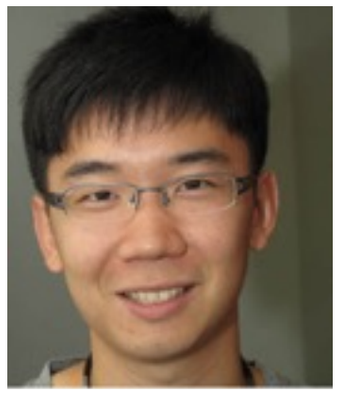
Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control

Jialin Li, Ellis Michael, Dan R. K. Ports

UNIVERSITY of WASHINGTON

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING



Web services and applications rely on distributed storage systems

Web services and applications rely on distributed storage systems

Web services and applications rely on distributed storage systems

Web services and applications rely on distributed storage systems

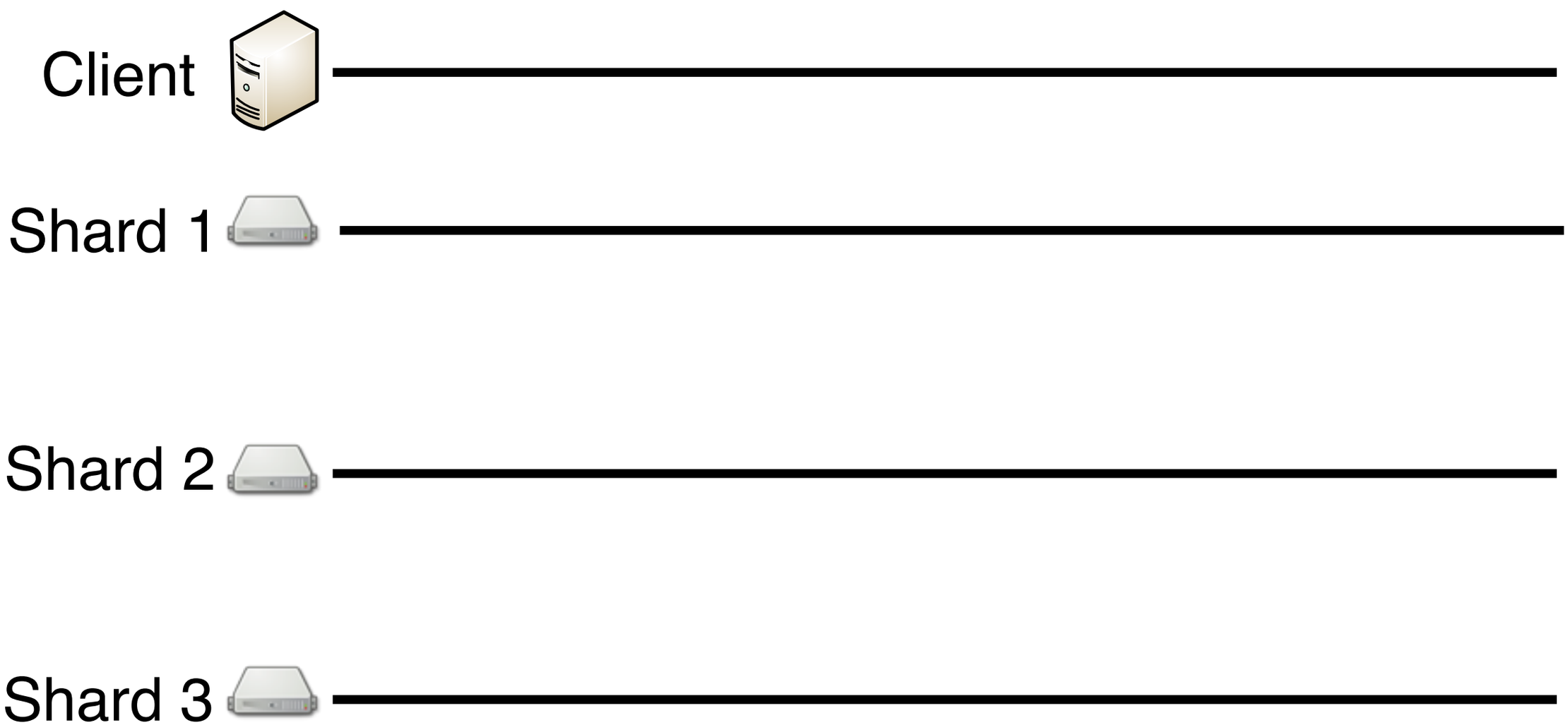
Web services and applications rely on distributed storage systems

Partitioned for Scalability, Replicated for Availability

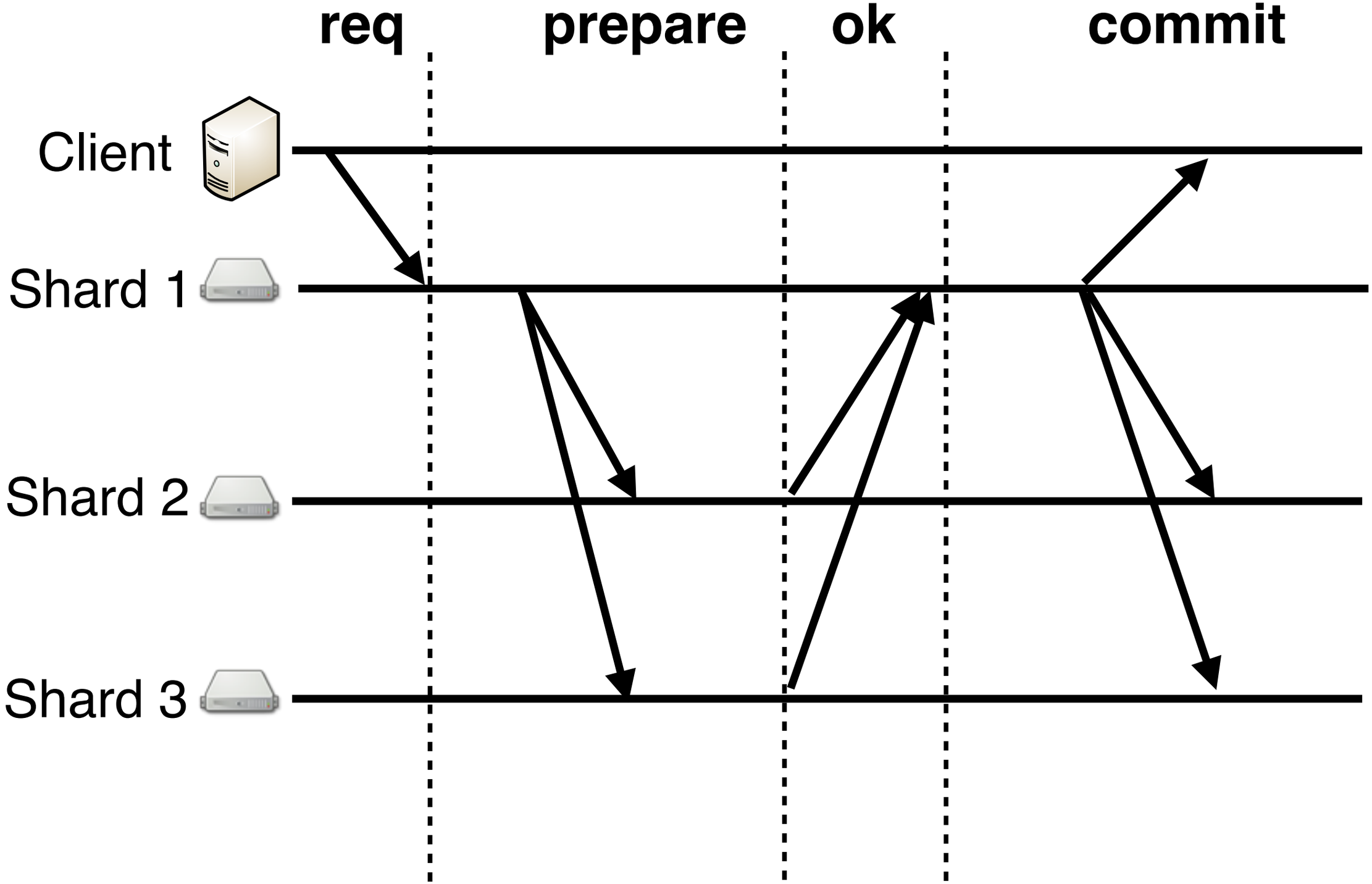
Partitioned for Scalability, Replicated for Availability

Partitioned for Scalability, Replicated for Availability

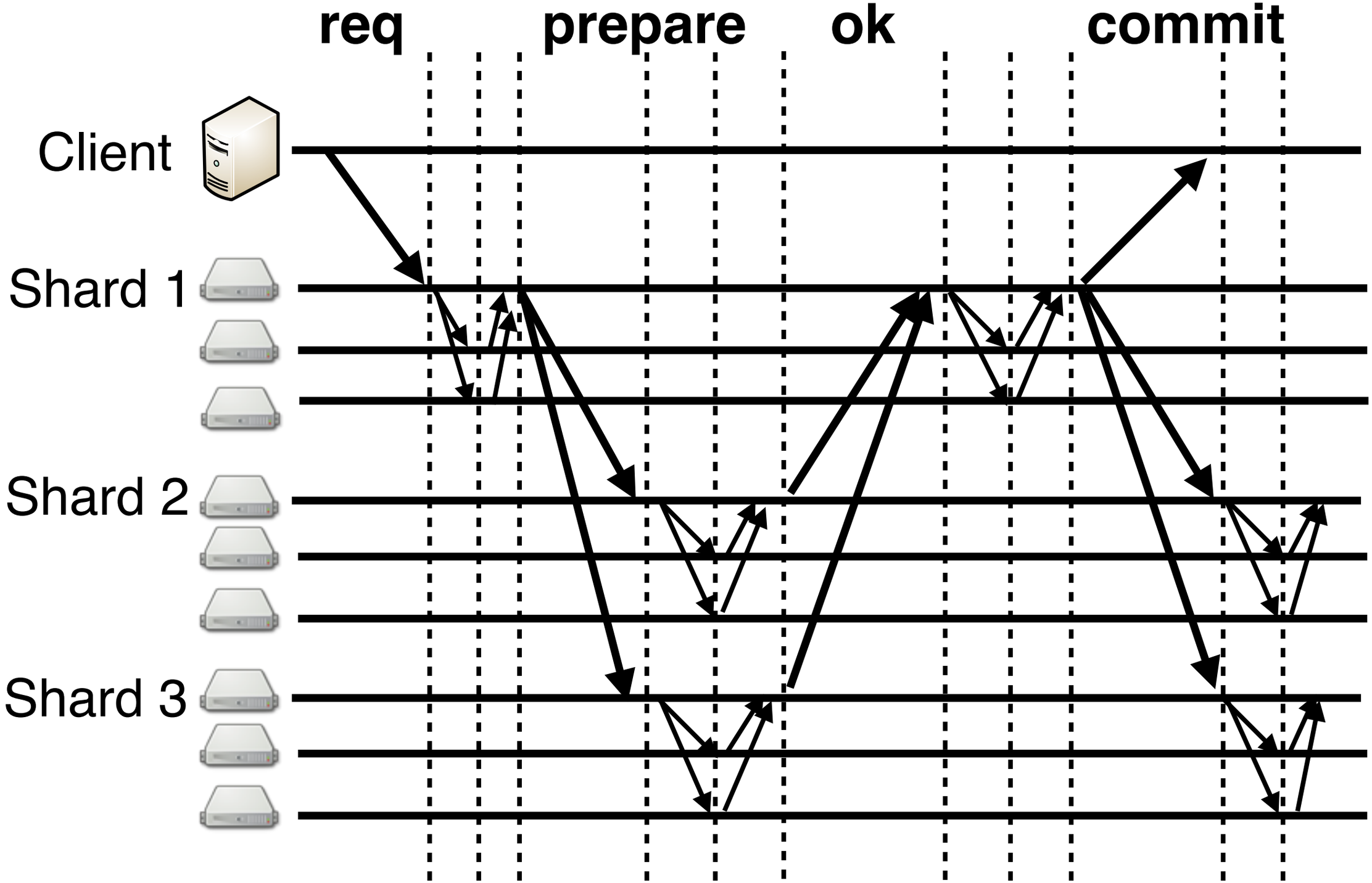
Existing transactional systems: extensive coordination



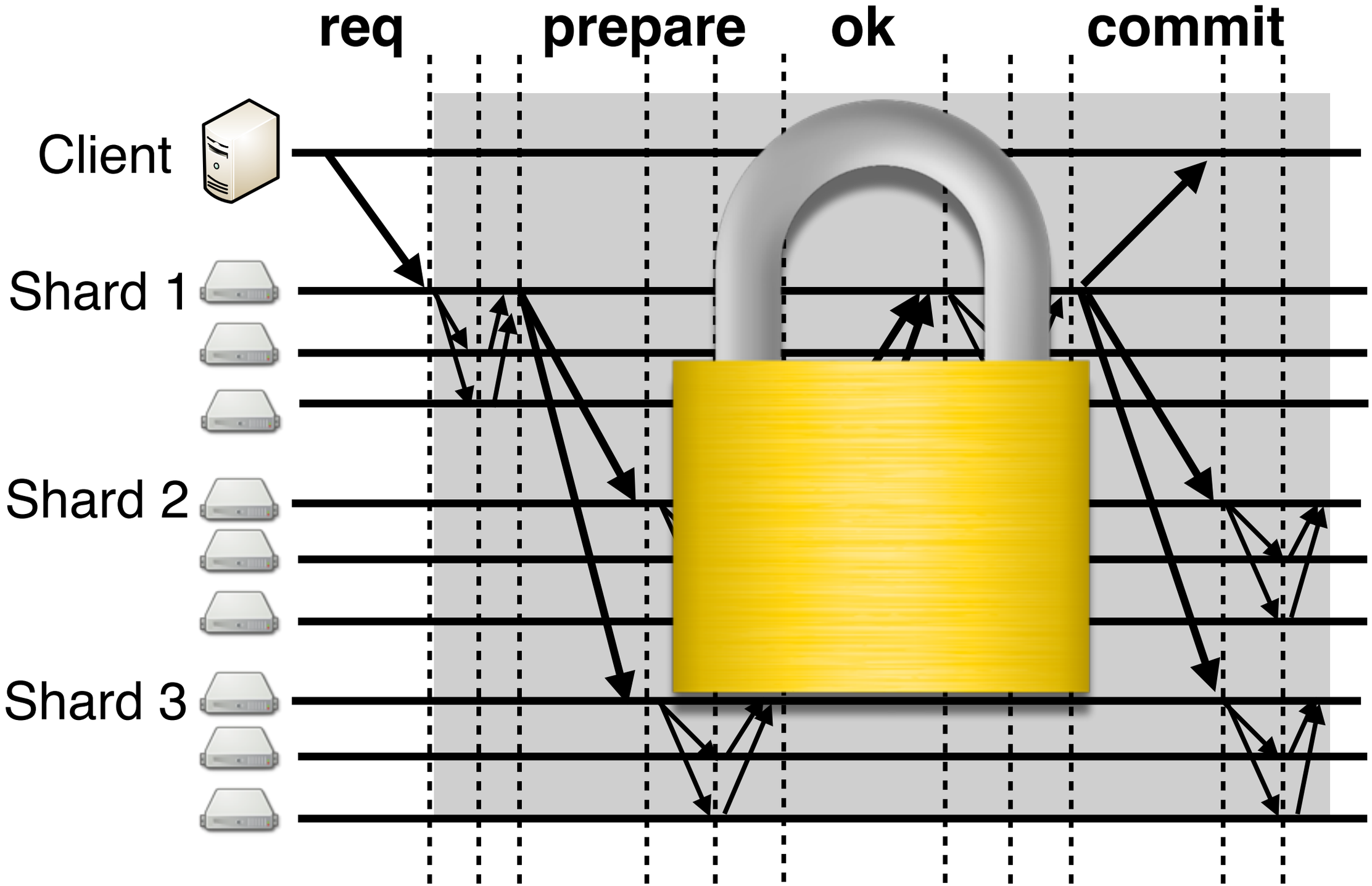
Existing transactional systems: extensive coordination



