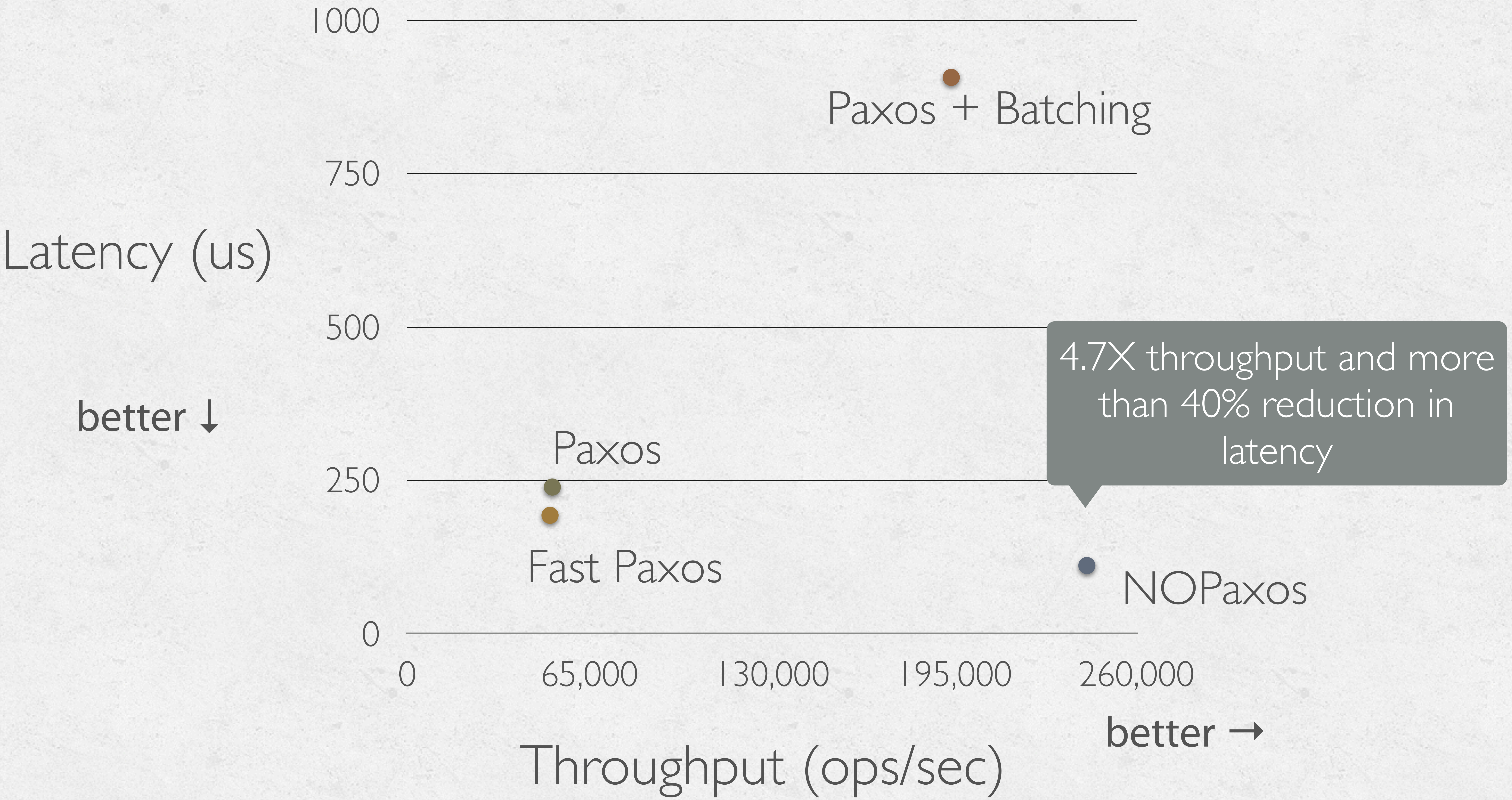


NOPaxos achieves better throughput and latency



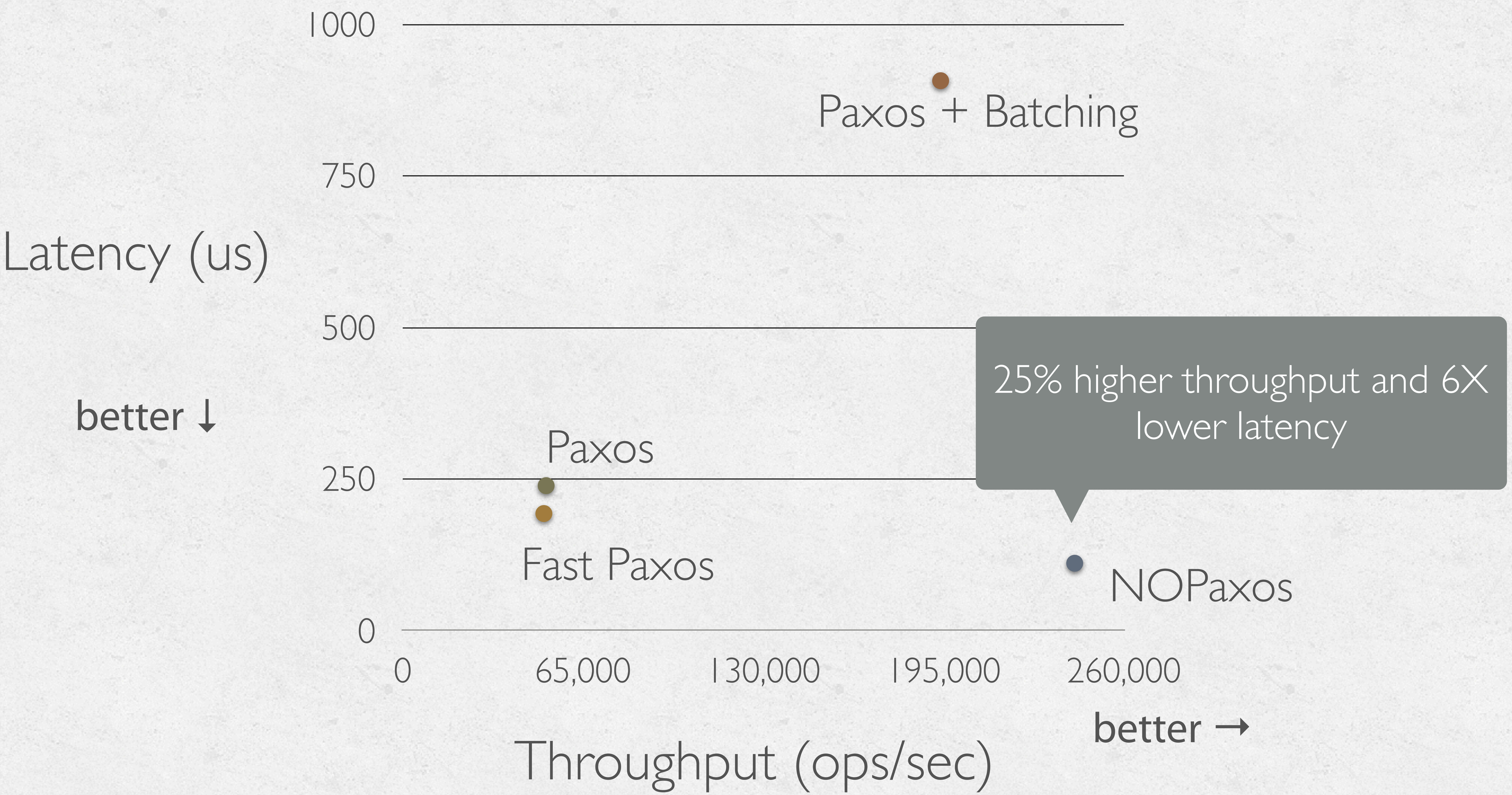
4.7X throughput and more than 40% reduction in latency

NOPaxos achieves better throughput and latency



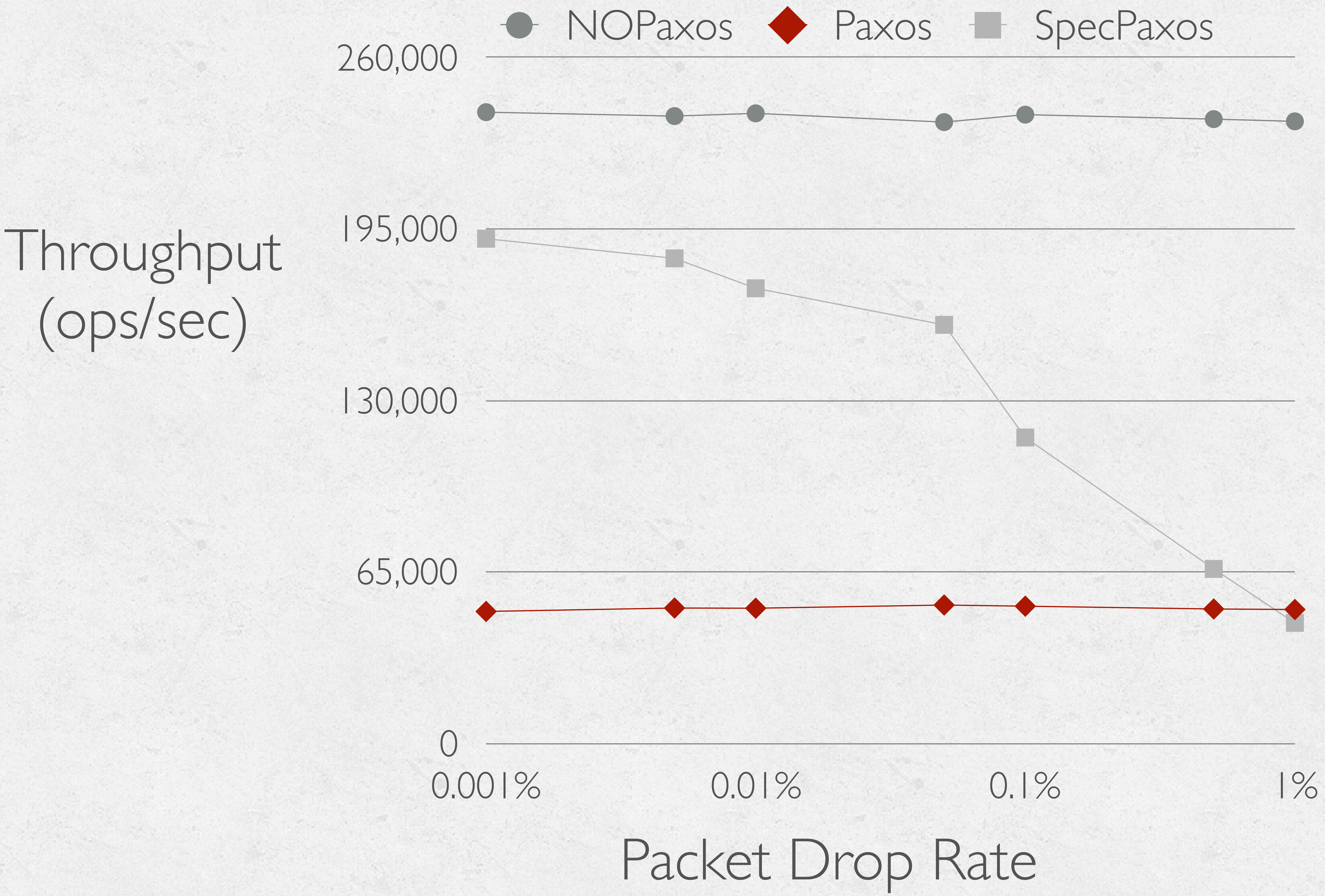
4.7X throughput and more than 40% reduction in latency

NOPaxos achieves better throughput and latency

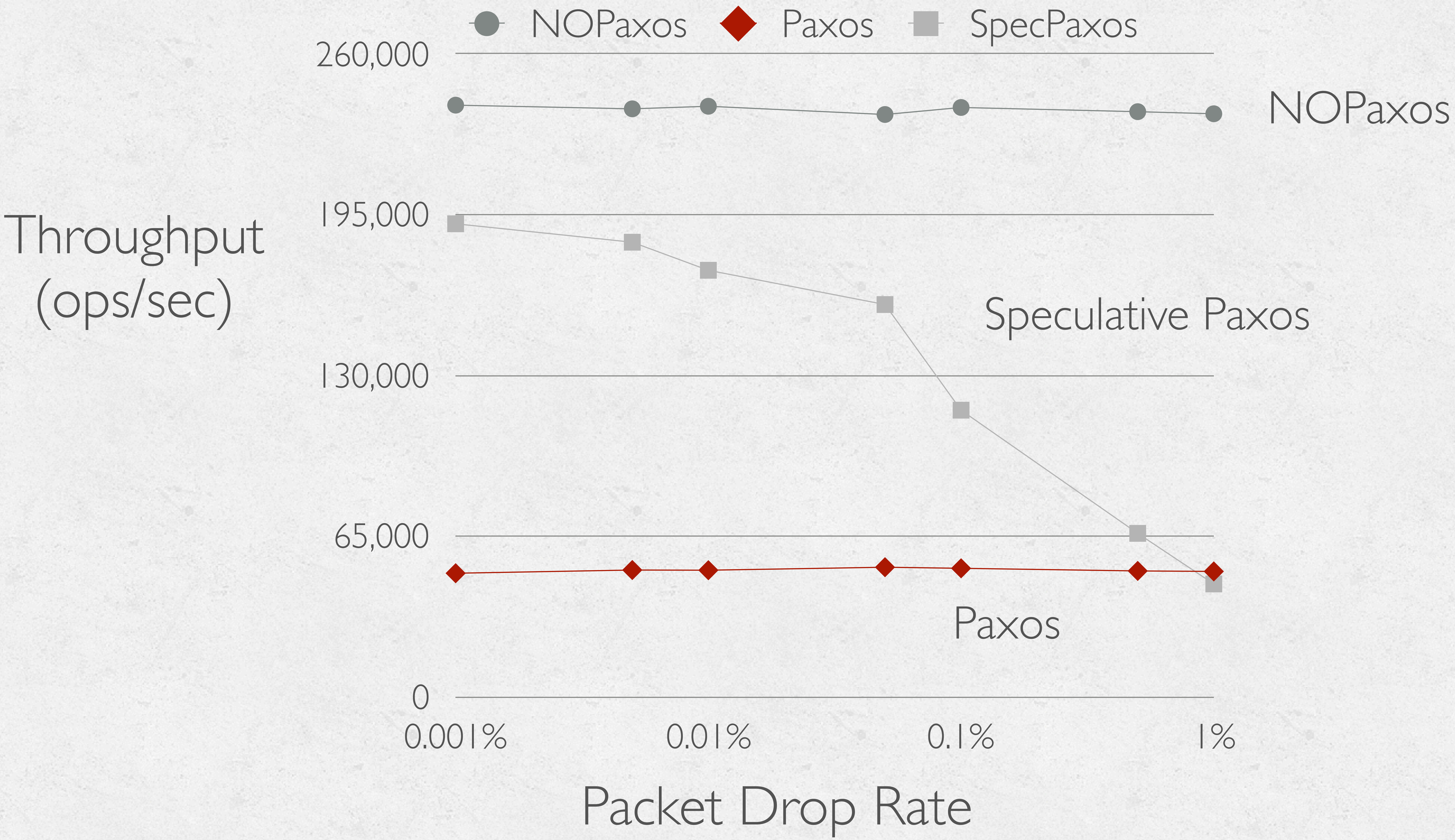


25% higher throughput and 6X lower latency

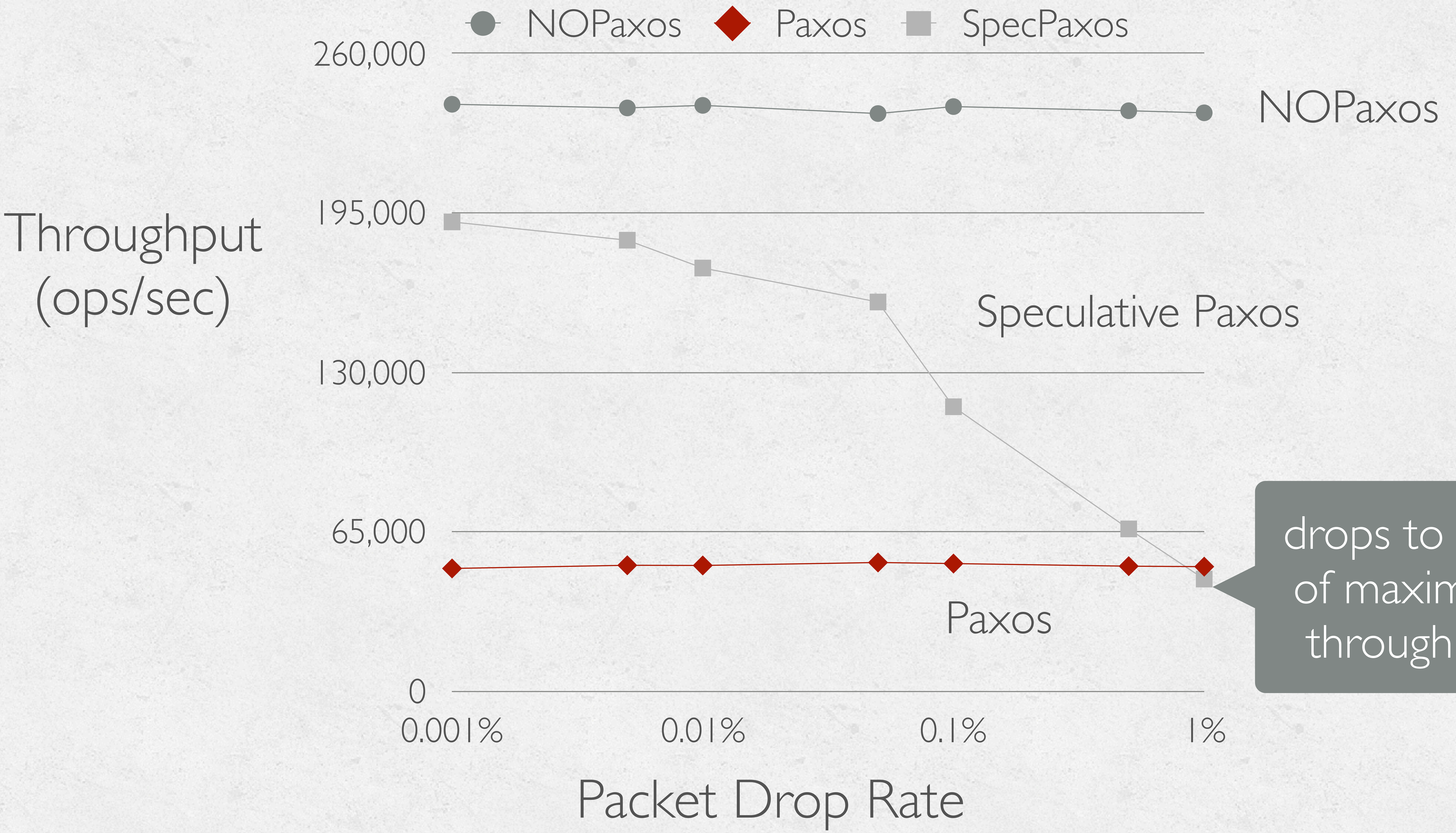
NOPaxos is resilient to network anomalies



NOPAXOS IS RESILIENT TO NETWORK ANOMALIES



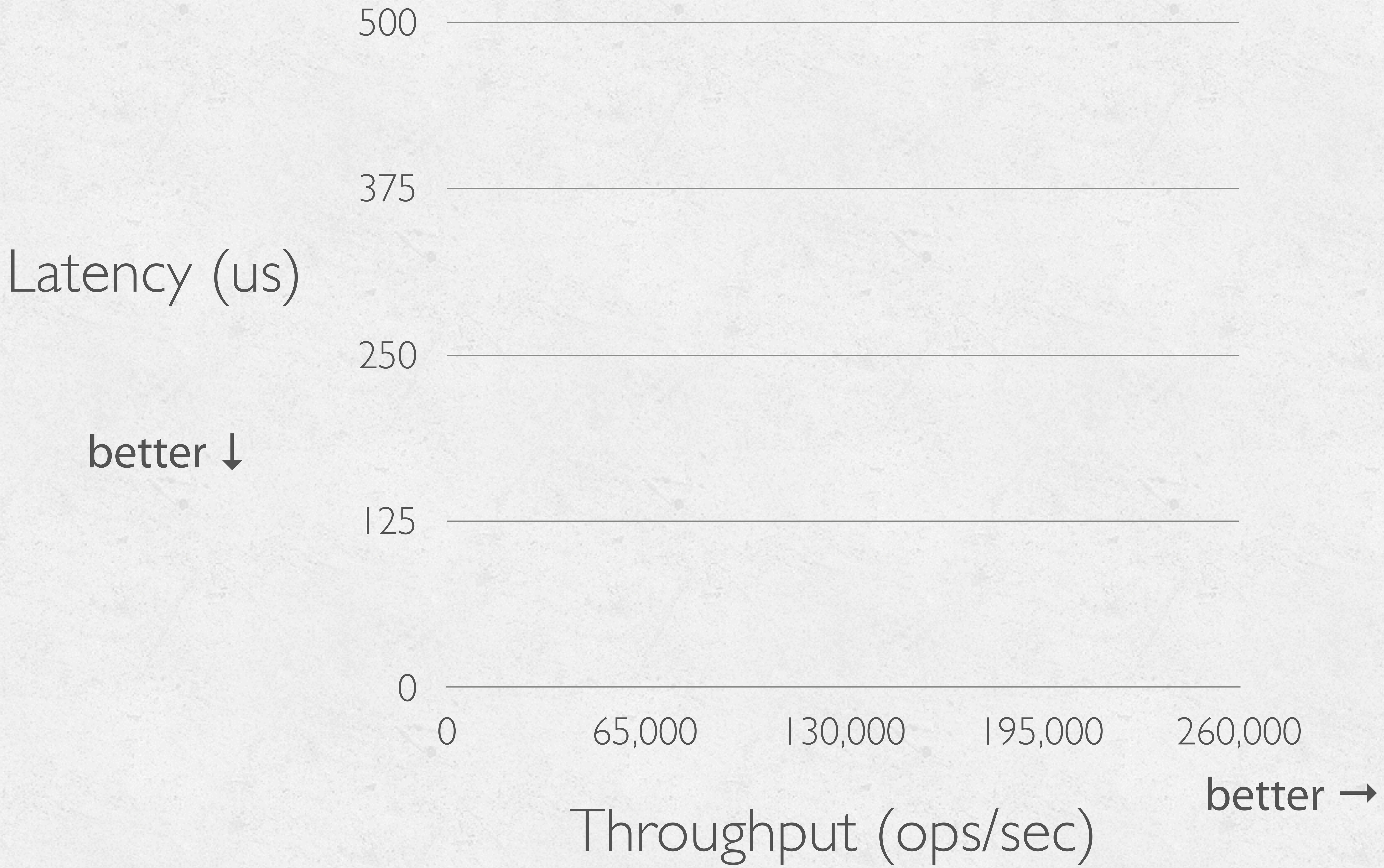
NOPaxos is resilient to network anomalies



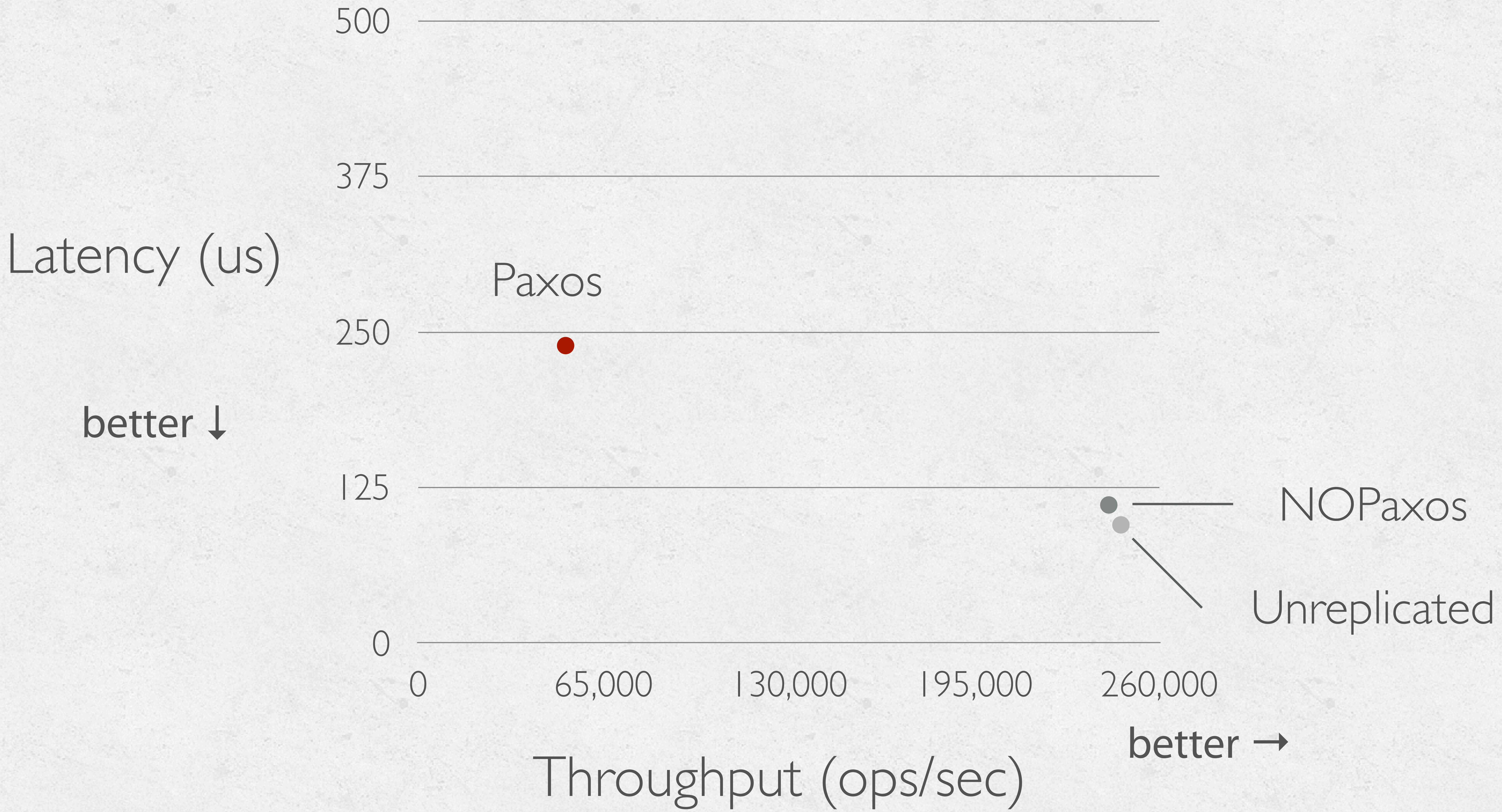
drops to 24% of maximum throughput

NOPaxos attains throughput within 2% of an unreplicated system

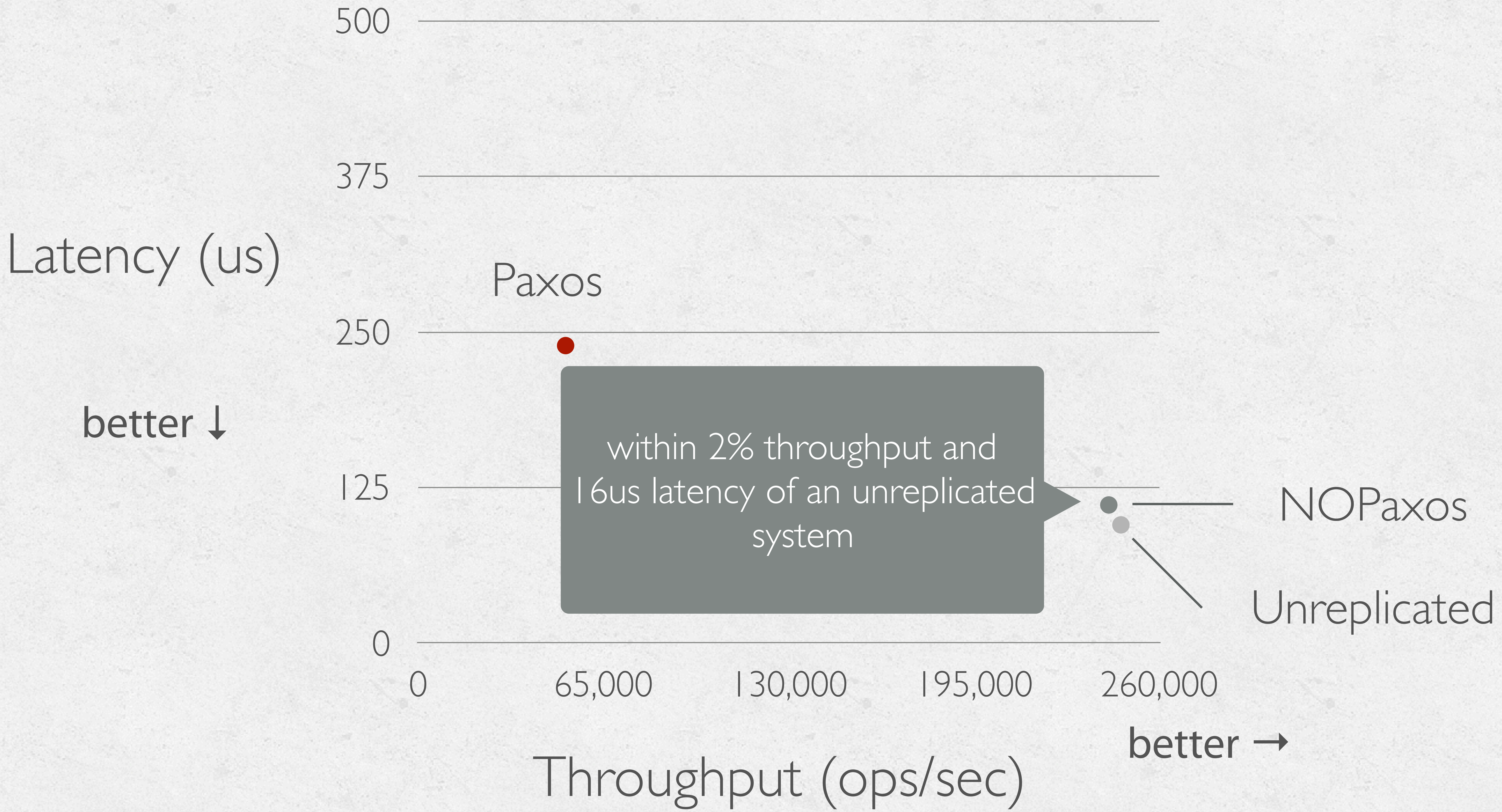
NOPAXOS ATTAINS THROUGHPUT WITHIN 2% OF AN UNREPLICATED SYSTEM



NOPaxos attains throughput within 2% of an unreplicated system

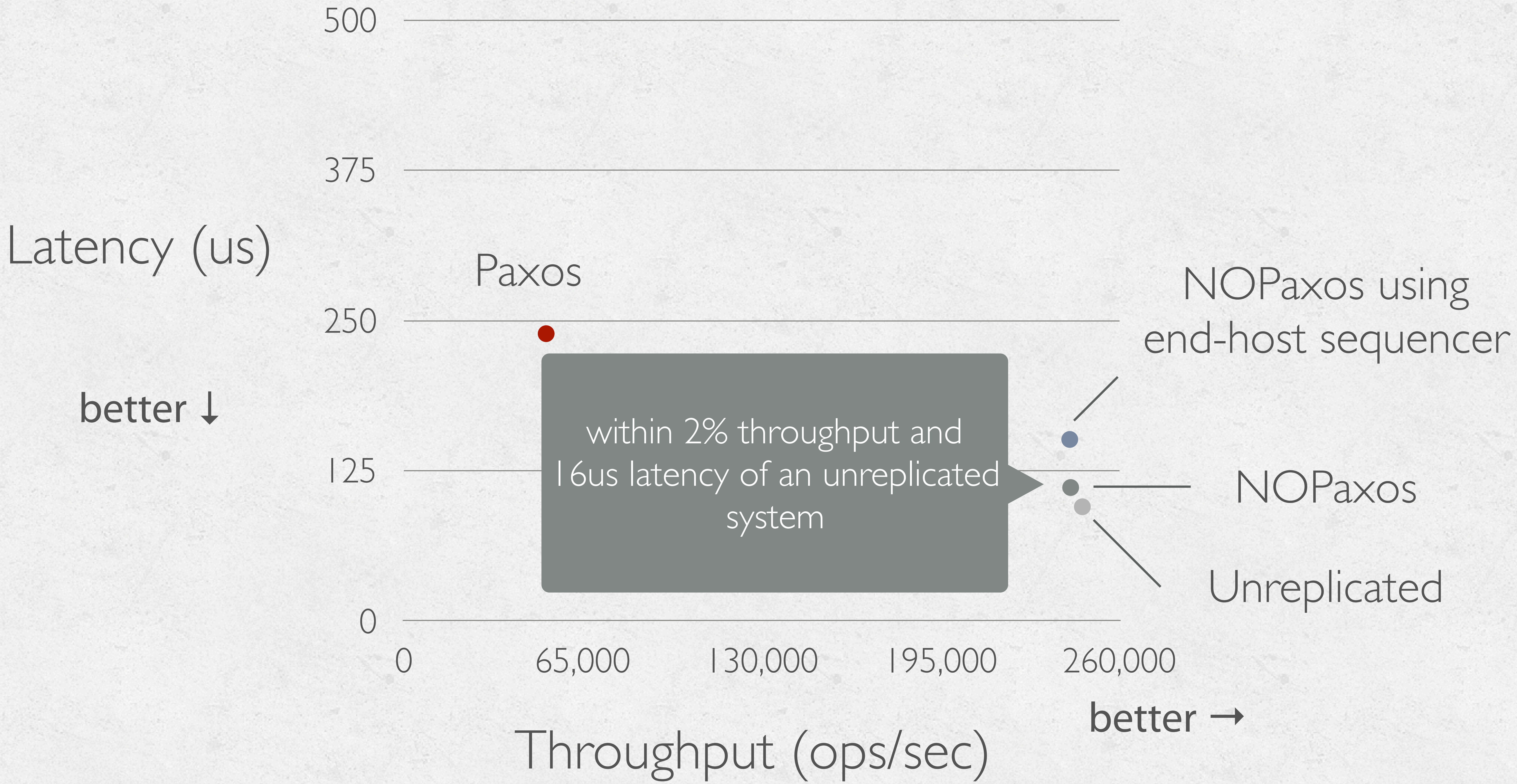


NOPaxos attains throughput within 2% of an unreplicated system



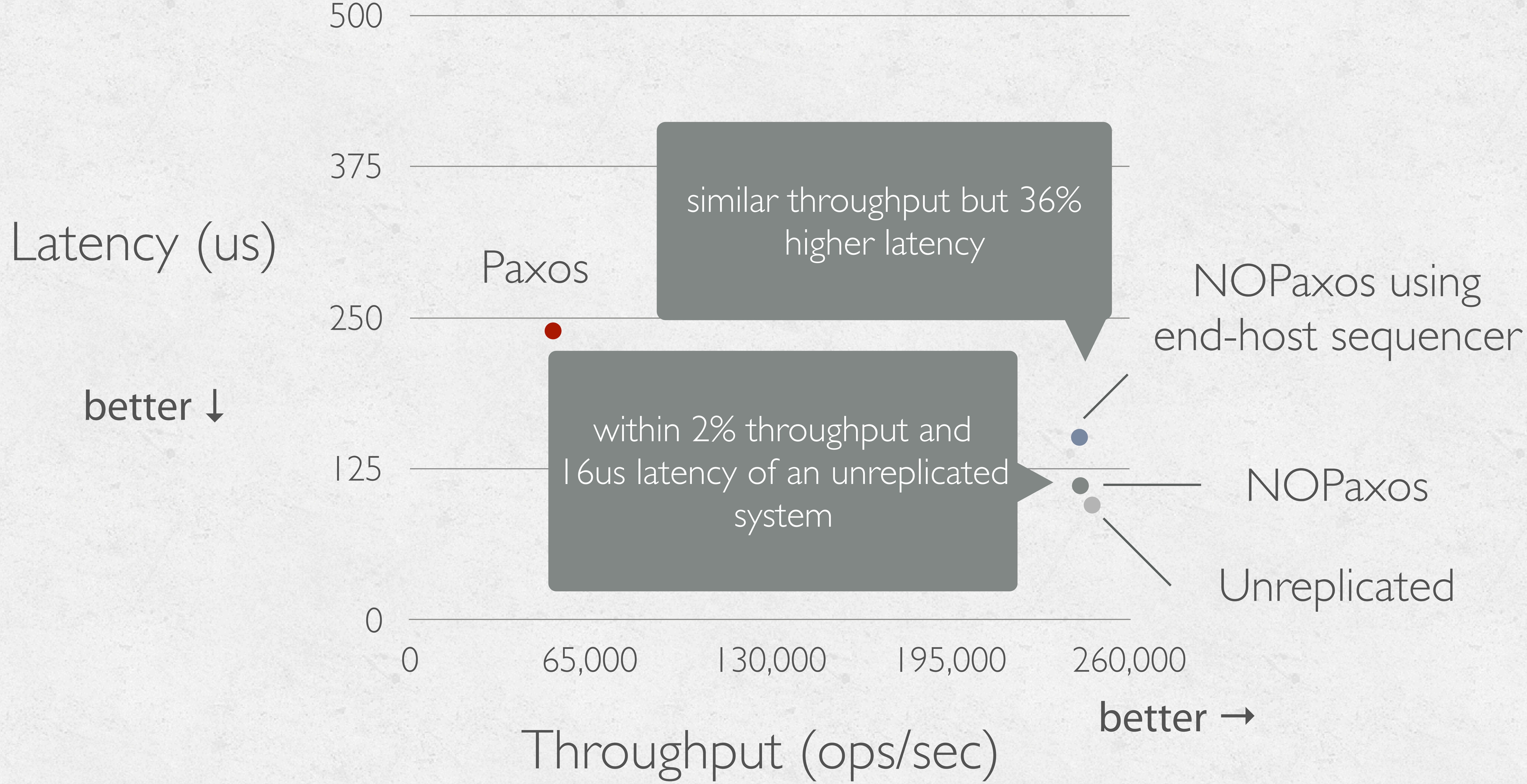
within 2% throughput and 16us latency of an unreplicated system

NOPaxos attains throughput within 2% of an unreplicated system



within 2% throughput and 16us latency of an unreplicated system

NOPaxos attains throughput within 2% of an unreplicated system



similar throughput but 36% higher latency

within 2% throughput and 16us latency of an unreplicated system

Summary

- Separate ordering from reliable delivery in state machine replication

- A network model OUM that provides ordered but unreliable message delivery

- A more efficient replication protocol NOPaxos that ensures reliable delivery

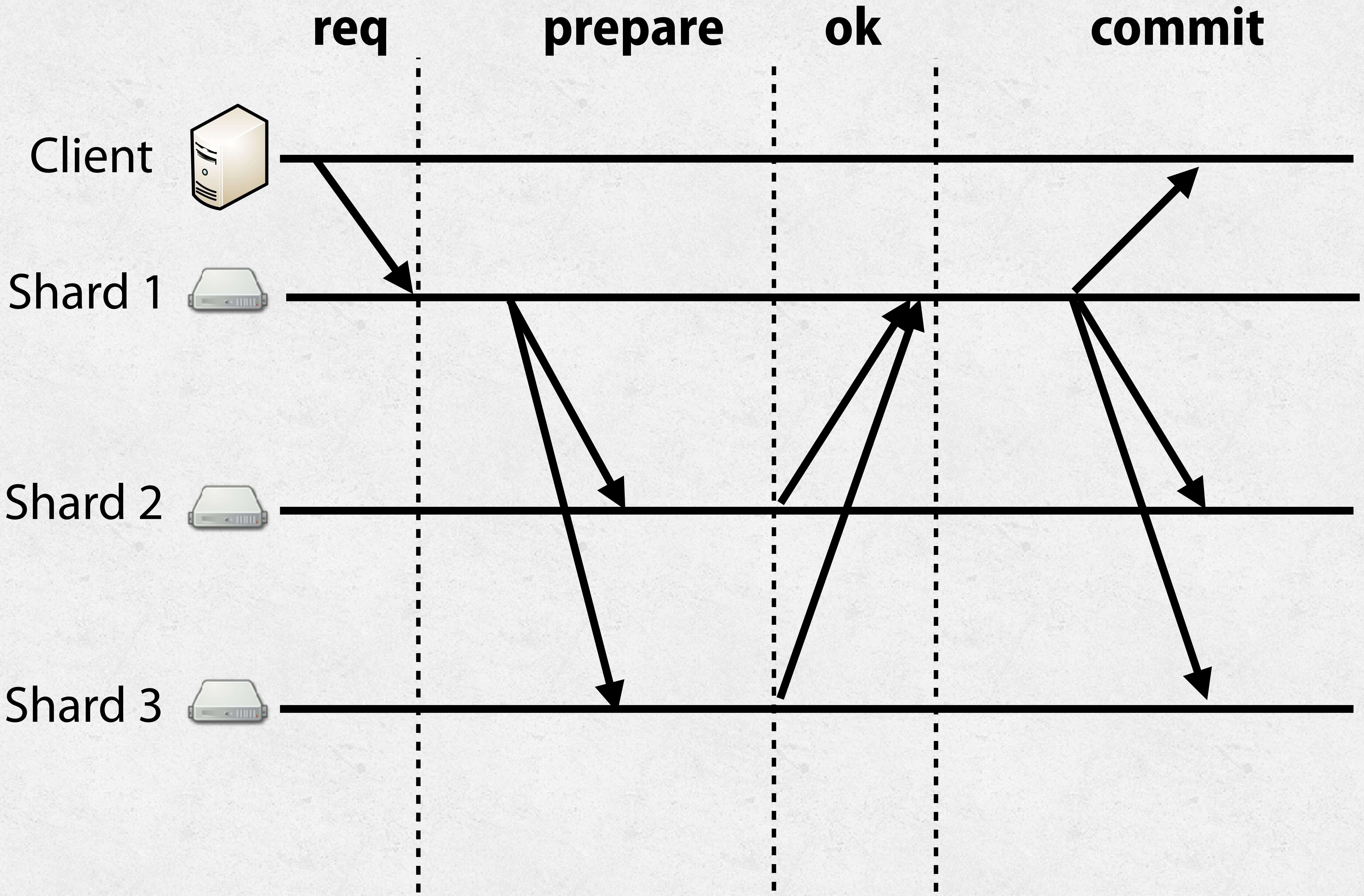
- The combined system achieves performance equivalent to an unreplicated system

The Eris Transaction Protocol

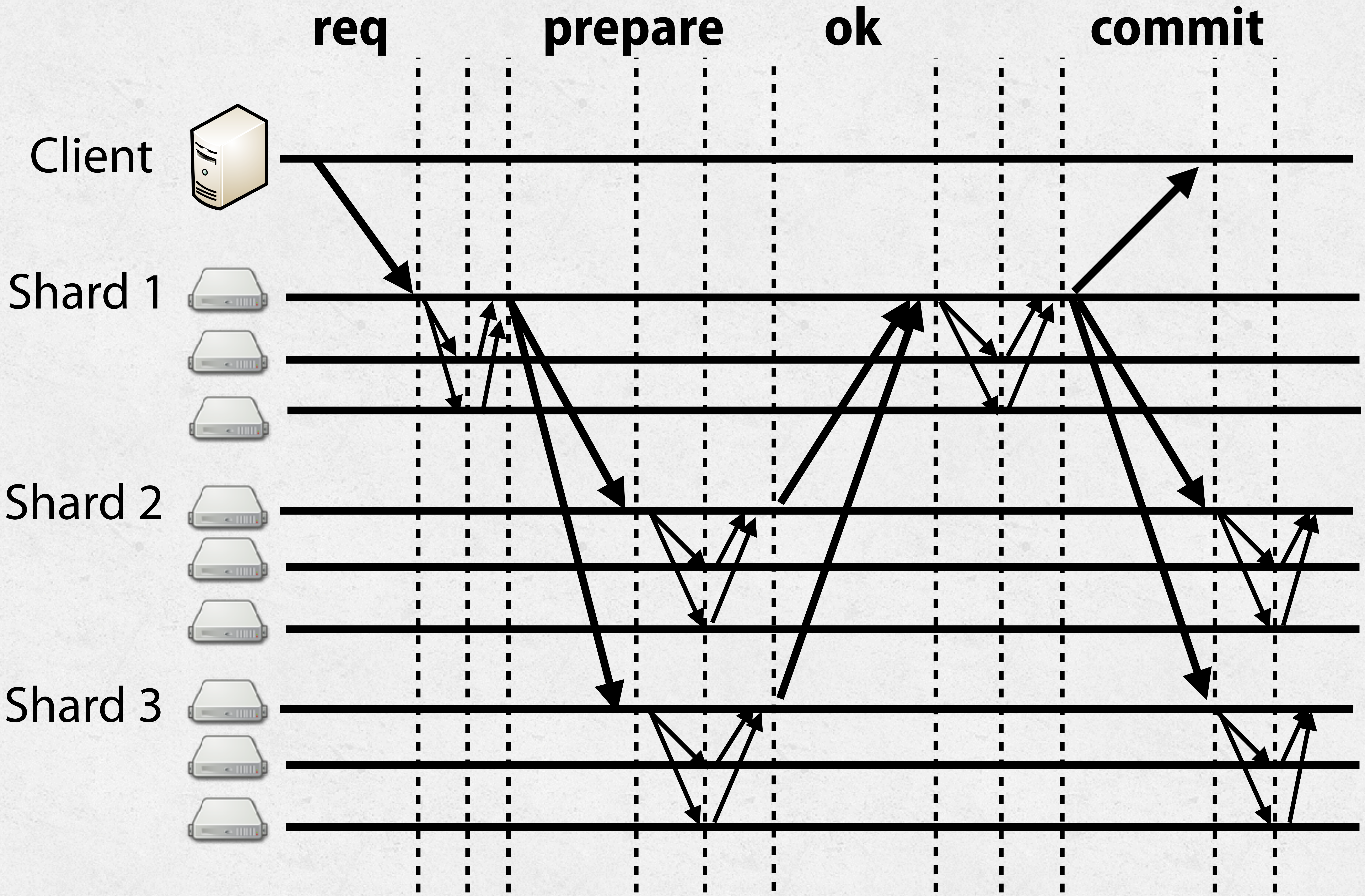
Existing transactional systems: extensive coordination

Client
Shard 1
Shard 2
Shard 3

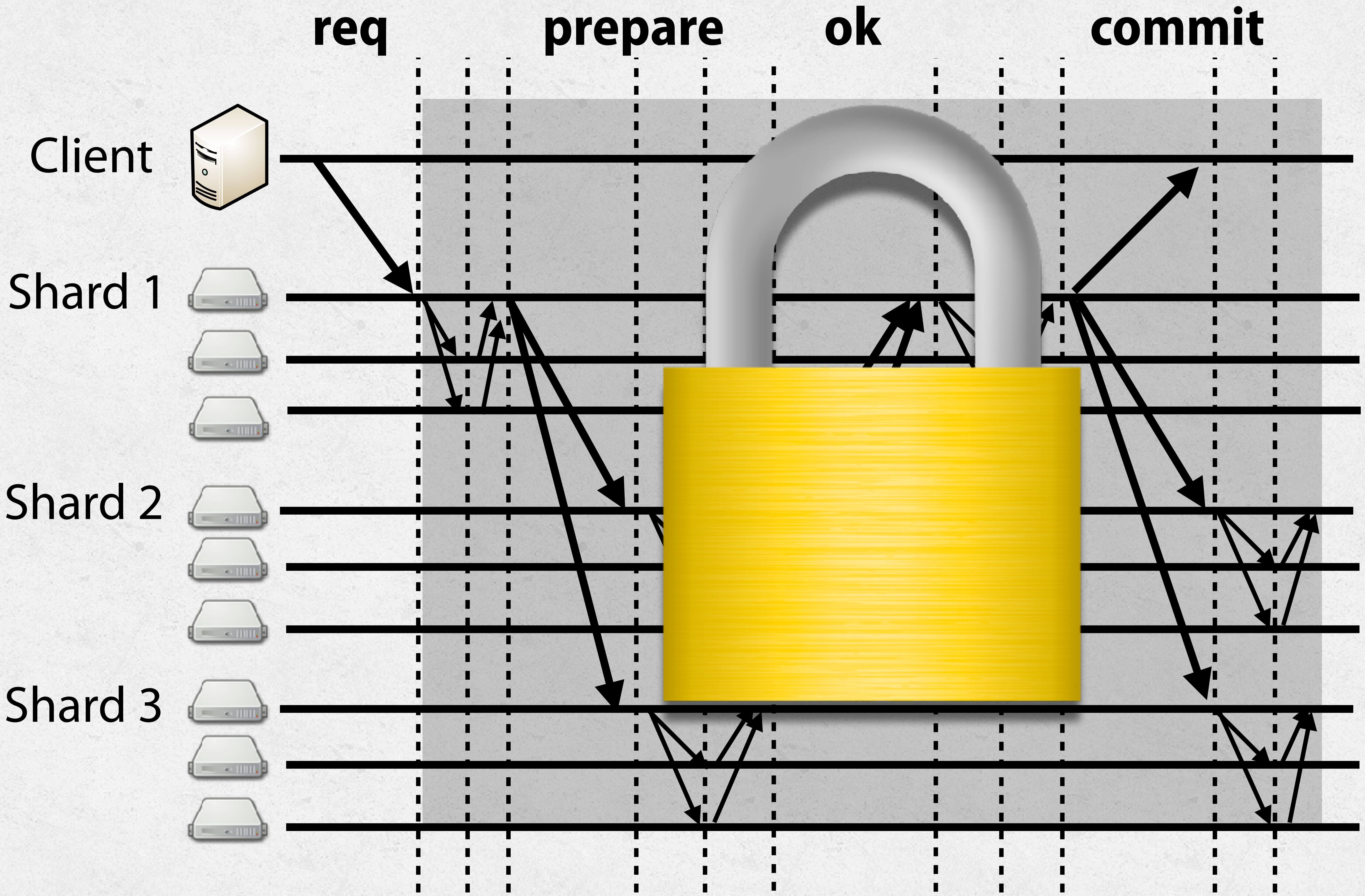
EXISTING TRANSACTIONAL SYSTEMS: EXTENSIVE COORDINATION



Existing transactional systems: extensive coordination



Existing transactional systems: extensive coordination



Eris

- Processes independent transactions **without coordination** in the normal case
- Performance within **3%** of a nontransactional, unreplicated system on TPC-C
- Strongly consistent, fault tolerant transactions with **minimal performance penalties**

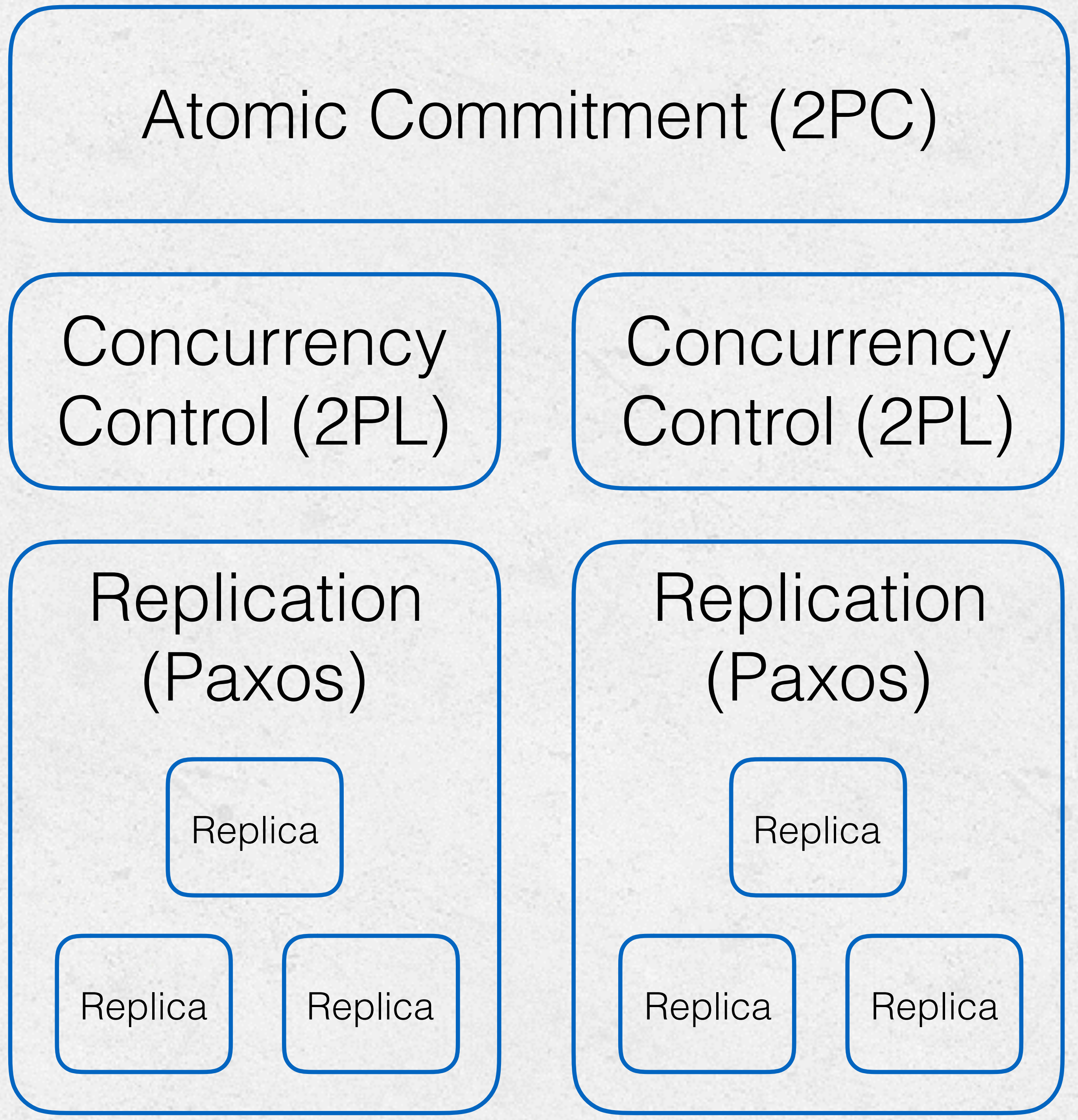
Key Contributions

A **new architecture** that divides the responsibility for transactional guarantees by

…leveraging the **datacenter network** to order messages within and across shards

…and a co-designed **transaction protocol** with minimal coordination.

Traditional Layered Approach



Atomic Commitment (2PC)

Concurrency Control (2PL)

Concurrency Control (2PL)

Replication (Paxos)

Replica

Replica

Replica

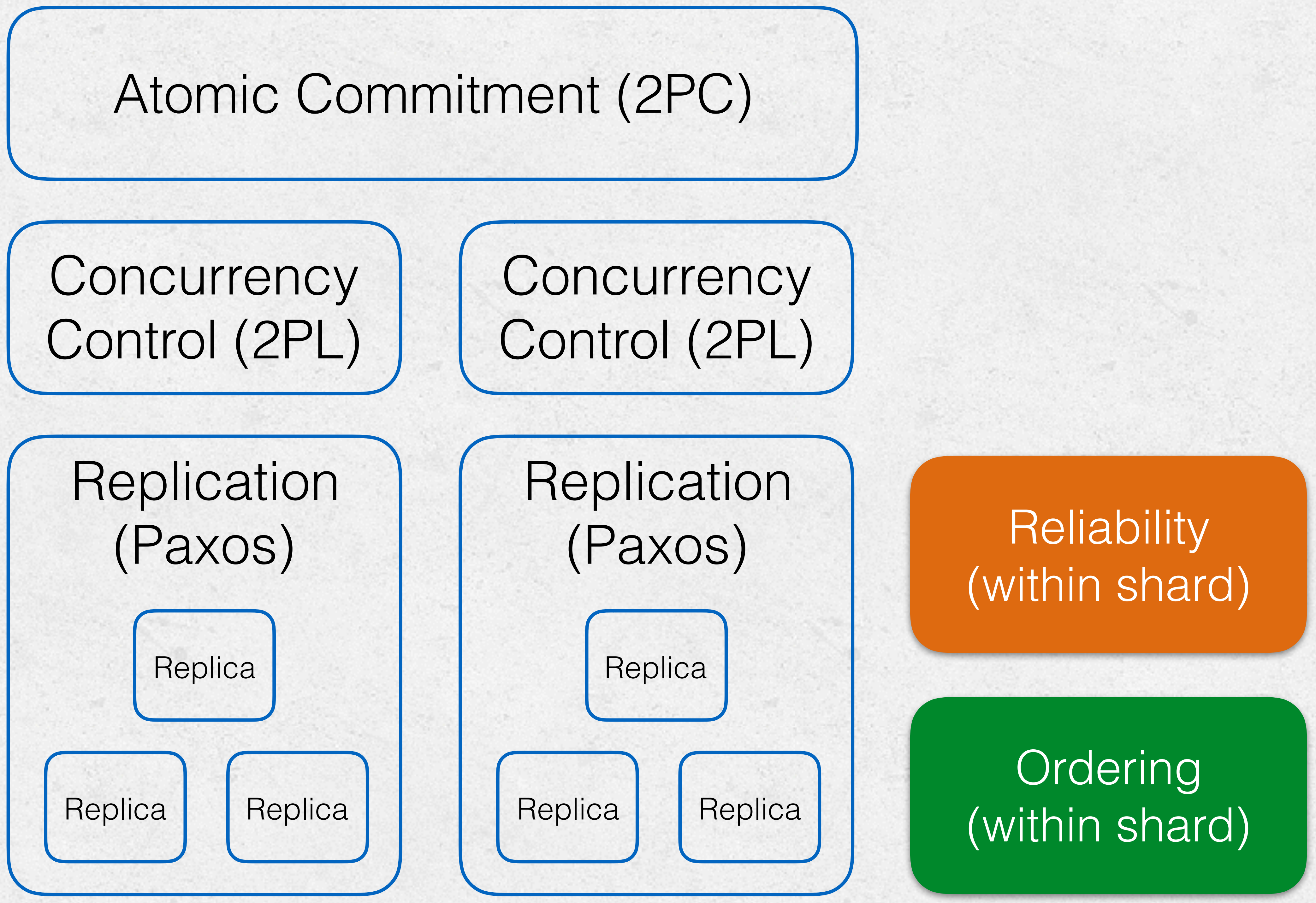
Replication (Paxos)

Replica

Replica

Replica

Traditional Layered Approach



Atomic Commitment (2PC)

Concurrency Control (2PL)

Concurrency Control (2PL)

Replication (Paxos)

Replica

Replica

Replica

Replication (Paxos)

Replica

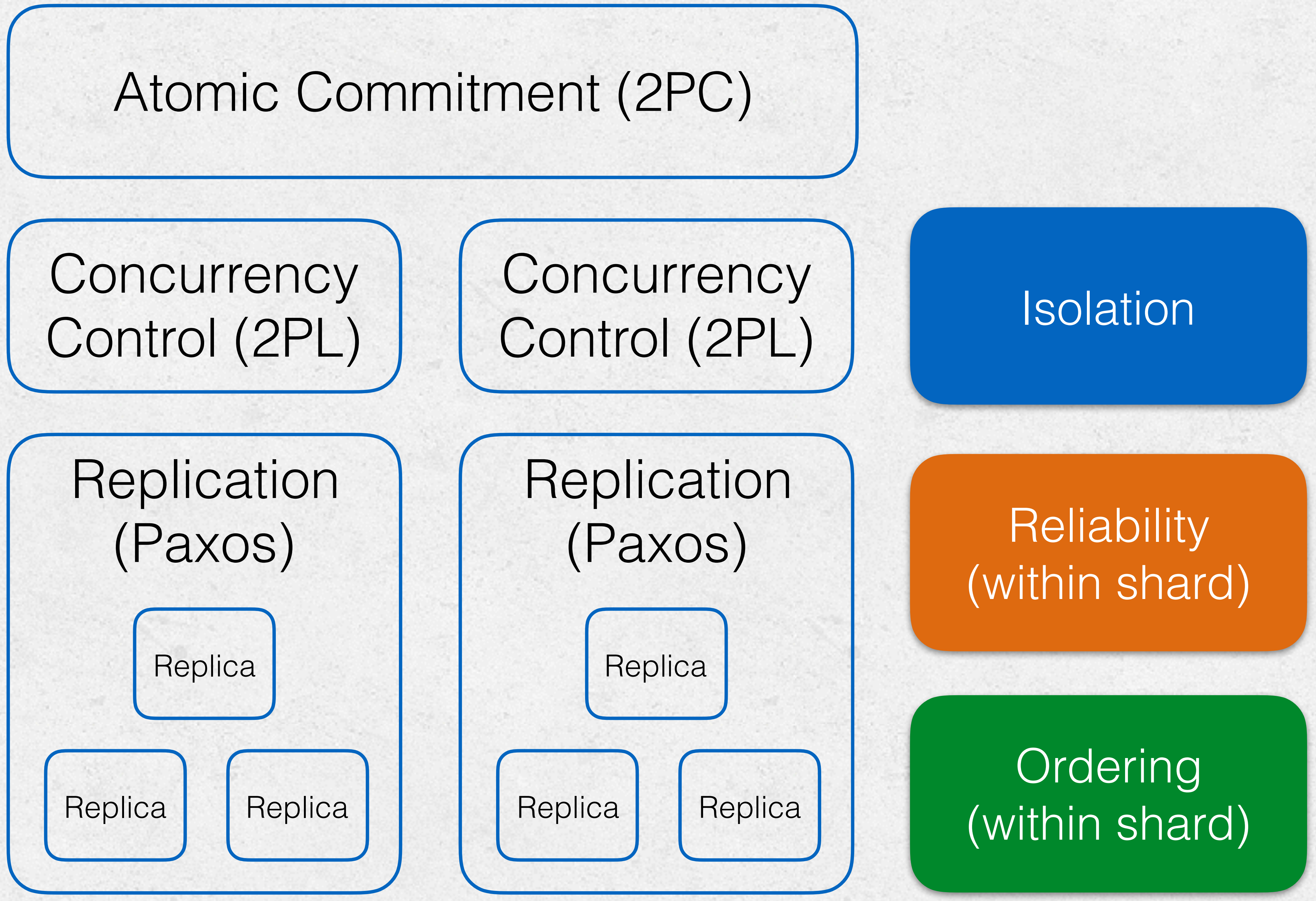
Replica

Replica

Reliability (within shard)

Ordering (within shard)

Traditional Layered Approach


Atomic Commitment (2PC)

Concurrency Control (2PL)

Concurrency Control (2PL)

Replication (Paxos)

Replica

Replica

Replica

Replication (Paxos)

Replica

Replica

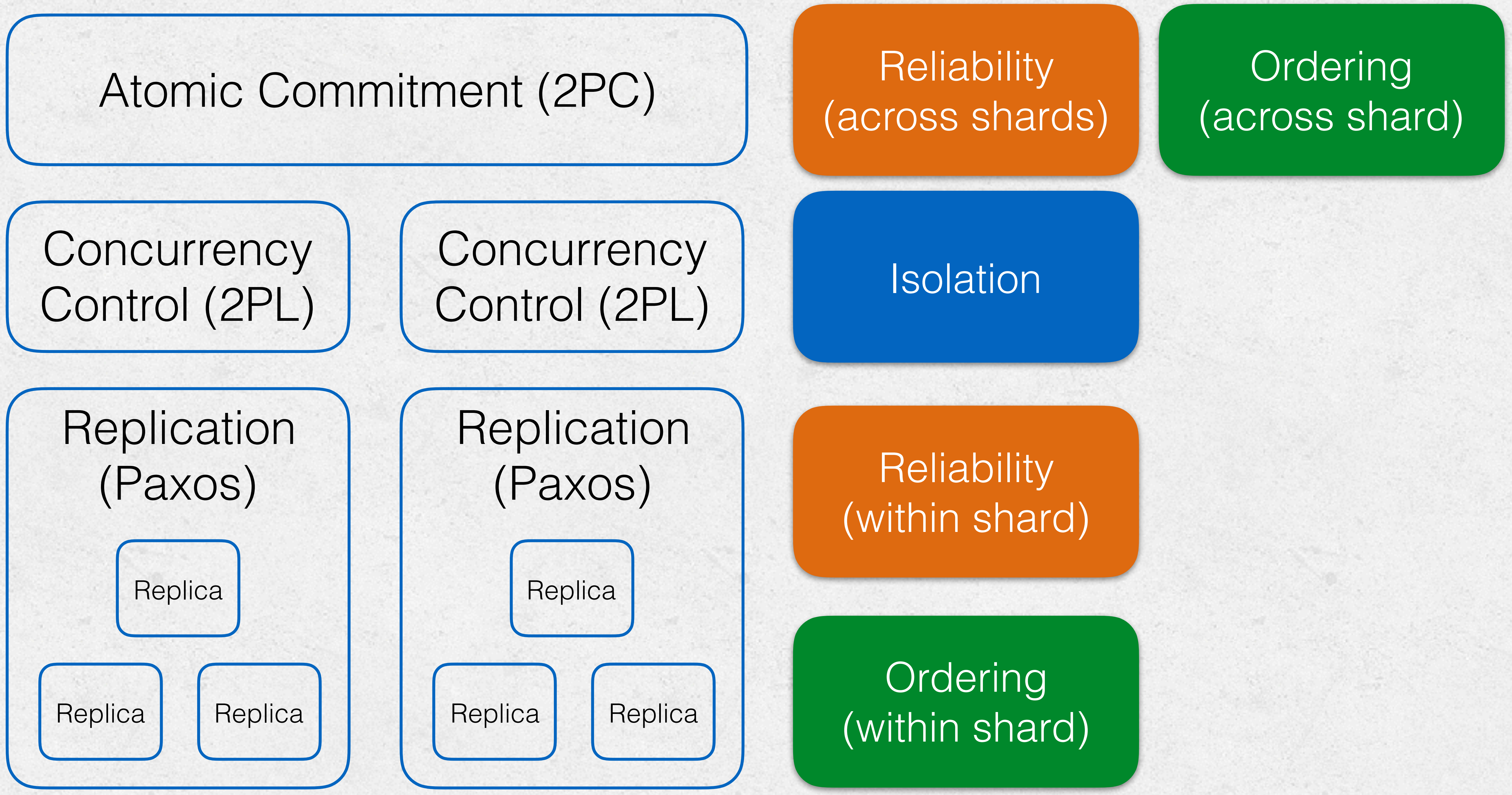
Replica

Isolation

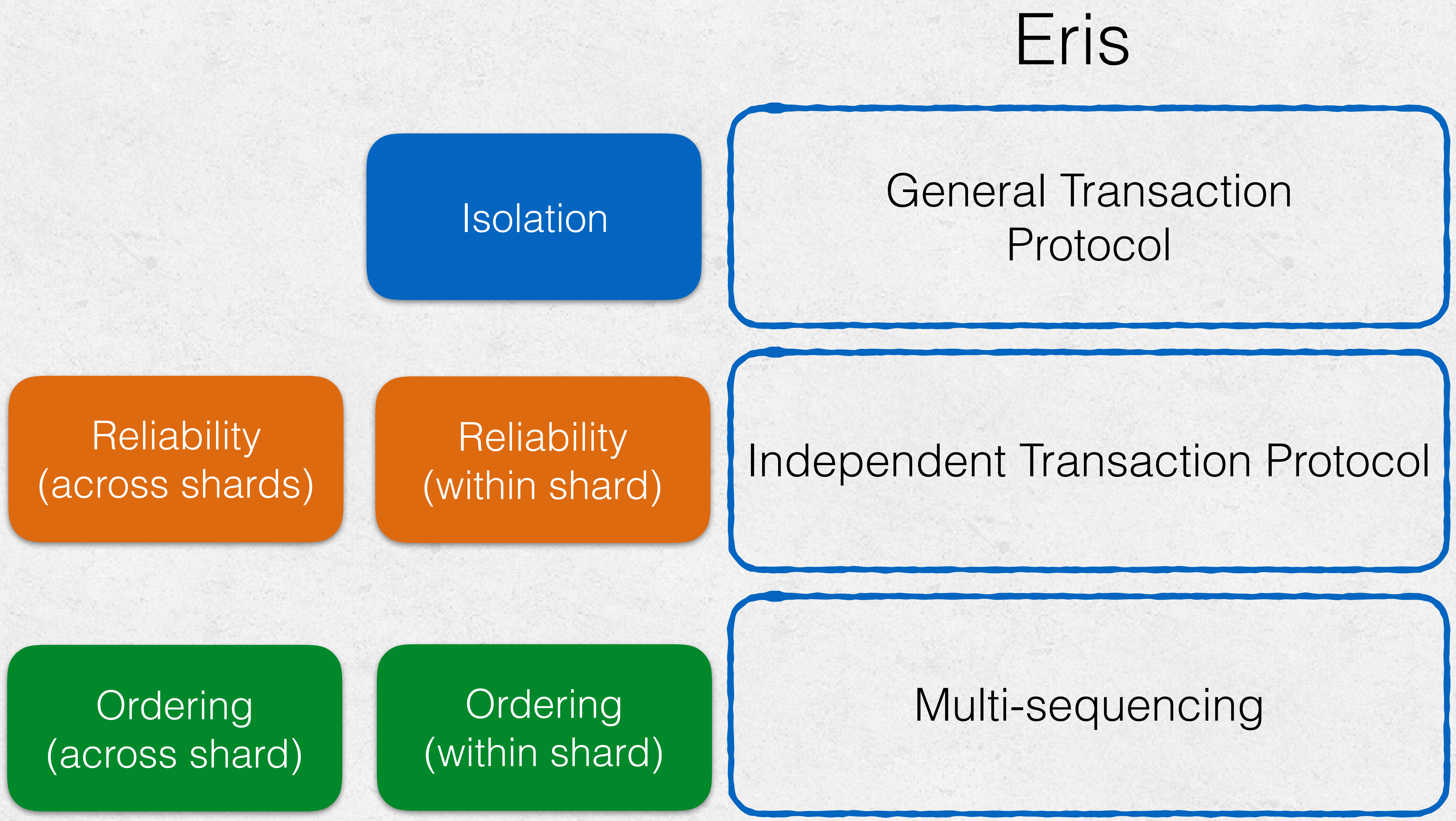
Reliability (within shard)

Ordering (within shard)

Traditional Layered Approach



A New Way to Divide Responsibilities


Eris

Isolation

General Transaction Protocol

Reliability (across shards)

Reliability (within shard)

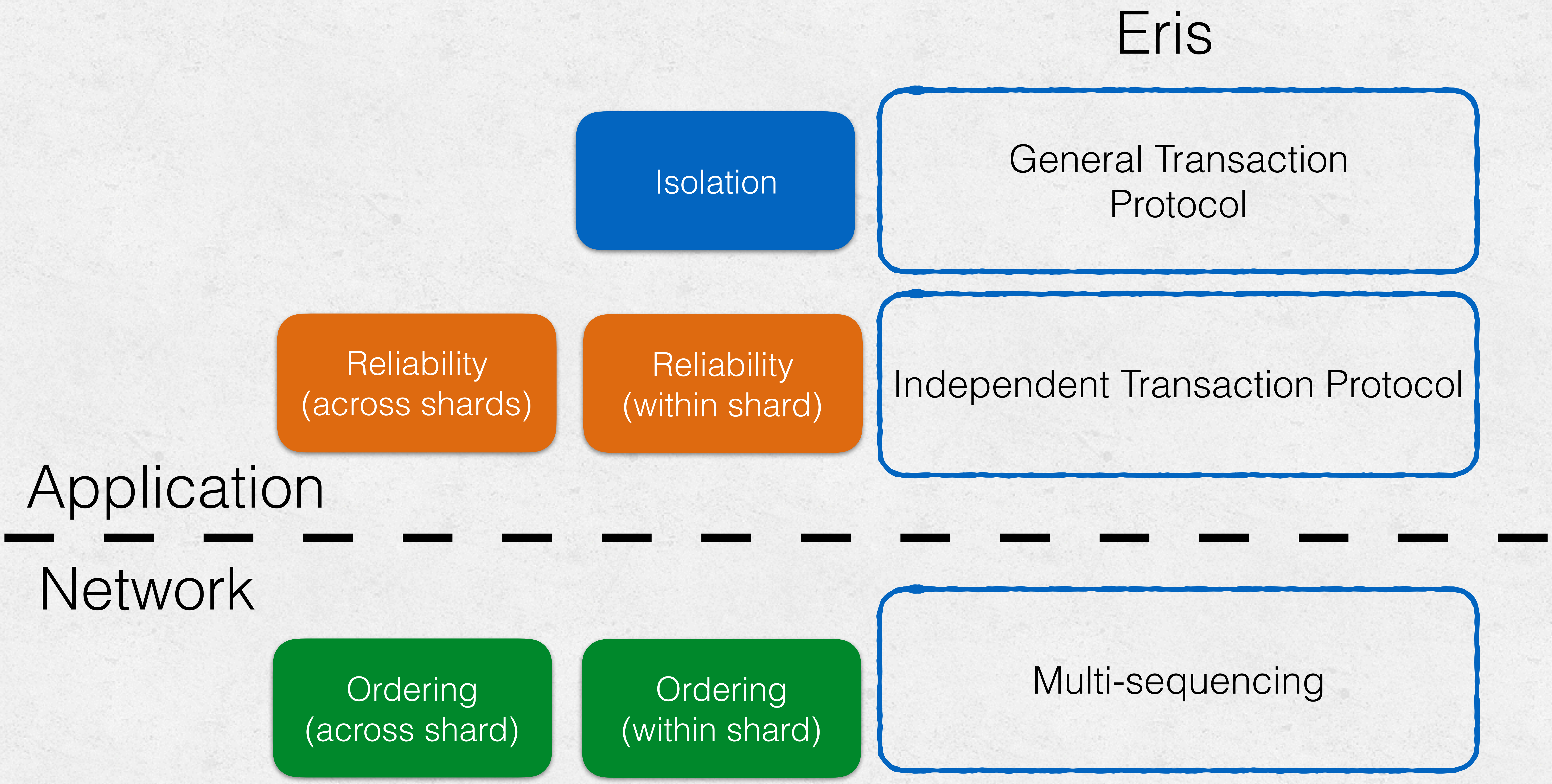
Independent Transaction Protocol

Ordering (across shard)

Ordering (within shard)

Multi-sequencing

A New Way to Divide Responsibilities



Eris

Isolation

General Transaction Protocol

Reliability (across shards)

Reliability (within shard)

Independent Transaction Protocol

Application

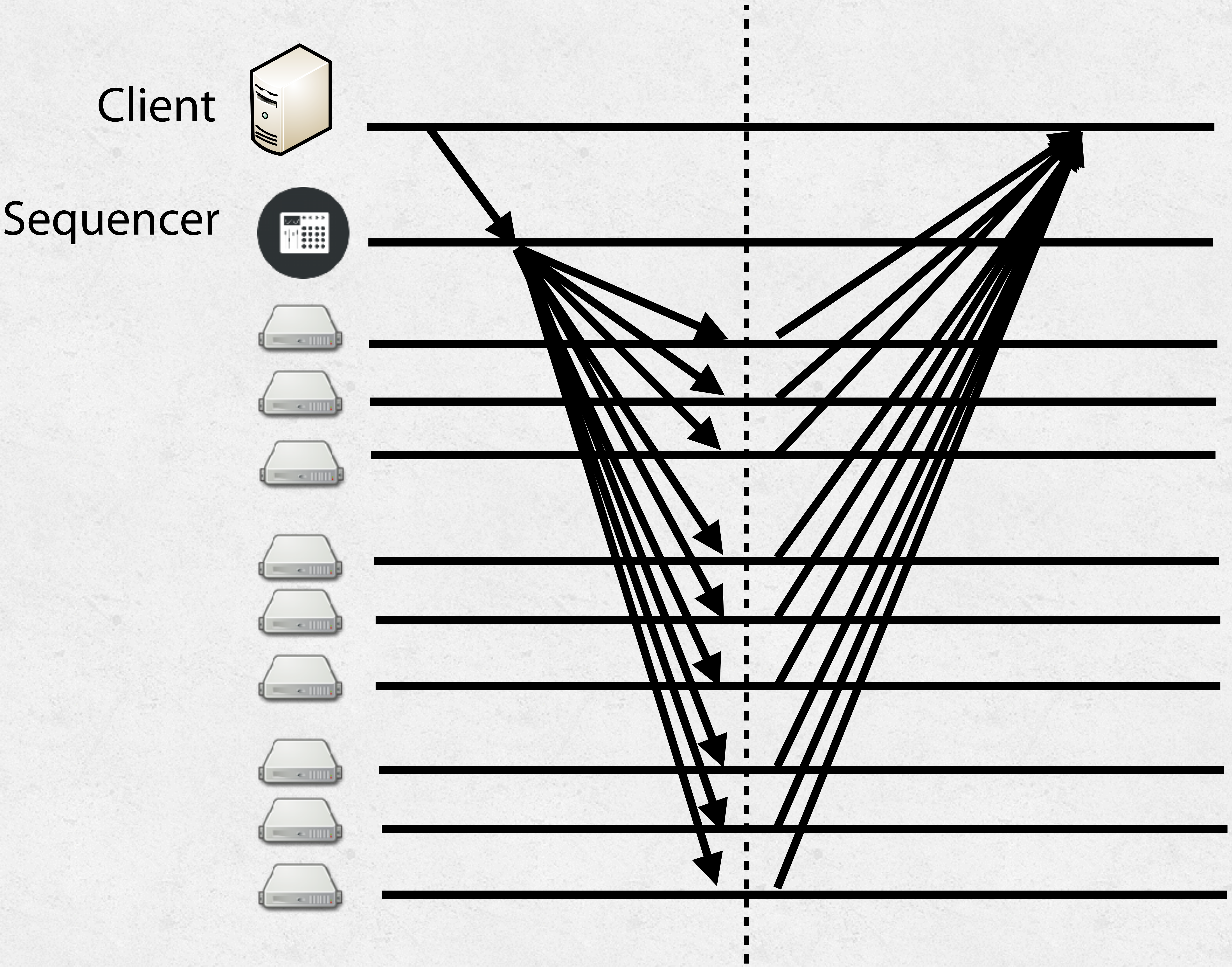
Network

Ordering (across shard)

Ordering (within shard)

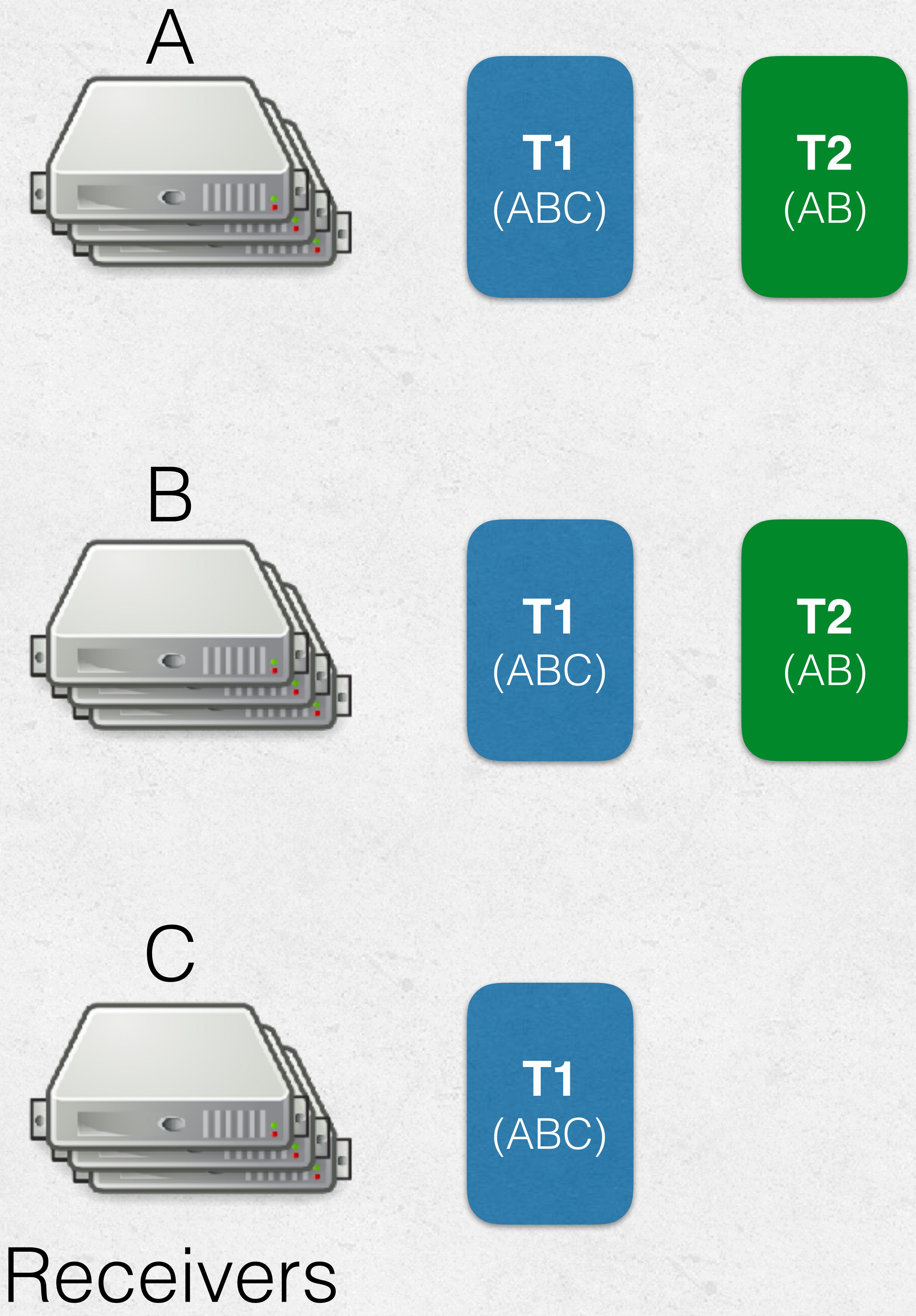
Multi-sequencing

Goal



In-Network Concurrency Control Goals

- **Globally consistent ordering** across messages delivered to multiple destination shards
- No reliable delivery guarantee
- Recipients can **detect dropped messages**



A

T1
(ABC)

T2
(AB)

B

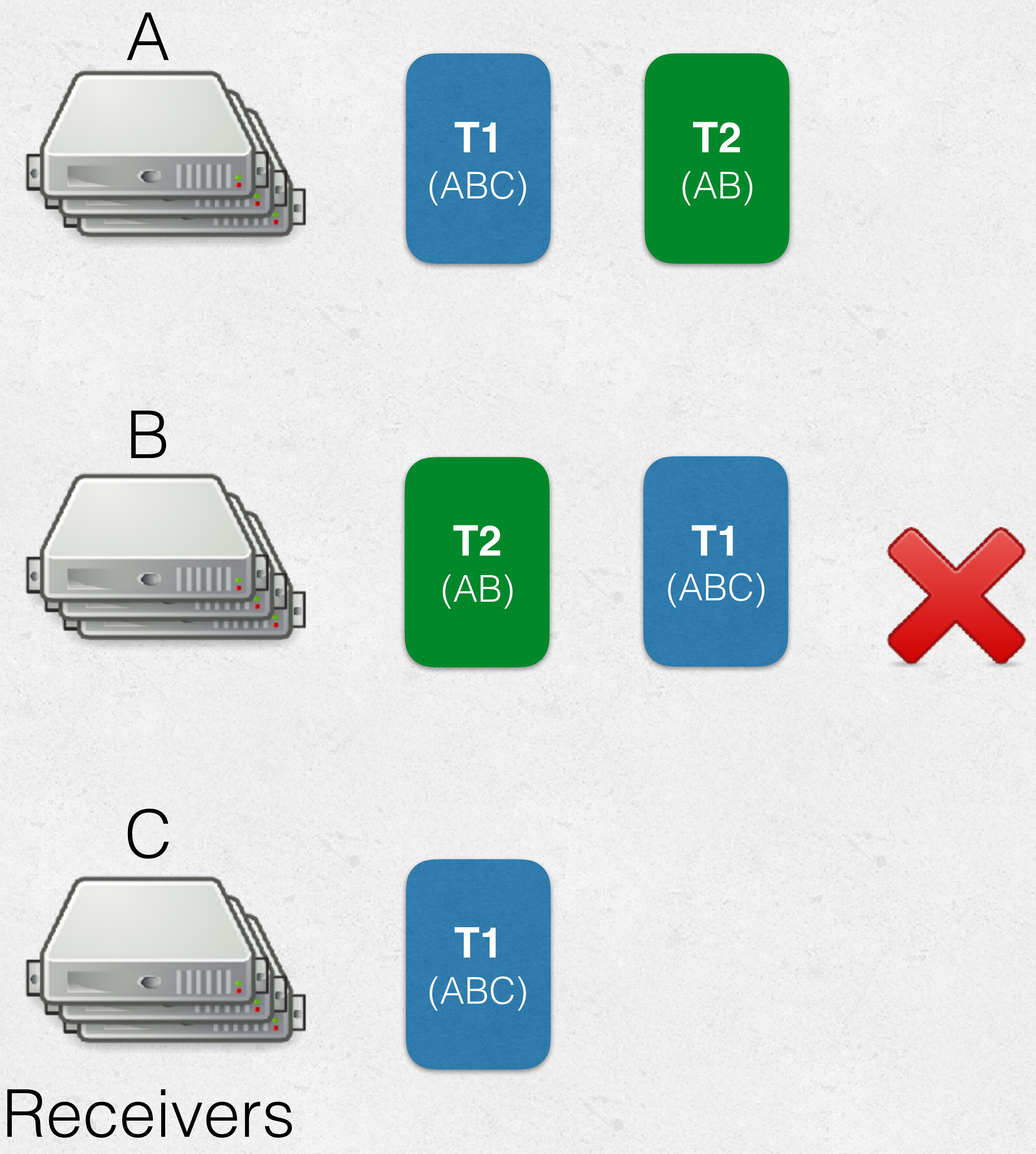
T1
(ABC)