Existing transactional systems: extensive coordination



Existing transactional systems: extensive coordination



In this talk … 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**

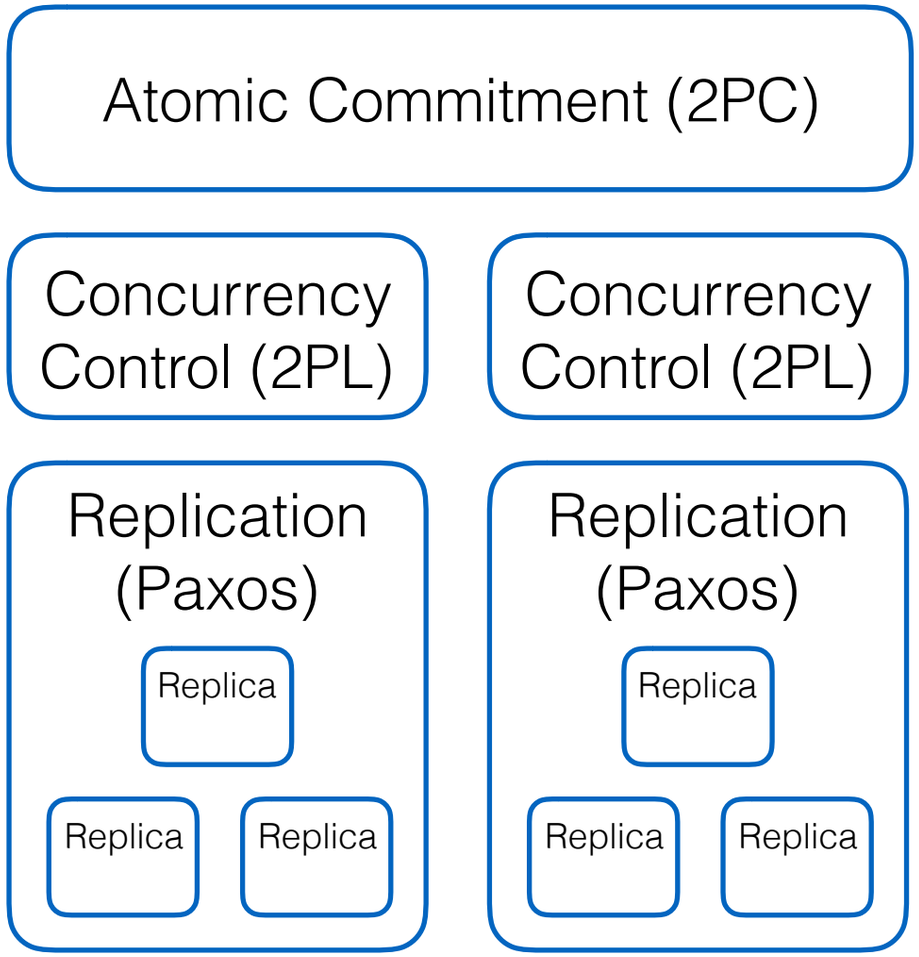
Key Contributions

A **new architecture** that divides the responsibility for transactional guarantees in a new way

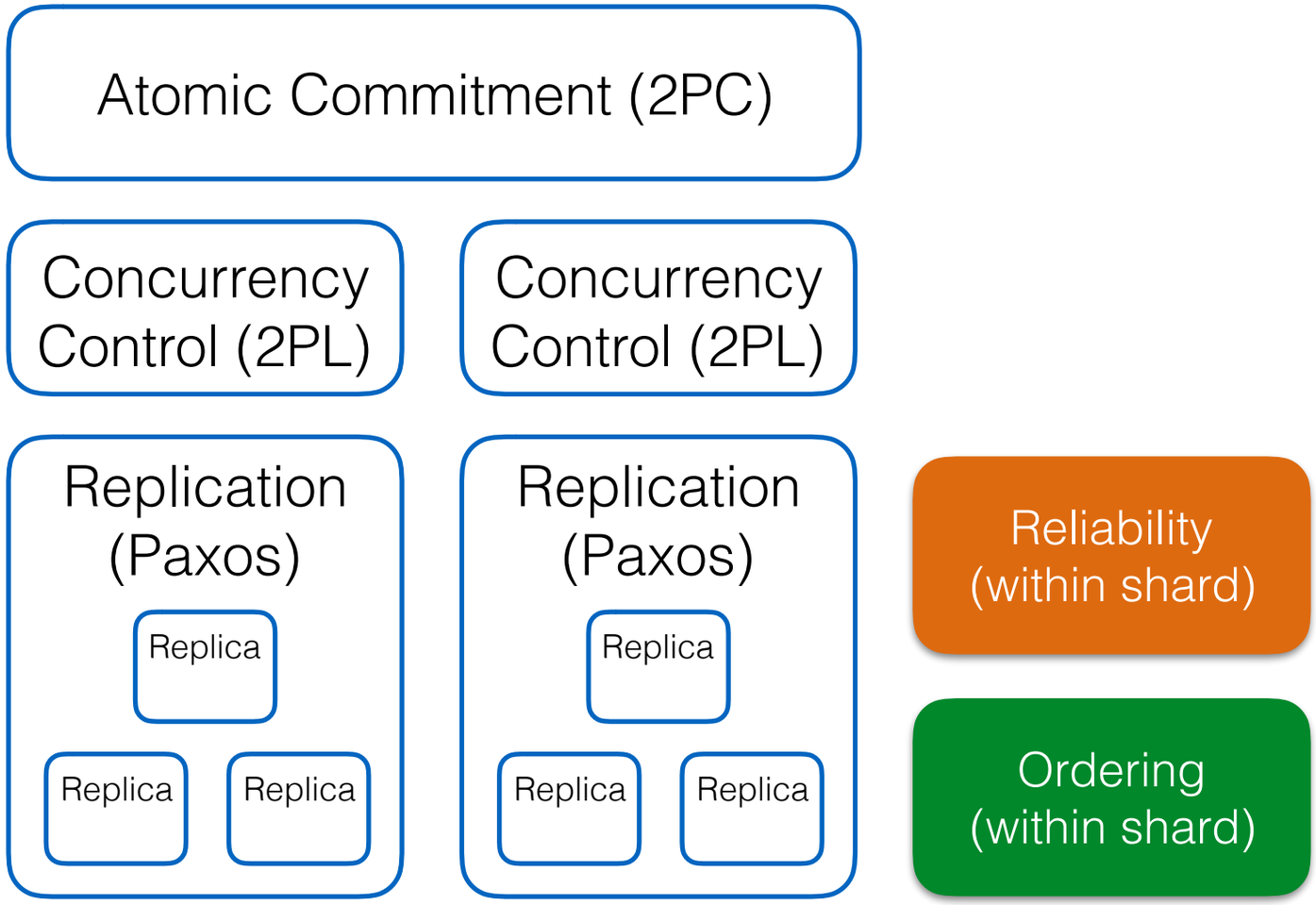
…leveraging the **datacenter network** to order messages within and across shards