T2
(AB)

C

T1
(ABC)

Receivers



A

T1 (ABC)

T2 (AB)

B

T2 (AB)

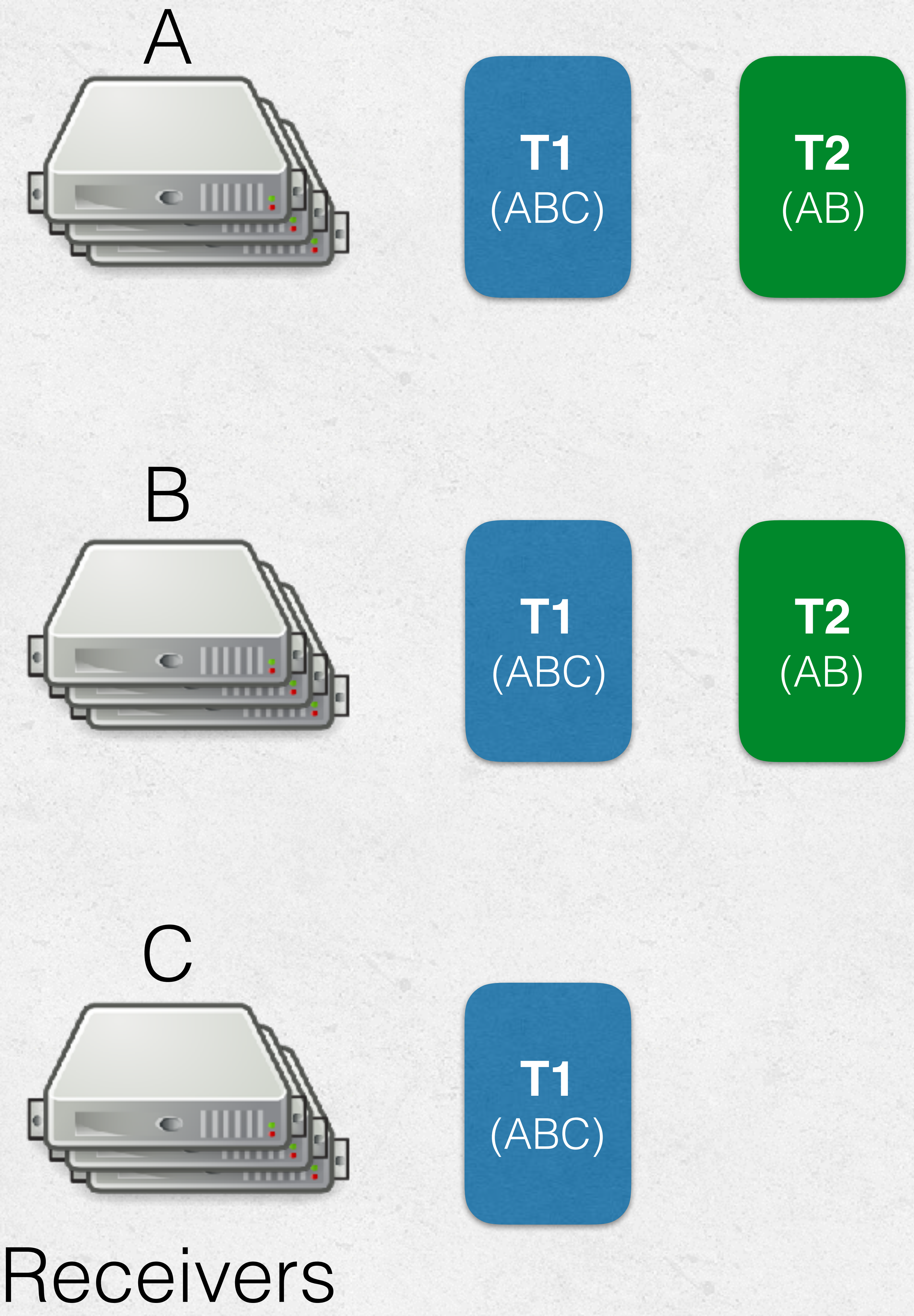
T1 (ABC)

✗

C

T1 (ABC)

Receivers



A

T1
(ABC)

T2
(AB)

B

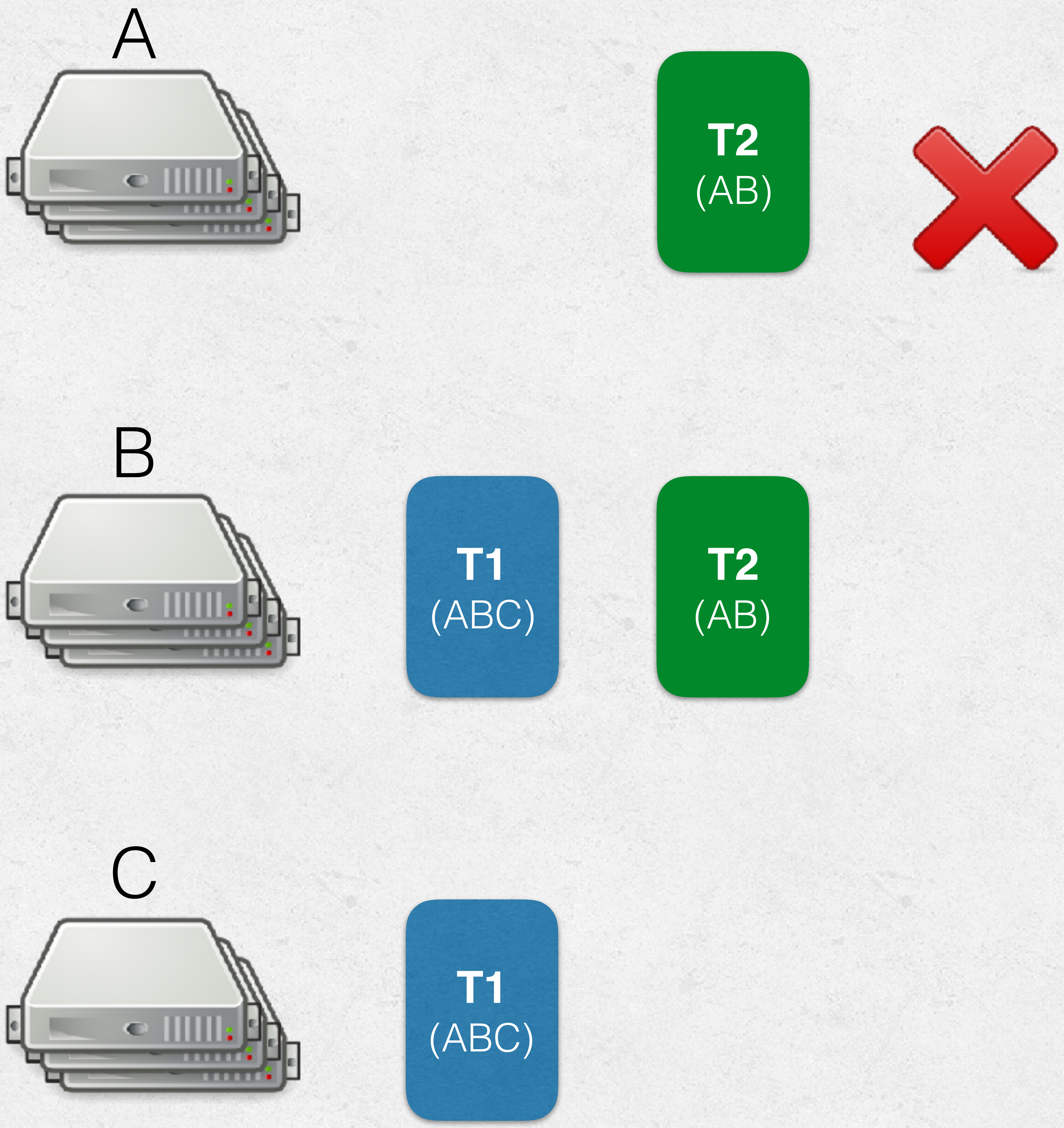
T1
(ABC)

T2
(AB)

C

T1
(ABC)

Receivers



A

T2
(AB)

✗

B

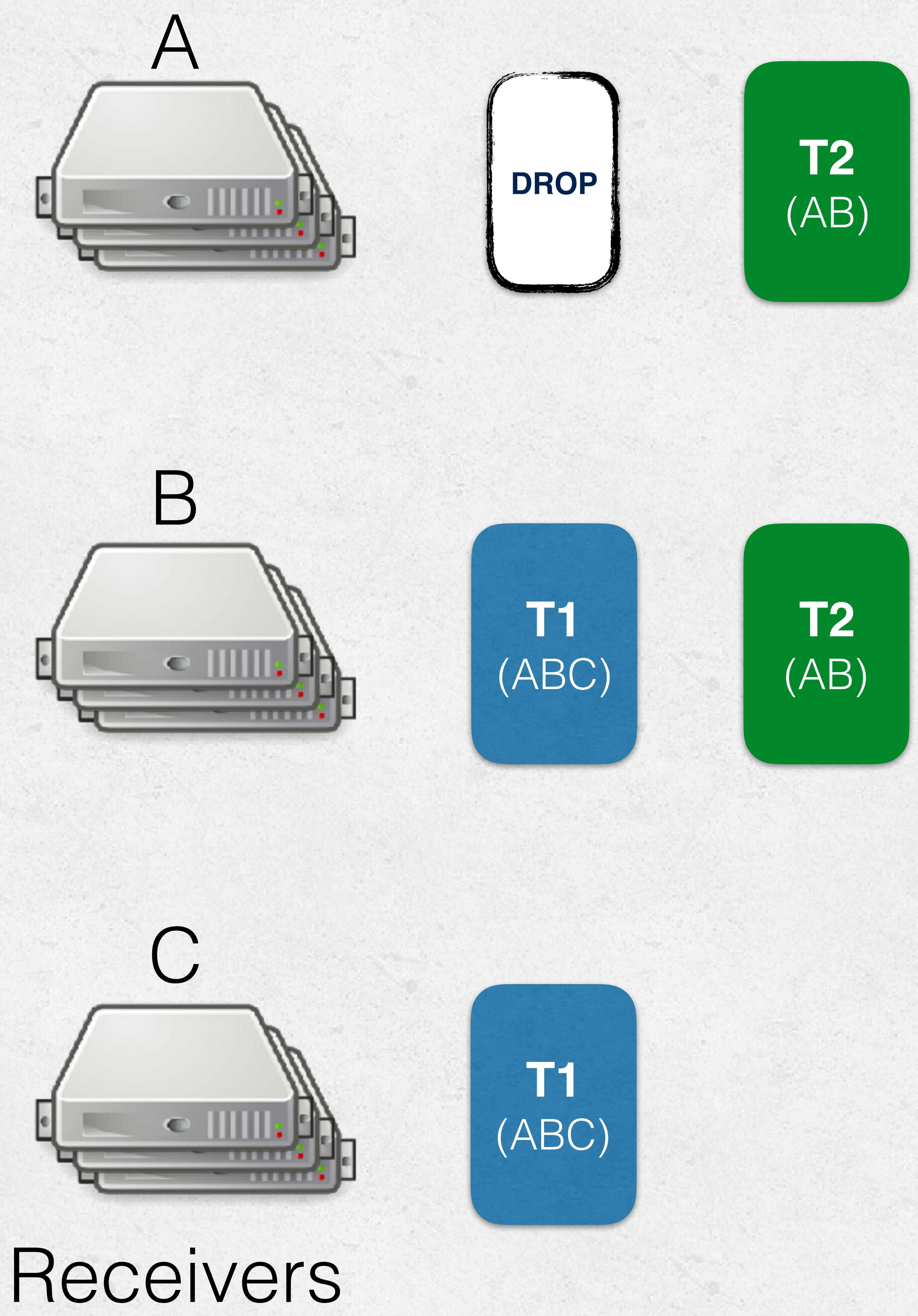
T1
(ABC)

T2
(AB)

C

T1
(ABC)

Receivers



A

DROP

T2 (AB)

B

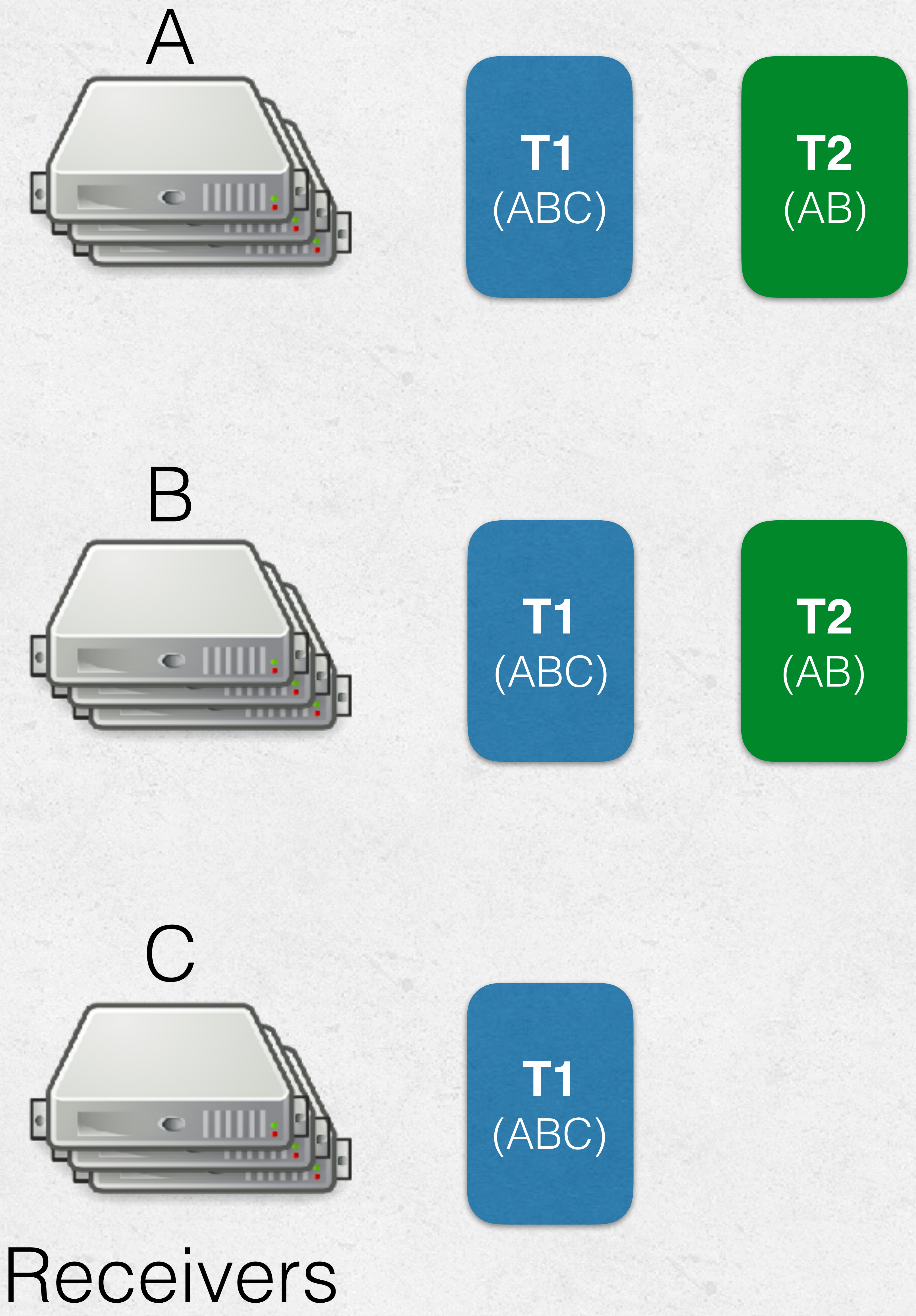
T1 (ABC)

T2 (AB)

C

T1 (ABC)

Receivers



A

T1
(ABC)

T2
(AB)

B

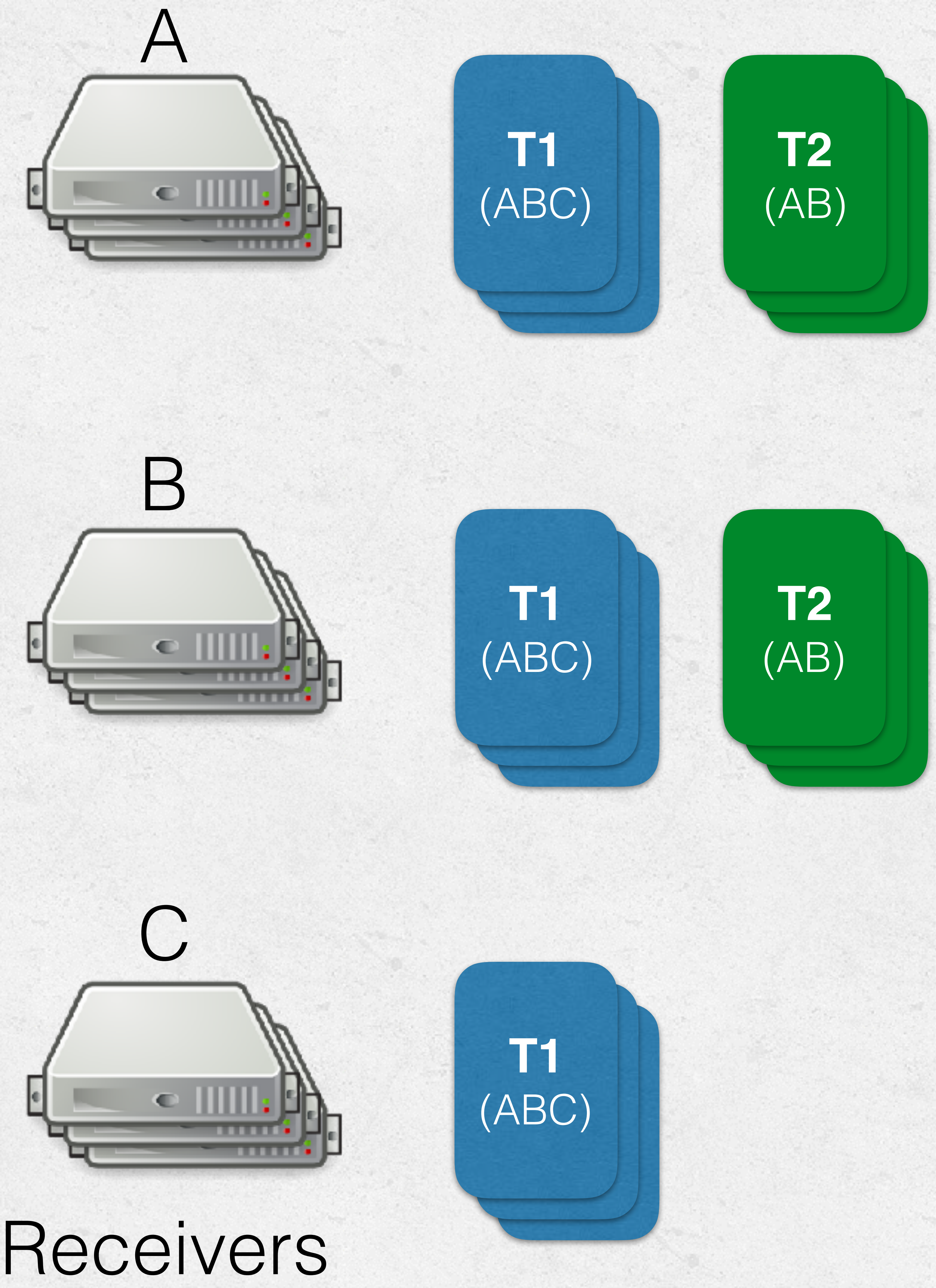
T1
(ABC)

T2
(AB)

C

T1
(ABC)

Receivers



A

T1
(ABC)

T2
(AB)

B

T1
(ABC)

T2
(AB)

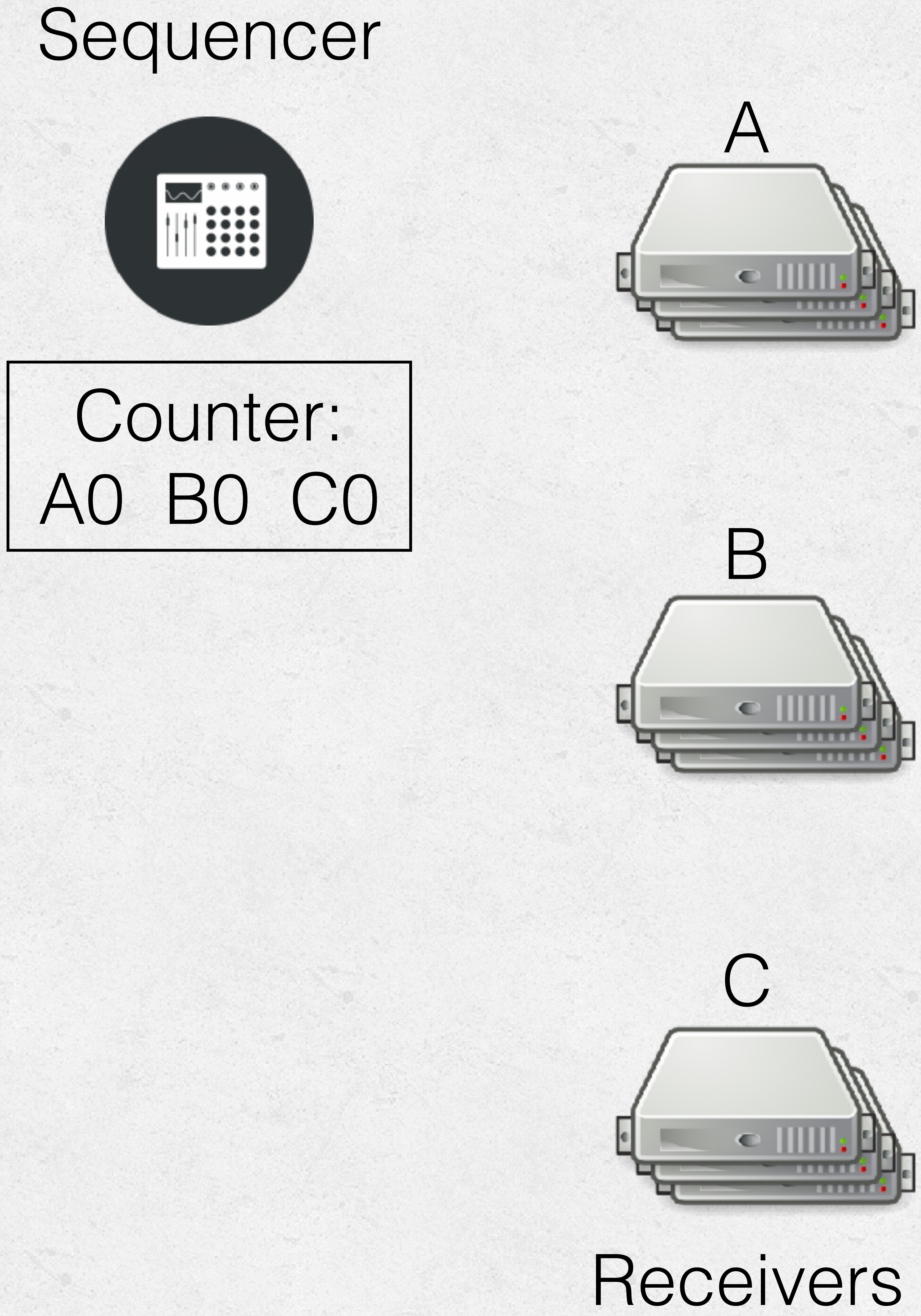
C

T1
(ABC)

Receivers

Multi-Sequenced Groupcast

- Groupcast: message header specifies a **set** of destination multicast groups
- Multi-sequenced groupcast: messages are sequenced **atomically** across all recipient groups
- Sequencer keeps a counter for each group
- Extends OUM in NOPaxos



Sequencer

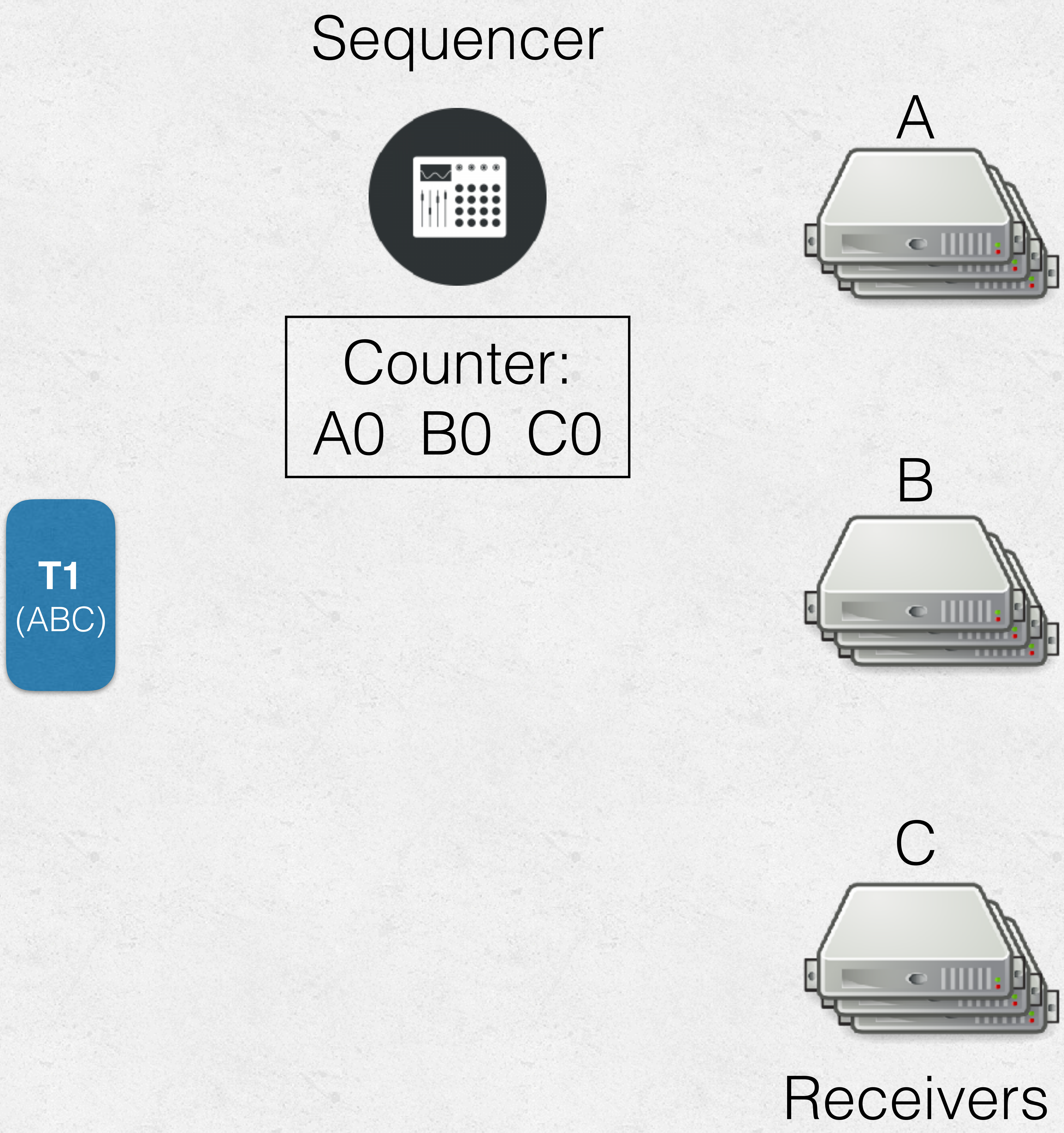
Counter:
A0 B0 C0

A

B

C

Receivers



Sequencer

Counter:
A0 B0 C0

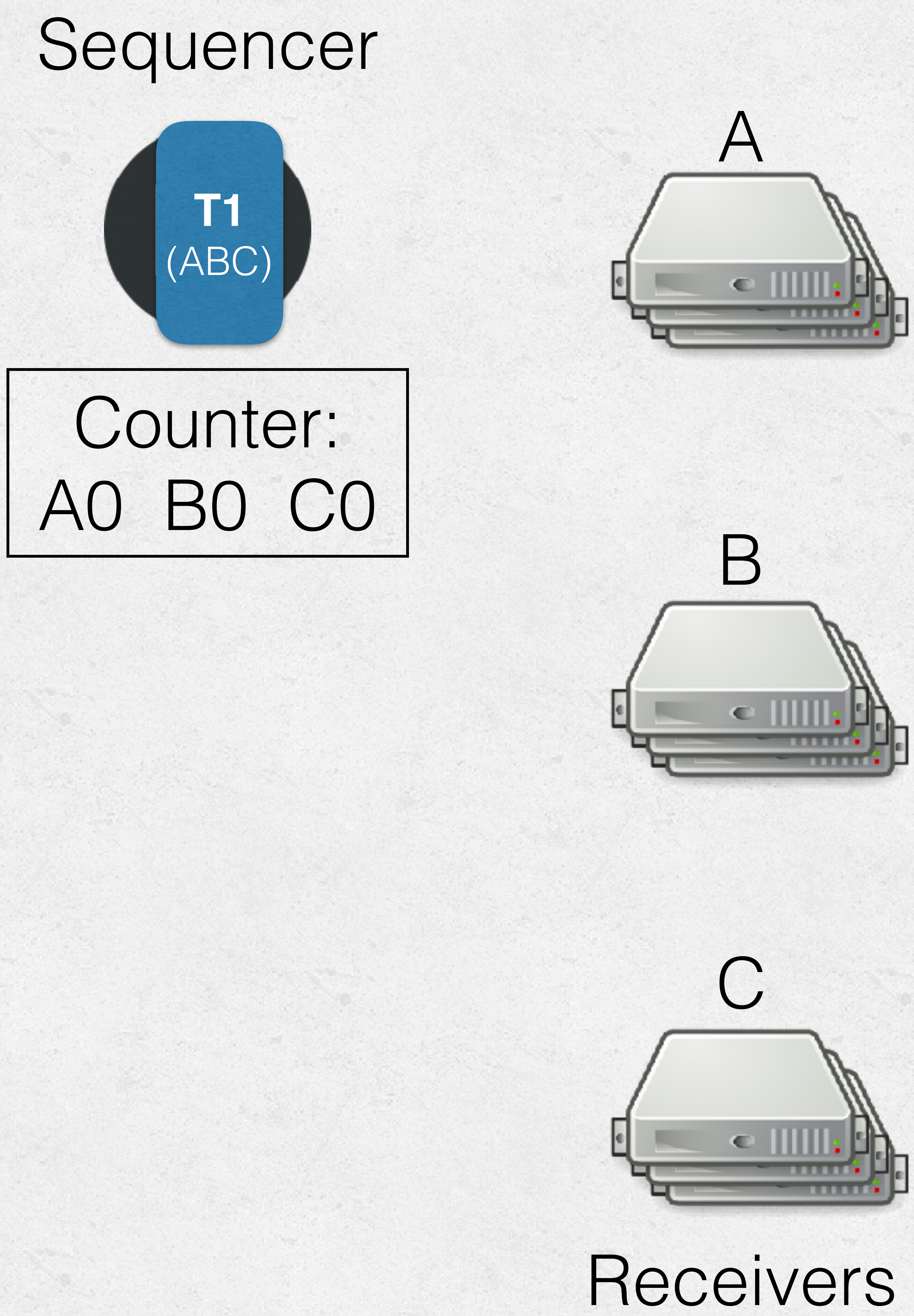
T1
(ABC)

A

B

C

Receivers



Sequencer

T1
(ABC)

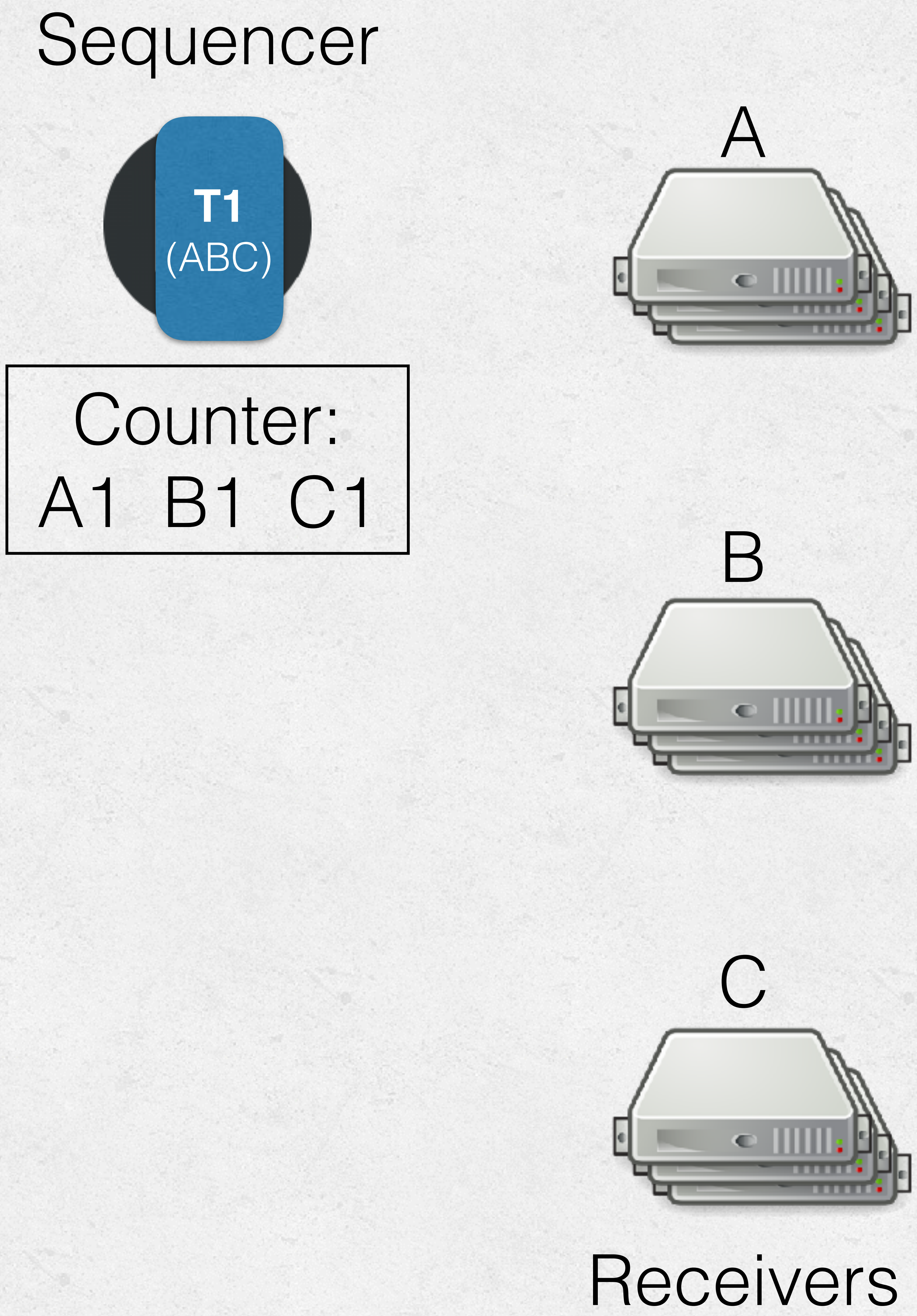
Counter:
A0 B0 C0

A

B

C

Receivers



Sequencer

T1
(ABC)

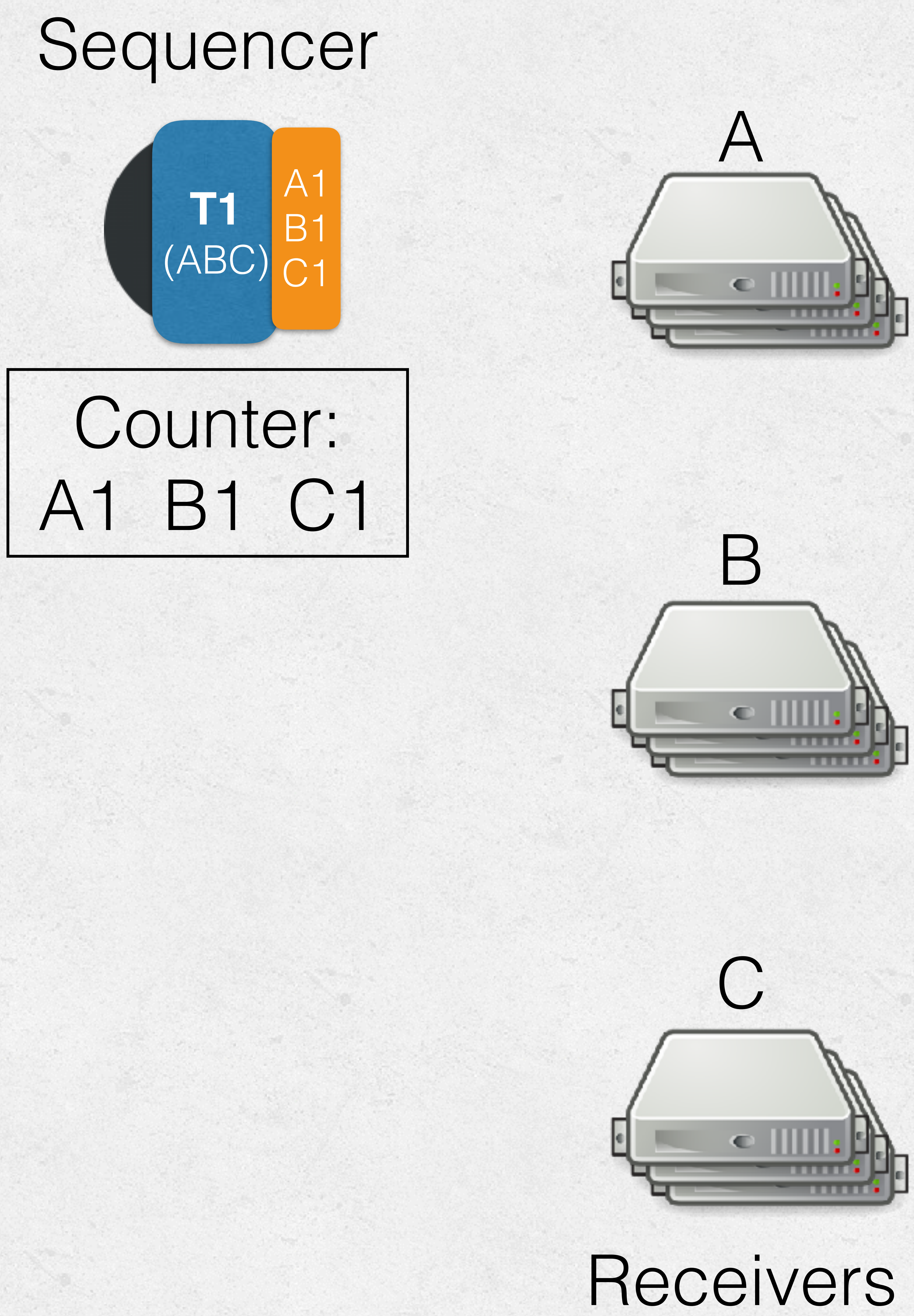
Counter:
A1 B1 C1

A

B

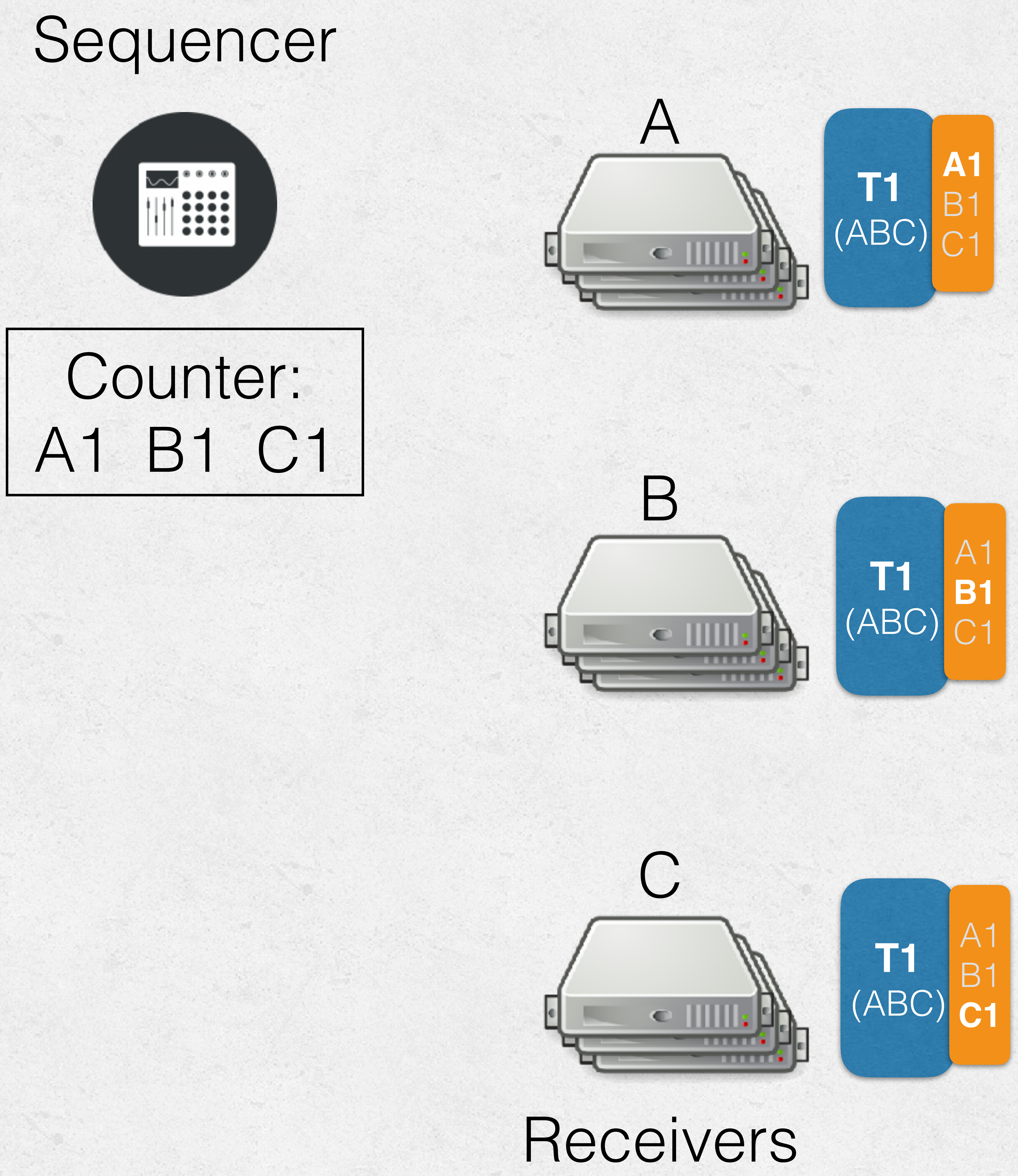
C

Receivers


Sequencer

Counter:
A1 B1 C1

Receivers



Sequencer

Counter:
A1 B1 C1

A

T1
(ABC)