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

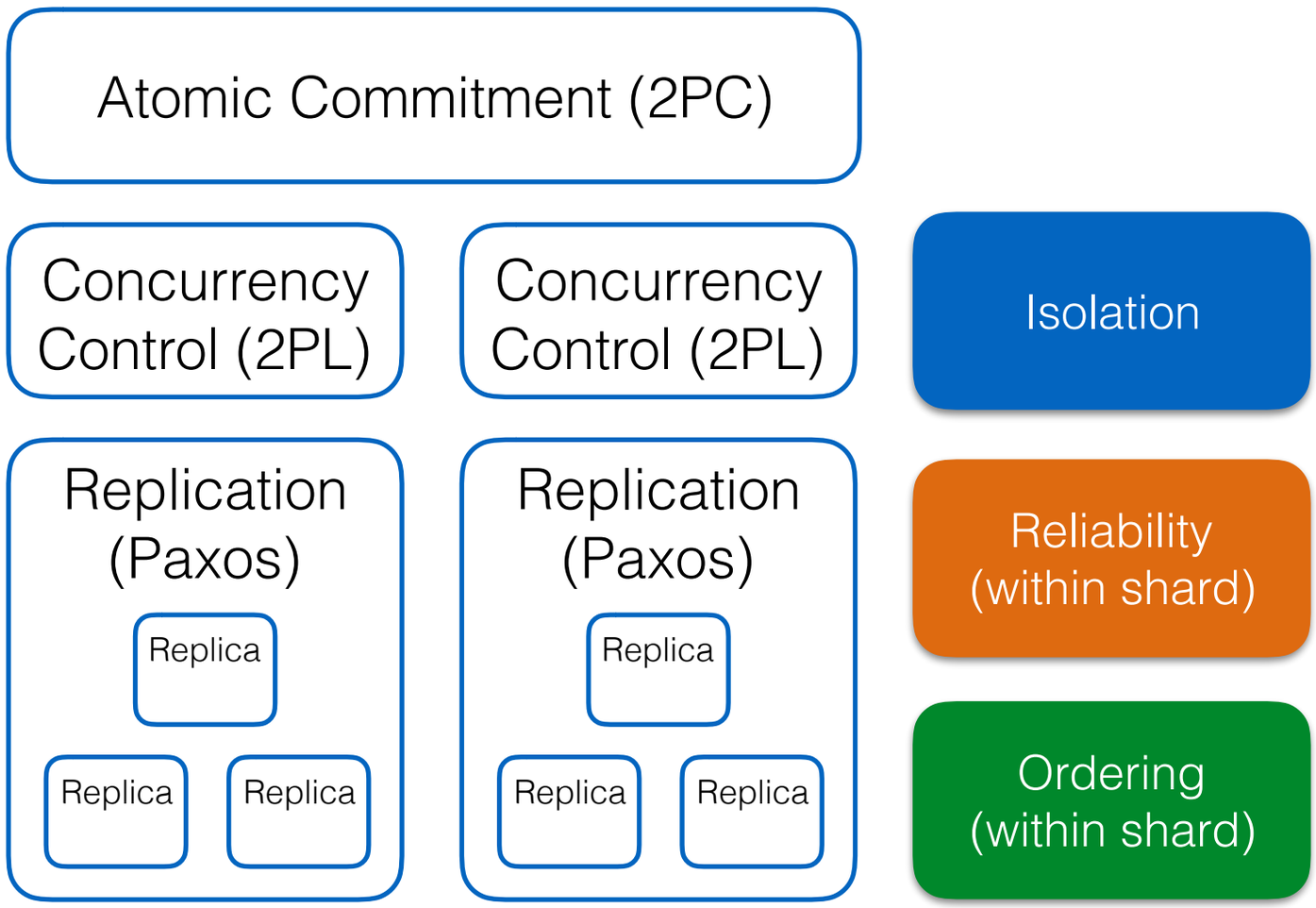
Traditional Layered Approach



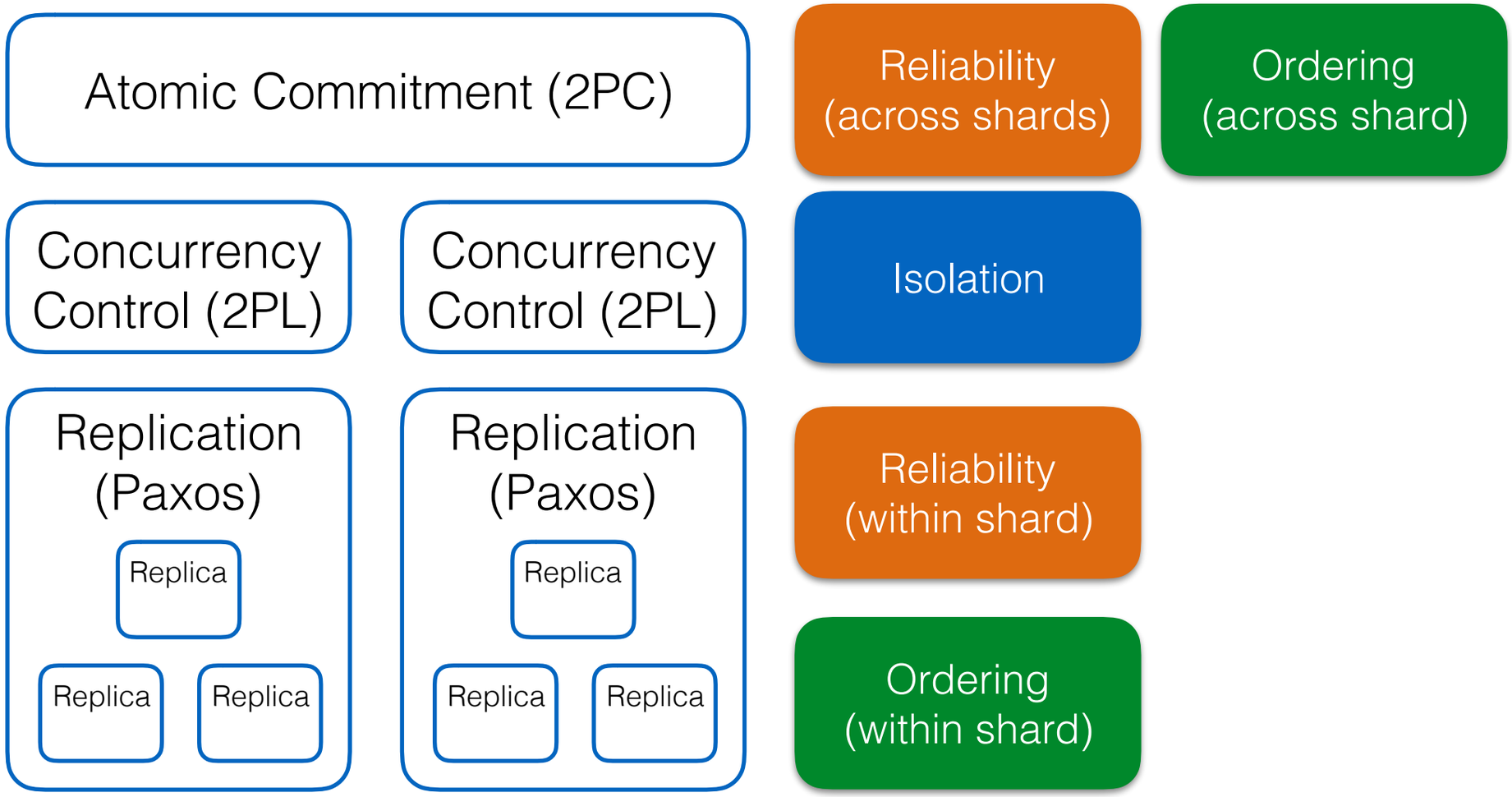
Traditional Layered Approach



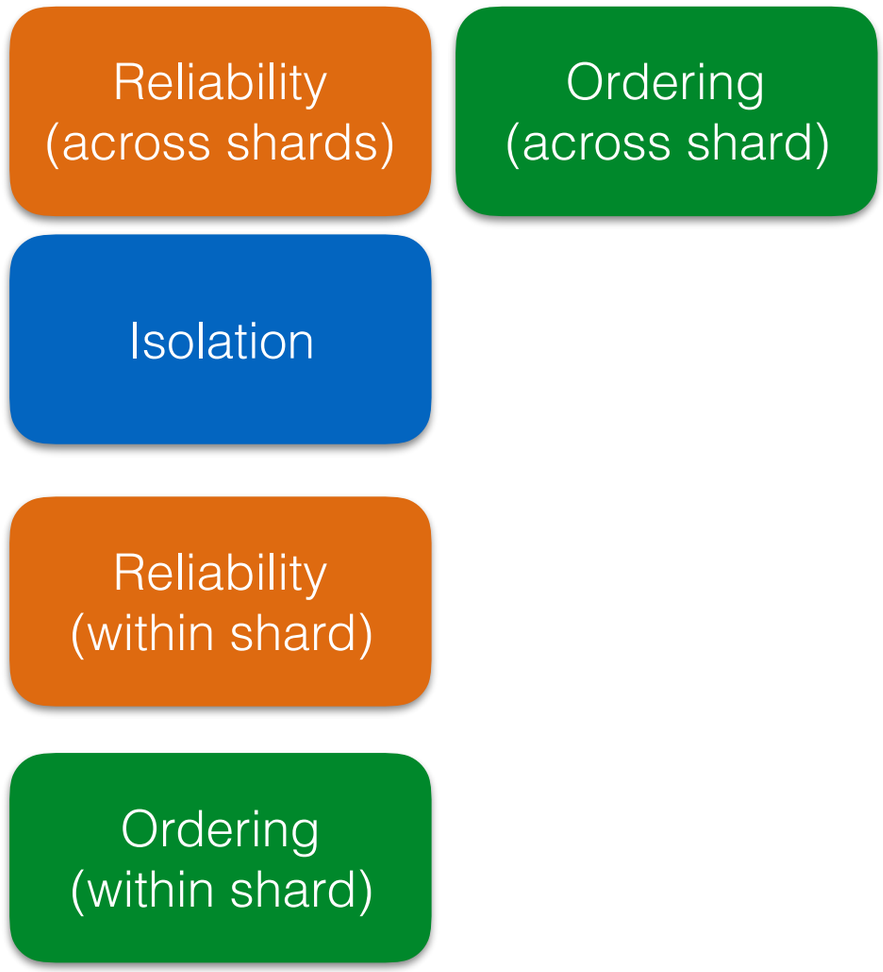
Traditional Layered Approach



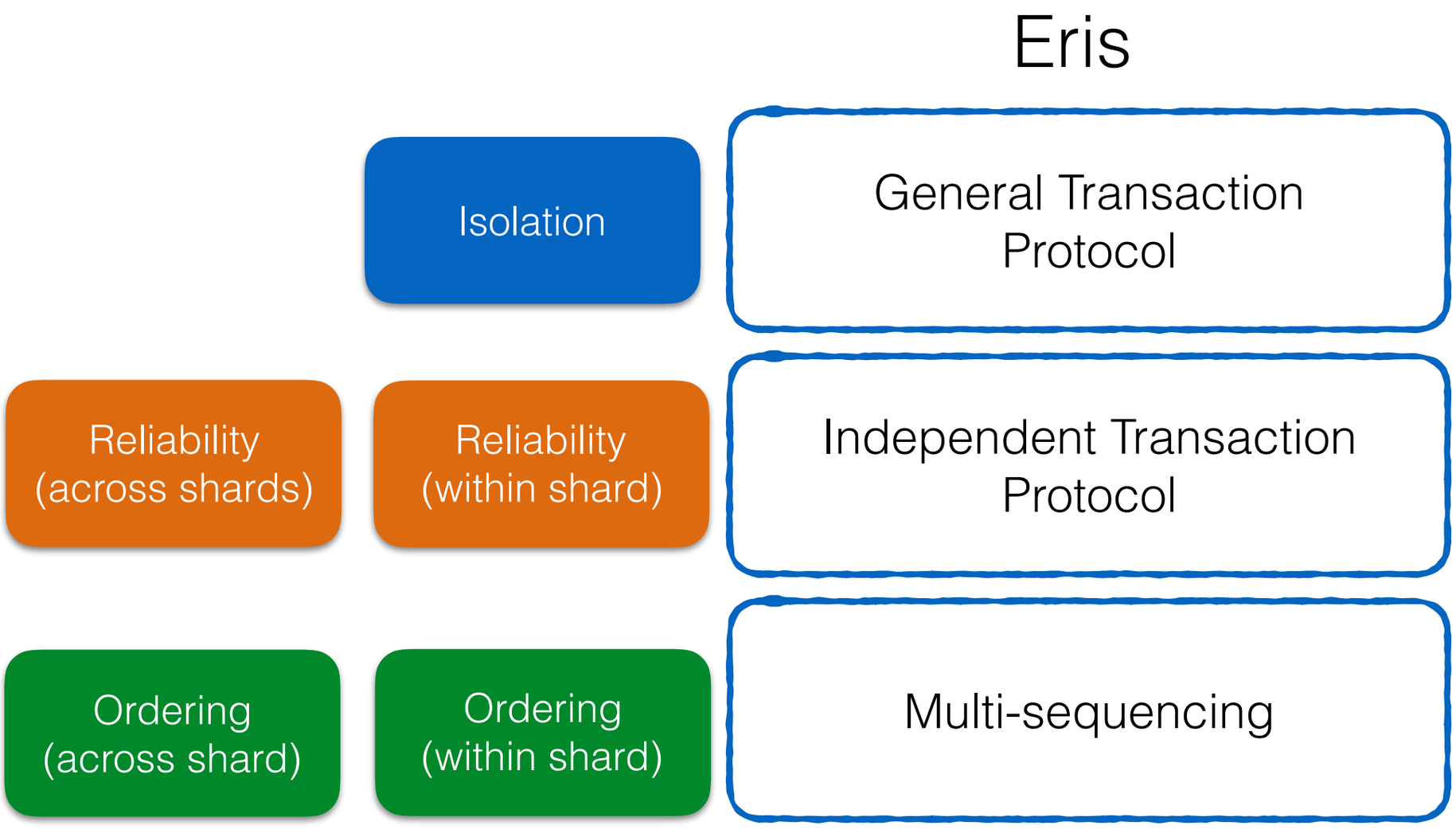
Traditional Layered Approach



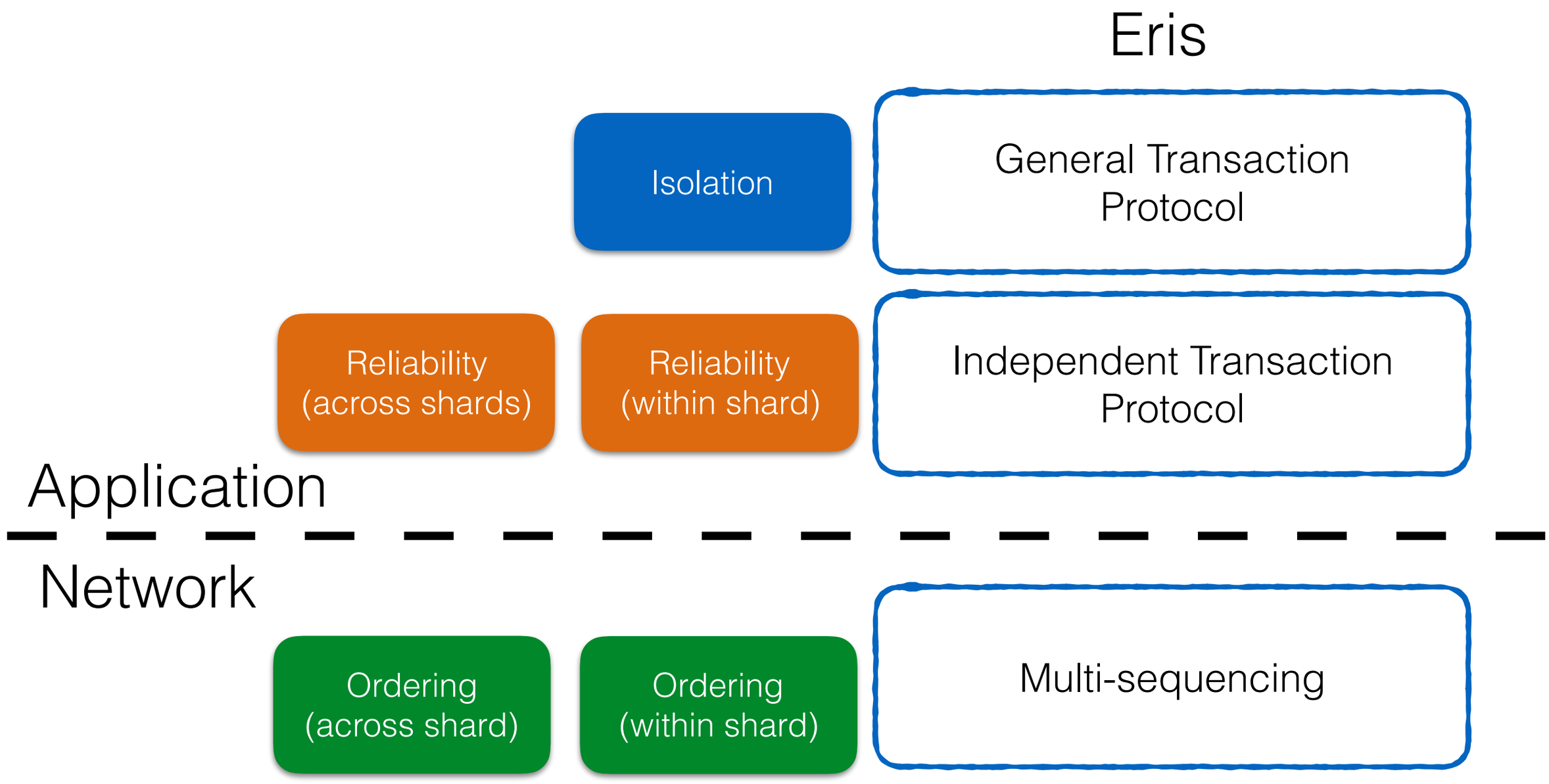
Traditional Layered Approach



A new way to divide the responsibilities for different guarantees



A new way to divide the responsibilities for different guarantees



Outline

1. Introduction
2. In-Network Concurrency Control
3. Transaction Model
4. Eris Protocol
5. Evaluation

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 [OSDI ’16]

Sequencer

Counter:
A0 B0 C0

A

B

C

Receivers

Sequencer

Counter:
A0 B0 C0

T1
(ABC)

A

B

C

Receivers

Sequencer

Counter:
A0 B0 C0

A

B

C

Receivers

Sequencer

Counter:
A1 B1 C1

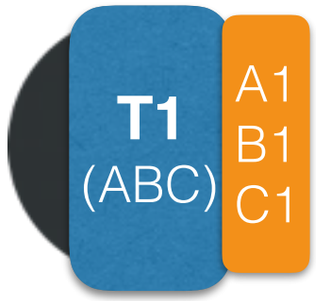
A

B

C

Receivers

Sequencer



Counter:
A1 B1 C1

A

B

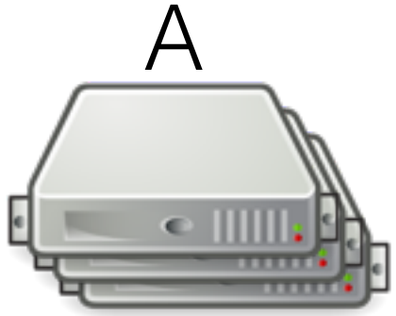
C

Receivers

Sequencer

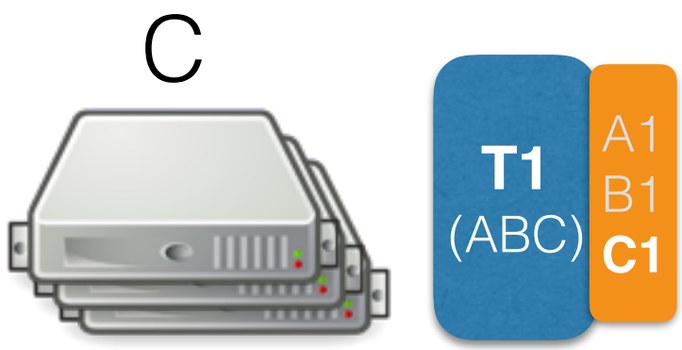
Counter:
A1 B1 C1

A



B

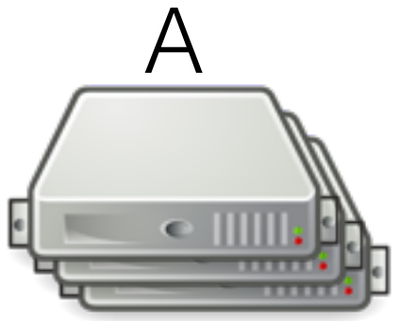
C

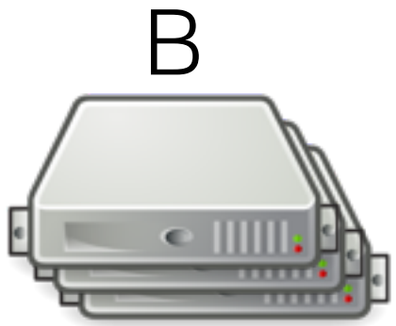


Receivers

Sequencer

Counter:
A1 B1 C1

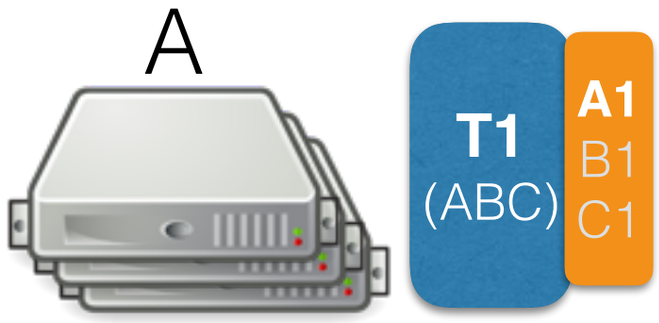


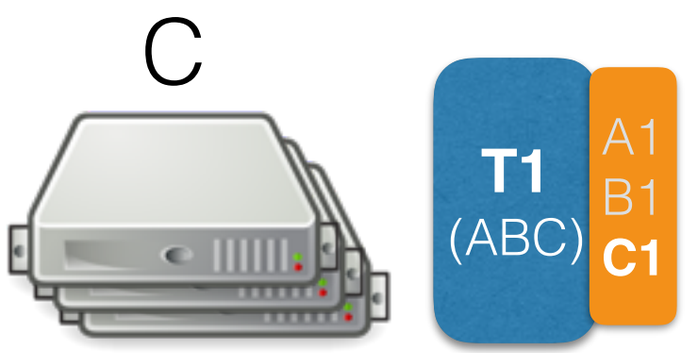


Receivers

Sequencer

Counter:
A1 B1 C1



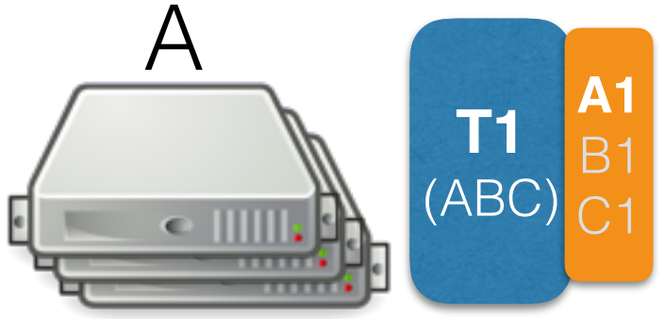


Receivers

Sequencer

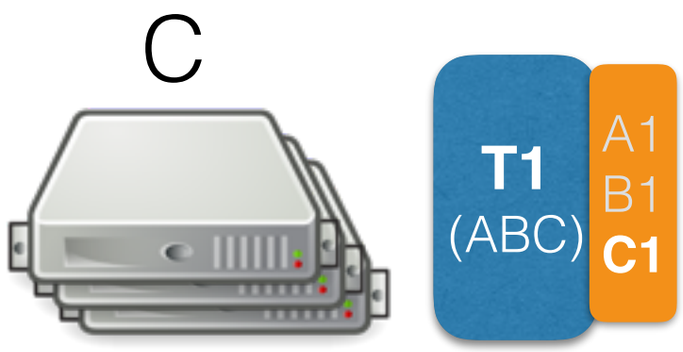
Counter:
A2 B2 C1

A



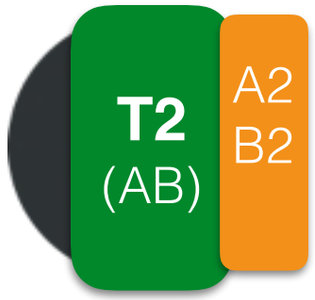
B

C



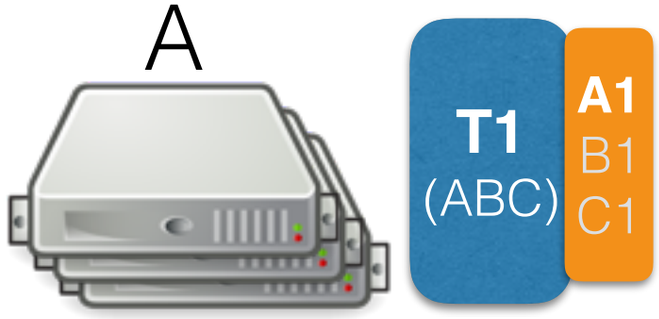
Receivers

Sequencer



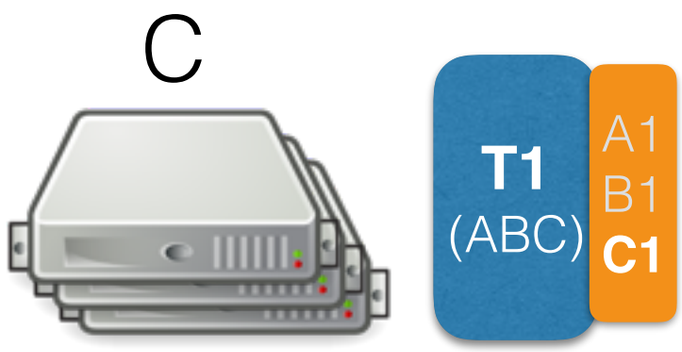
Counter:
A2 B2 C1

A



B

C

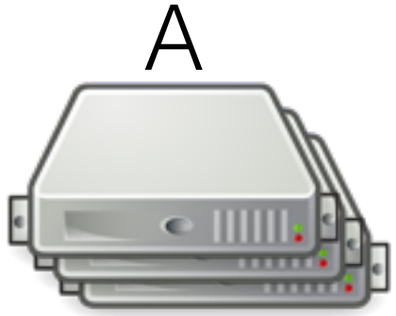


Receivers

Sequencer

Counter:
A2 B2 C1

A



T1
(ABC)
A1
B1
C1

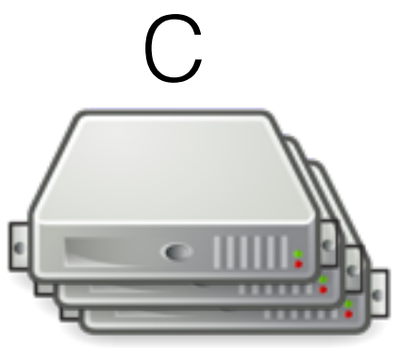
T2
(AB)
A2
B2

B

T1
(ABC)
A1
B1
C1

T2
(AB)
A2
B2

C



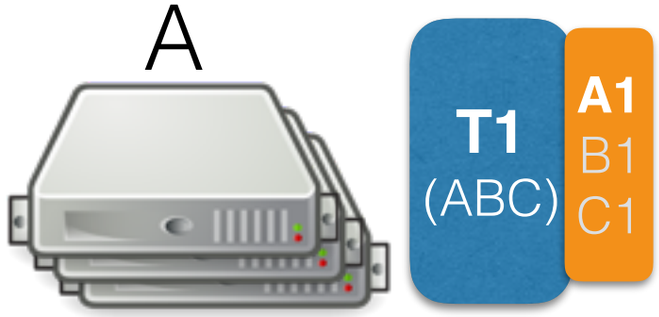
T1
(ABC)
A1
B1
C1

Receivers

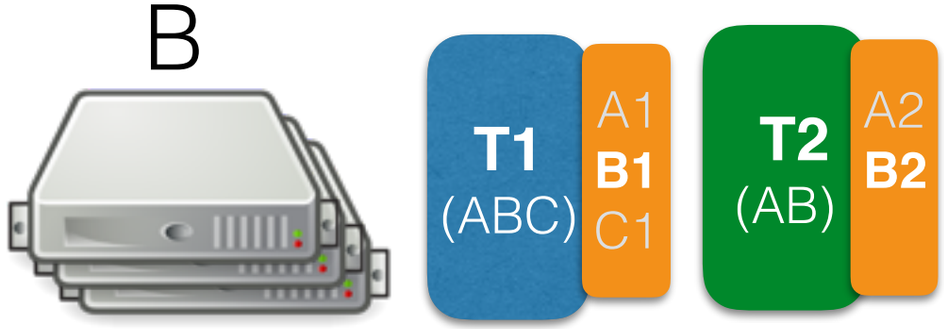
Sequencer

Counter:
A2 B2 C1

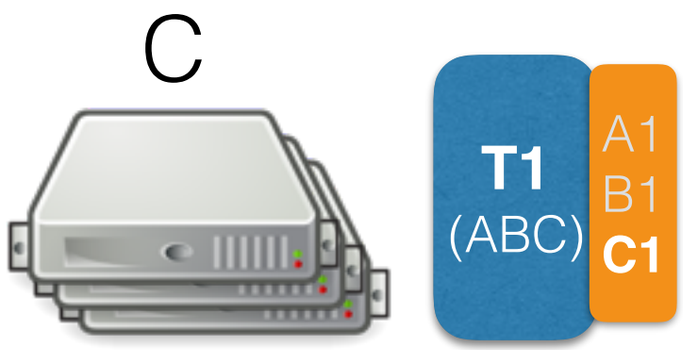
A



B



C

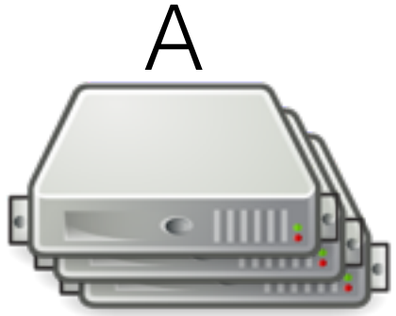


Receivers

Sequencer

Counter:
A2 B2 C1

A



T1
(ABC)
A1
B1
C1

B

T1
(ABC)
A1
B1
C1

T2
(AB)
A2
B2

C

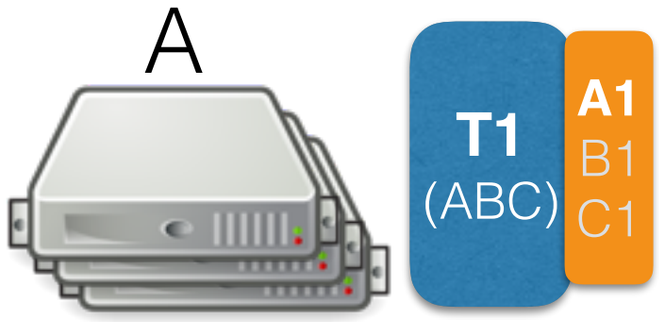
T1
(ABC)
A1
B1
C1

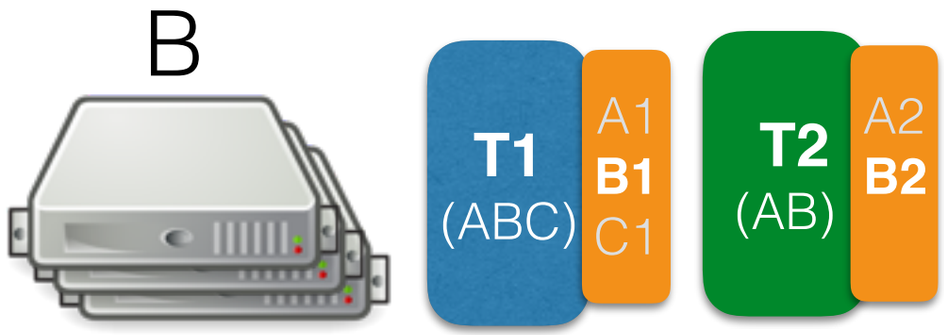
T3
(A)