A1
B1
C1

B

T1
(ABC)

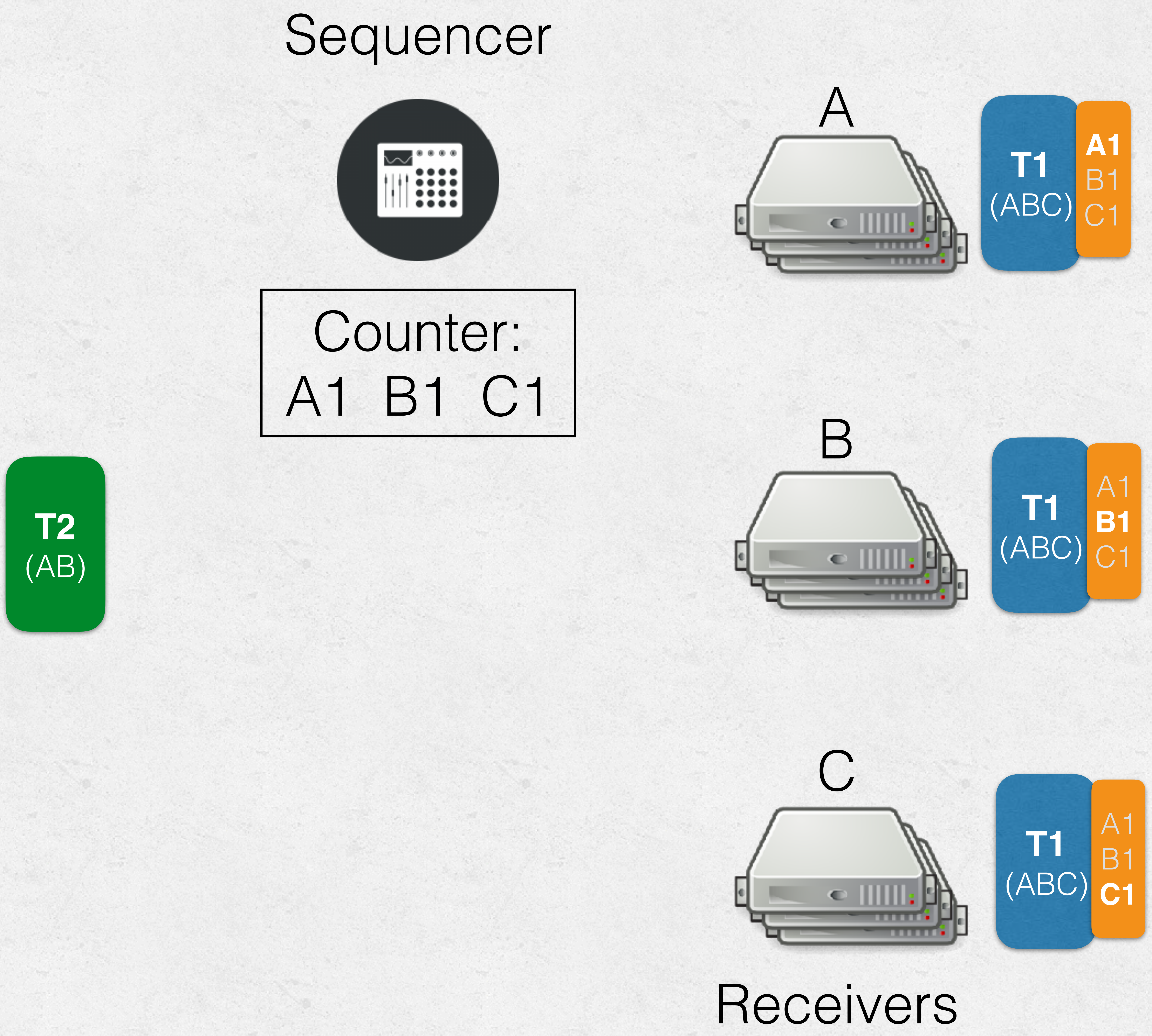
A1
B1
C1

C

T1
(ABC)

A1
B1
C1

Receivers



Sequencer

Counter:
A1 B1 C1

T2
(AB)

A

T1
(ABC)

A1
B1
C1

B

T1
(ABC)

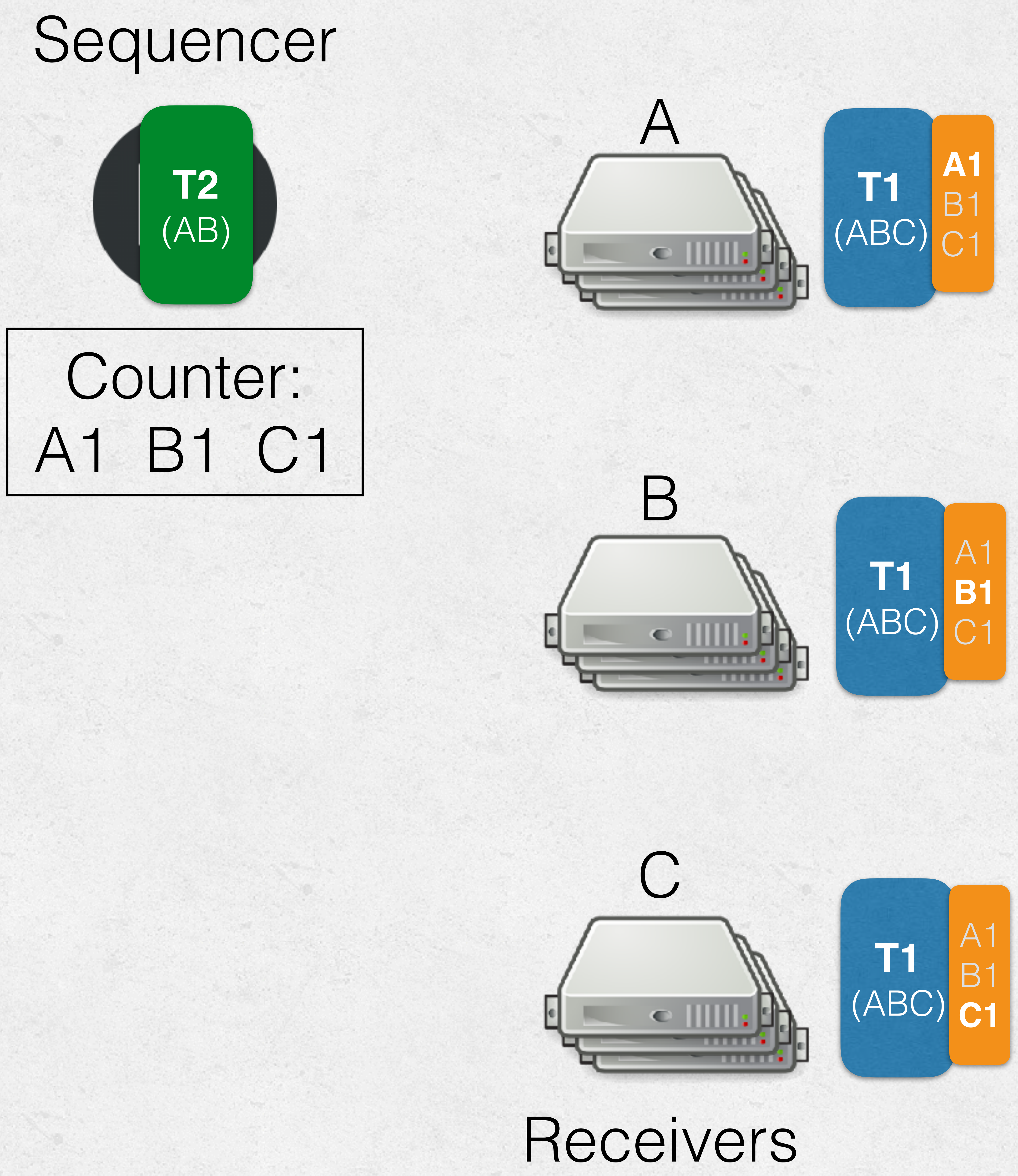
A1
B1
C1

C

T1
(ABC)

A1
B1
C1

Receivers



Sequencer

T2
(AB)

Counter:
A1 B1 C1

A

T1
(ABC)

A1
B1
C1

B

T1
(ABC)

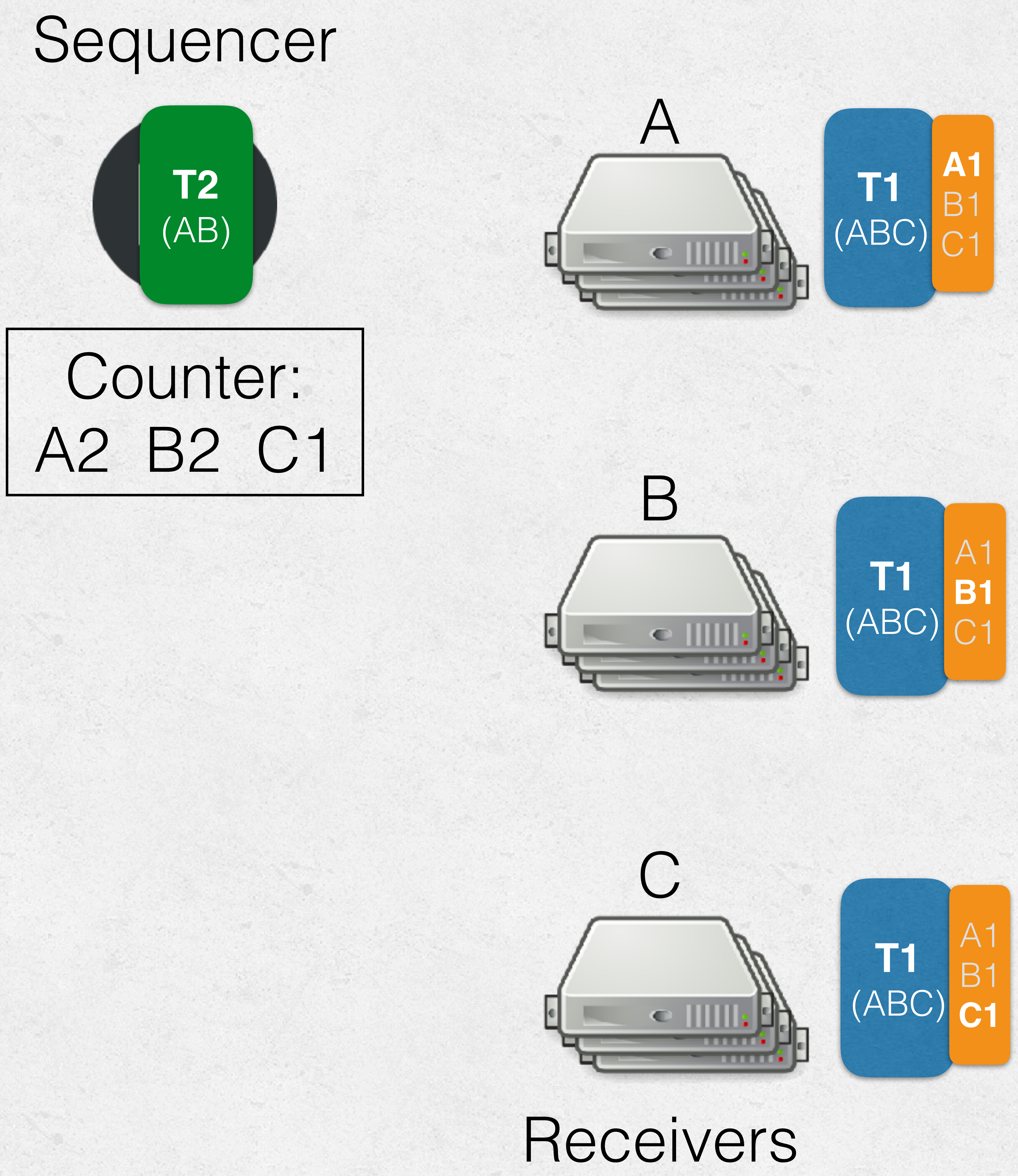
A1
B1
C1

C

T1
(ABC)

A1
B1
C1

Receivers



Sequencer

T2
(AB)

Counter:
A2 B2 C1

A

T1
(ABC)

A1
B1
C1

B

T1
(ABC)

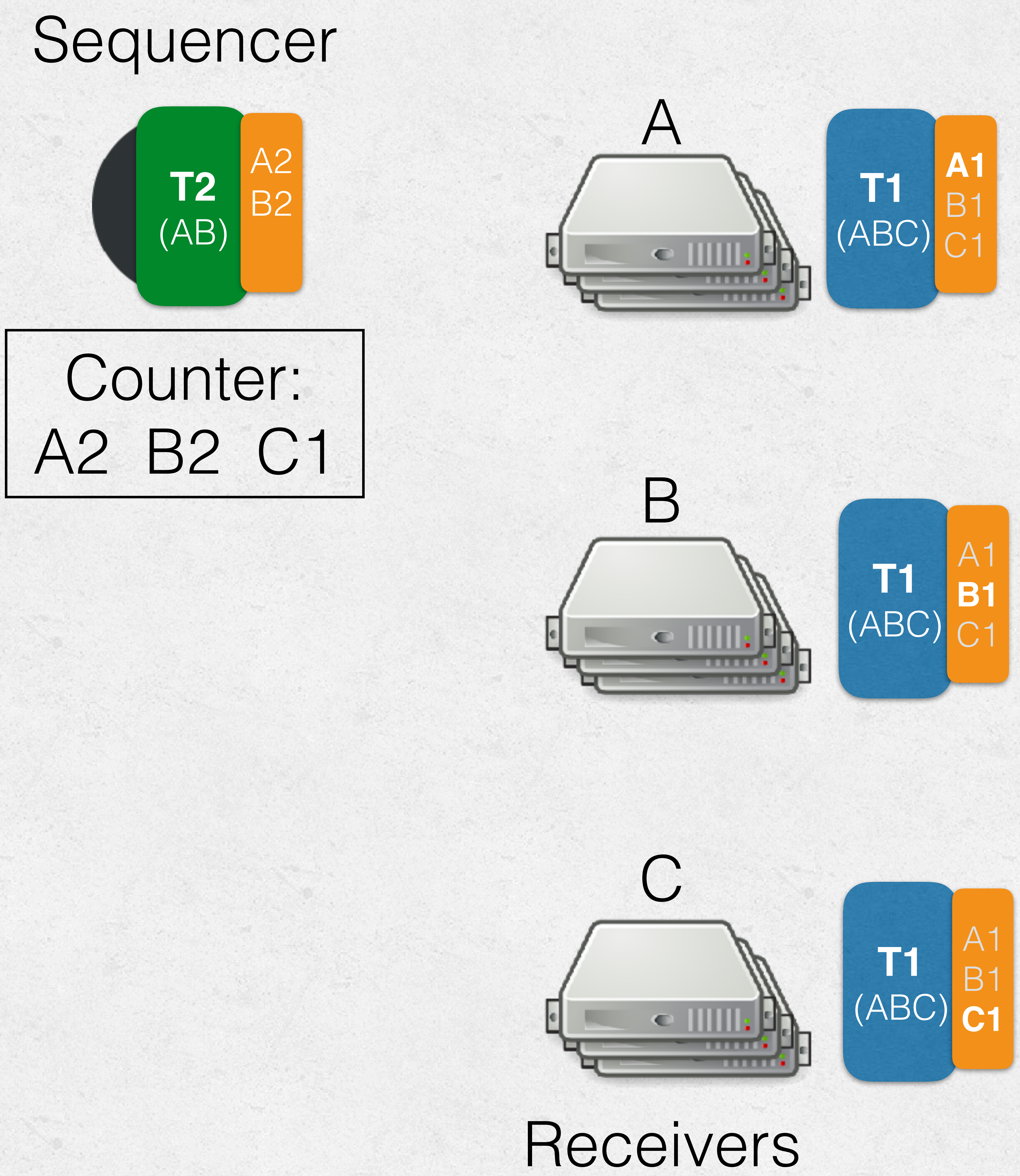
A1
B1
C1

C

T1
(ABC)

A1
B1
C1

Receivers



Sequencer

T2
(AB)
A2
B2

Counter:
A2 B2 C1

A

T1
(ABC)
A1
B1
C1

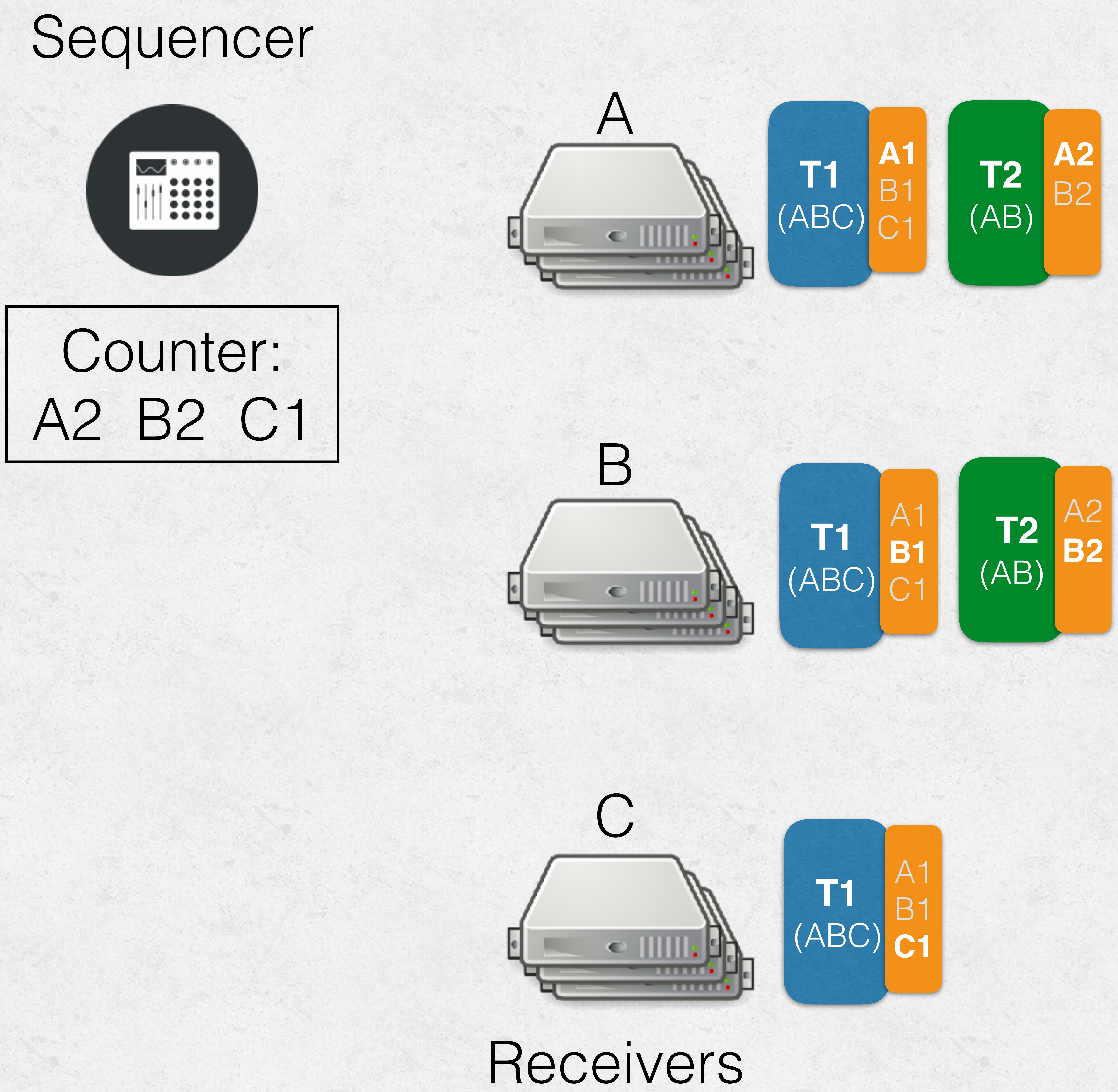
B

T1
(ABC)
A1
B1
C1

C

T1
(ABC)
A1
B1
C1

Receivers



Sequencer

Counter:
A2 B2 C1

A

T1 (ABC) | **A1** B1 C1

T2 (AB) | **A2** B2

B

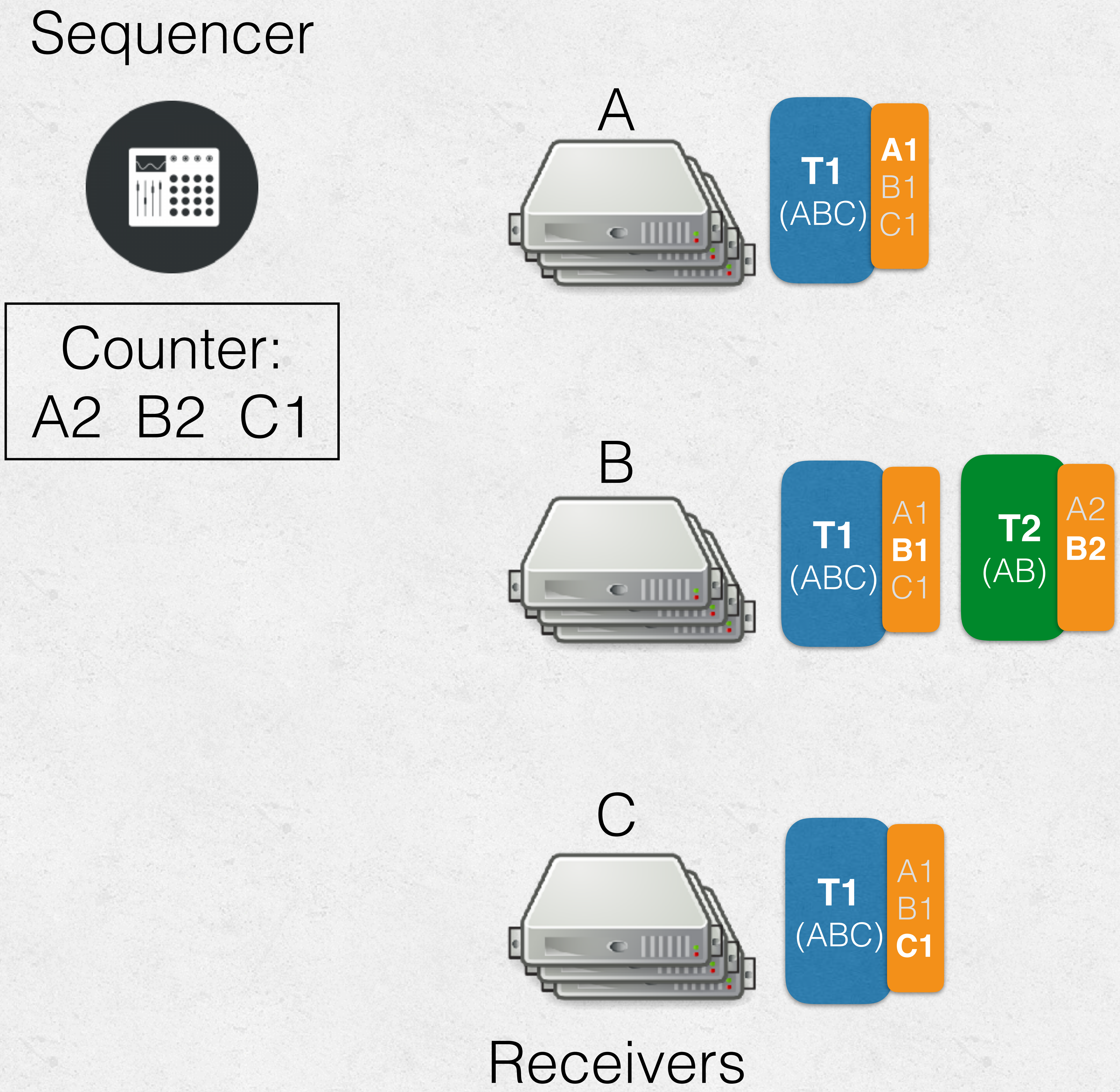
T1 (ABC) | A1 **B1** C1

T2 (AB) | A2 **B2**

C

T1 (ABC) | A1 B1 **C1**

Receivers



Sequencer

Counter:
A2 B2 C1

A

T1 (ABC) **A1** B1 C1

B

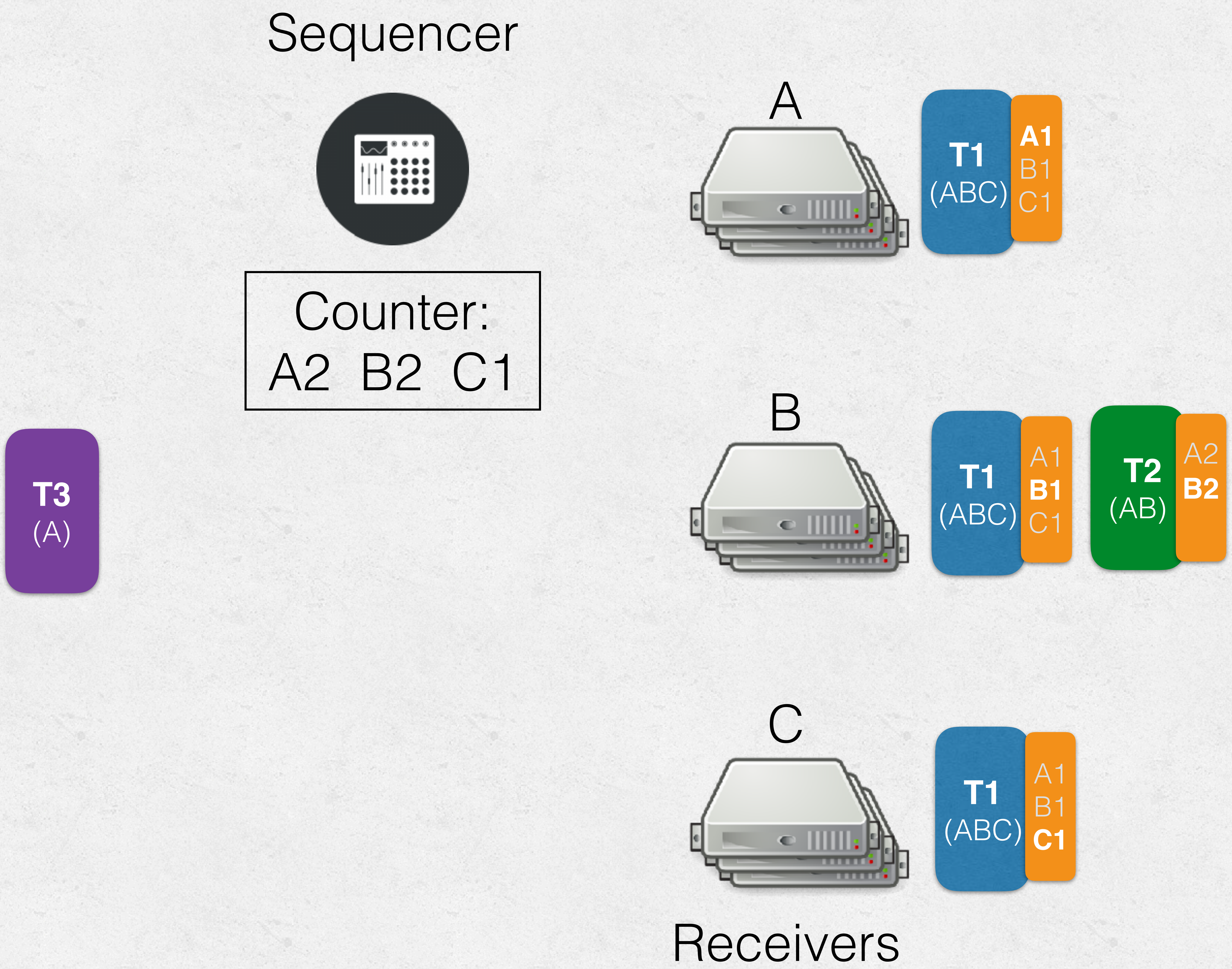
T1 (ABC) A1 **B1** C1

T2 (AB) A2 **B2**

C

T1 (ABC) A1 B1 **C1**

Receivers



Sequencer

Counter:
A2 B2 C1

T3
(A)

A

T1
(ABC)

A1
B1
C1

B

T1
(ABC)

A1
B1
C1

T2
(AB)

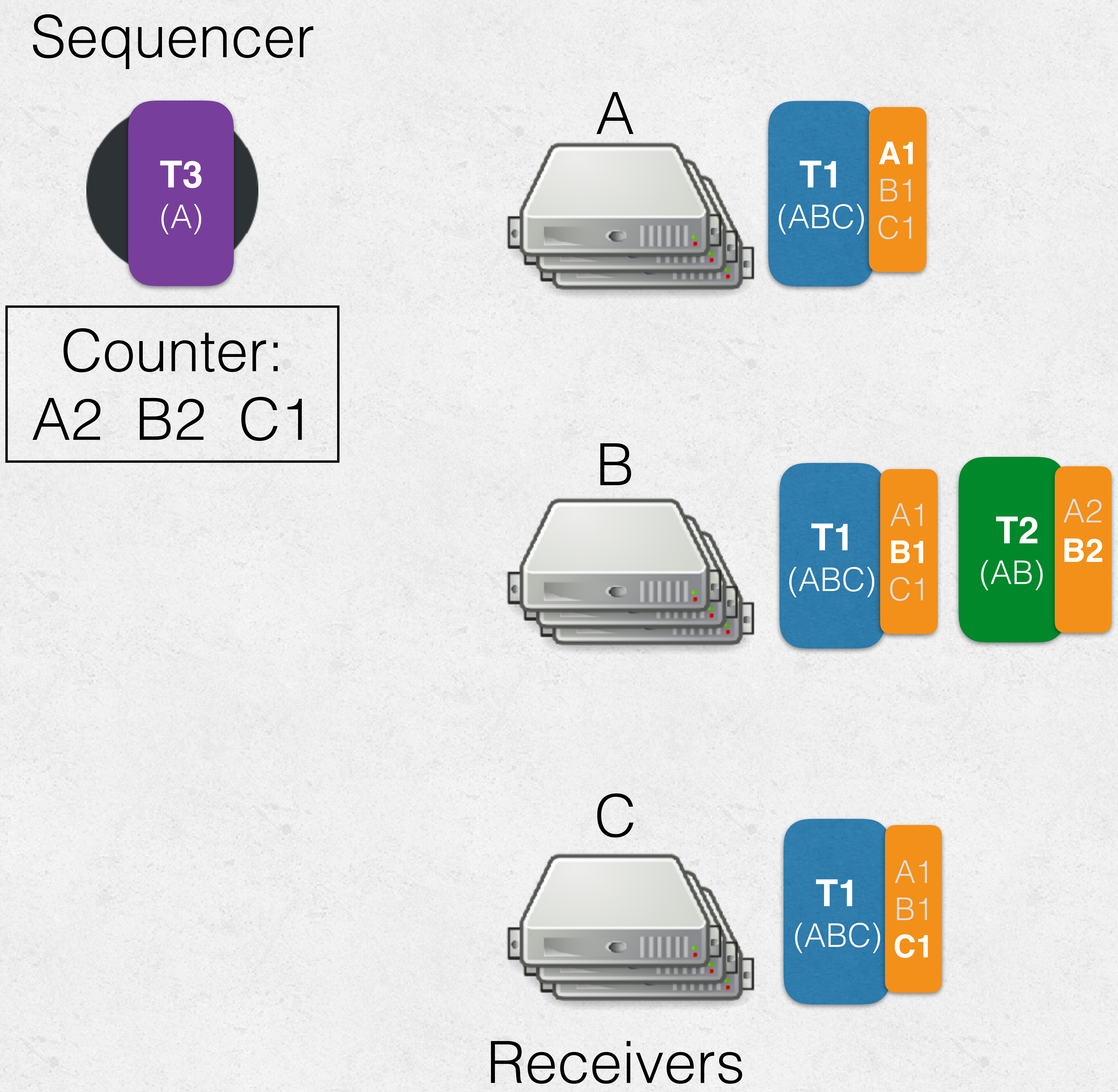
A2
B2

C

T1
(ABC)

A1
B1
C1

Receivers



Sequencer

T3
(A)

Counter:
A2 B2 C1

A

T1
(ABC)

A1
B1
C1

B

T1
(ABC)

A1
B1
C1

T2
(AB)

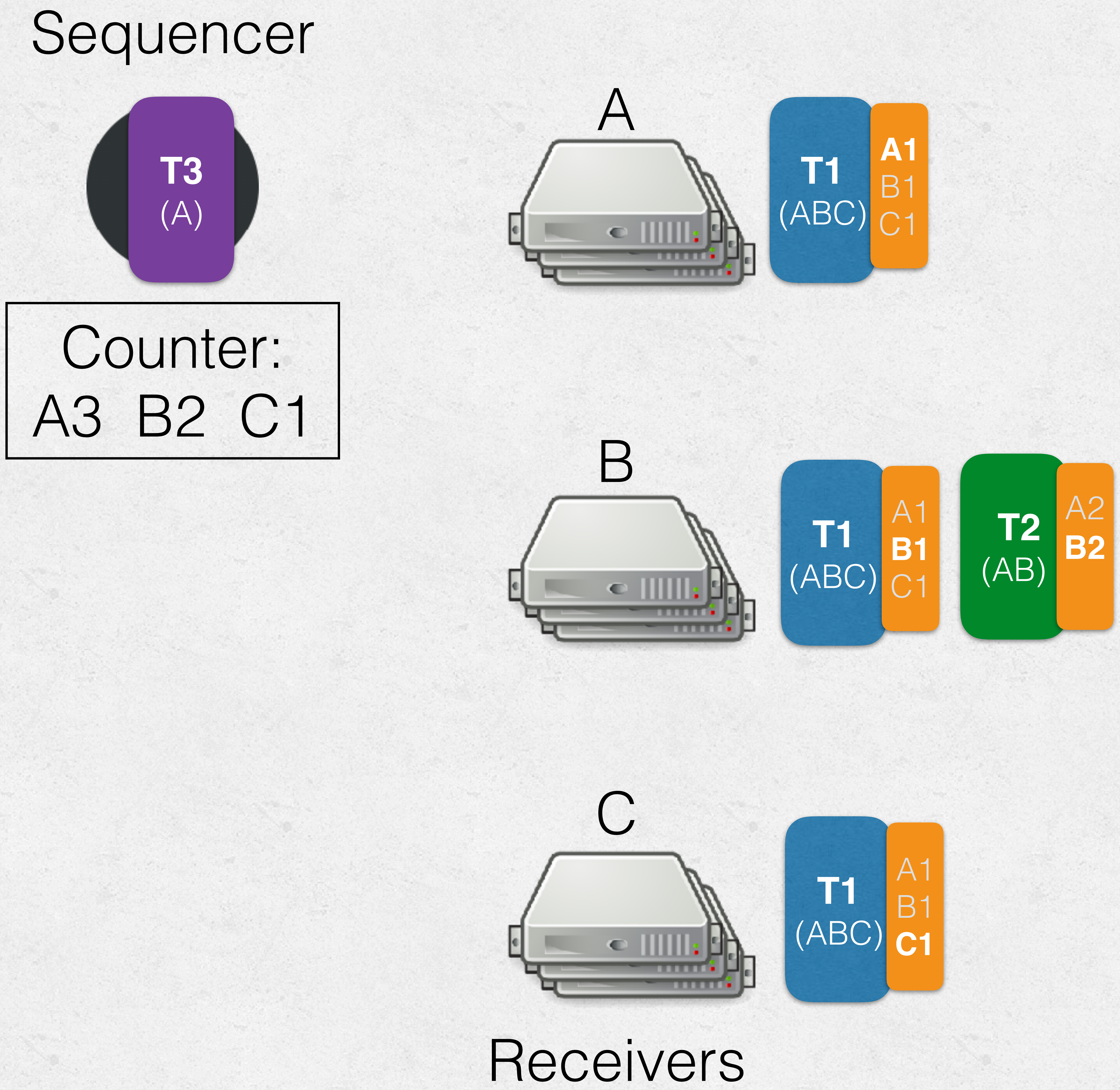
A2
B2

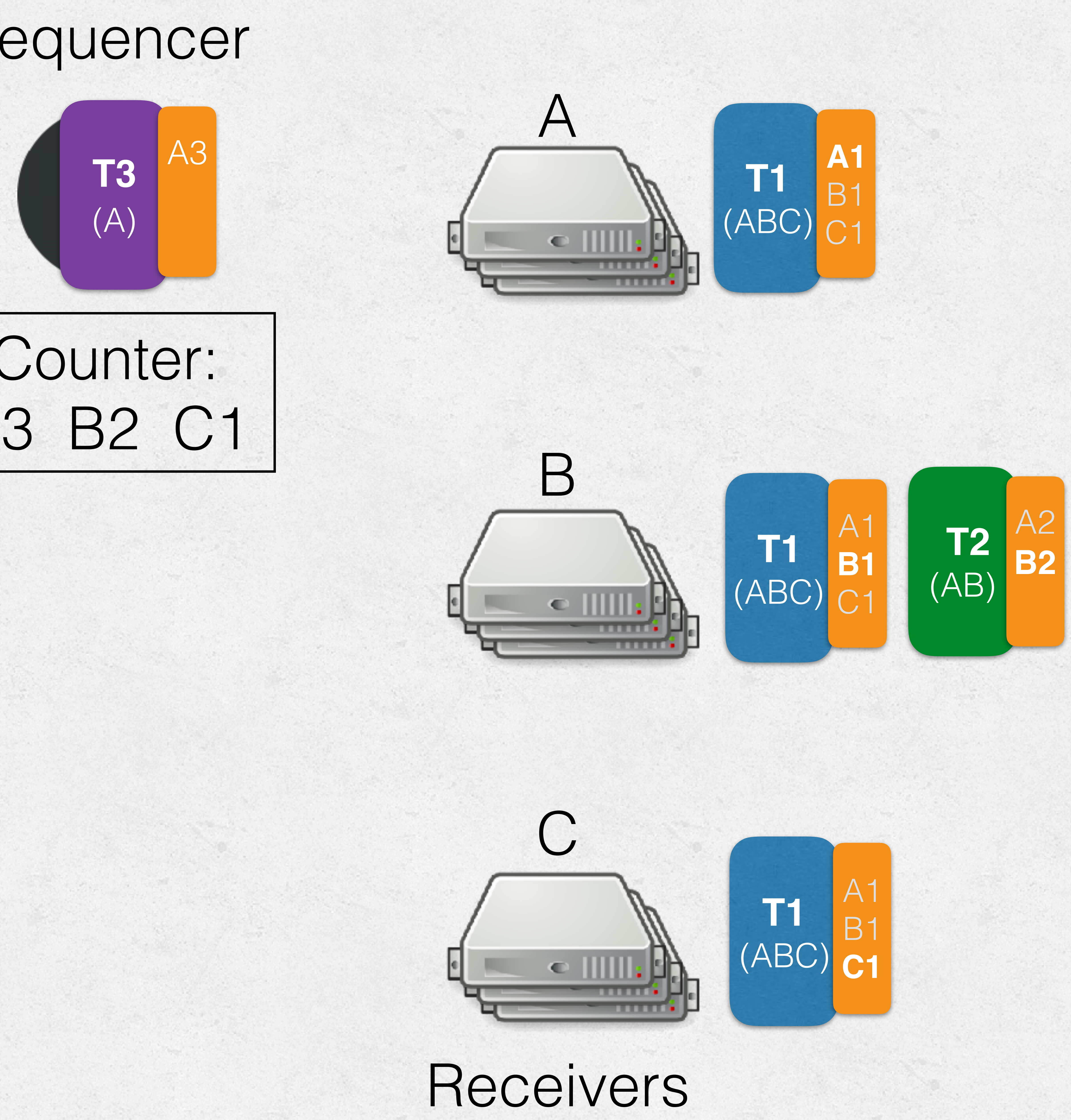
C

T1
(ABC)

A1
B1
C1

Receivers





Sequencer

T3
(A)
A3

Counter:
A3 B2 C1

A

T1
(ABC)
A1
B1
C1

B

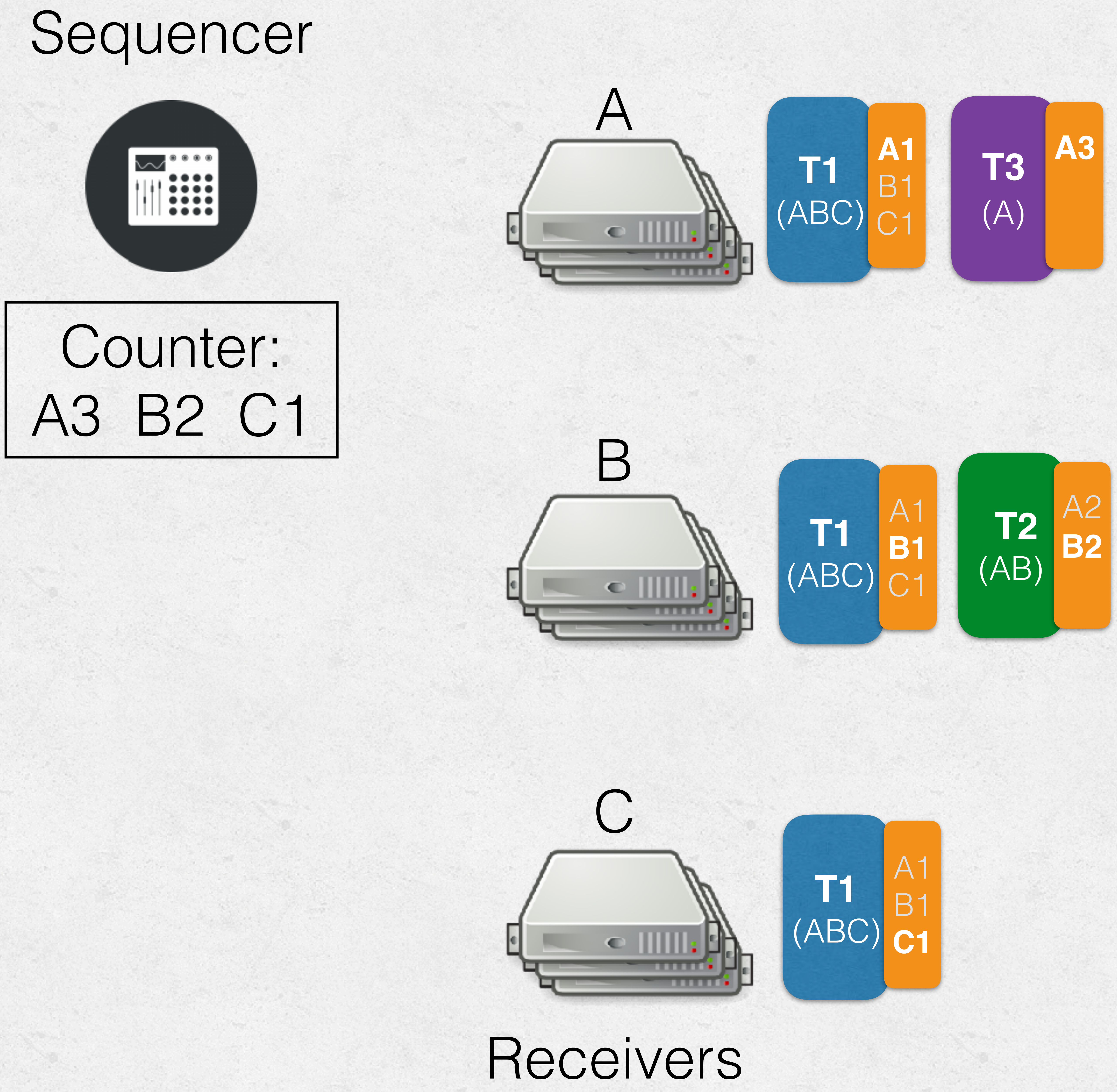
T1
(ABC)
A1
B1
C1

T2
(AB)
A2
B2

C

T1
(ABC)
A1
B1
C1

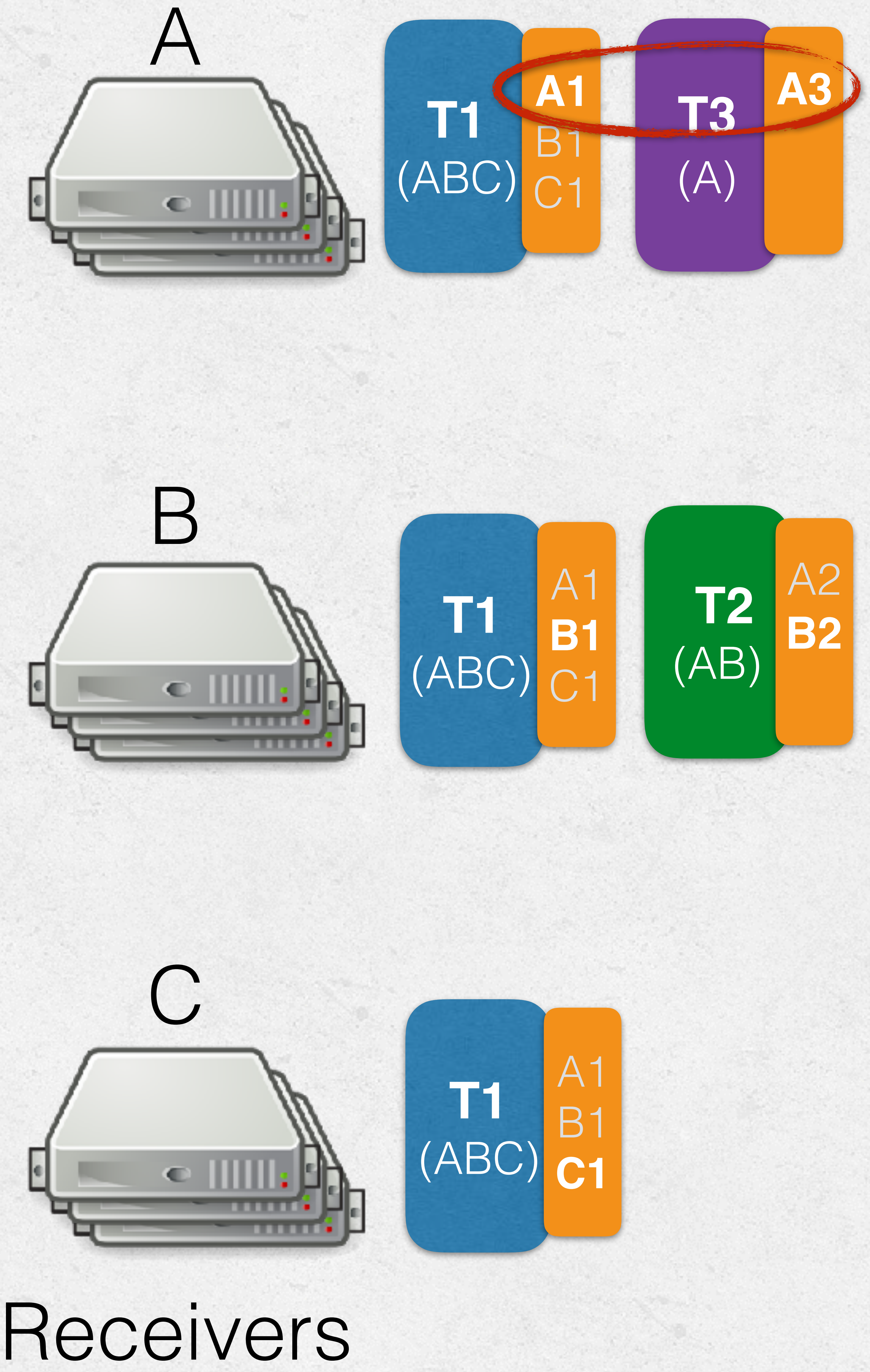
Receivers

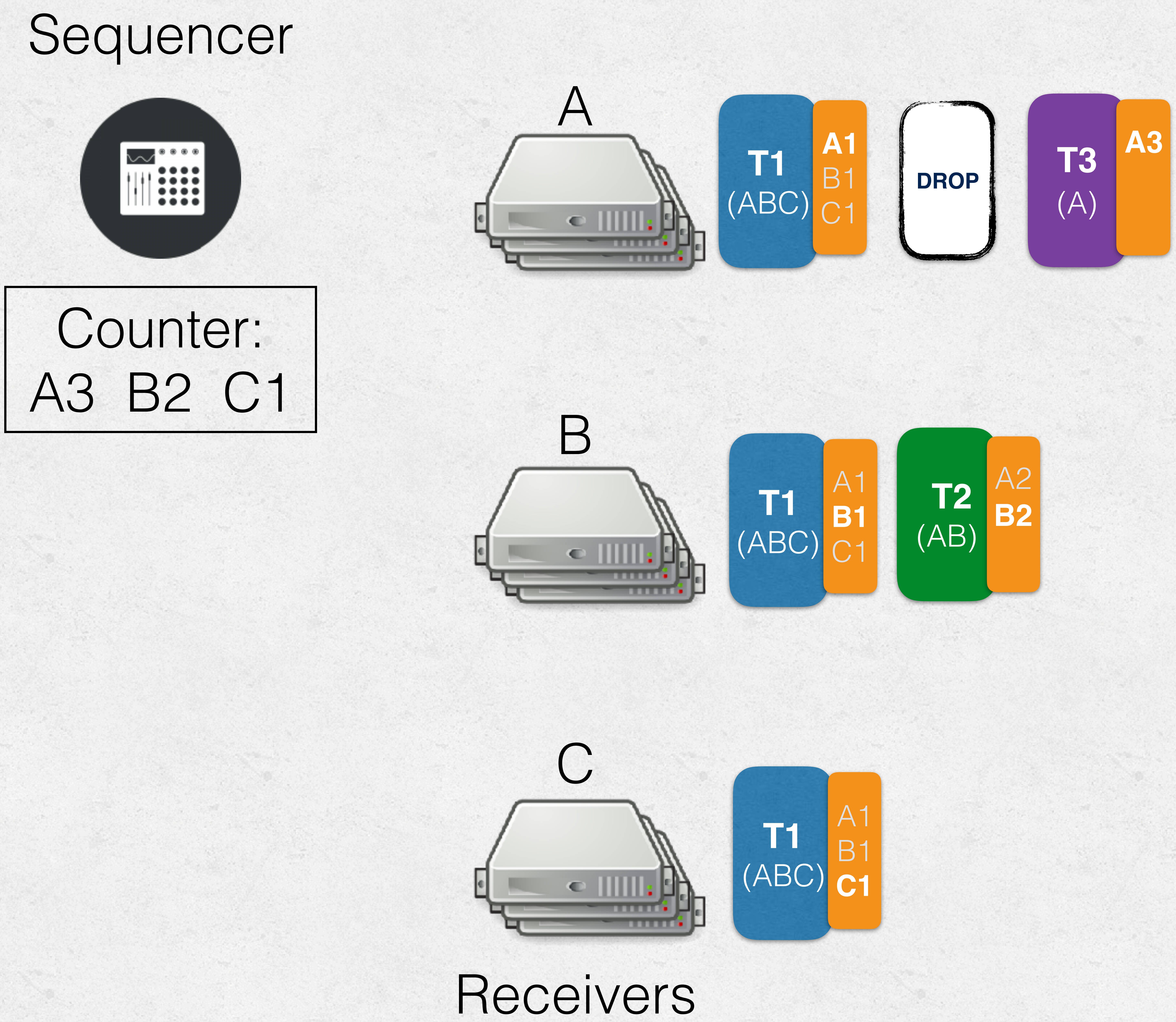


Sequencer
Counter:
A3 B2 C1
A
T1
(ABC)
A1
B1
C1
T3
(A)
A3
B
T1
(ABC)
A1
B1
C1
T2
(AB)
A2
B2
C
T1
(ABC)
A1
B1
C1
Receivers

Sequencer

Counter:
A3 B2 C1





What have we accomplished so far?

- Consistently ordered groupcast primitive with drop detection

- How do we go from multi-sequenced groupcast to transactions?

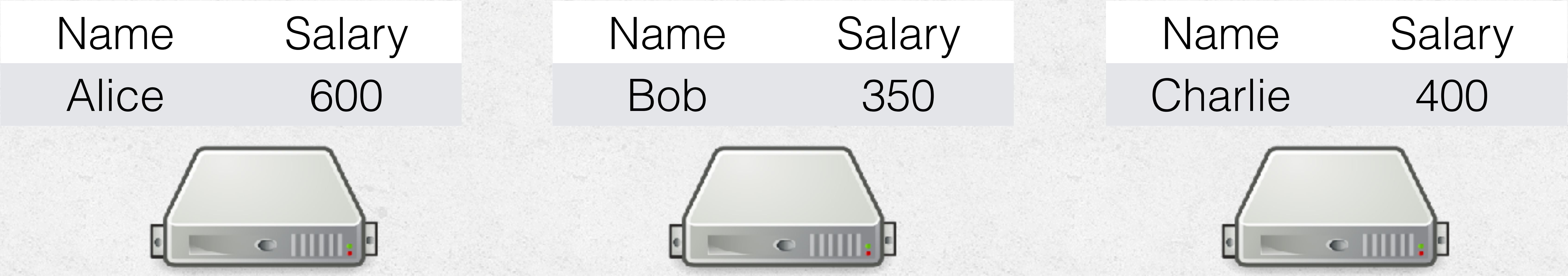
Transaction Model

Eris supports two types of transactions

- **Independent transactions**:
  - One-shot (stored procedures)
  - No cross-shard dependencies
  - Proposed by H-Store [VLDB ’07] and Granola [ATC ’12]
- Fully general transactions

INDEPENDENT TRANSACTION

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```



Name	Salary
Alice	600

Name	Salary
Bob	350

Name	Salary
Charlie	400

Independent Transaction

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Alice	600

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Bob	350

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Charlie	400

Independent Transaction

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Alice	600

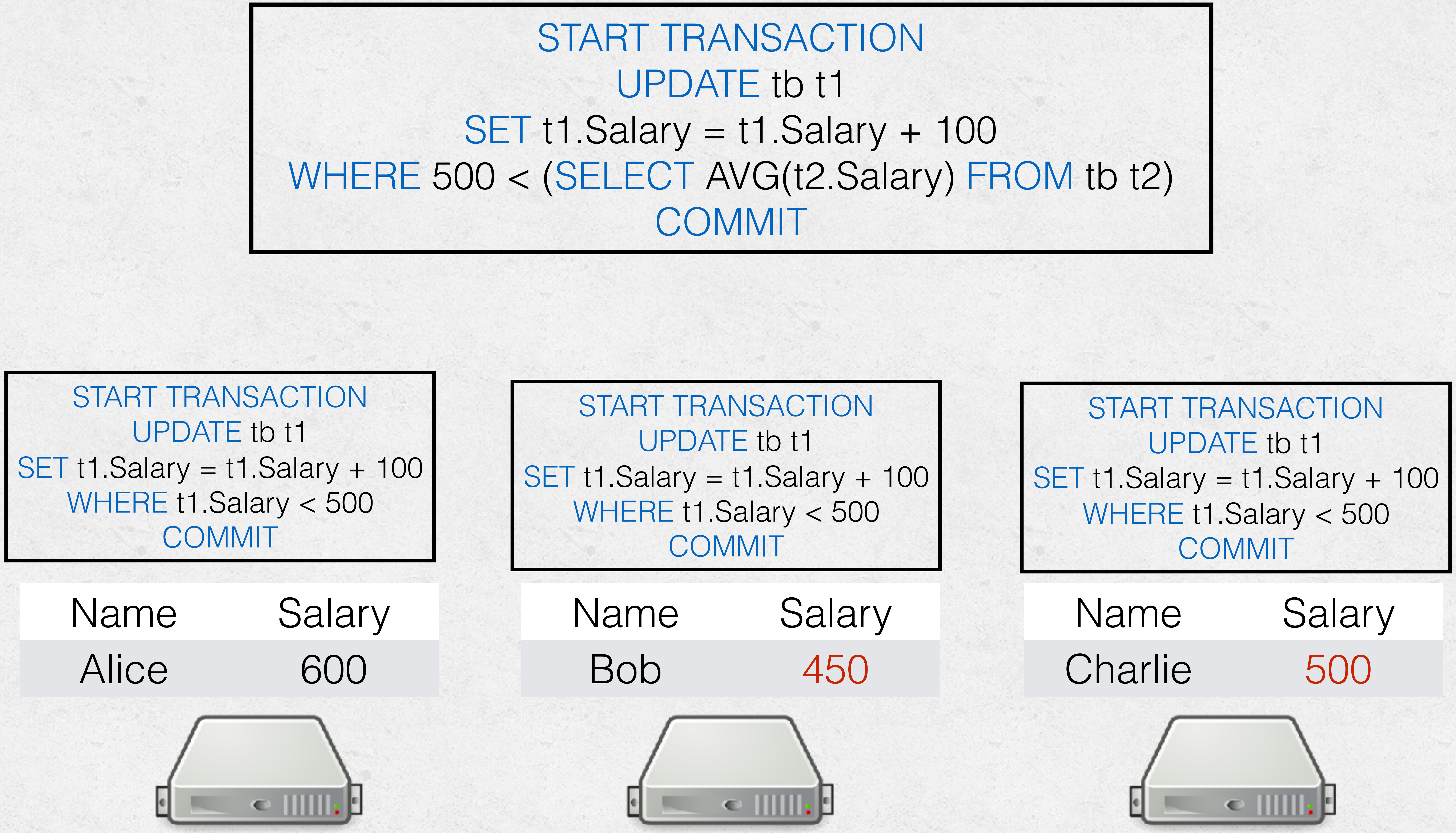
```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Bob	450

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Charlie	500

Independent Transaction



```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE 500 < (SELECT AVG(t2.Salary) FROM tb t2)
COMMIT
```

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Alice	600

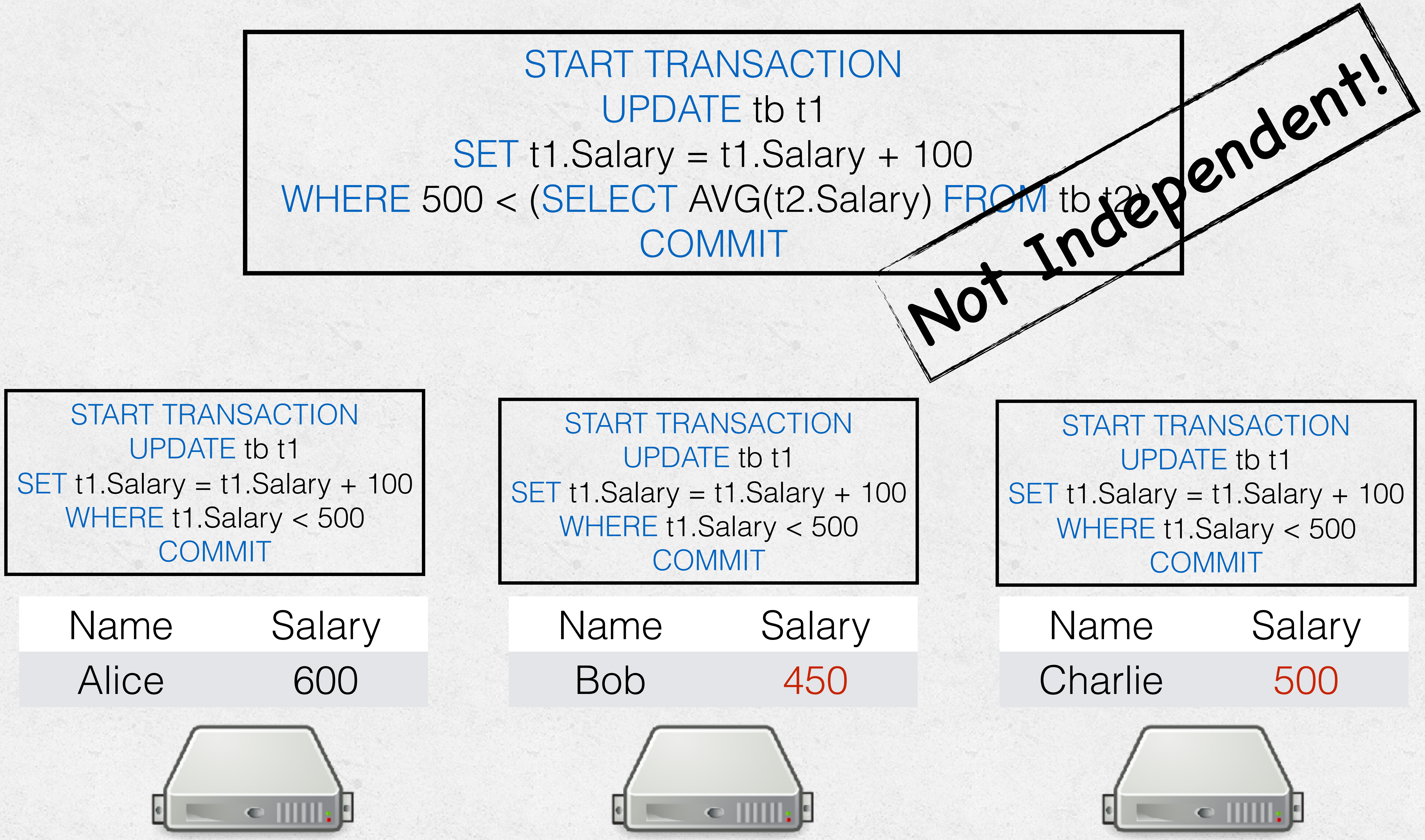
```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Bob	450

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Charlie	500

Independent Transaction



```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE 500 < (SELECT AVG(t2.Salary) FROM tb t2)
COMMIT
```

Not Independent!

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Alice	600

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Bob	450

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Charlie	500

Independent Transaction

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Alice	600

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

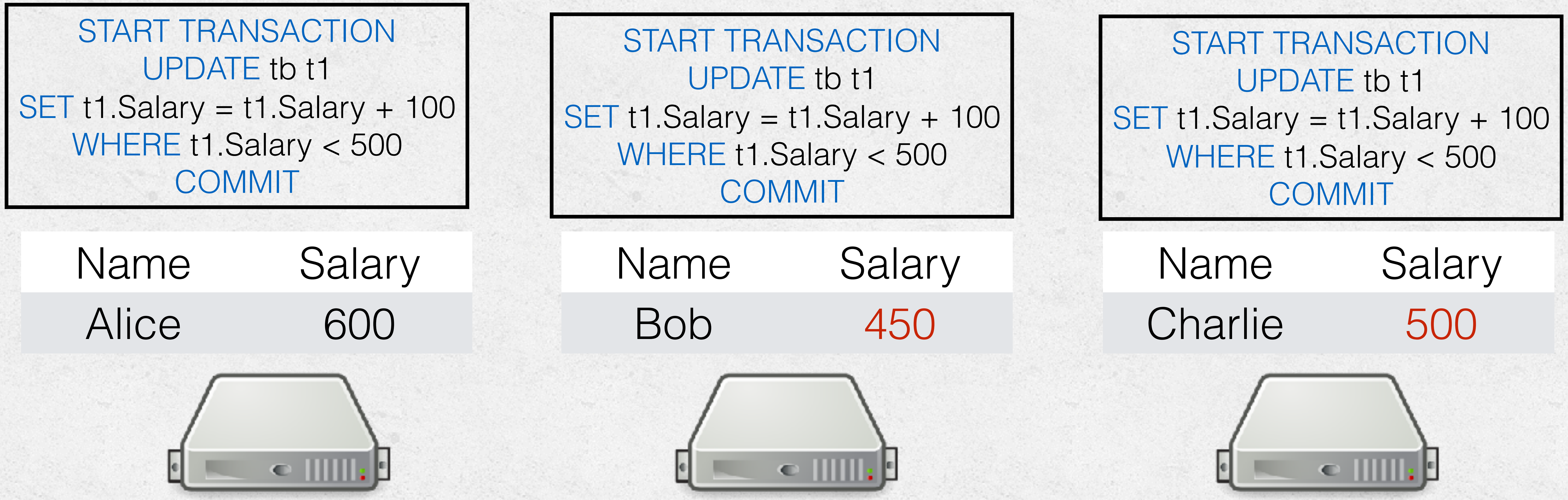
Name	Salary
Bob	450

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Charlie	500

Independent Transaction

Many applications consist **entirely** of independent transactions



```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Alice	600

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Bob	450

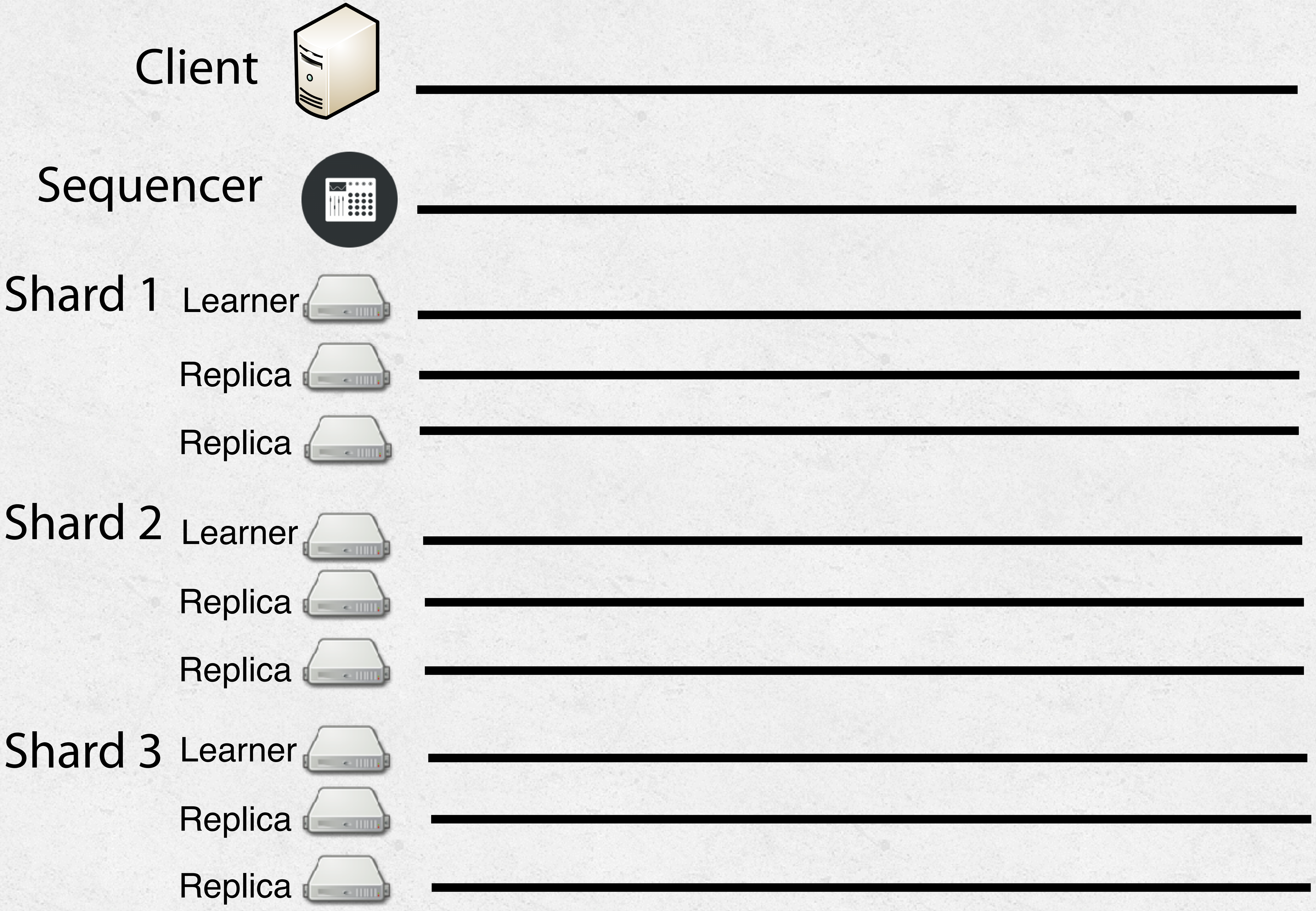
```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Charlie	500

Why independent transactions?

- **No** coordination/communication across shards
- Executing them serially at each shard in a **consistent order** guarantees **serializability**
- Multi-sequenced groupcast establishes such an order
- How to handle **message drops** and sequencer/ server **failures**?

NORMAL CASE



Client

Sequencer

Shard 1 Learner

Replica

Replica

Shard 2 Learner

Replica

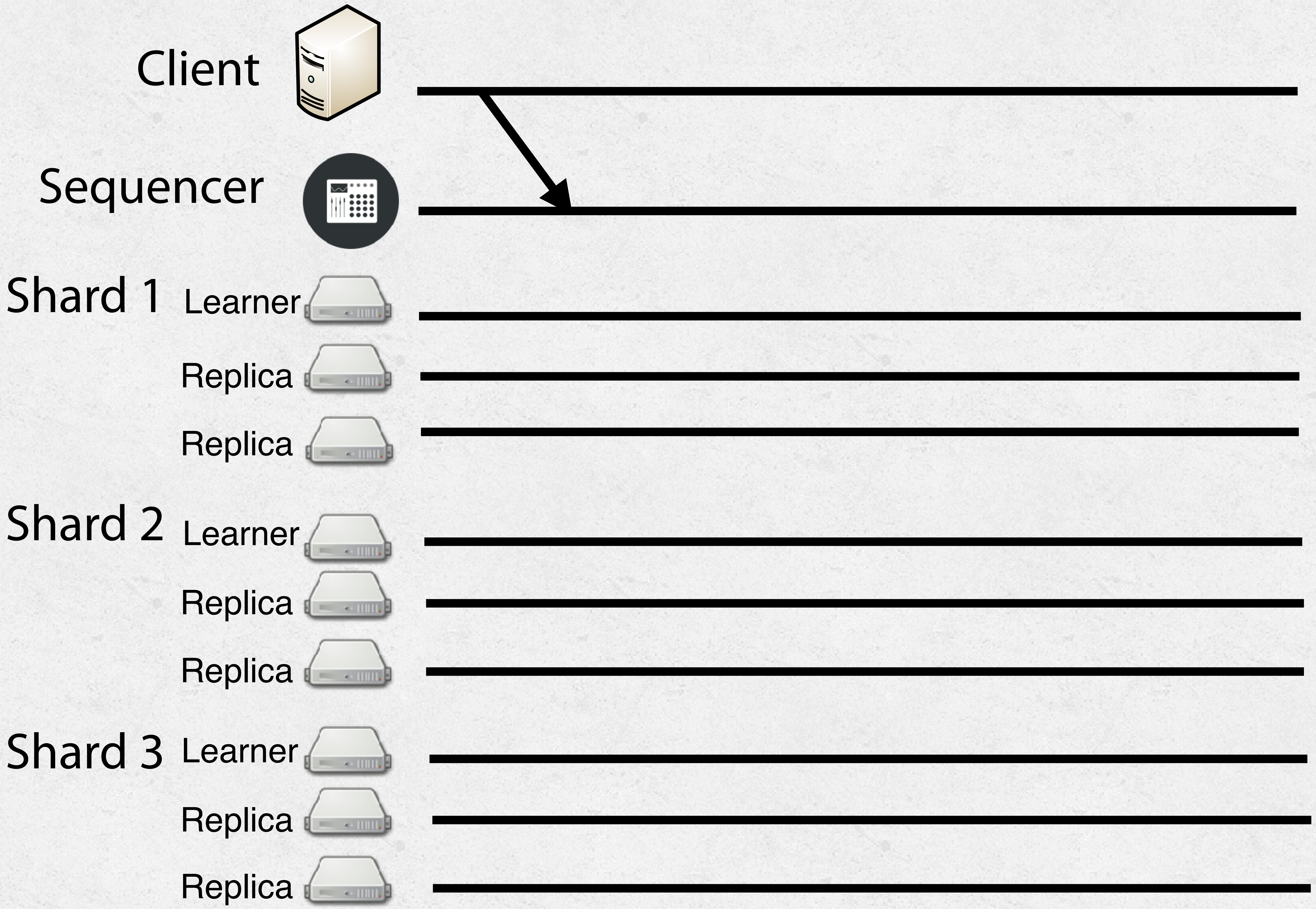
Replica

Shard 3 Learner

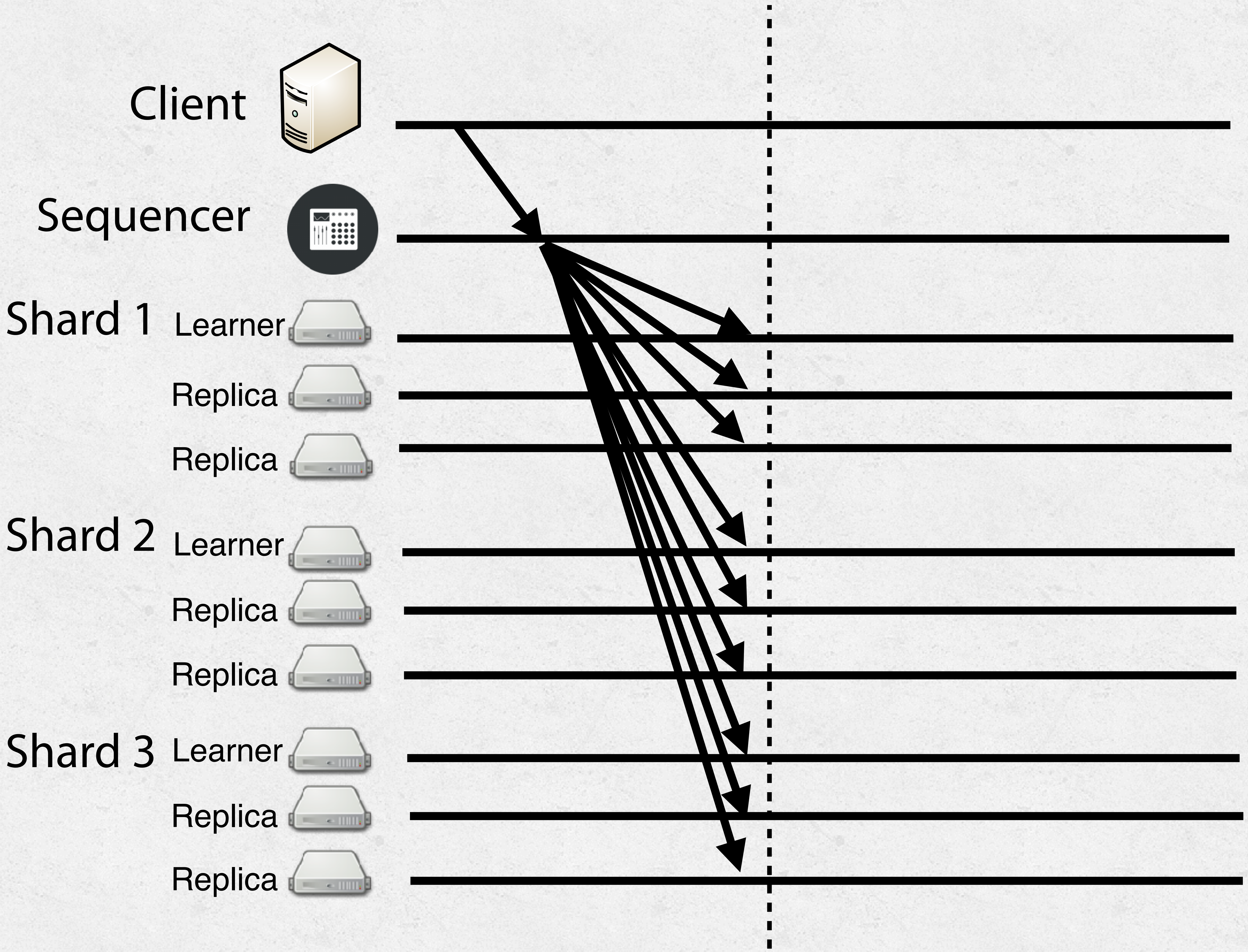
Replica

Replica

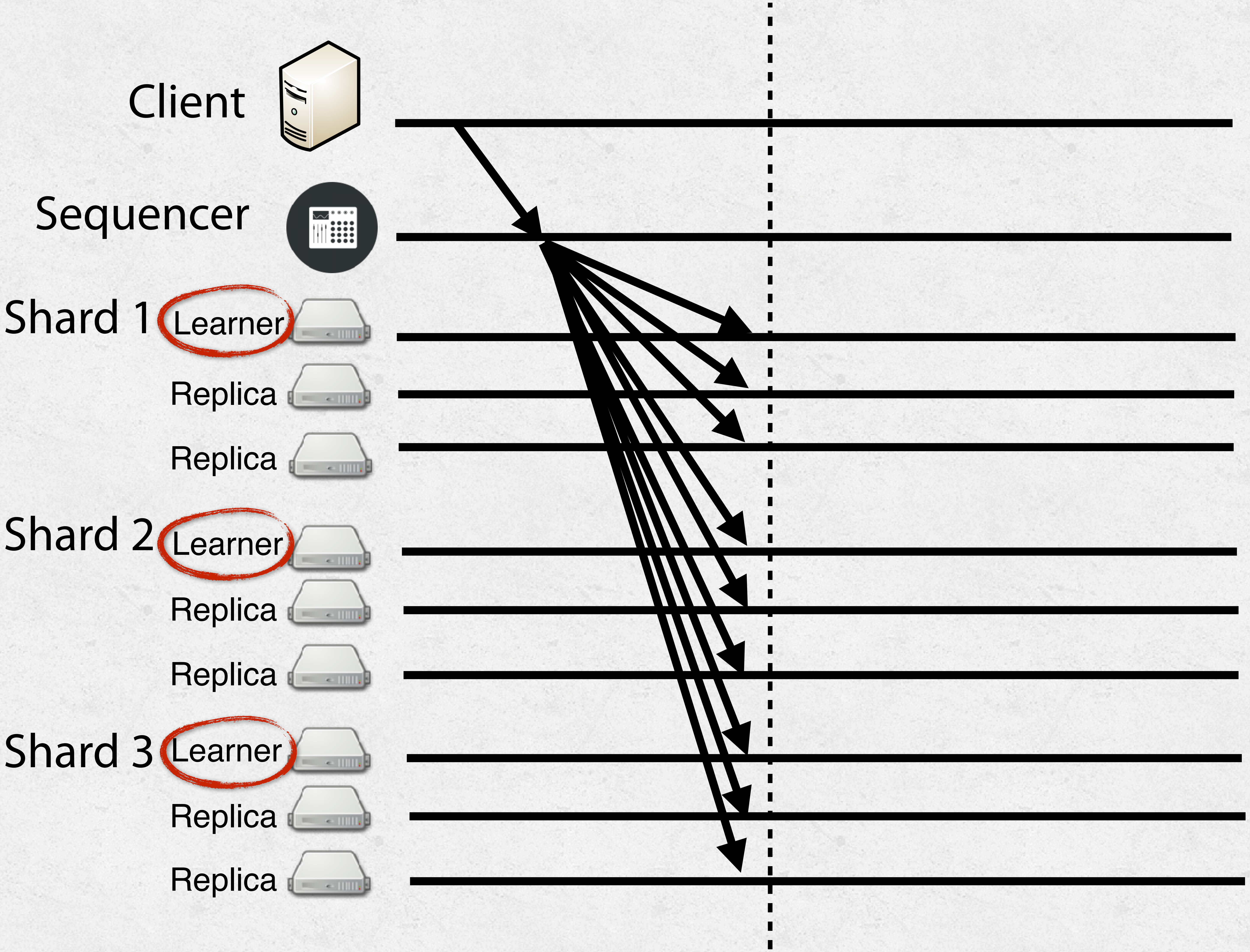
Normal Case


Client
Sequencer
Shard 1
Learner
Replica
Replica
Shard 2
Learner
Replica
Replica
Shard 3
Learner
Replica
Replica

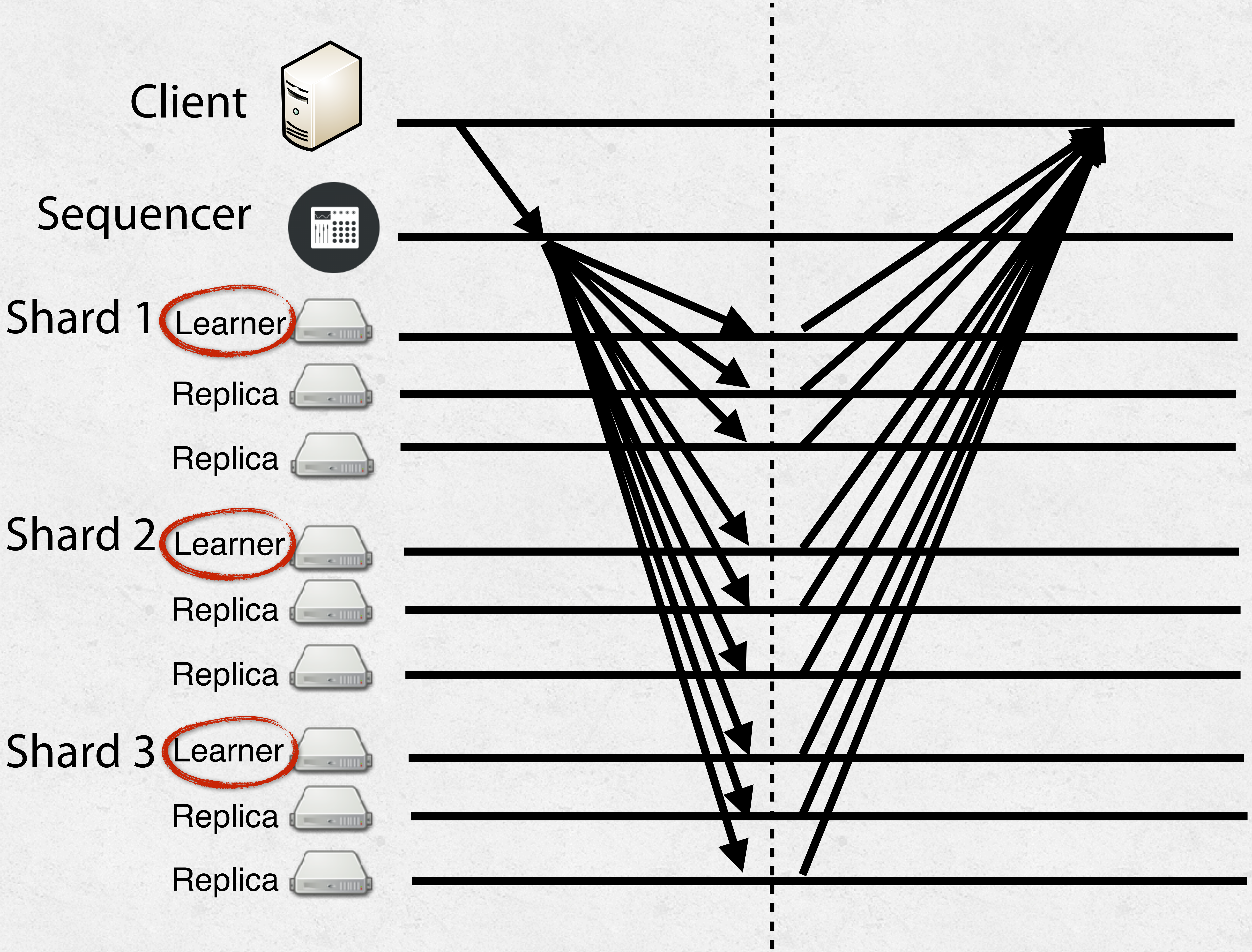
Normal Case


Client
Sequencer
Shard 1
Learner
Replica
Replica
Shard 2
Learner
Replica
Replica
Shard 3
Learner
Replica
Replica