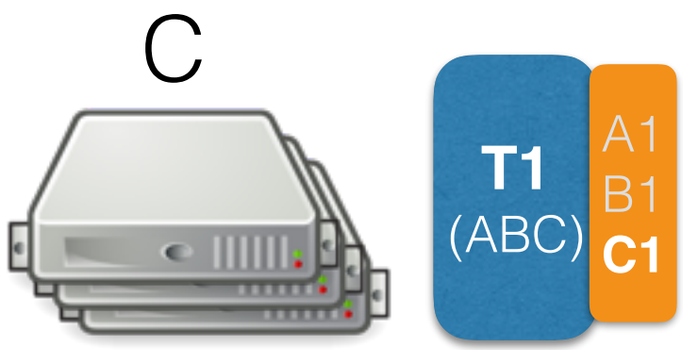
Receivers

Sequencer

Counter:
A2 B2 C1

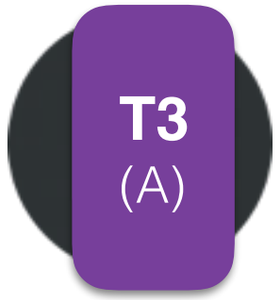






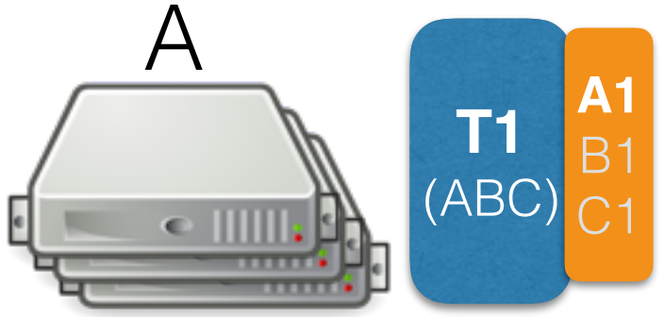
Receivers

Sequencer

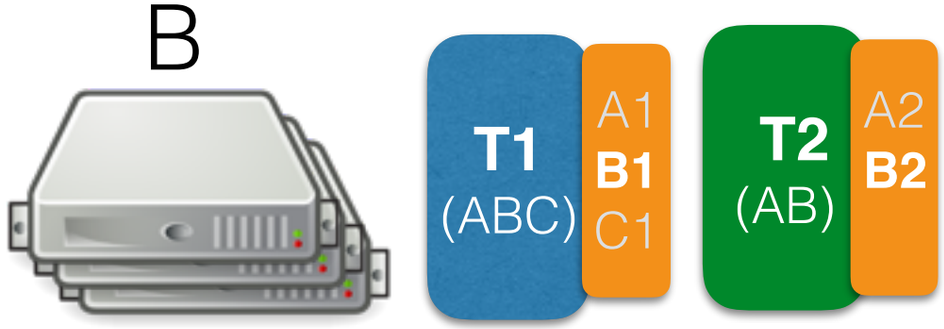


Counter:
A3 B2 C1

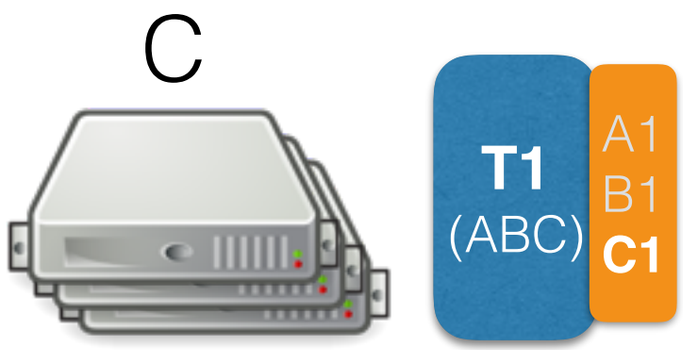
A



B



C

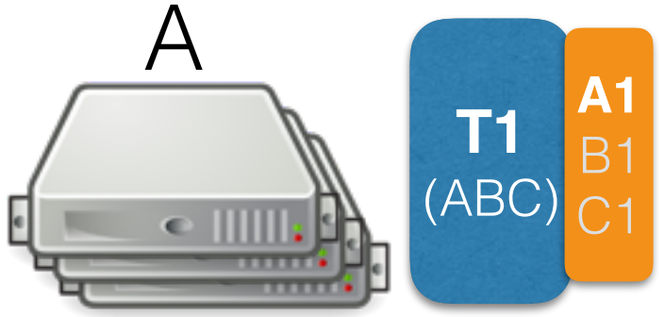


Receivers

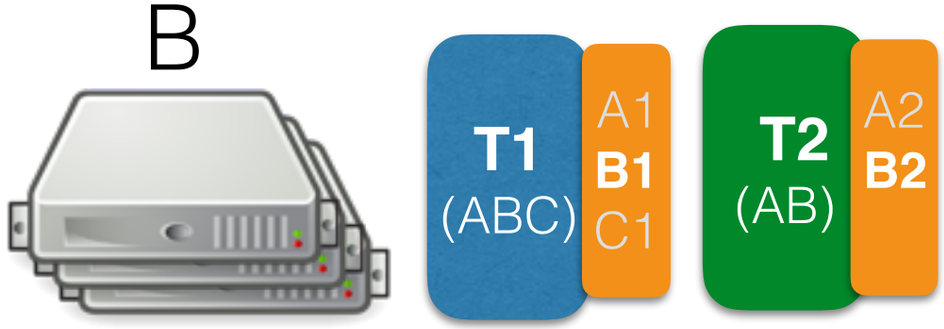
Sequencer

Counter:
A3 B2 C1

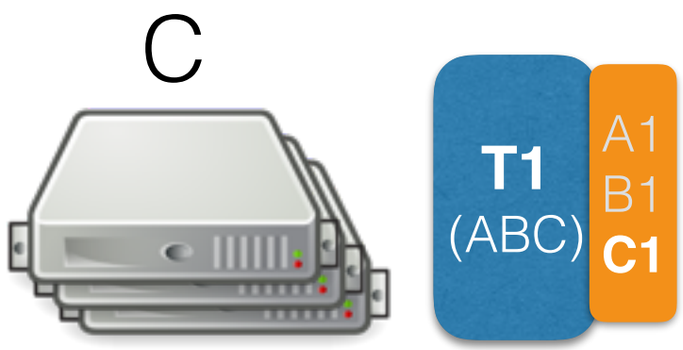
A



B



C

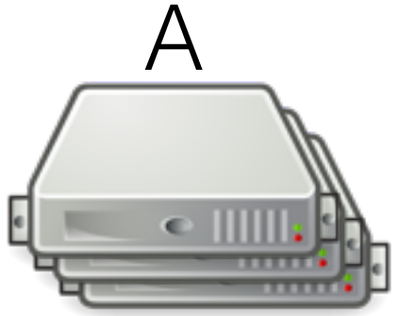


Receivers

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

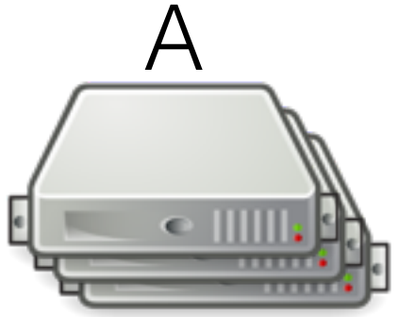
T1 (ABC) A1 B1 **C1**

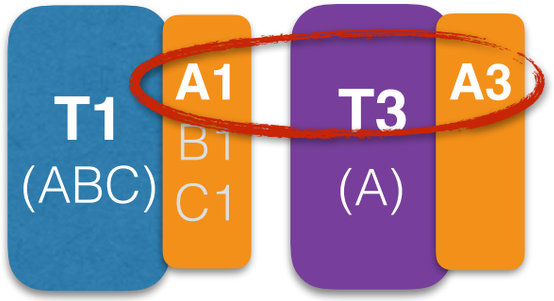
Receivers

Sequencer

Counter:
A3 B2 C1

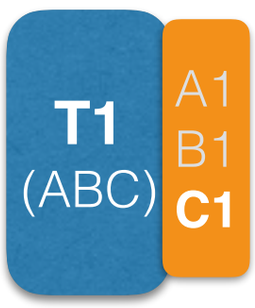
A





B

C

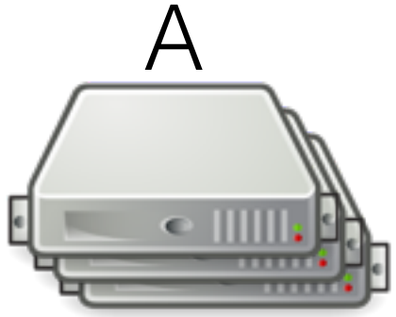


Receivers

Sequencer

Counter:
A3 B2 C1

A



T1 (ABC) A1 B1 C1

DROP

T3 (A) A3

B

T1 (ABC) A1 **B1** C1

T2 (AB) A2 **B2**

C

T1 (ABC) A1 B1 **C1**

Receivers

Network Implementation

- Groupcast routing using OpenFlow
- Sequencer implementations:
  - Programmable switches, written in P4
  - Middlebox prototype using network processors
- Global epoch number for sequencer failures

What have we accomplished so far?

- Consistently ordered groupcast primitive with drop detection
- How do we go from multi-sequenced groupcast to transactions?