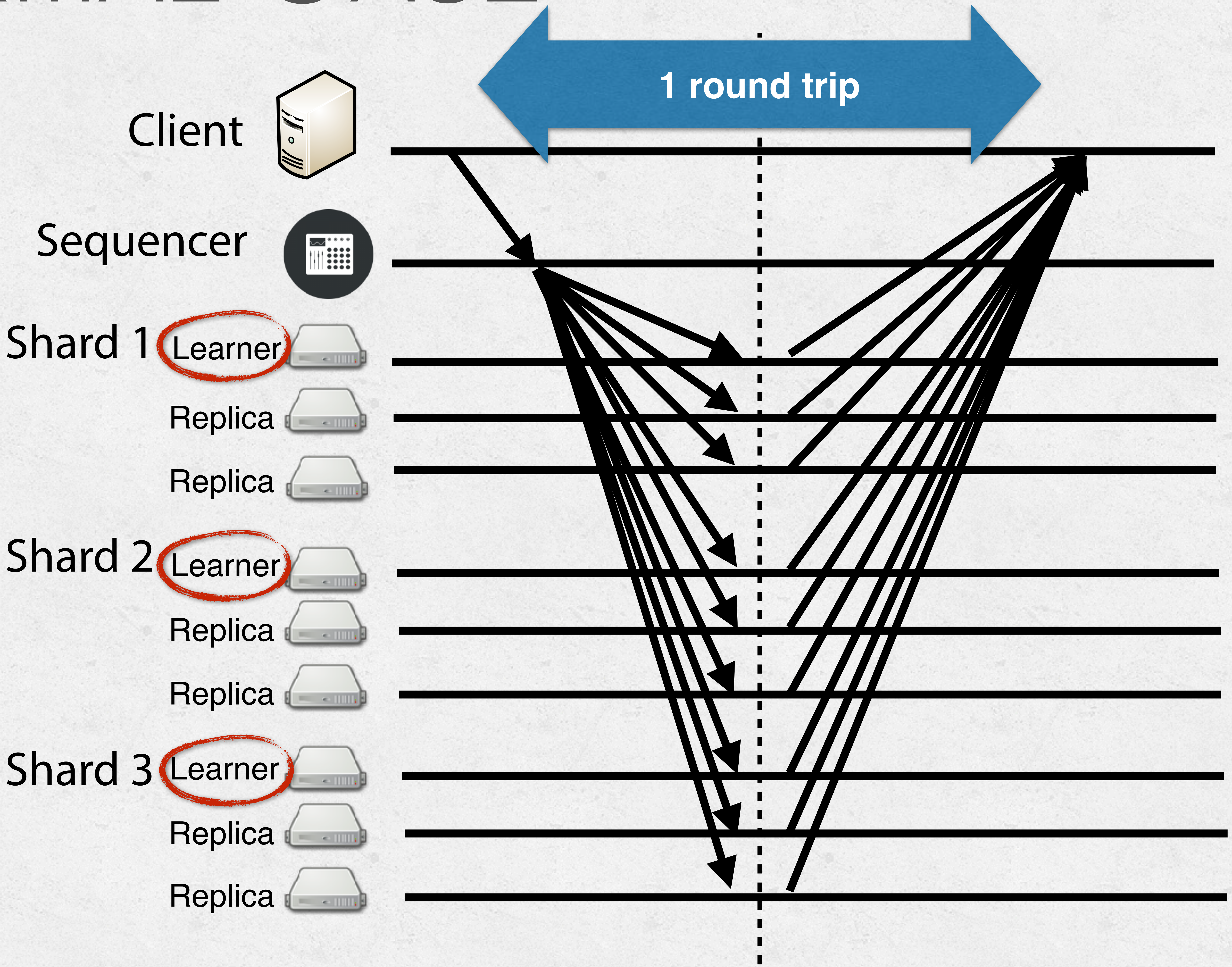
Normal Case


Client
Sequencer
Shard 1
Learner
Replica
Replica
Shard 2
Learner
Replica
Replica
Shard 3
Learner
Replica
Replica

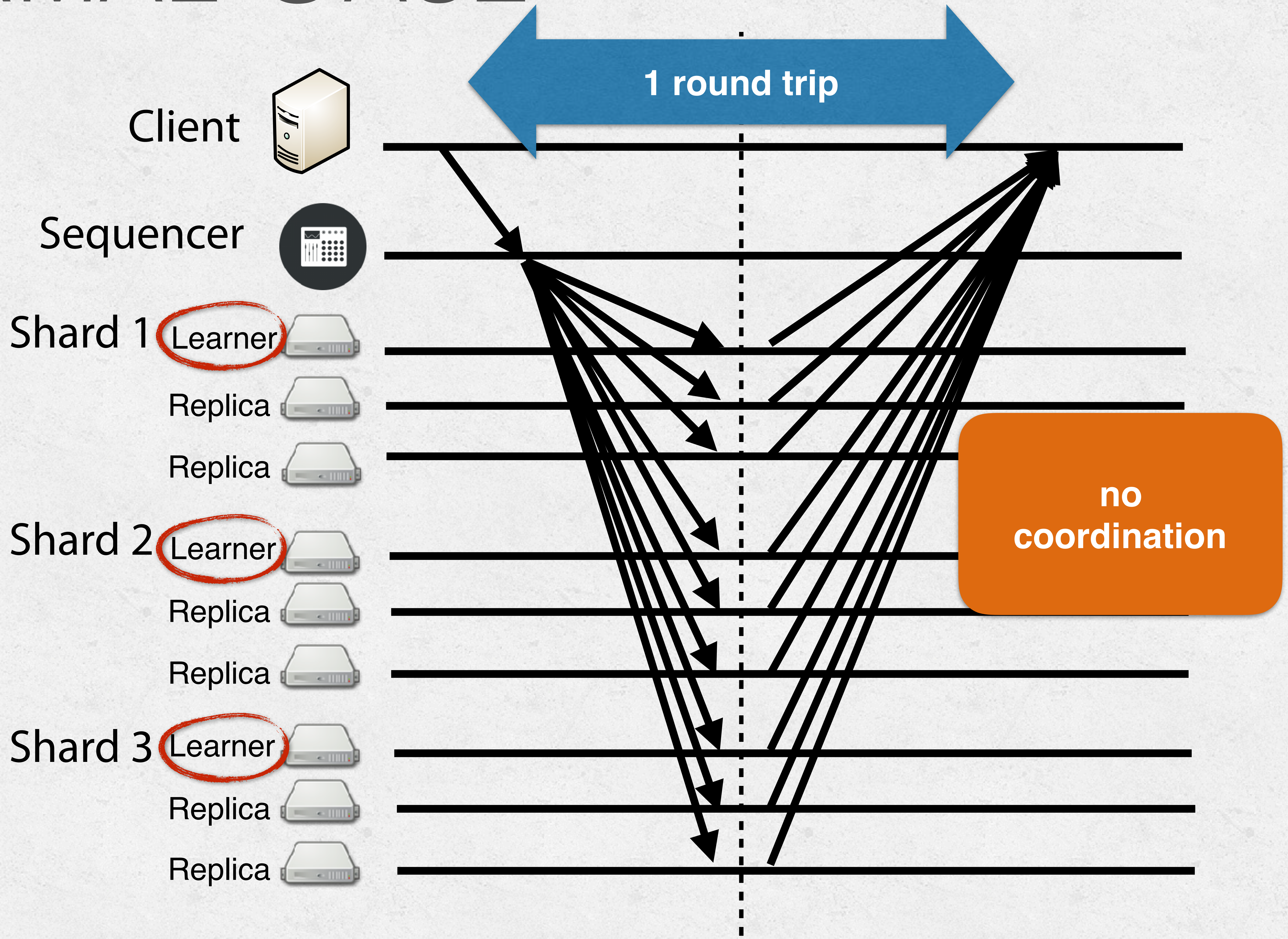
Normal Case


Client
Sequencer
Shard 1
Learner
Replica
Replica
Shard 2
Learner
Replica
Replica
Shard 3
Learner
Replica
Replica

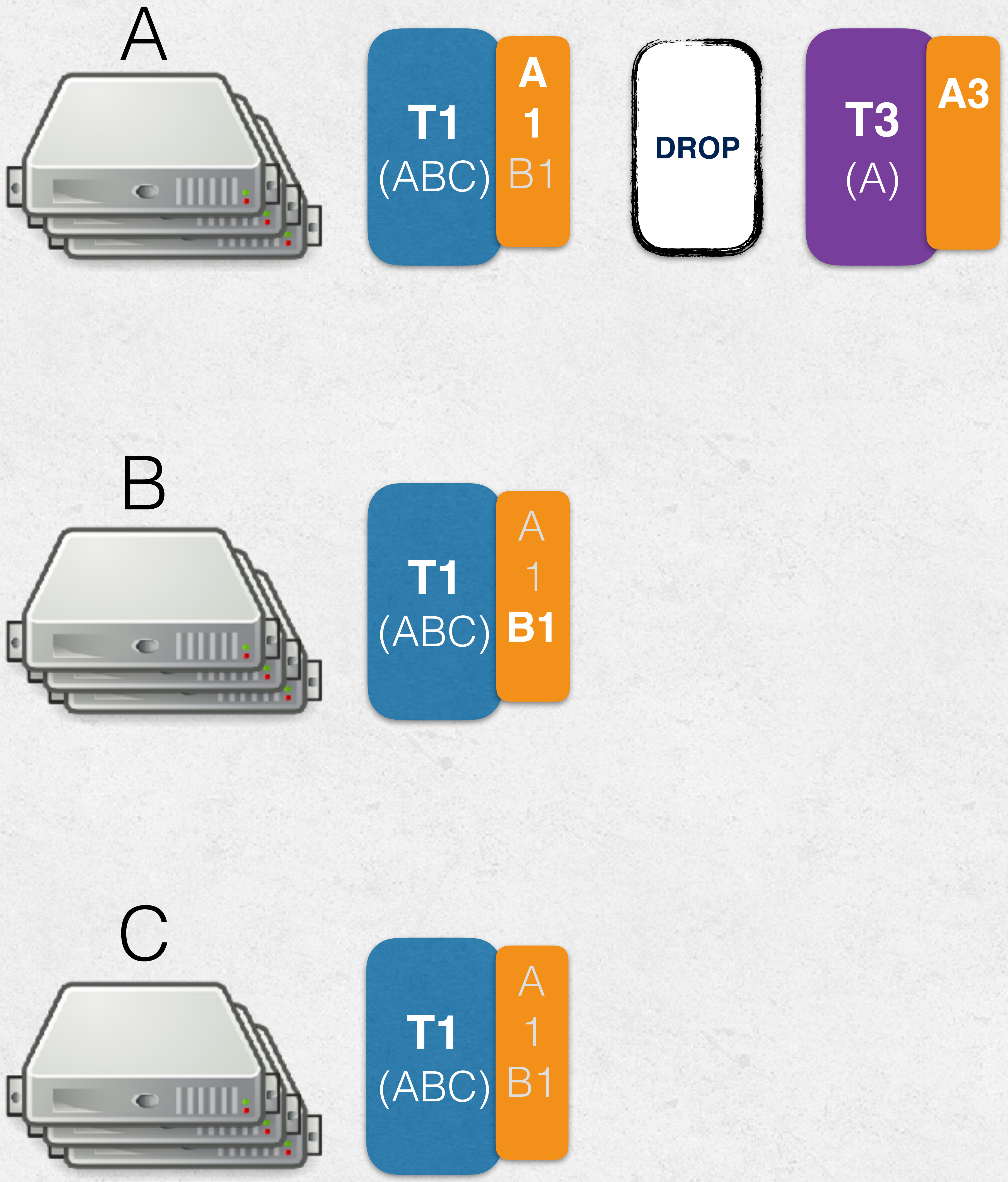
Normal Case


1 round trip
Client
Sequencer
Shard 1
Learner
Replica
Replica
Shard 2
Learner
Replica
Replica
Shard 3
Learner
Replica
Replica

Normal Case


1 round trip
Client
Sequencer
Shard 1
Learner
Replica
Replica
Shard 2
Learner
Replica
Replica
Shard 3
Learner
Replica
Replica
no coordination

How to handle dropped messages?



A

T1 (ABC) A 1 B1

DROP

T3 (A) A3

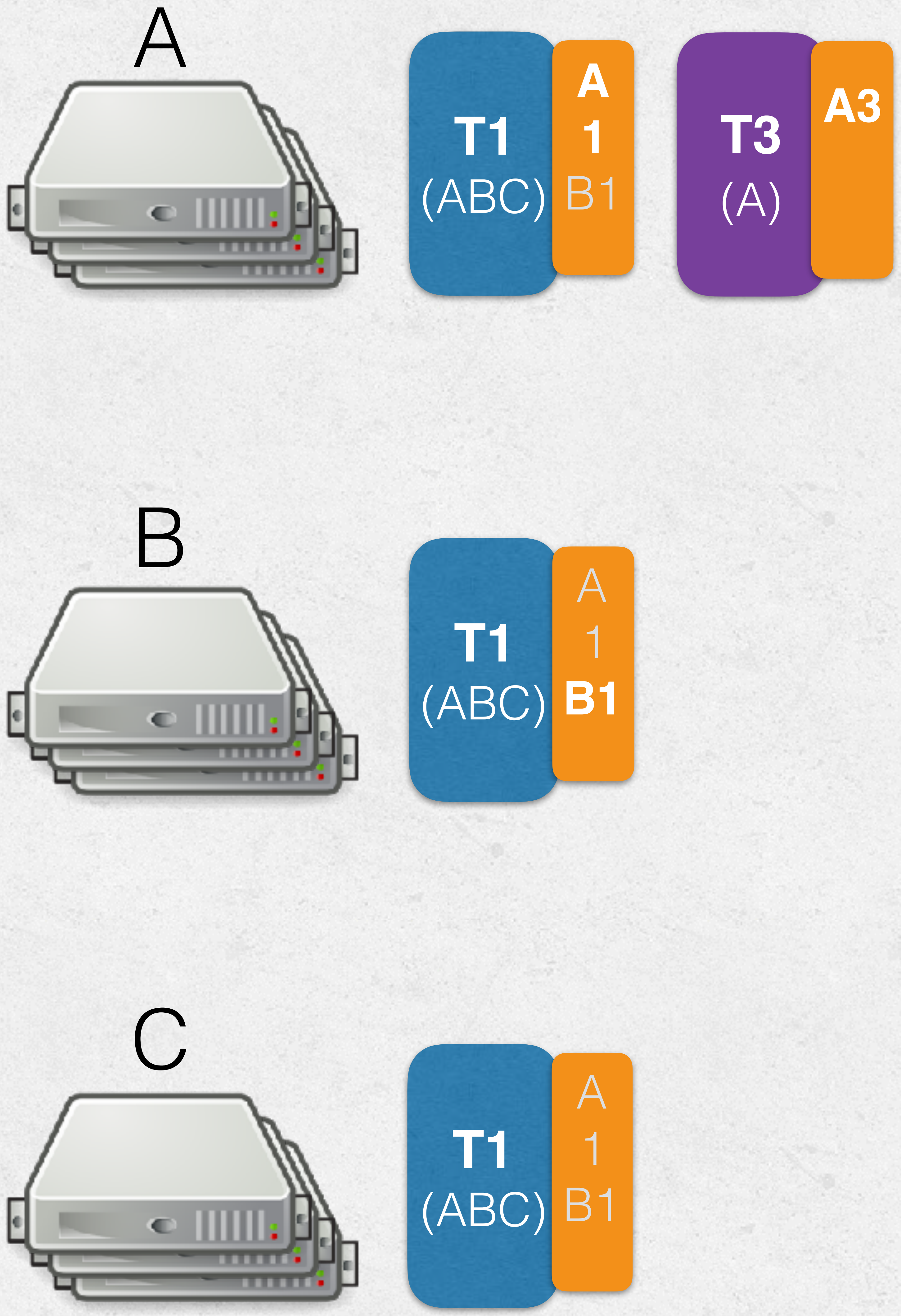
B

T1 (ABC) A 1 B1

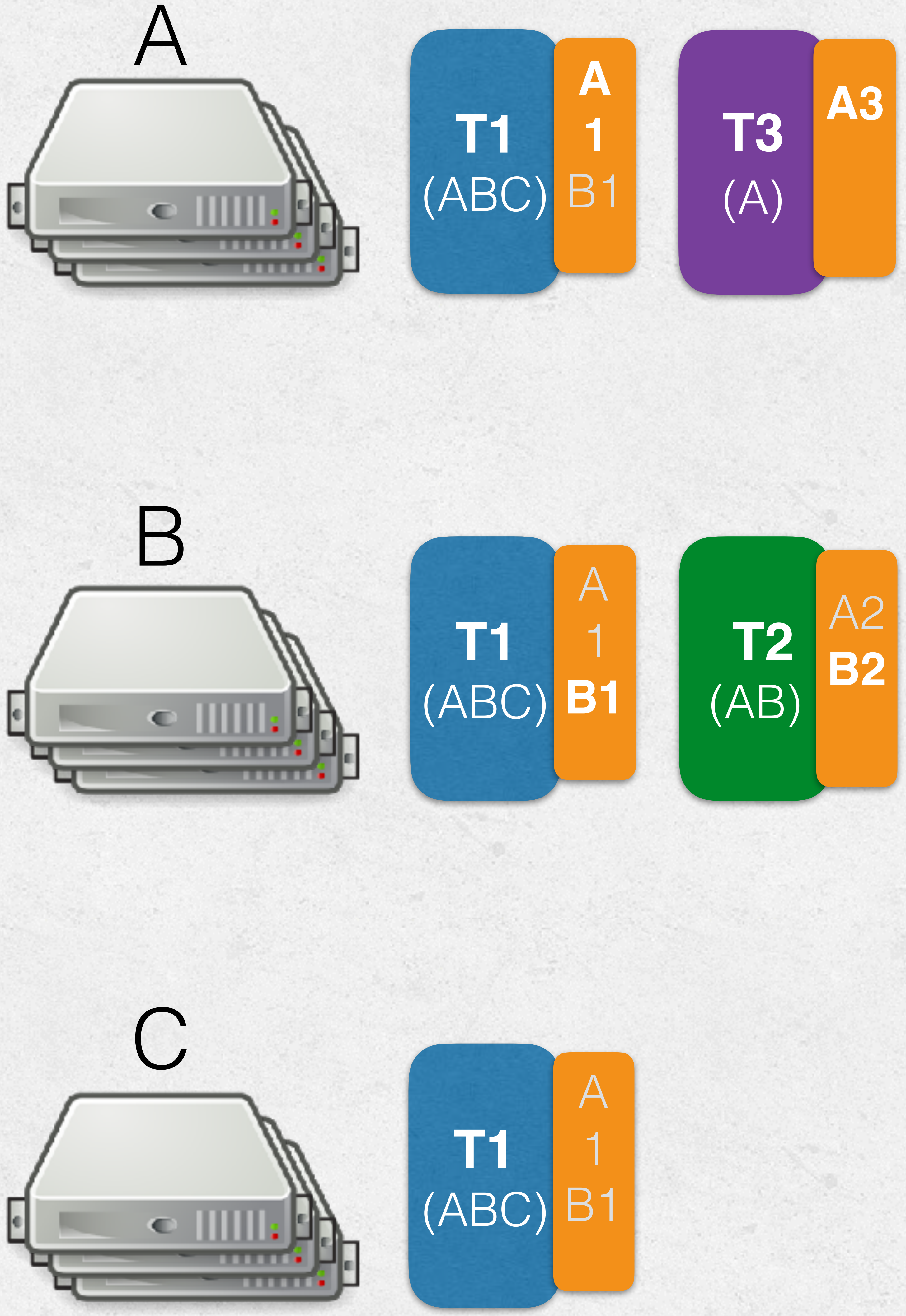
C

T1 (ABC) A 1 B1

How to handle dropped messages?



How to handle dropped messages?



A

T1 (ABC) **A** **1** B1

T3 (A) **A3**

B

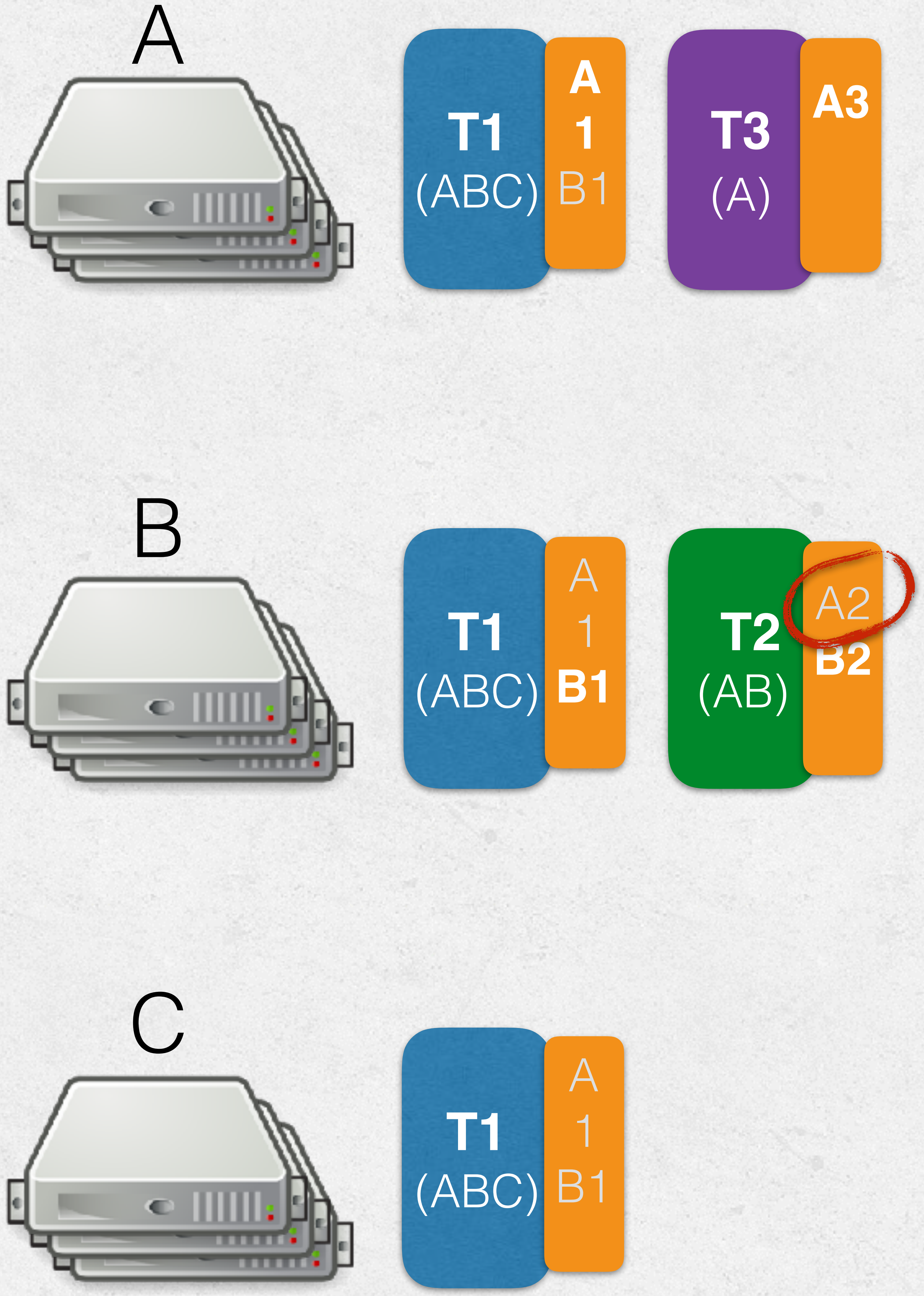
T1 (ABC) A 1 **B1**

T2 (AB) A2 **B2**

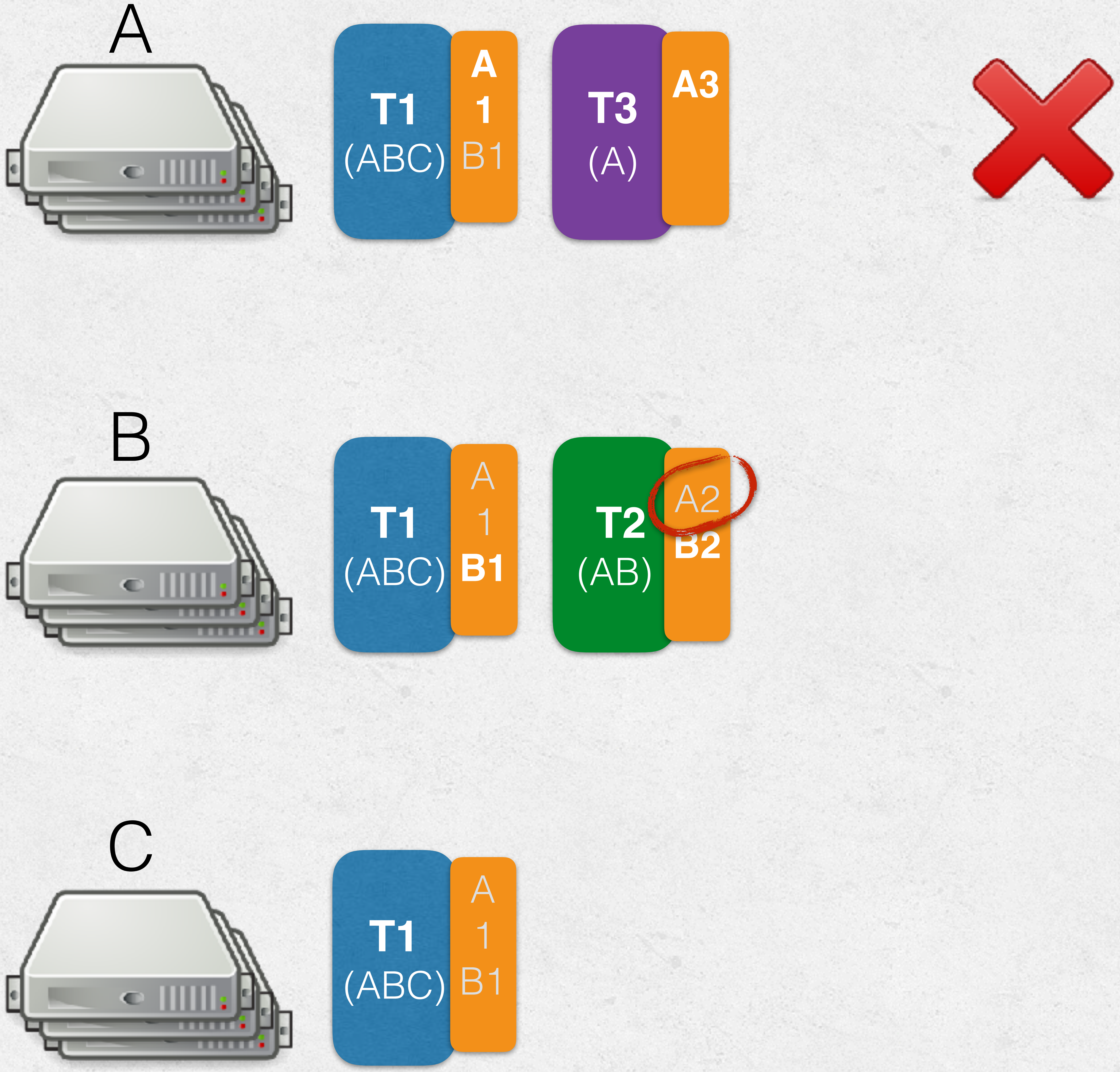
C

T1 (ABC) A 1 B1

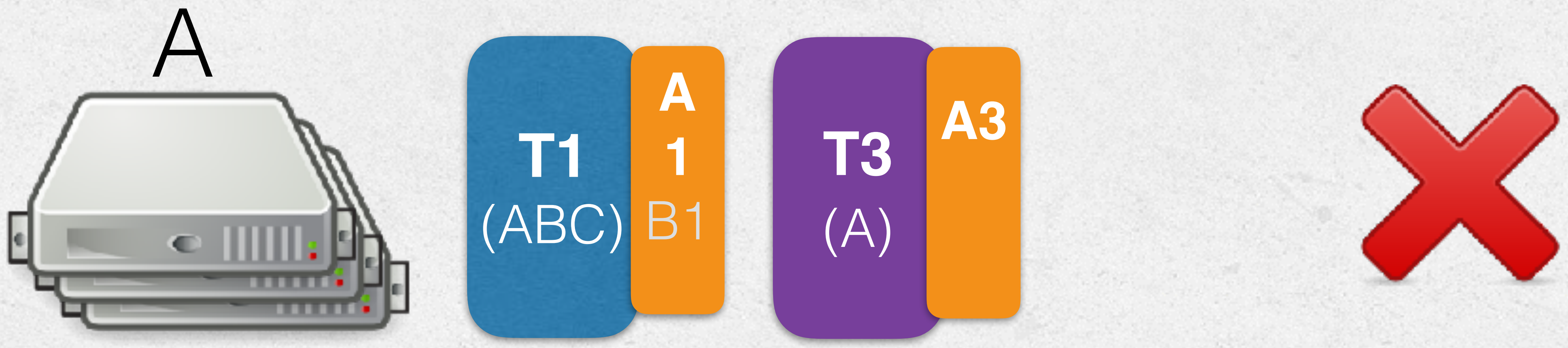
How to handle dropped messages?



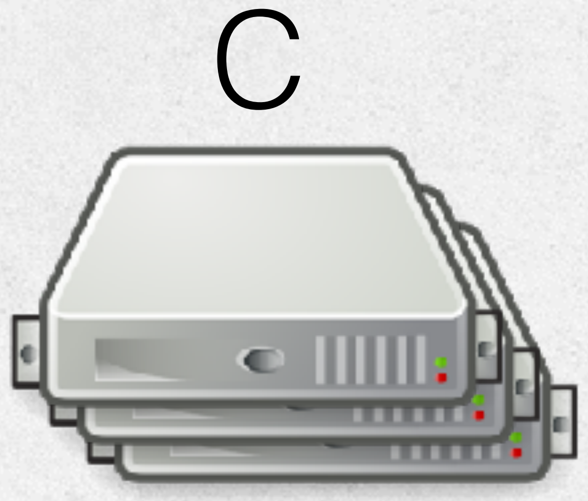
How to handle dropped messages?



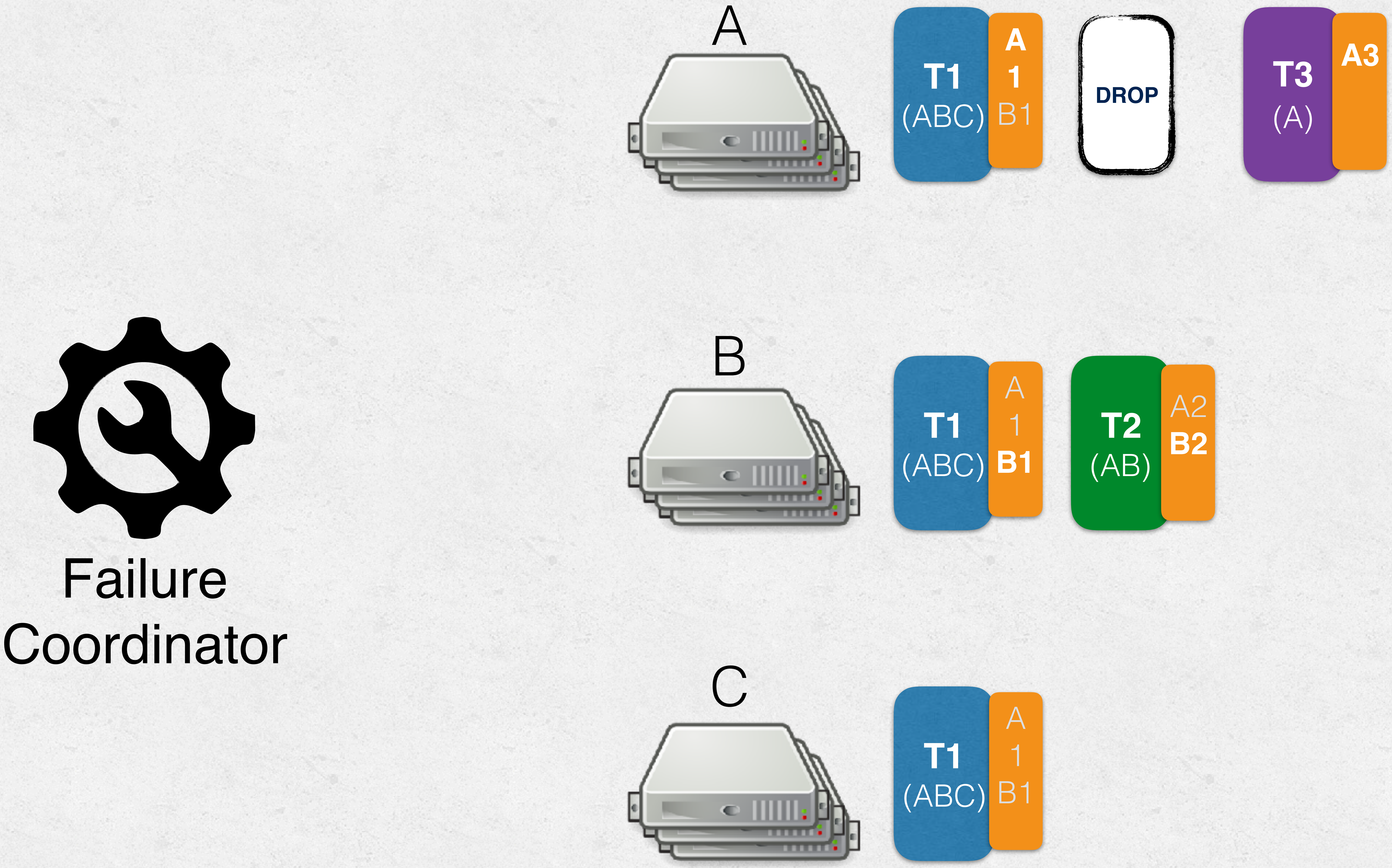
How to handle dropped messages?



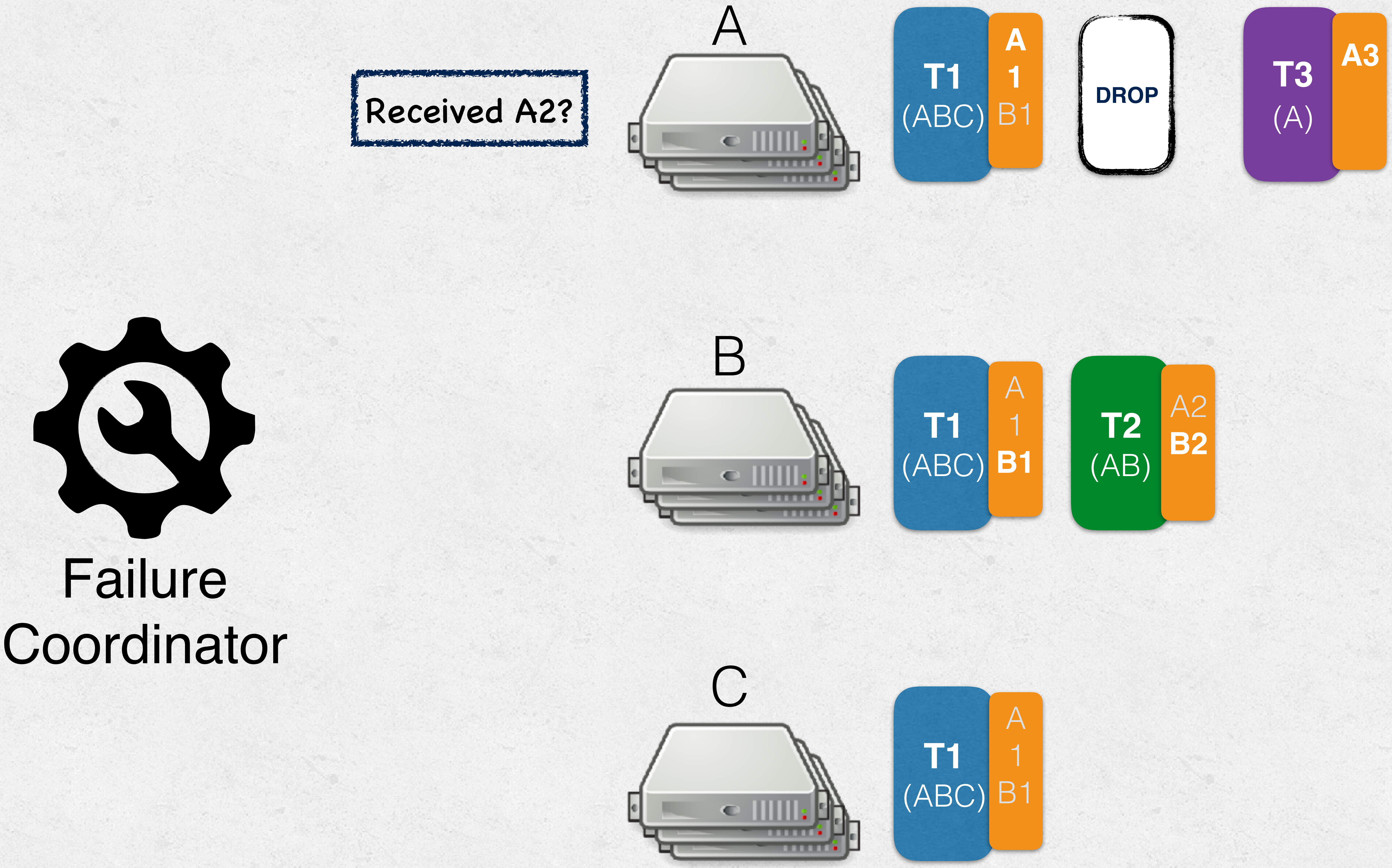
Global coordination problem



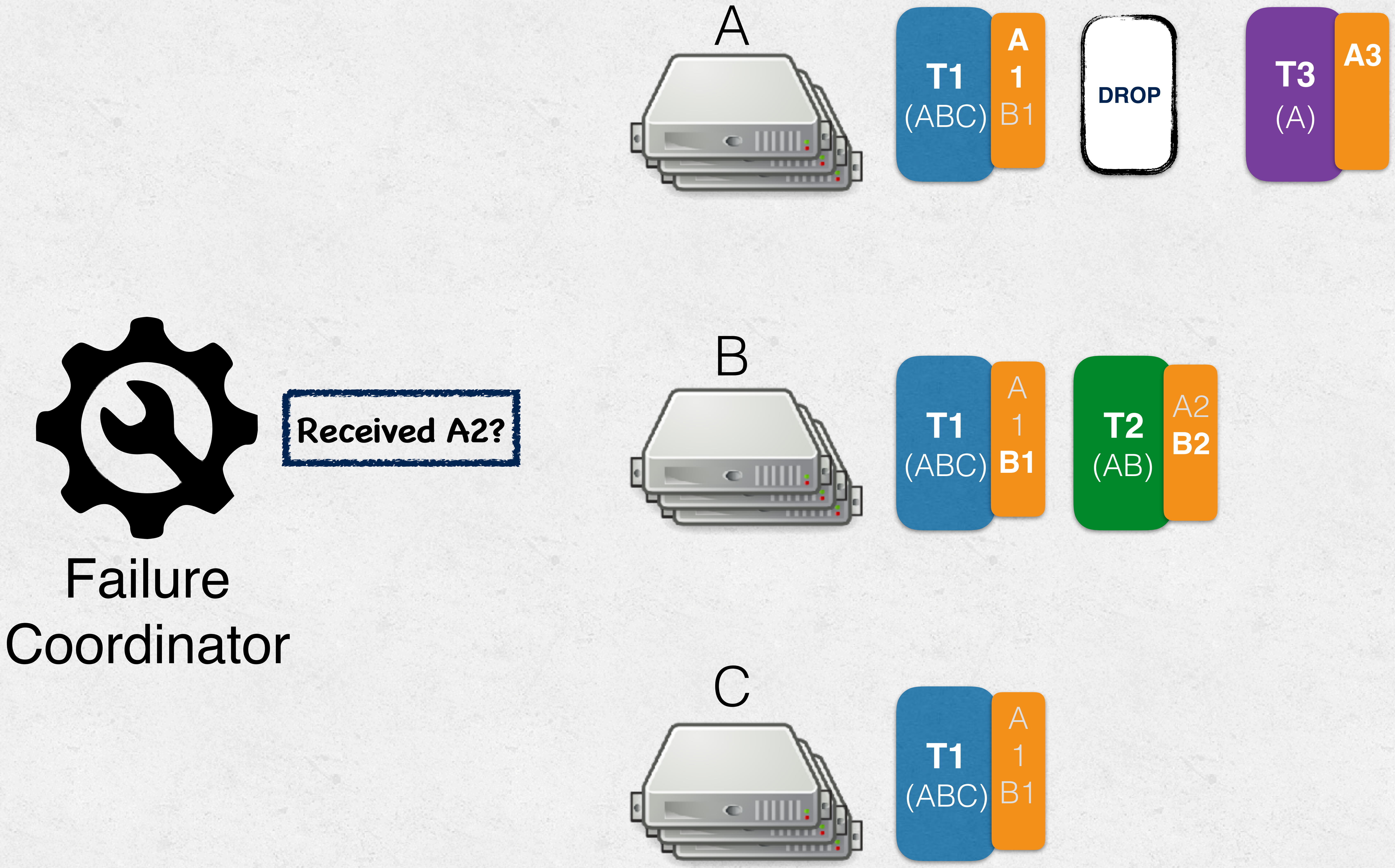
The Failure Coordinator



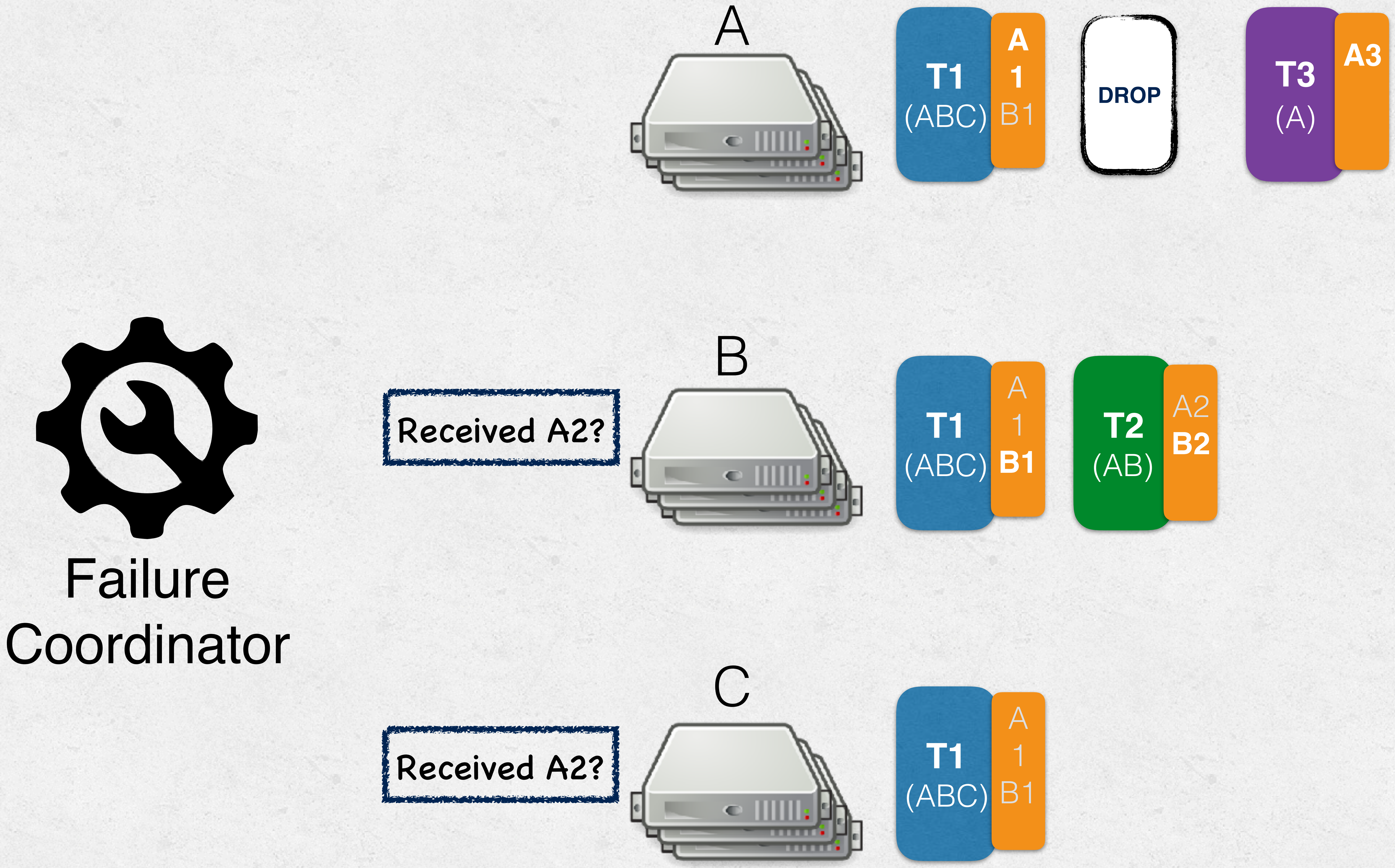
The Failure Coordinator



The Failure Coordinator



The Failure Coordinator



A

T1 (ABC) A 1 B1

DROP

T3 (A) A3

Failure Coordinator

Received A2?

B

T1 (ABC) A 1 B1

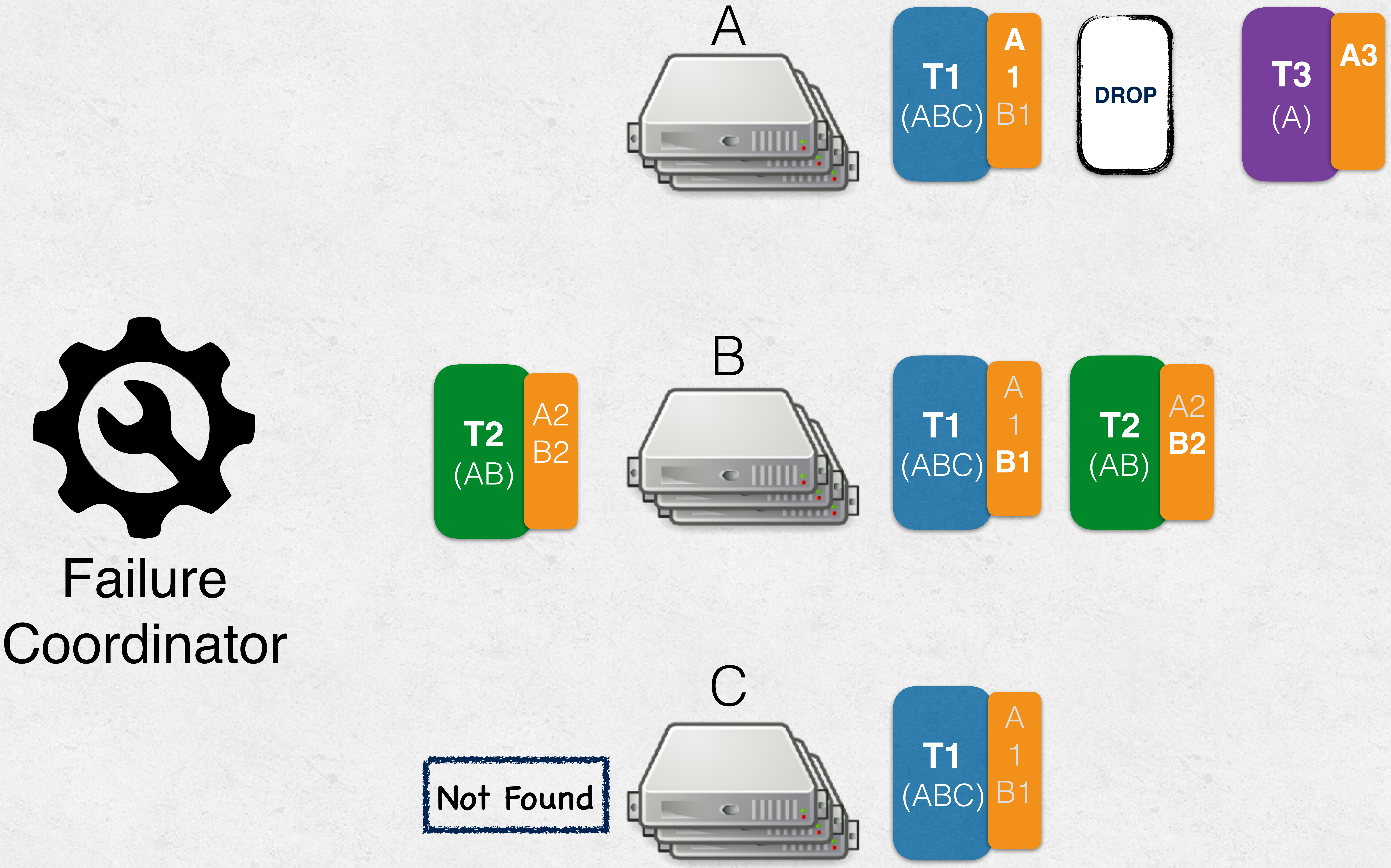
T2 (AB) A2 B2

Received A2?

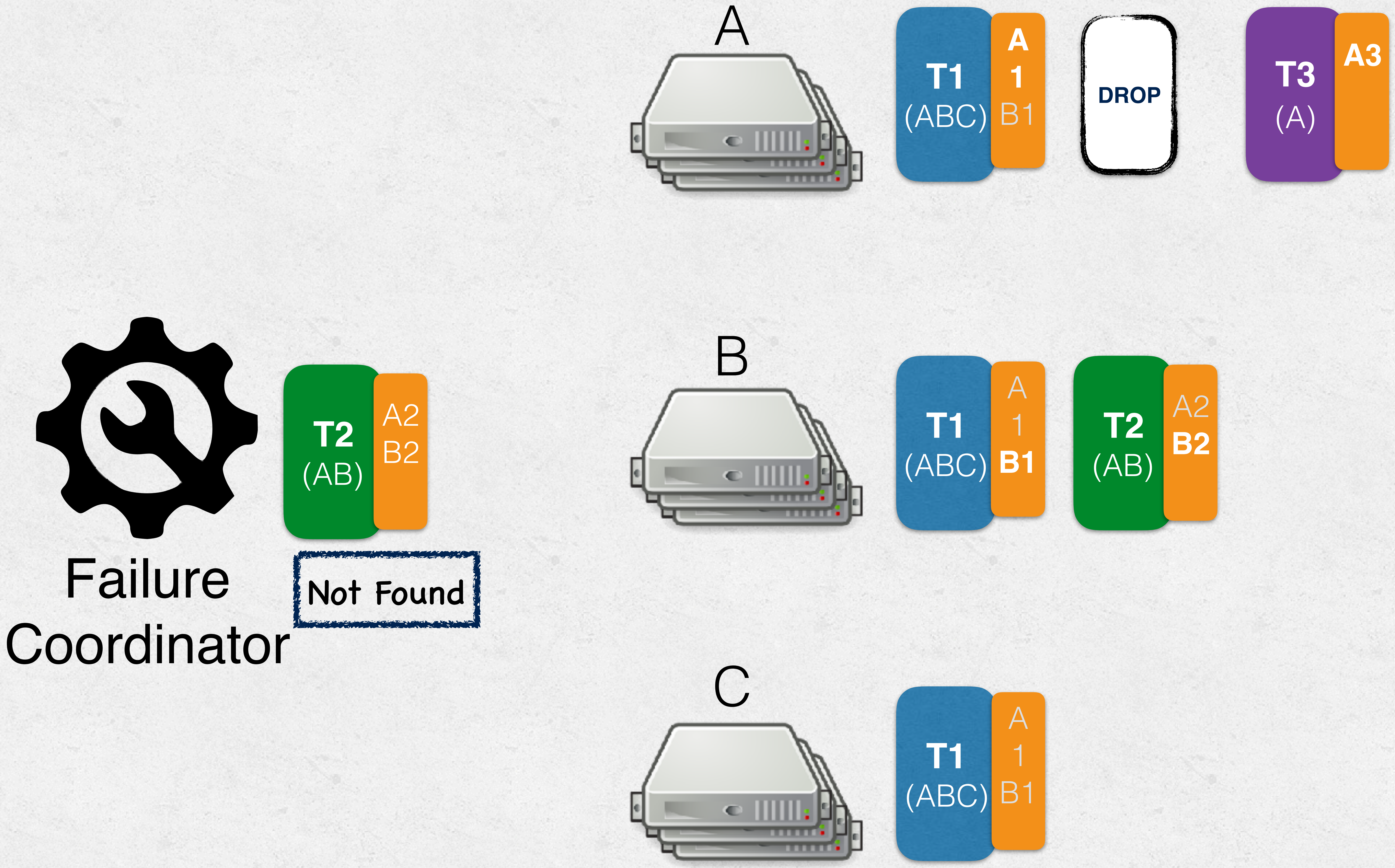
C

T1 (ABC) A 1 B1

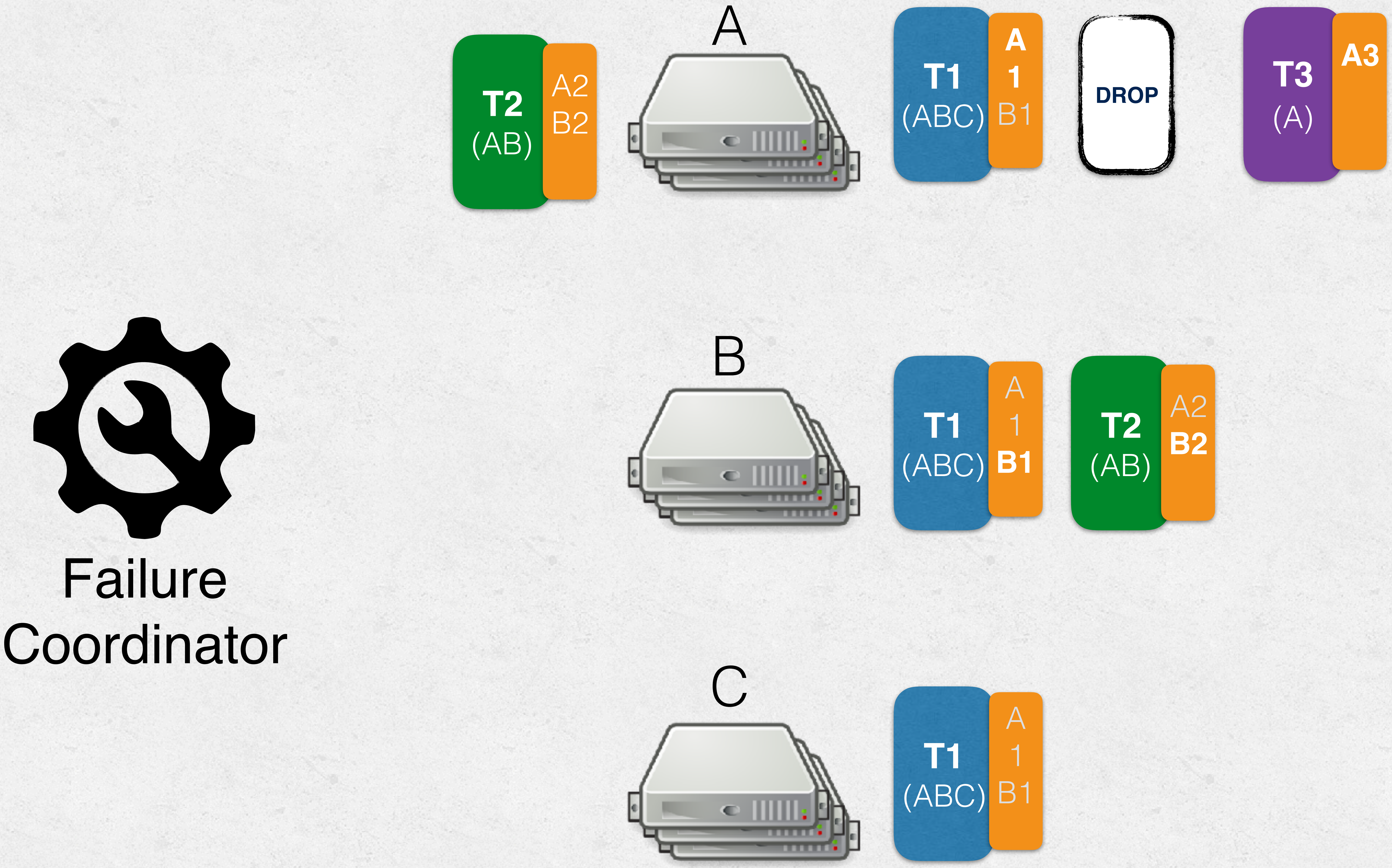
The Failure Coordinator



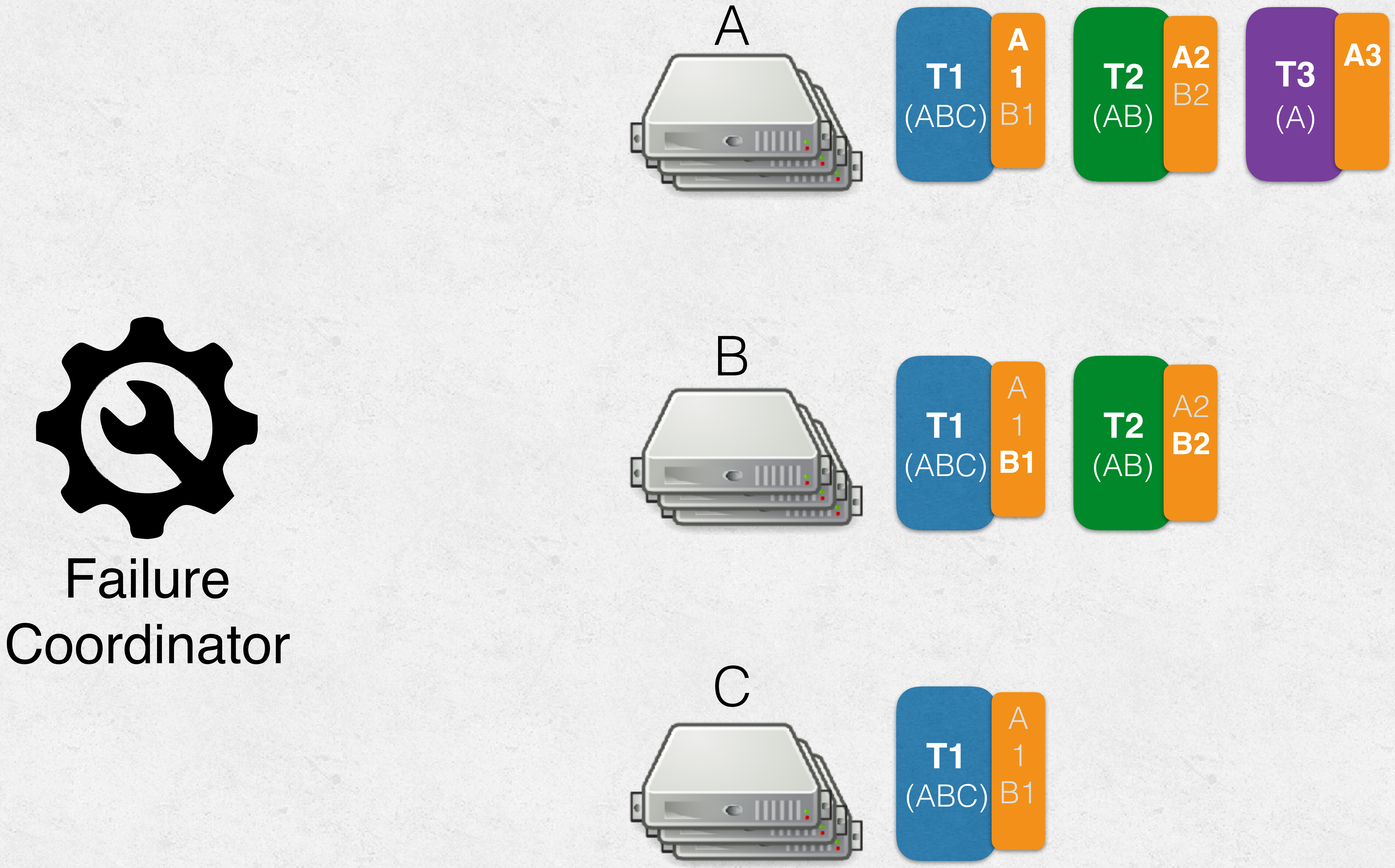
THE FAILURE COORDINATOR



The Failure Coordinator



The Failure Coordinator



Failure Coordinator

A

T1 (ABC) **A 1** B1

T2 (AB) **A2** B2

T3 (A) **A3**

B

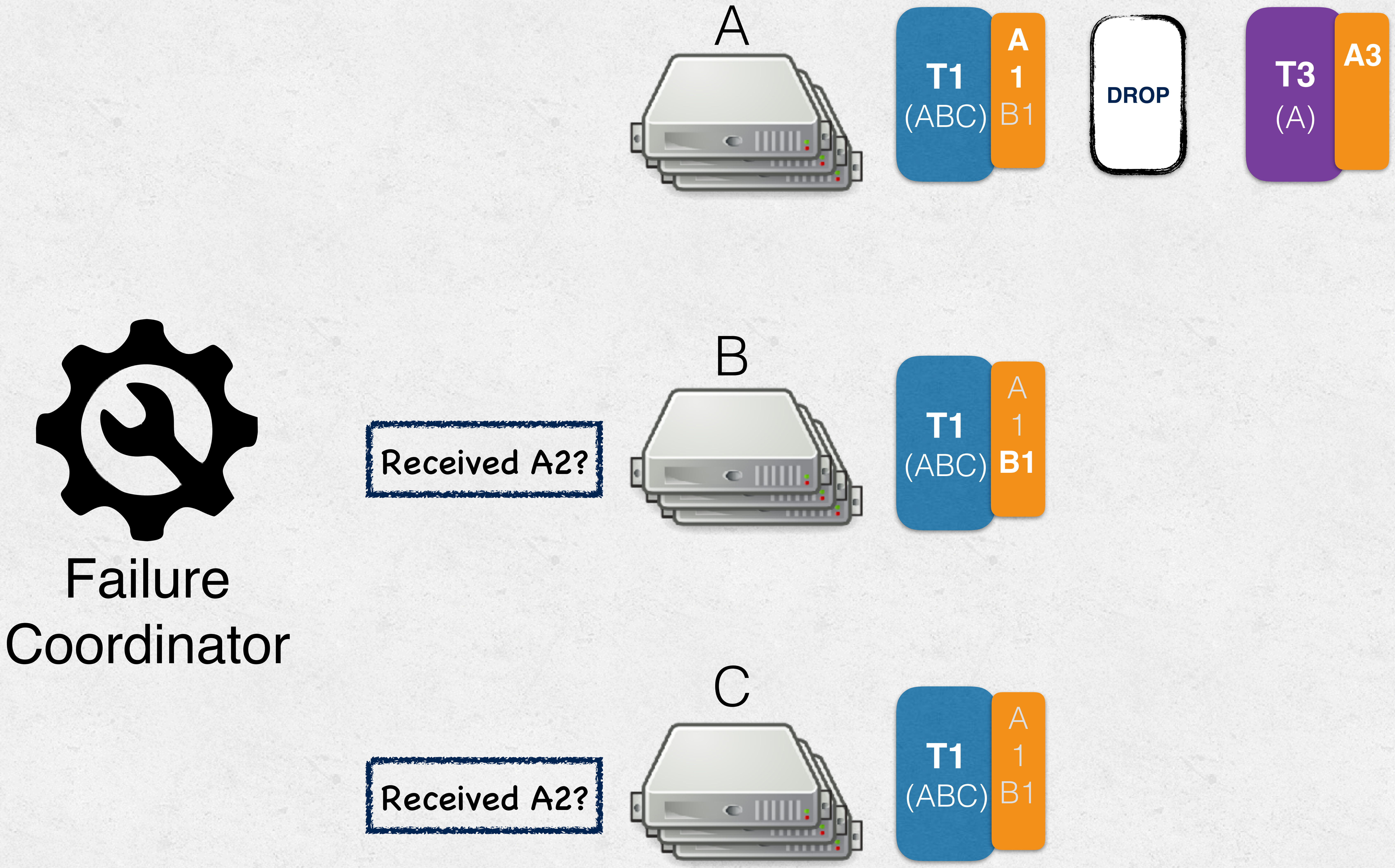
T1 (ABC) A 1 **B1**

T2 (AB) A2 **B2**

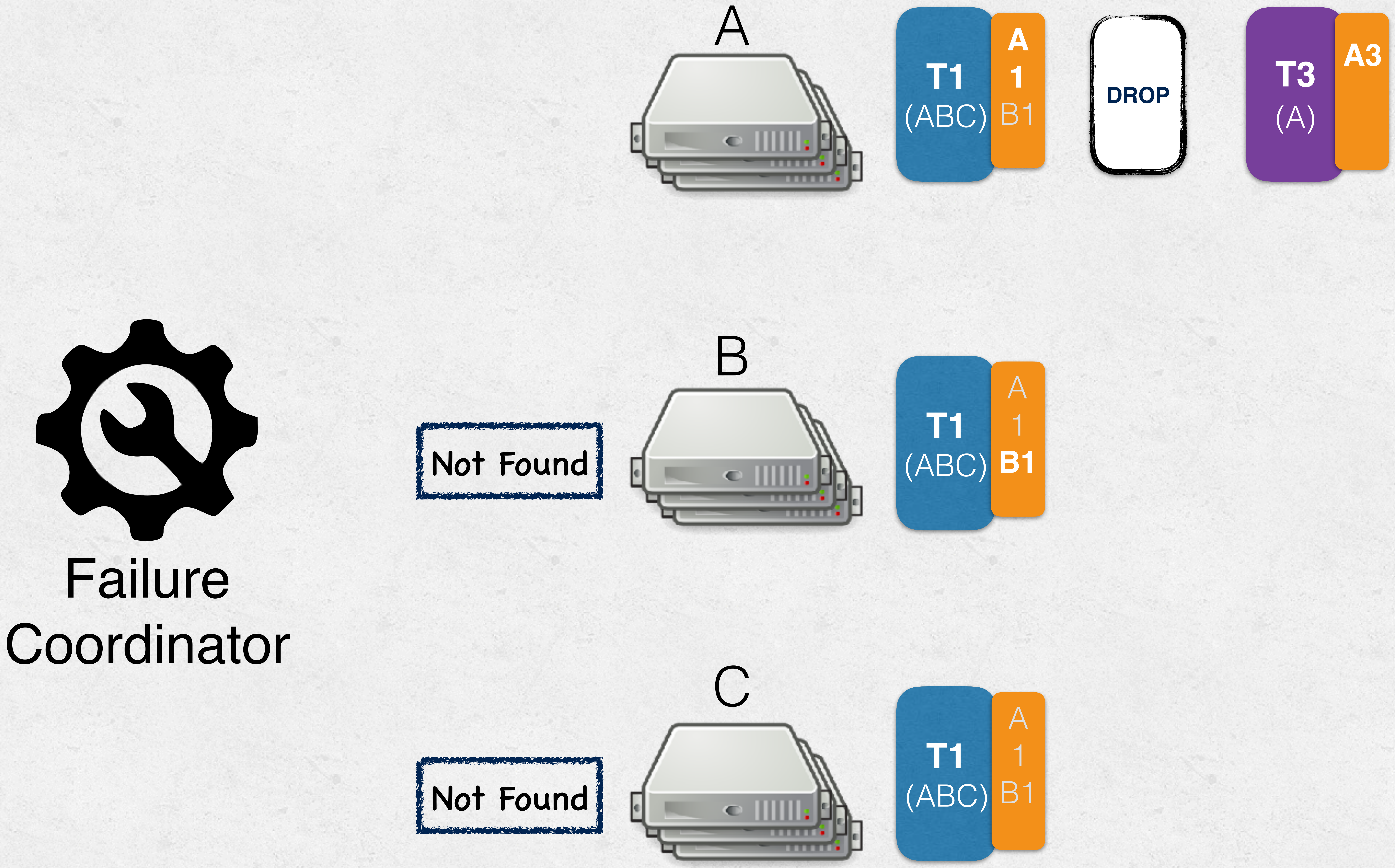
C

T1 (ABC) A 1 B1

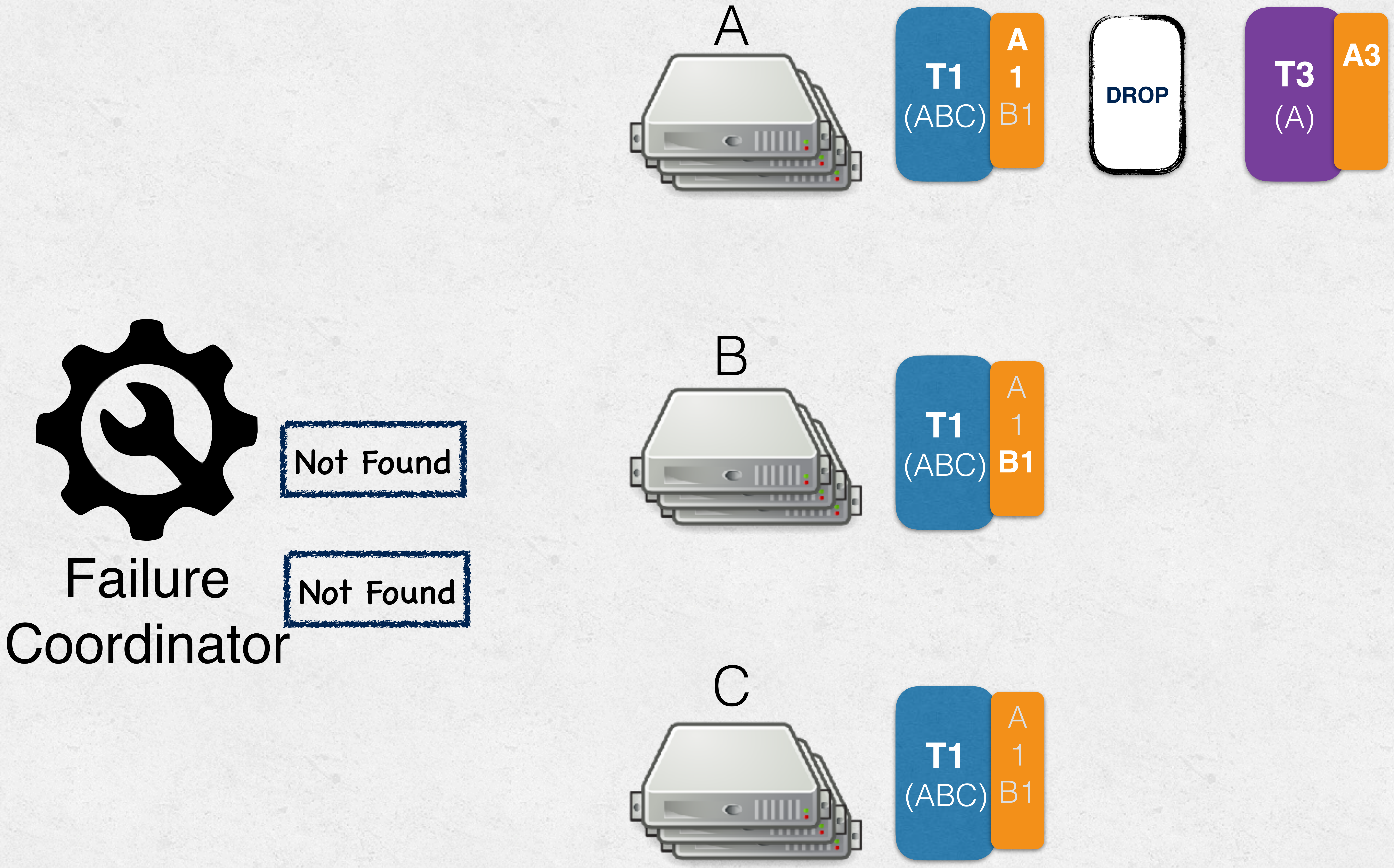
The Failure Coordinator



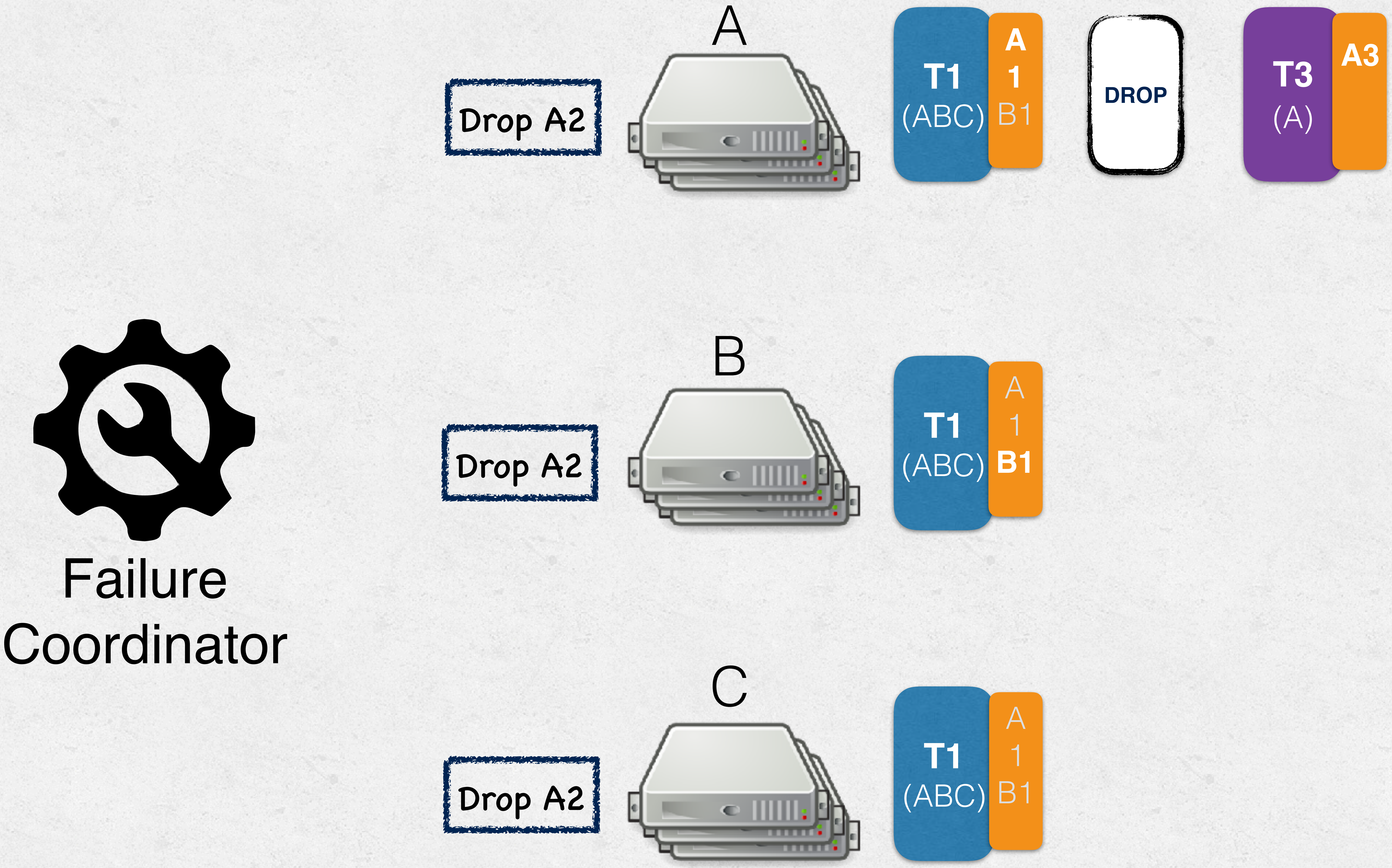
The Failure Coordinator



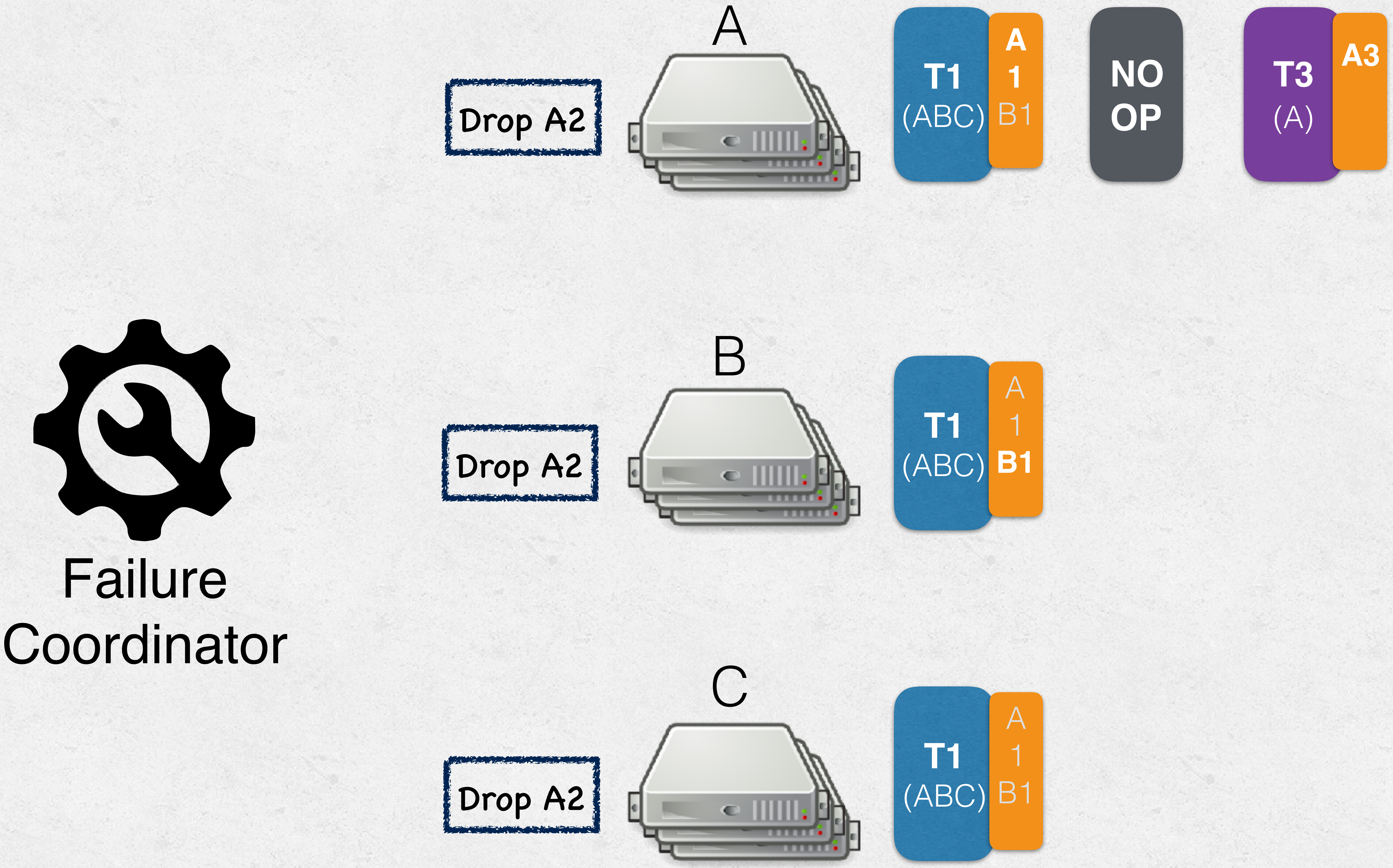
The Failure Coordinator



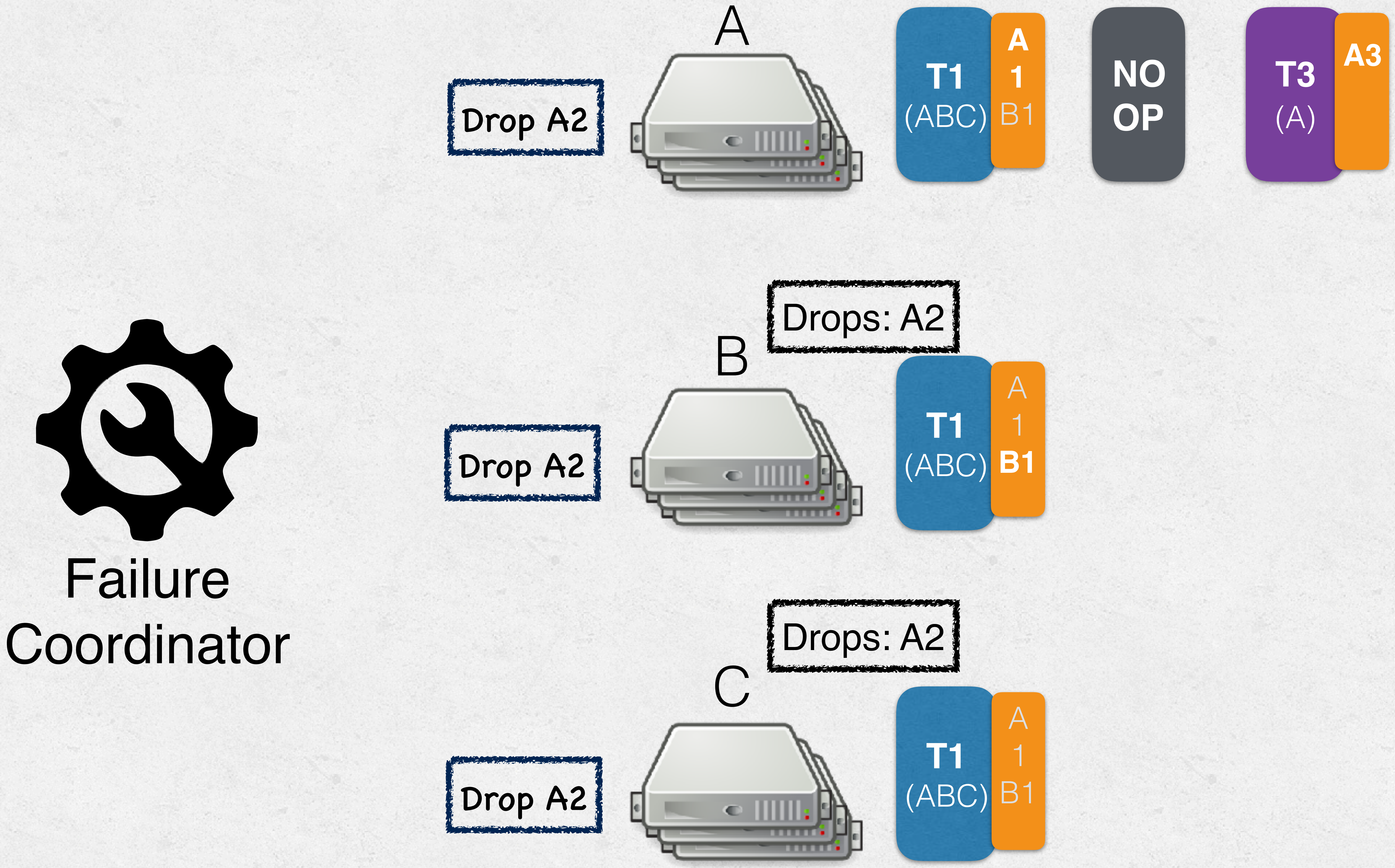
The Failure Coordinator



THE FAILURE COORDINATOR



THE FAILURE COORDINATOR



A

Drop A2

T1 (ABC) | A 1 B1 | NO OP | T3 (A) | A3

Drops: A2

B

Drop A2

T1 (ABC) | A 1 B1

Failure Coordinator

Drops: A2

C

Drop A2

T1 (ABC) | A 1 B1

Designated Learner and Sequencer Failures

Designated learner (DL) failure:

- View change based protocol
- Ensures new DL learns all **committed transactions** from previous views

Sequencer failure:

- Higher epoch number from the new sequencer
- Epoch change ensures all replicas across all shards start the new epoch in **consistent states**. They should all agree on the exact set of transactions completed in the previous epoch.

Can we process non-independent transactions efficiently?

Approach: Divide Into Independent Transactions

- Relies on the **linearizable execution** of independent transactions
- This means that we have the abstraction of a single, correct machine that processes independent transactions only.
- Uses **locks** to provide strong isolation
- Two phases:
  - Independent transaction 1: execute reads and acquire locks
  - Independent transaction 2: commit/abort changes and release locks

Benefits of our layered architecture

- Simple solution to handle client failures: if the client fails, any server can **unilaterally** send the abort command for its general transactions *as an independent transaction.*

- No deadlocks/deadlock detection. Locks are acquired in a single step.