Outline

1. Introduction
2. In-Network Concurrency Control
3. Transaction Model
4. Eris Protocol
5. Evaluation

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 (e.g. TPC-C)

```
START TRANSA
UPDATE tb t1
SET t1.Salary =
WHERE t1.Salary < 500
COMMIT
```

```
CTION
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```

Name	Salary
Alice	600

Name	Salary
Bob	450

Name	Salary
Charlie	500

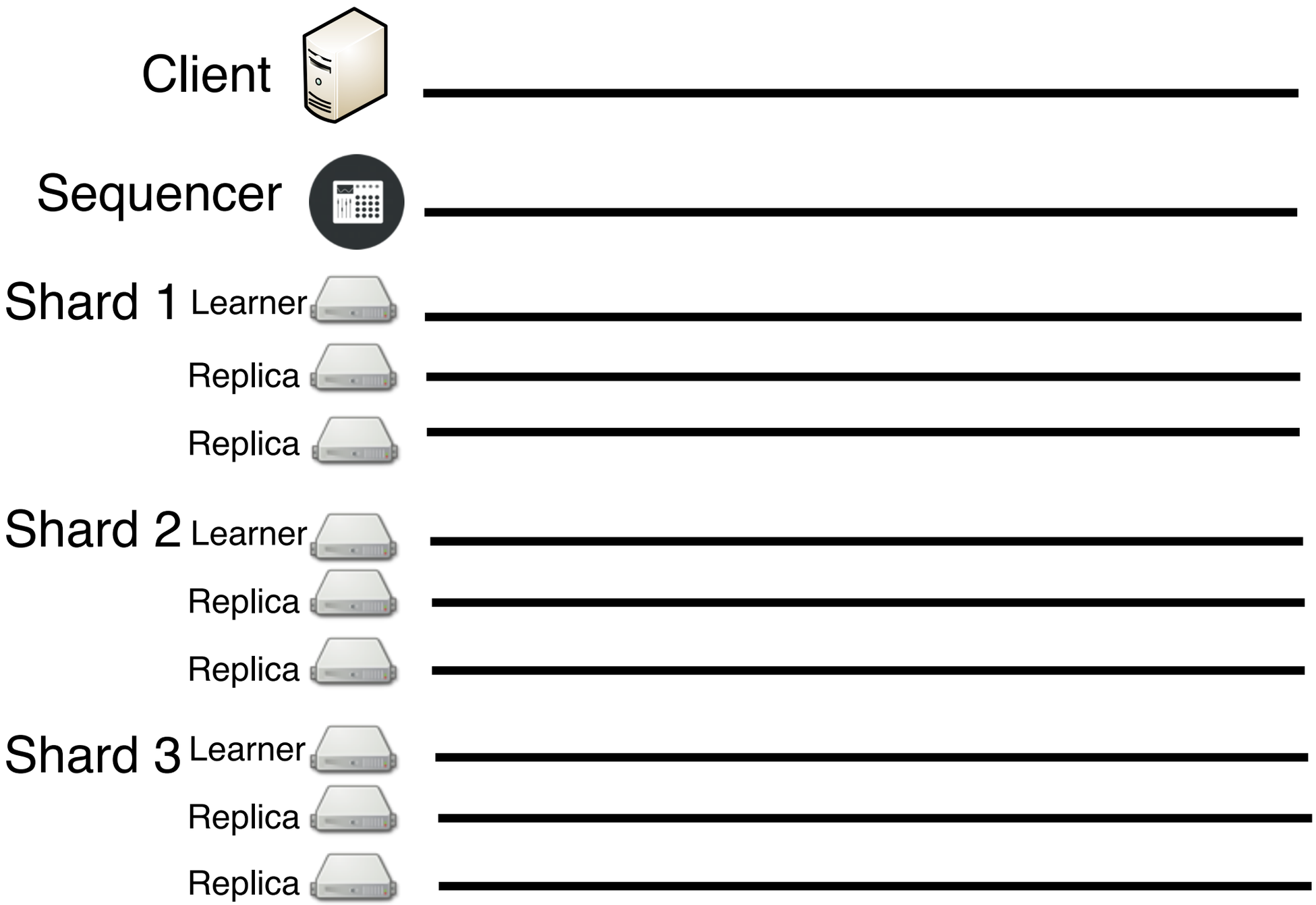
Why independent transactions?

- **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**?

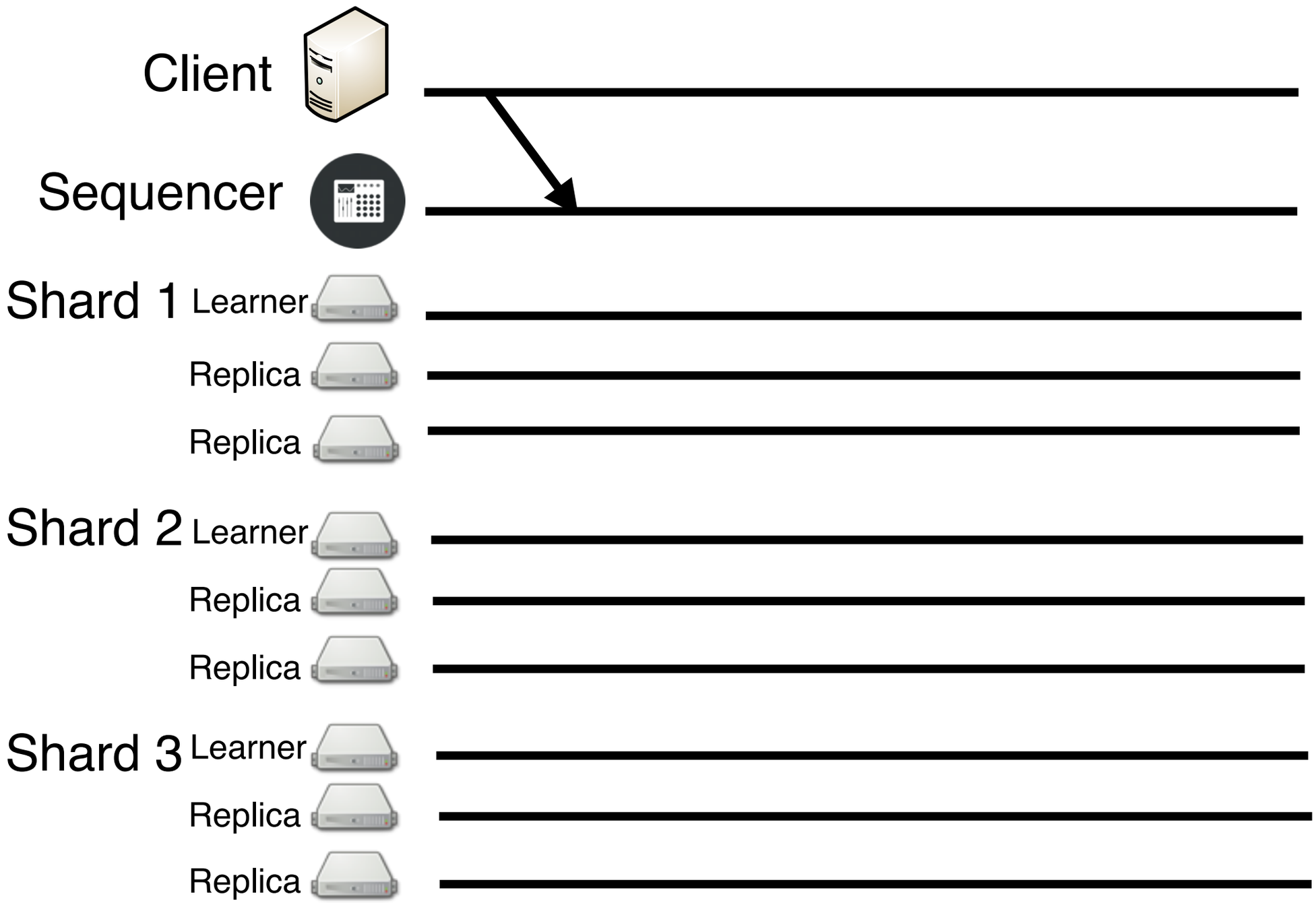
Outline

1. Introduction
2. In-Network Concurrency Control
3. Transaction Model
4. Eris Protocol
5. Evaluation

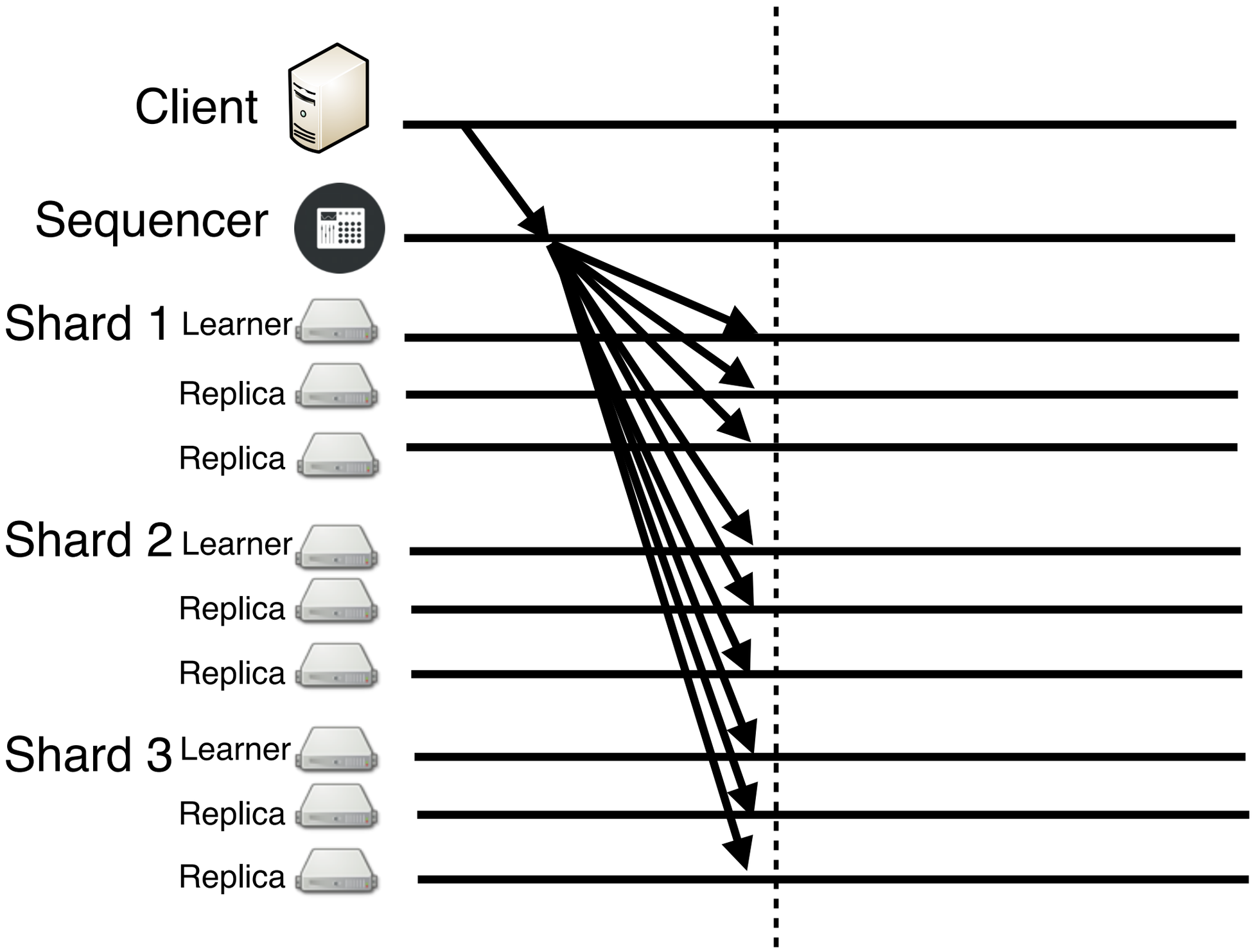
Normal Case


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

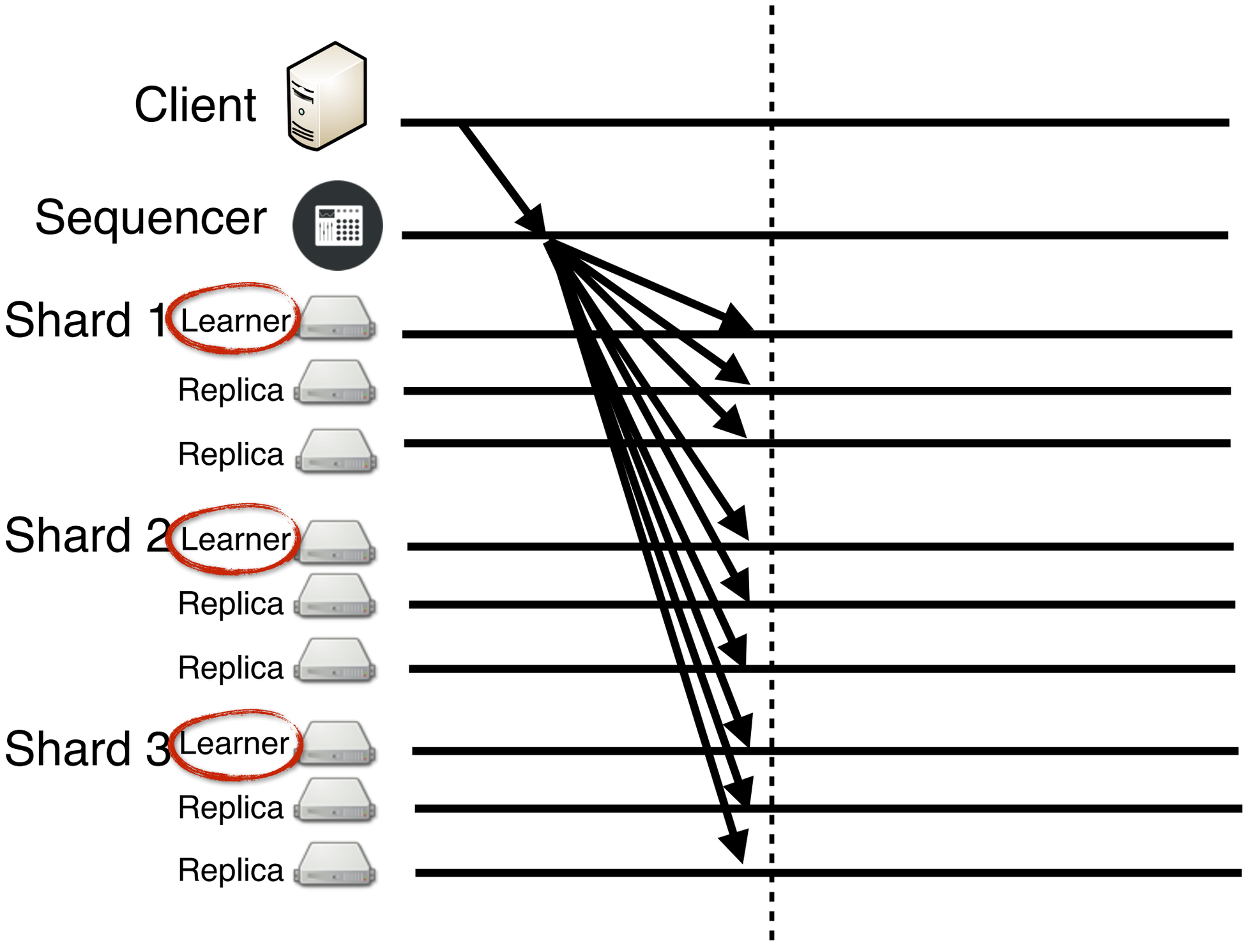
Normal Case


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

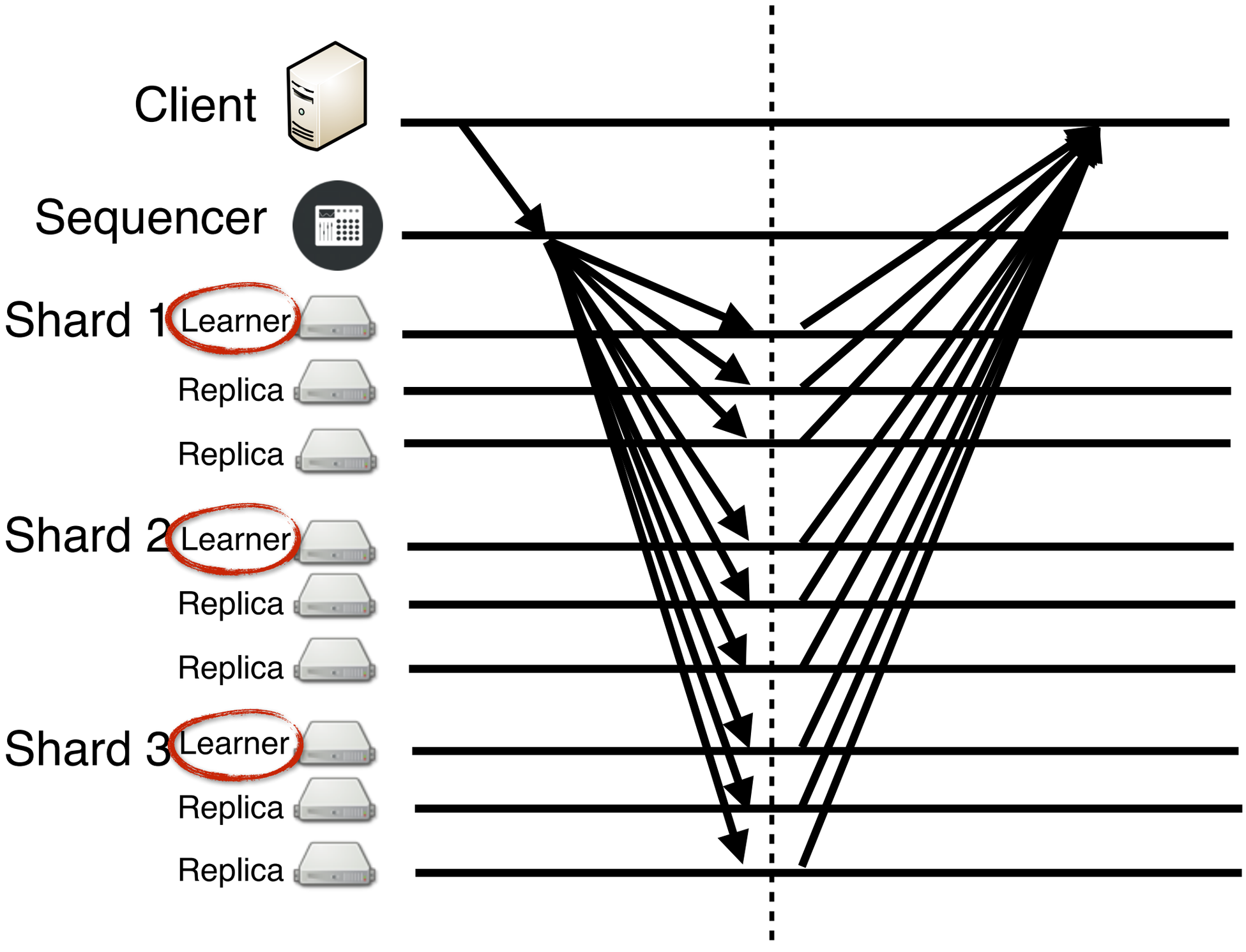
Normal Case


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

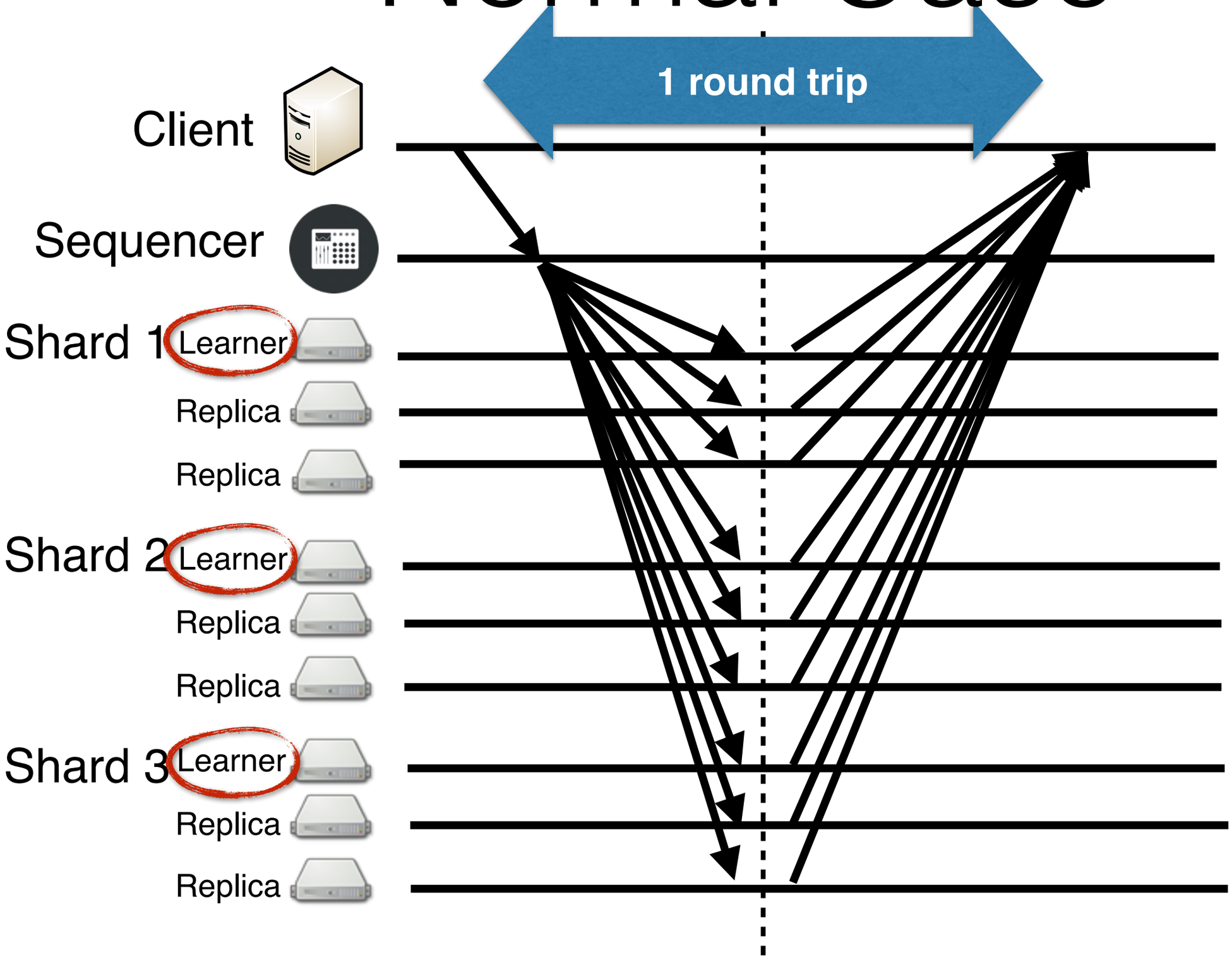
Normal Case


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

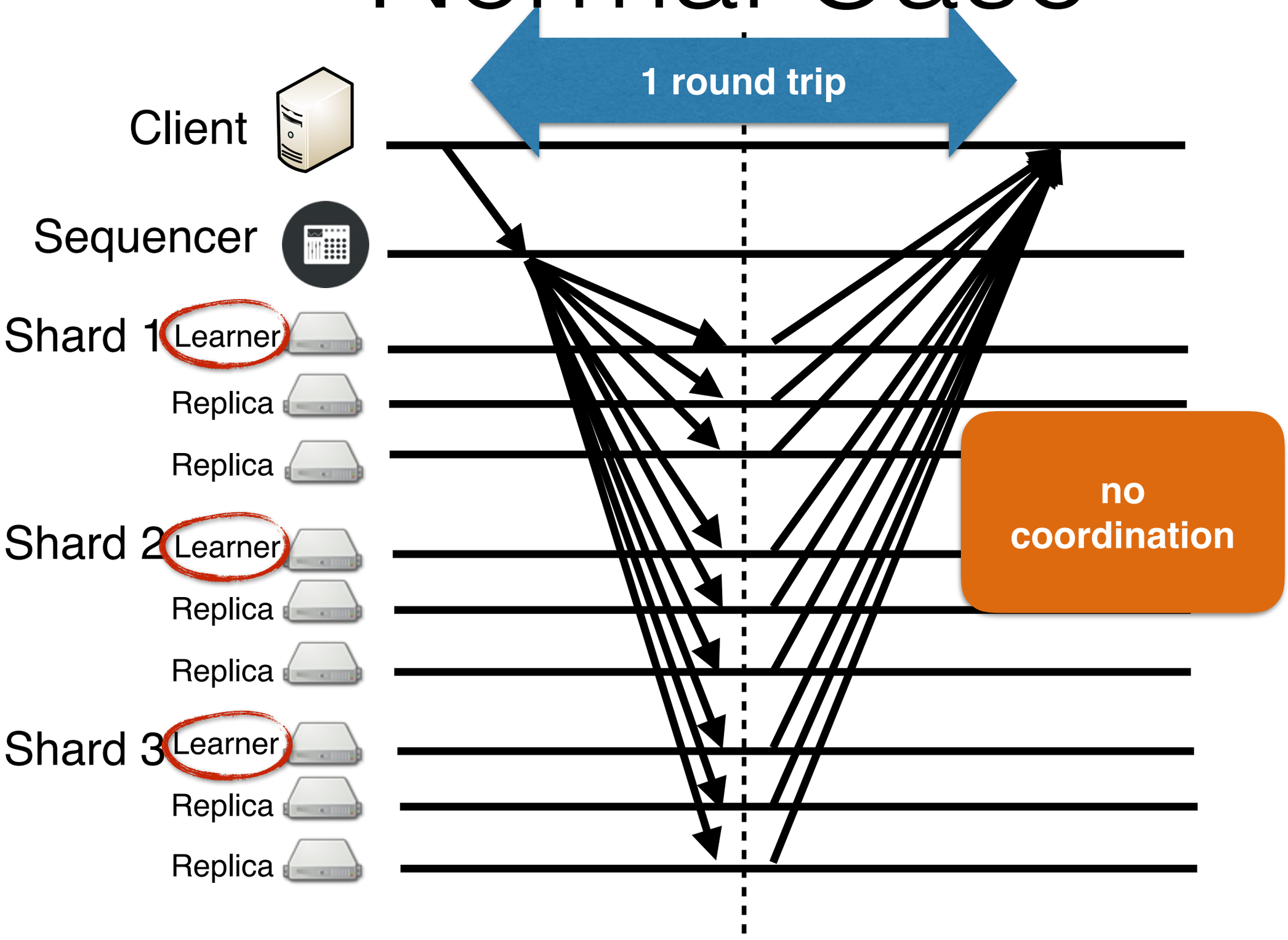
Normal Case


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

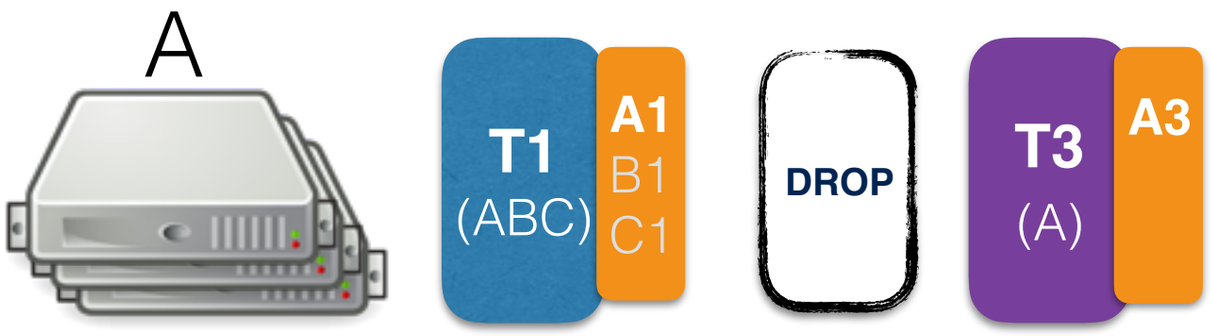
Normal Case


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

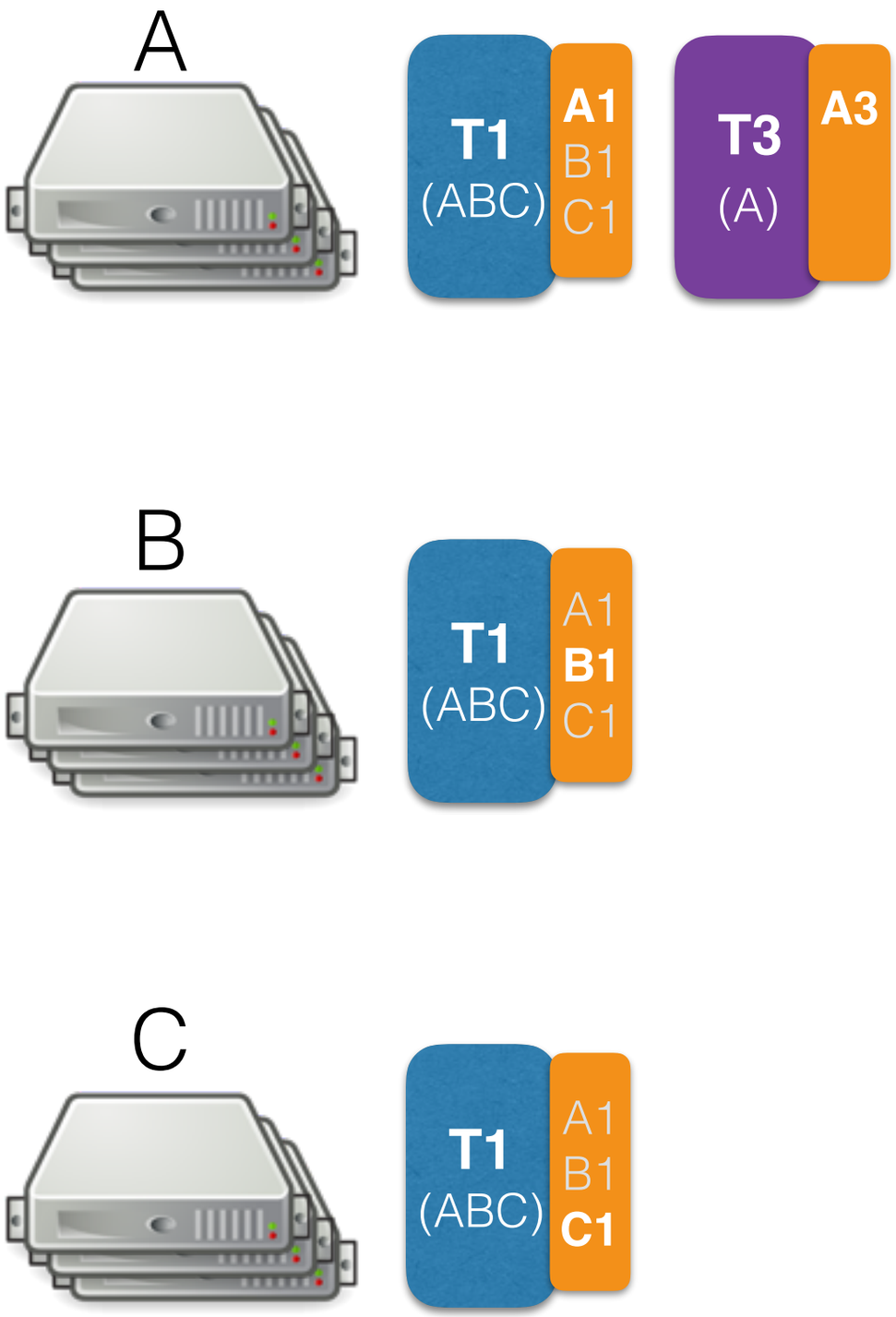
Normal Case


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

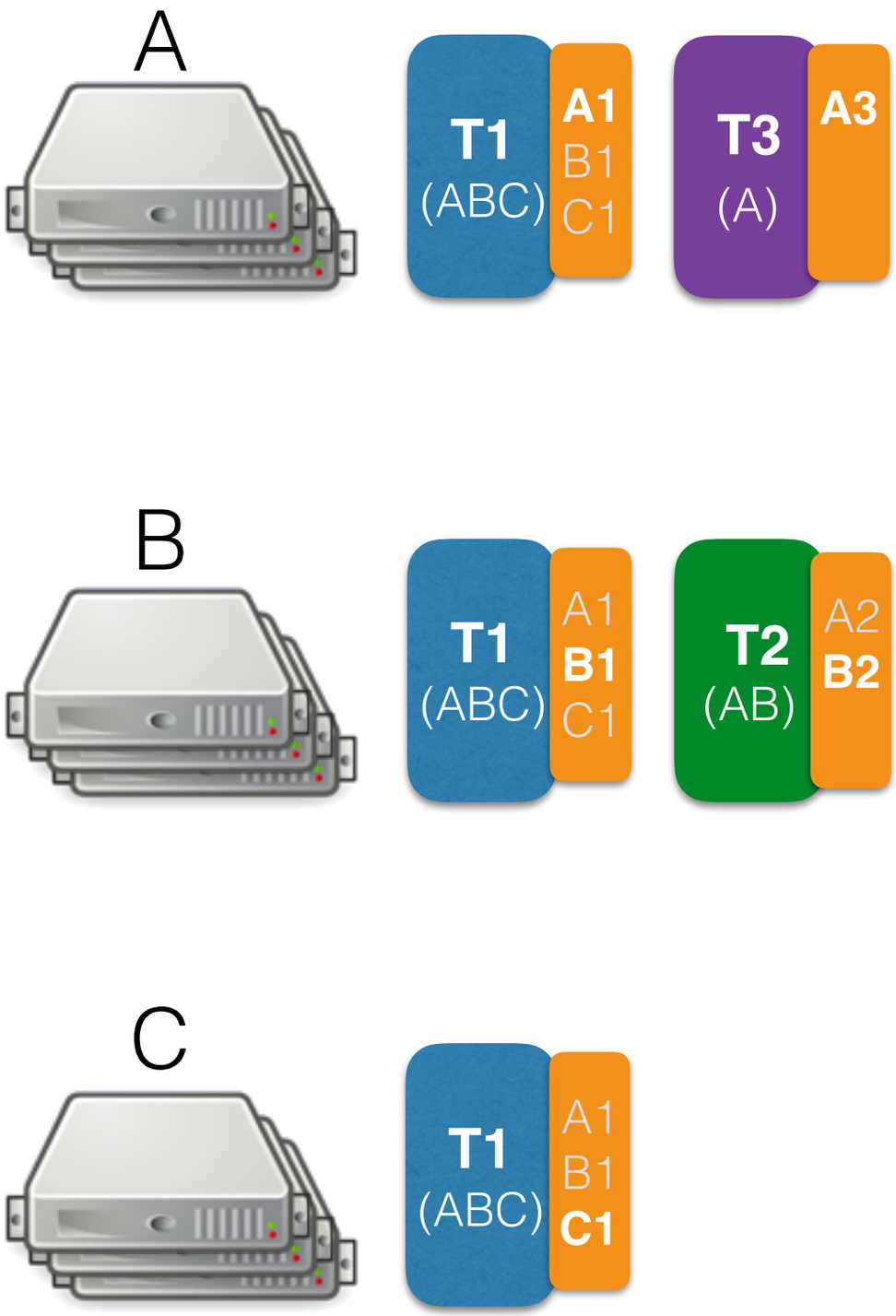
How to handle dropped messages?



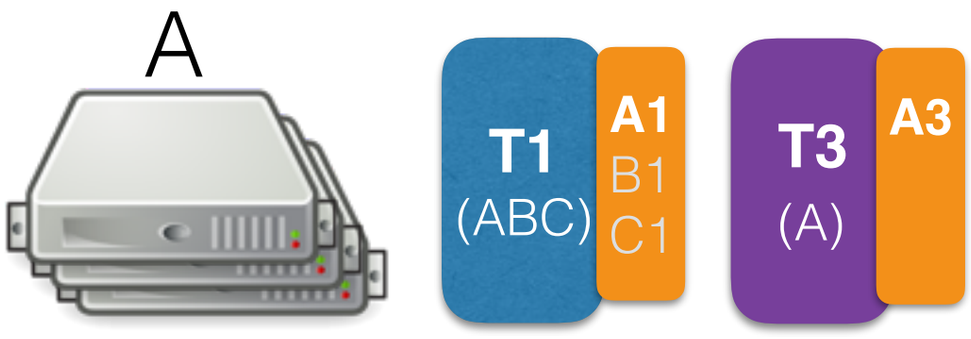
How to handle dropped messages?



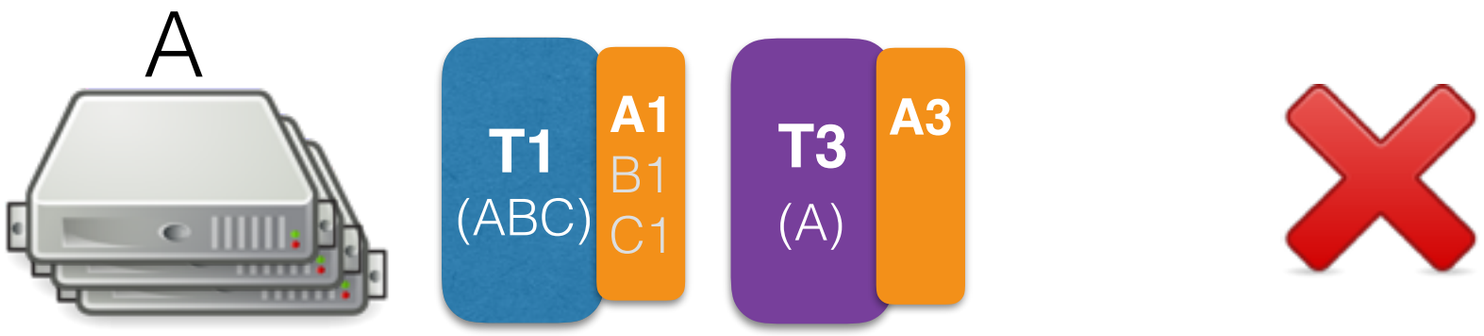
How to handle dropped messages?