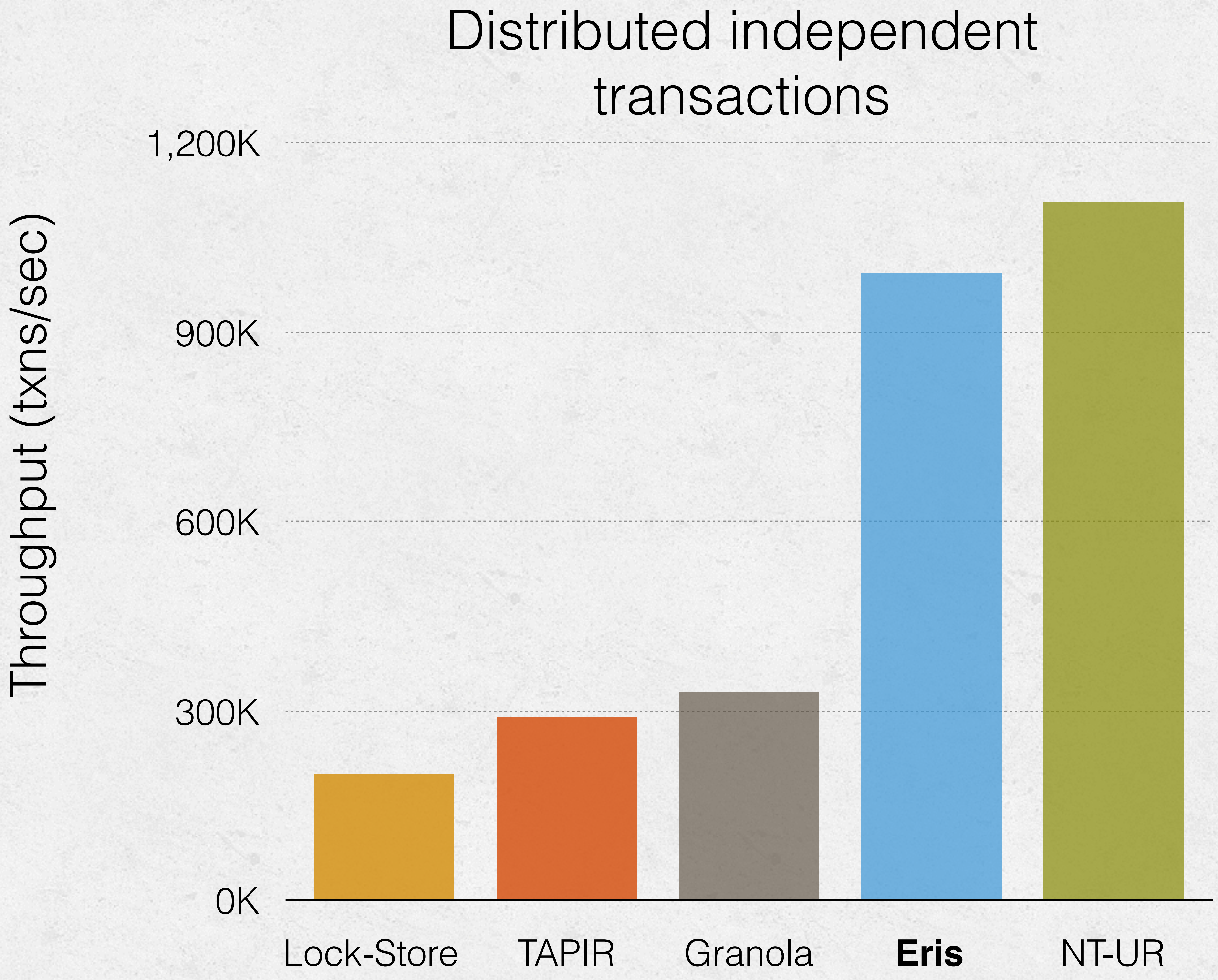
- Furthermore, we don't even need aborts! Wait queues are easy.

- Takes advantage of the efficient independent transaction processing layer. General transactions are processed in two round trips in the normal case.

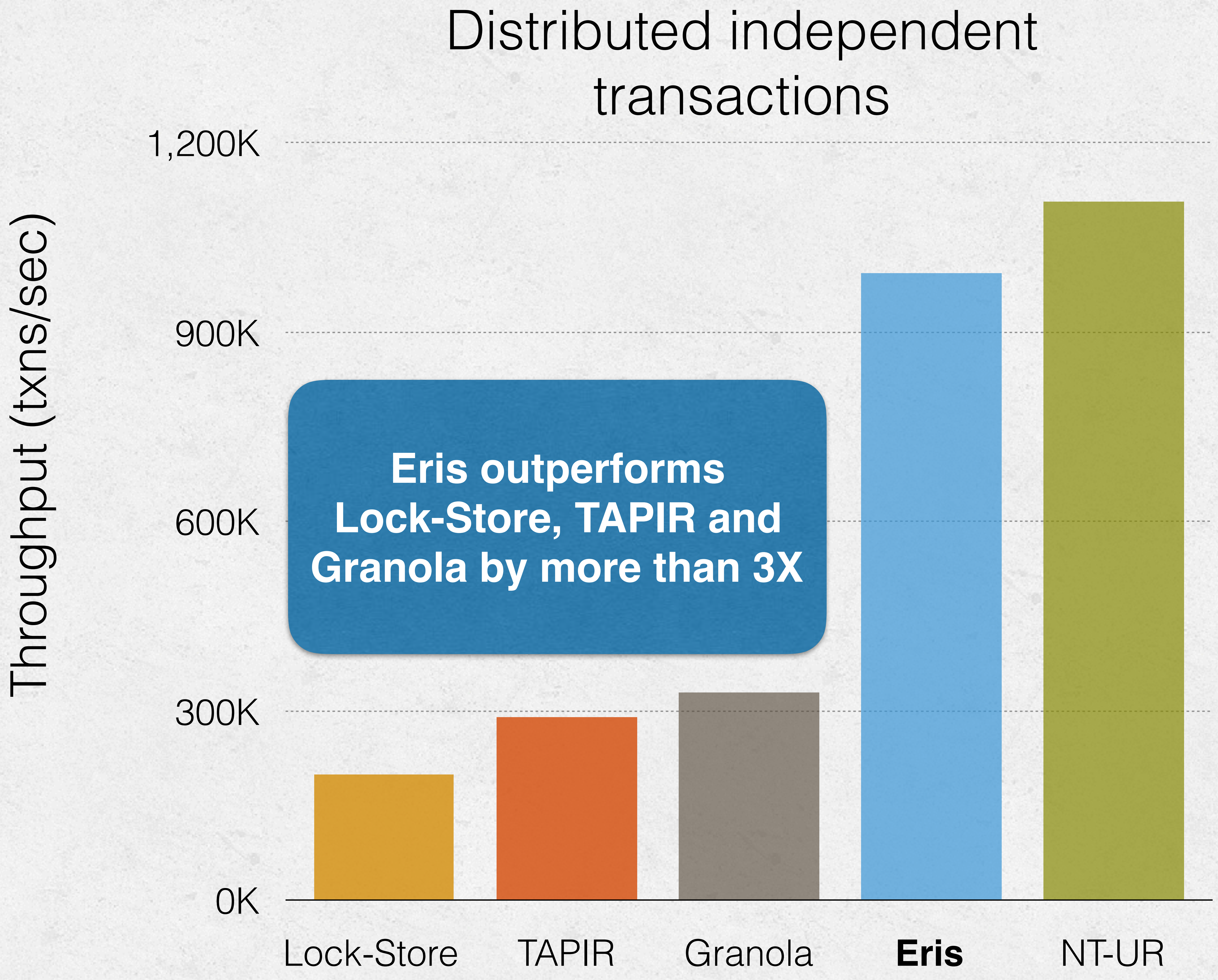
Evaluation Comparison Systems

- Lock-Store (2PC + 2PL + Paxos)
- TAPIR [SOSP ’15]
- Granola [ATC ‘12]
- Non-transactional, unreplicated (NT-UR)

Eris performs well on independent transactions

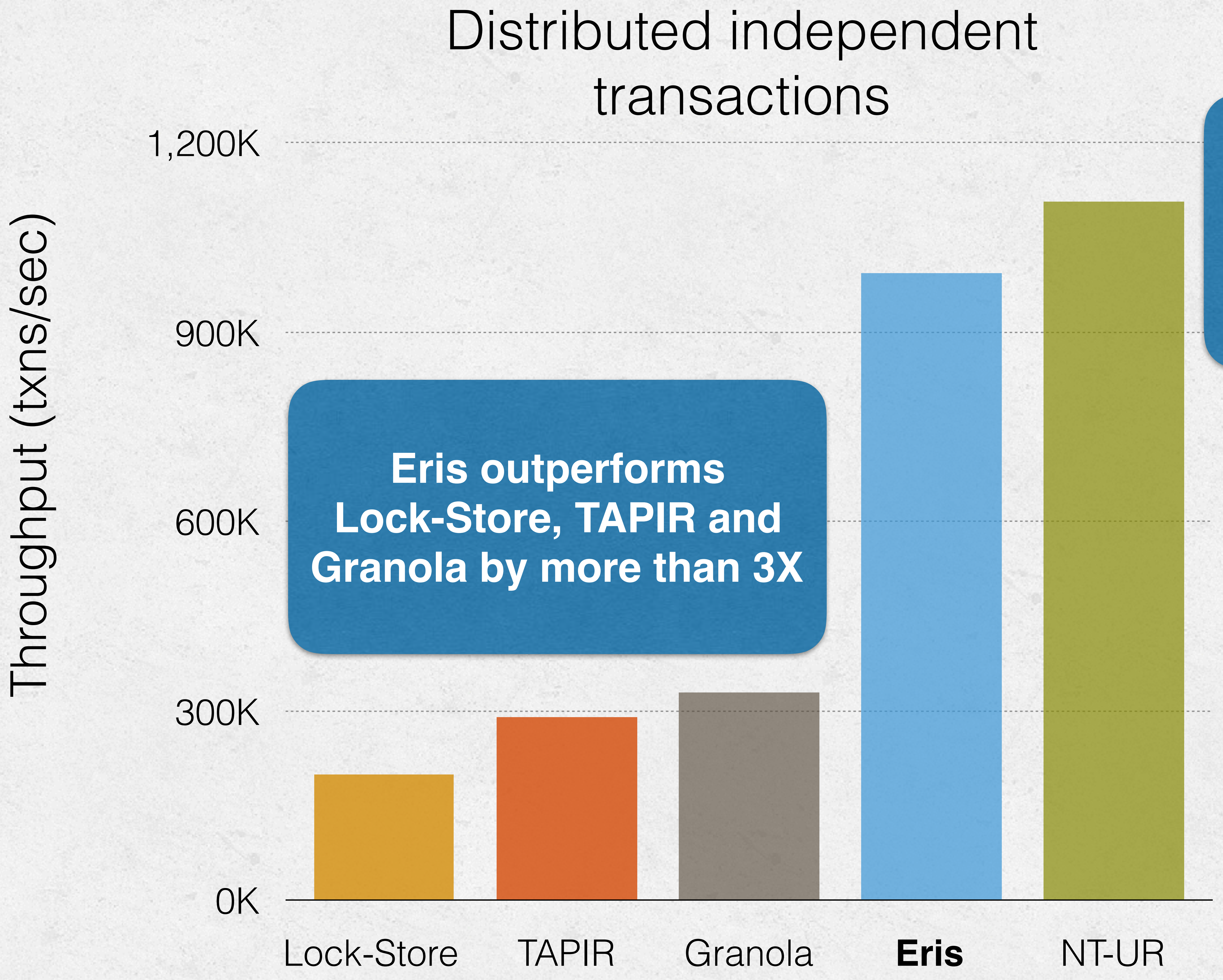


ERIS PERFORMS WELL ON INDEPENDENT TRANSACTIONS



Eris outperforms Lock-Store, TAPIR and Granola by more than 3X

ERIS PERFORMS WELL ON INDEPENDENT TRANSACTIONS

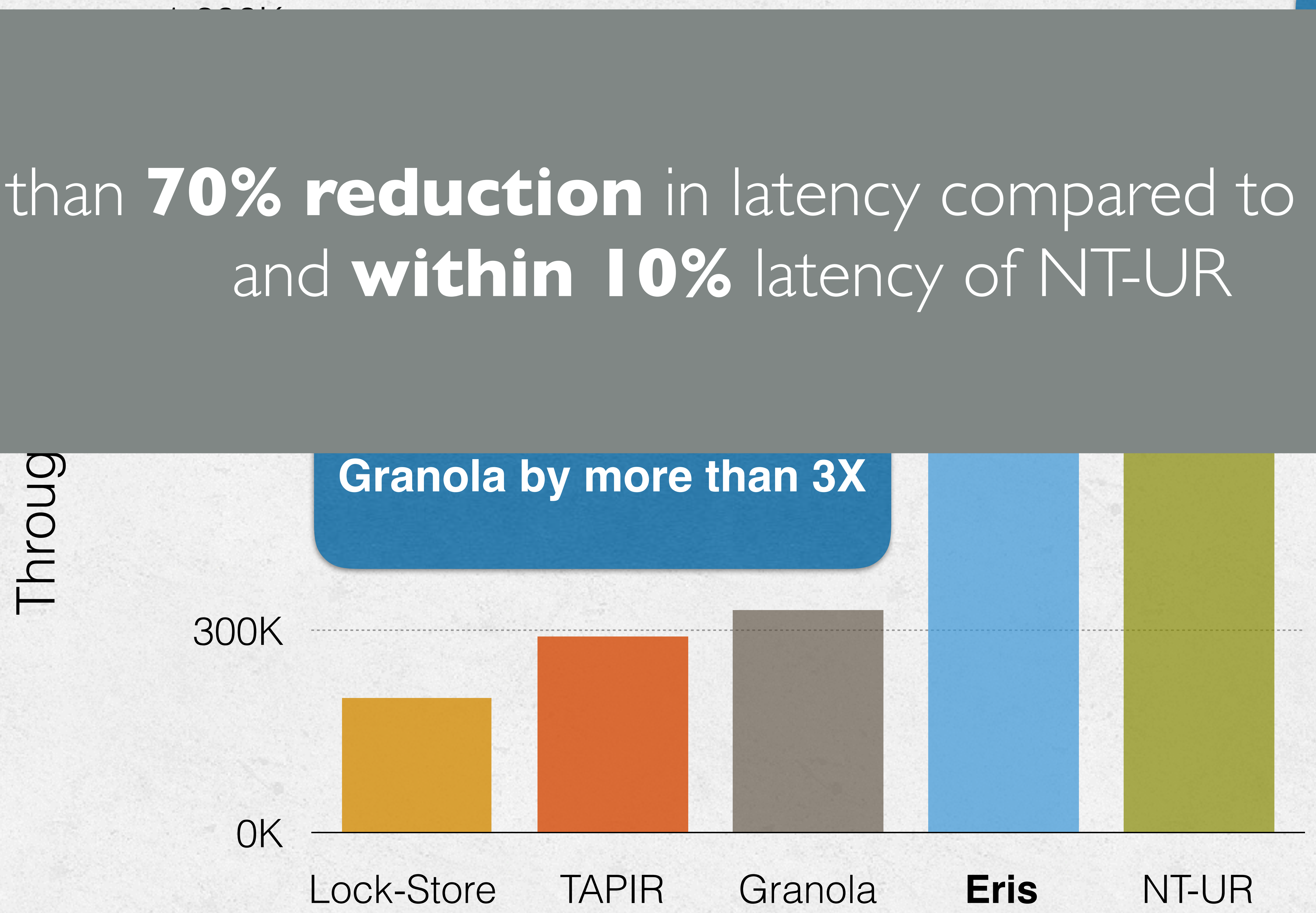


Eris outperforms Lock-Store, TAPIR and Granola by more than 3X

Eris achieves throughput within 10% of NT-UR

Eris performs well on independent transactions

Distributed independent transactions

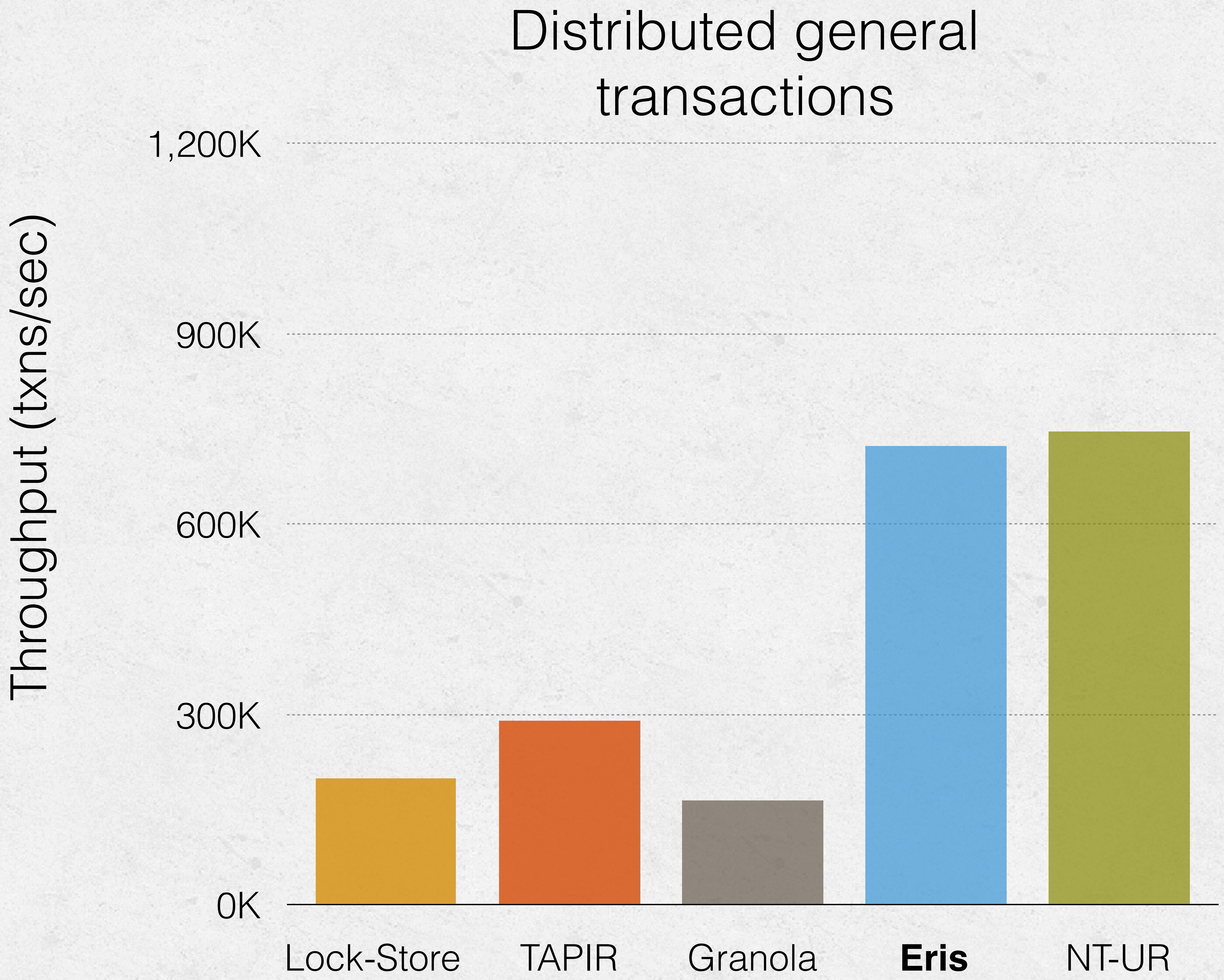


Granola by more than 3X

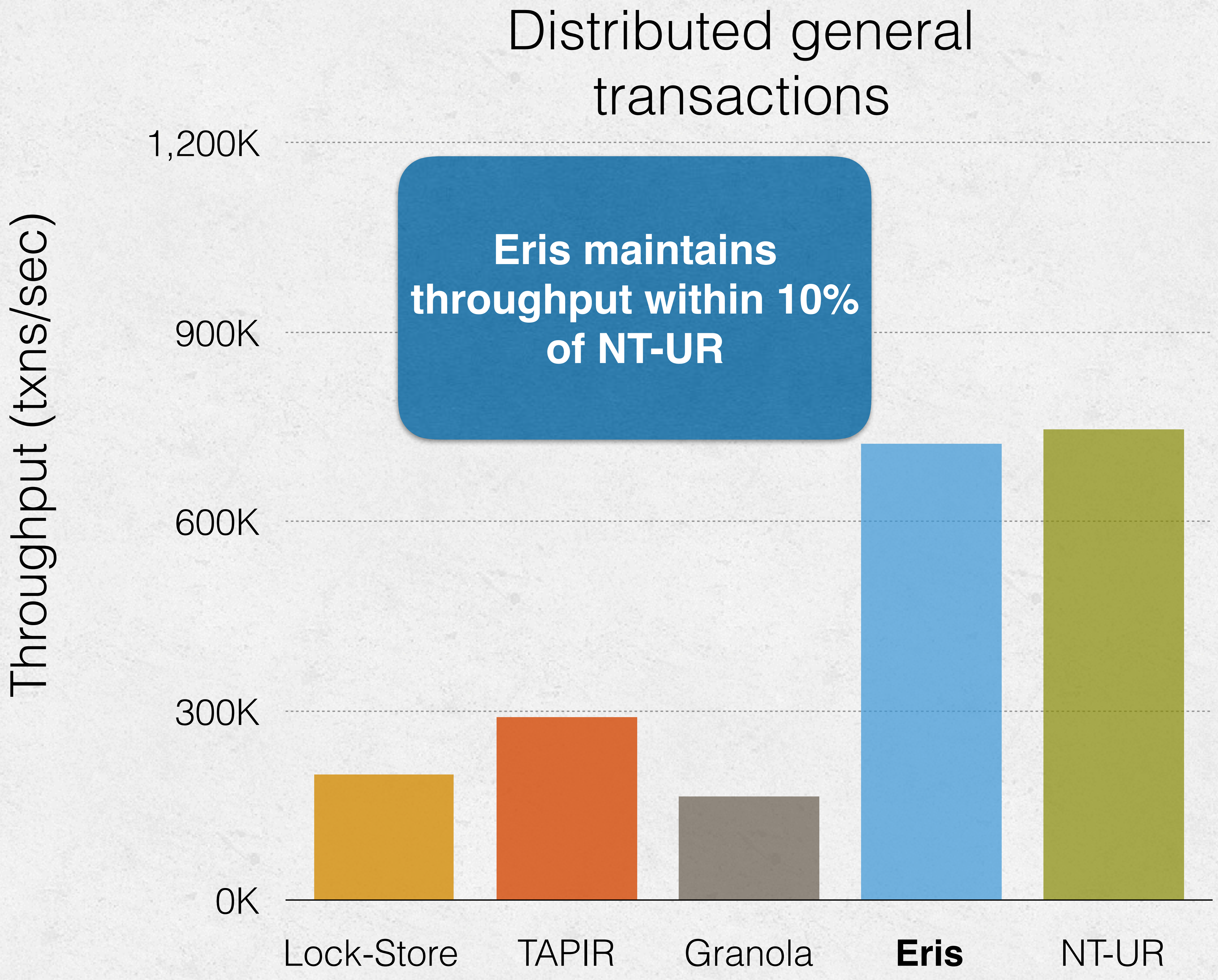
in

More than **70% reduction** in latency compared to Lock-Store, and **within 10%** latency of NT-UR

ERIS ALSO PERFORMS WELL ON GENERAL TRANSACTIONS

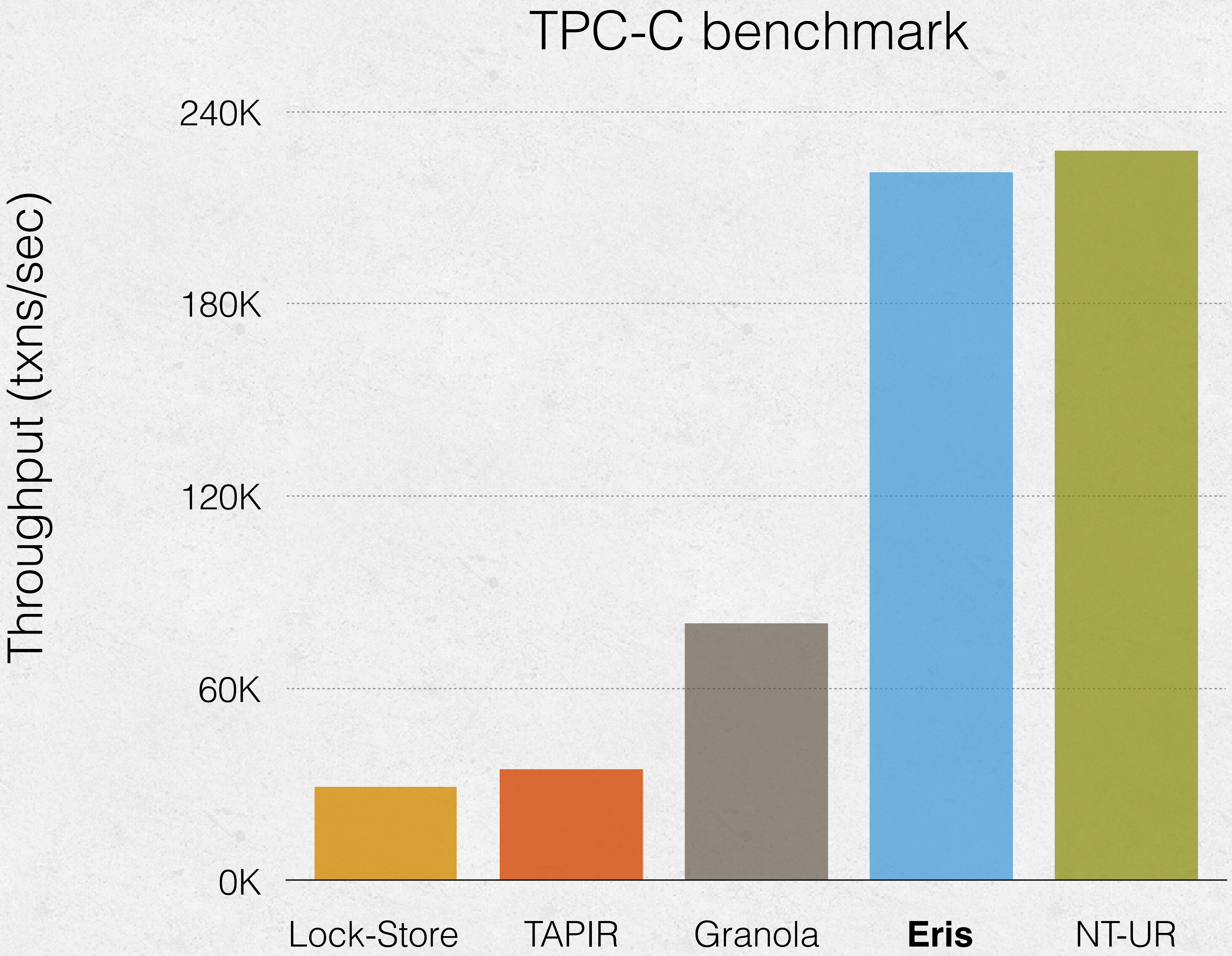


Eris also performs well on general transactions

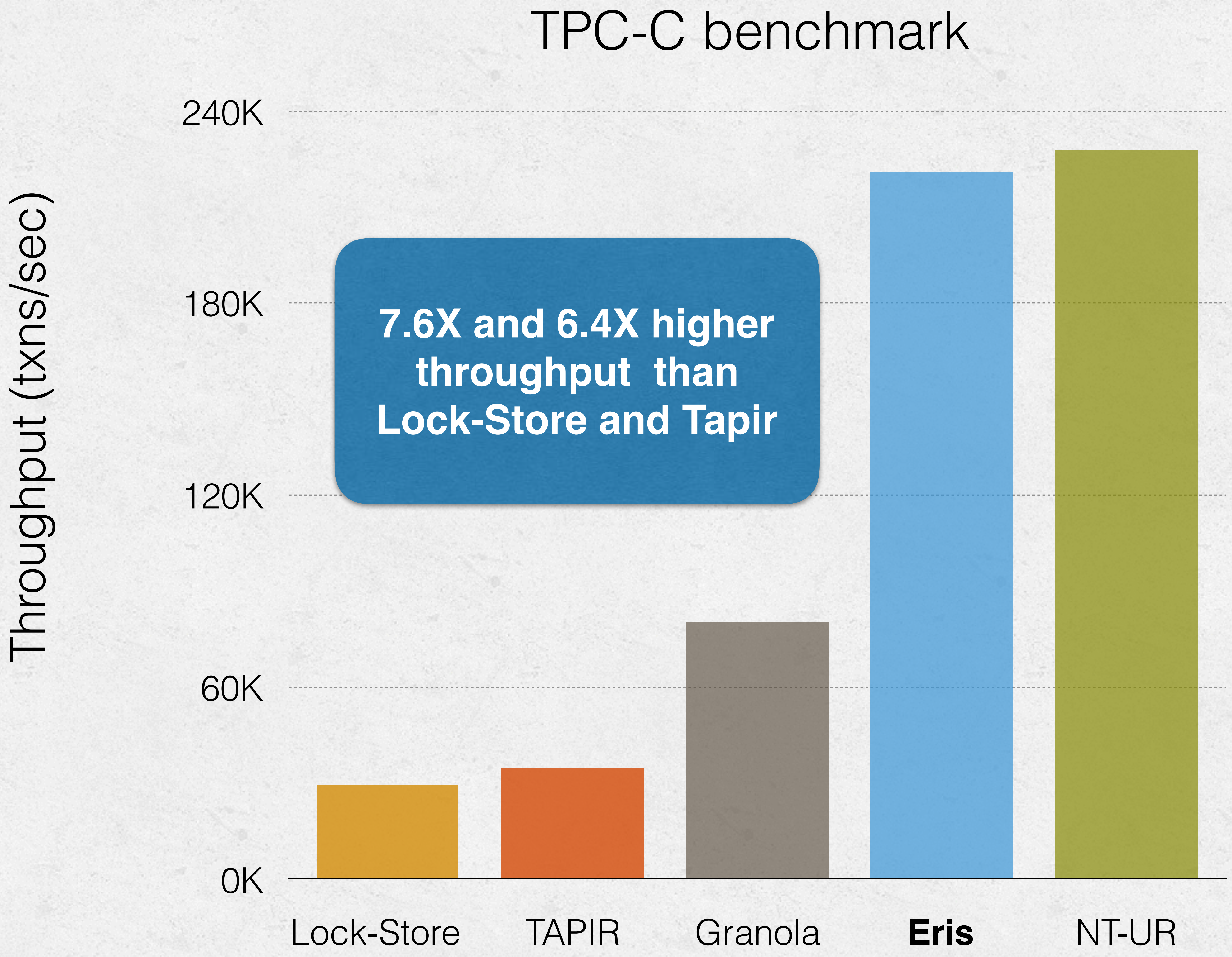


Eris maintains throughput within 10% of NT-UR

Eris excels at complex transactional applications

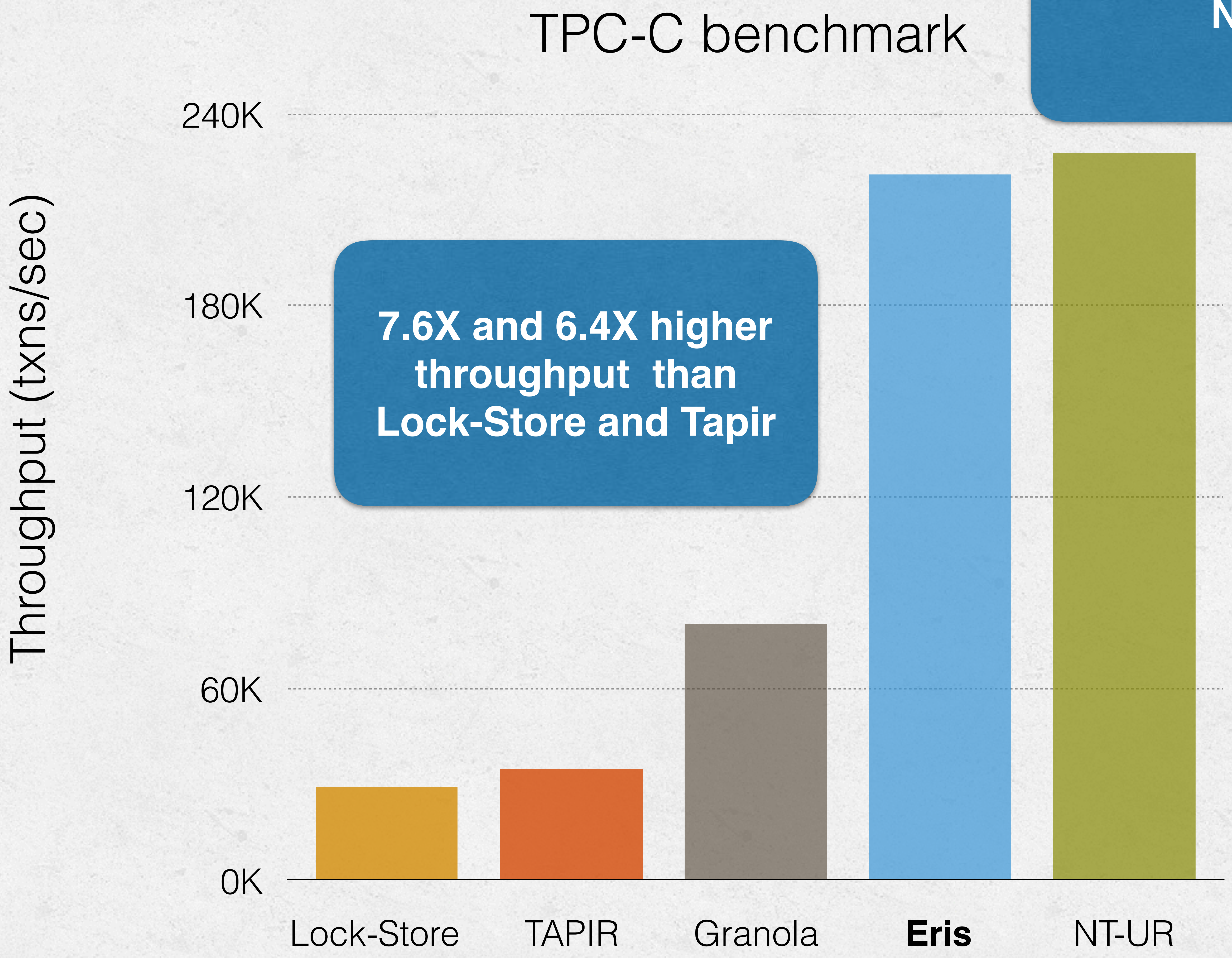


Eris excels at complex transactional applications



7.6X and 6.4X higher throughput than Lock-Store and Tapir

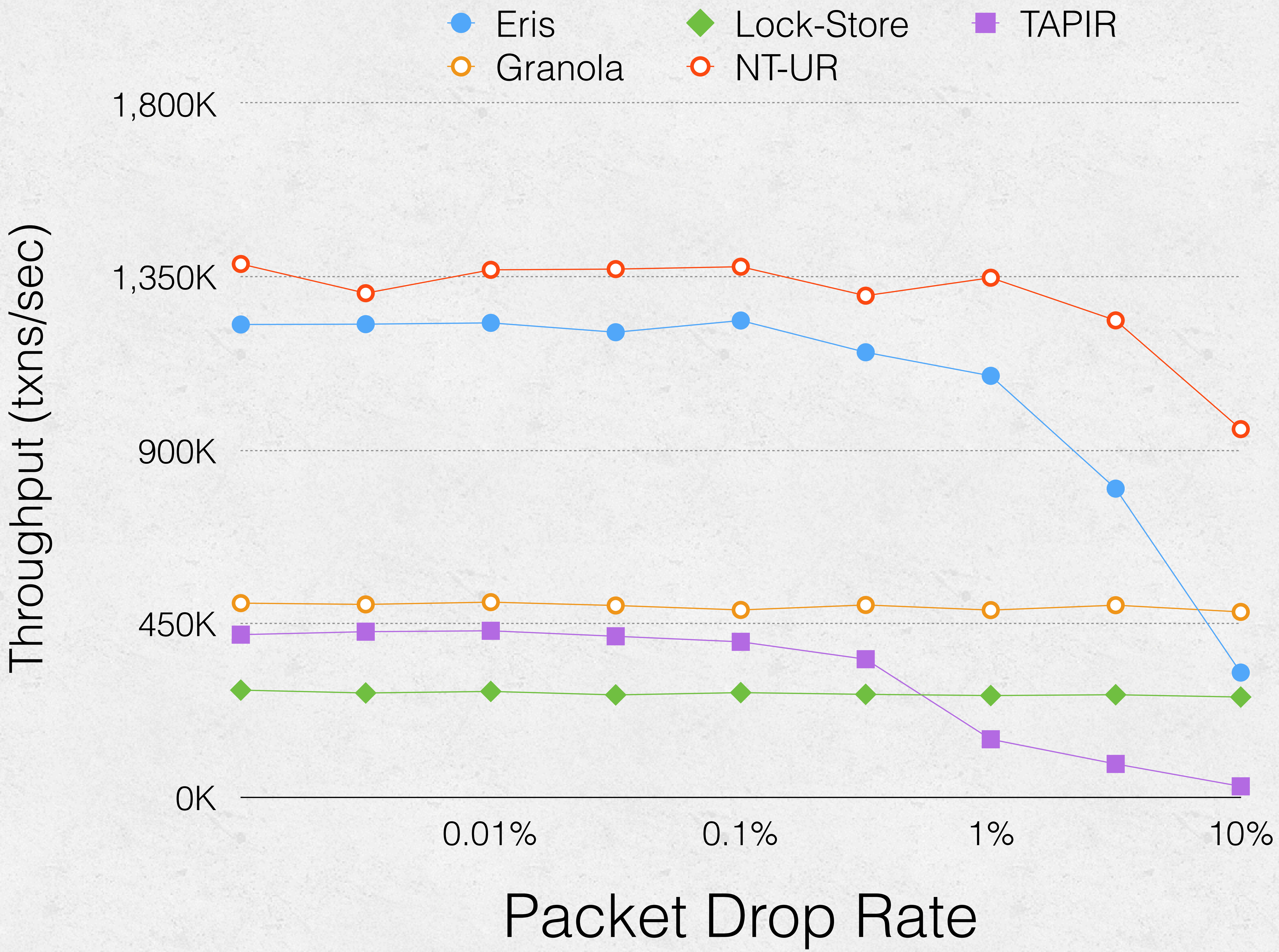
Eris excels at complex transactional applications



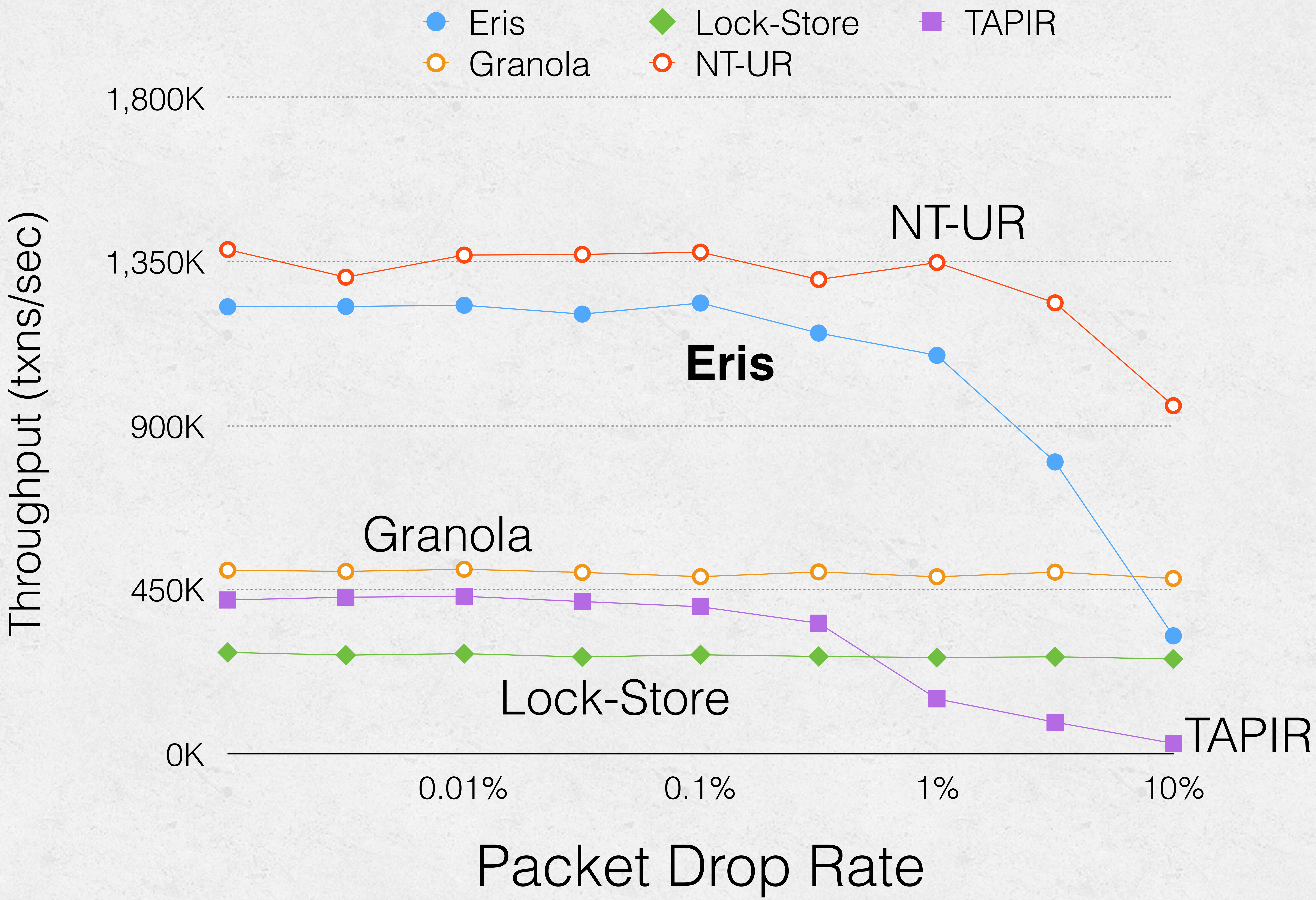
within 3% throughput of NT-UR

7.6X and 6.4X higher throughput than Lock-Store and Tapir

Eris is resilient to network anomalies



Eris is resilient to network anomalies



Eris Recap

- A new division of responsibility for transaction processing
  - An in-network concurrency control mechanism that establishes a **consistent order** of transactions across shards
  - An efficient protocol that ensures **reliable delivery** of independent transactions
  - A **general transaction** layer atop independent transaction processing
- Result: strongly consistent, fault-tolerant transactions with **minimal performance overhead**

Eris and NOPaxos Discussion

- Can we use an end-host sequencer for Eris? In NOPaxos, it's not a problem.

- What properties are important to NOPaxos's "scalability"?

- How deployable are these approaches?

- How scalable is Eris compared to two-phase commit?