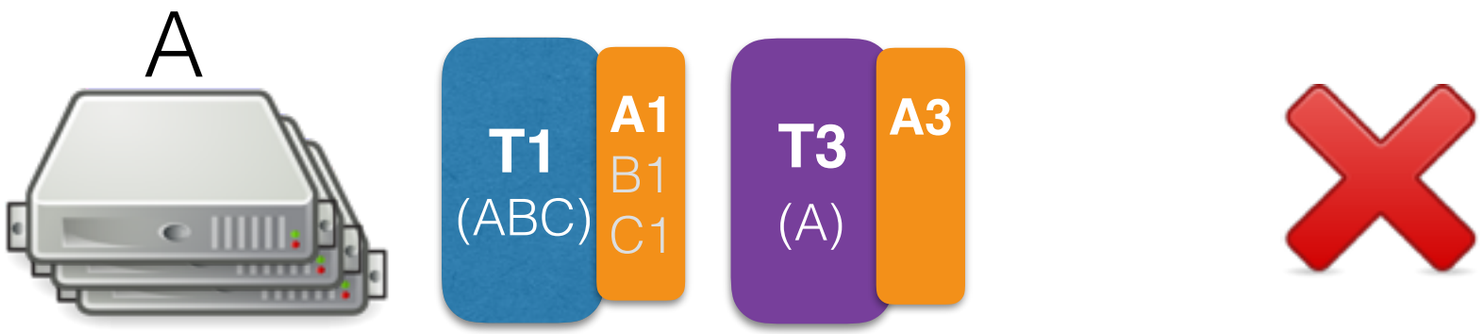
How to handle dropped messages?



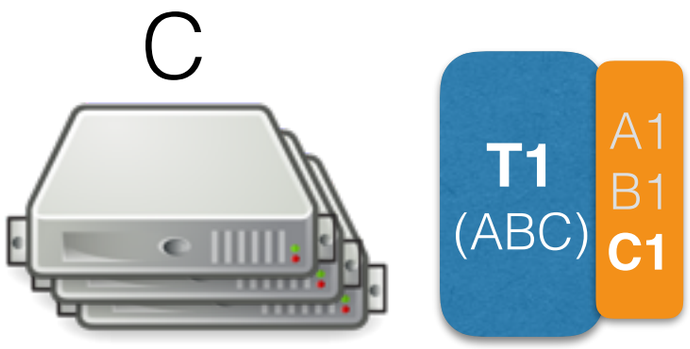
How to handle dropped messages?



How to handle dropped messages?

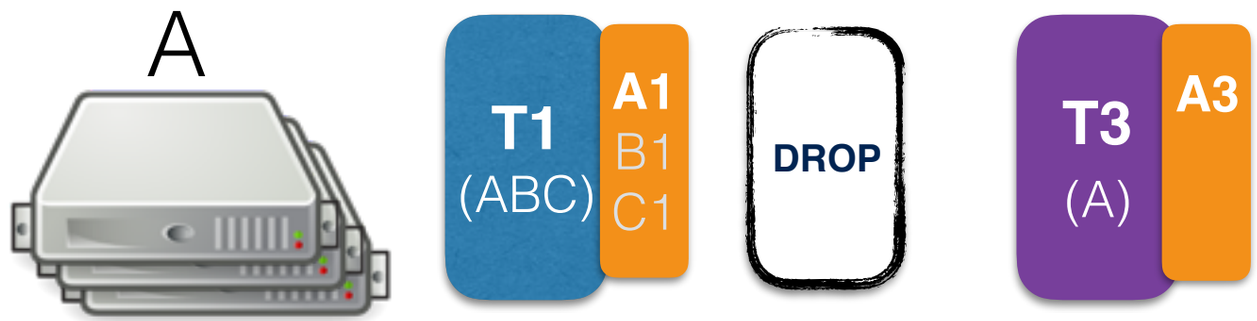


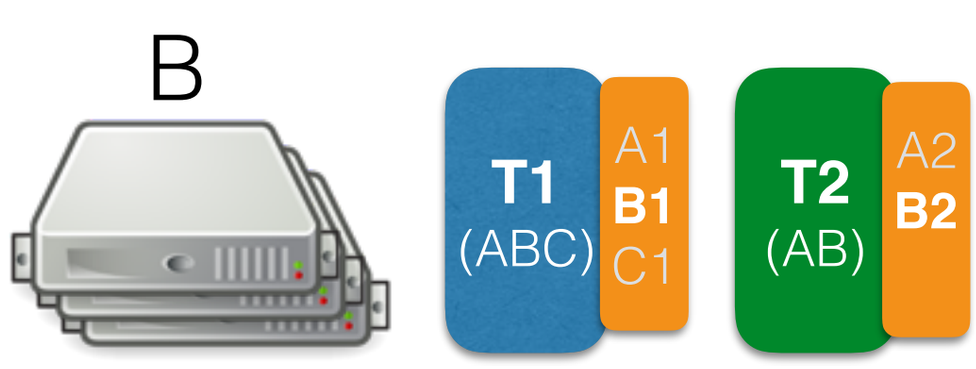
Global coordination problem

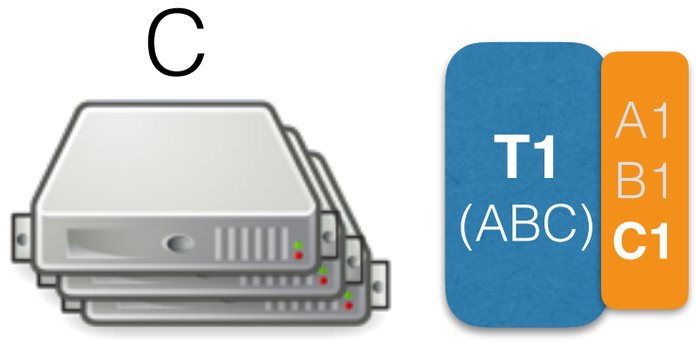


The Failure Coordinator

Failure Coordinator



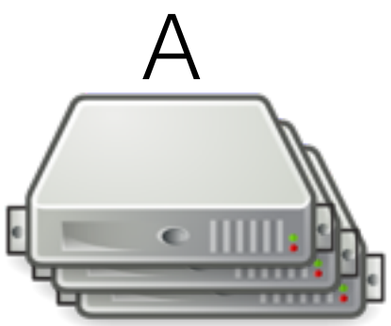


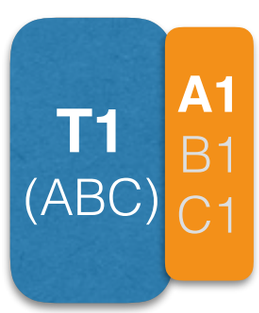


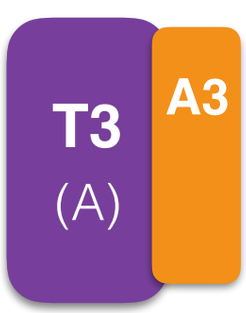
The Failure Coordinator

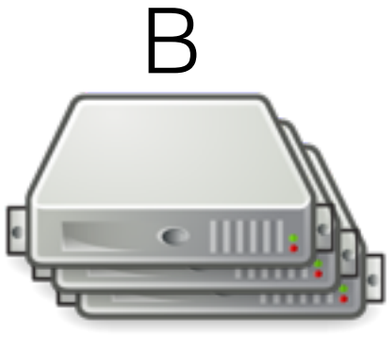
Failure Coordinator

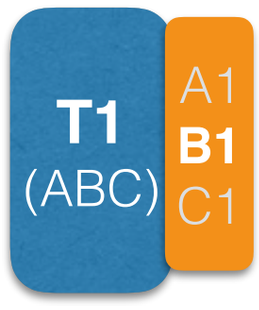
Received A2?

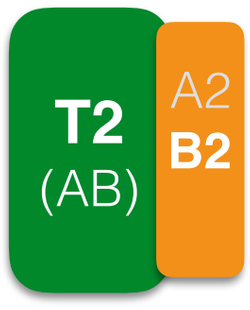


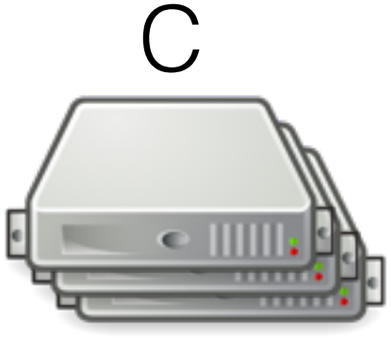


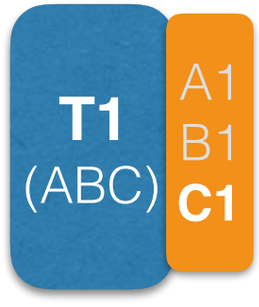








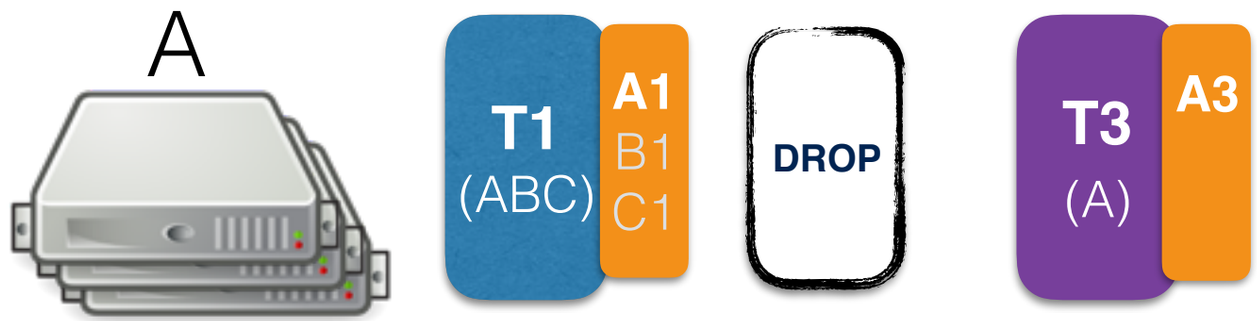


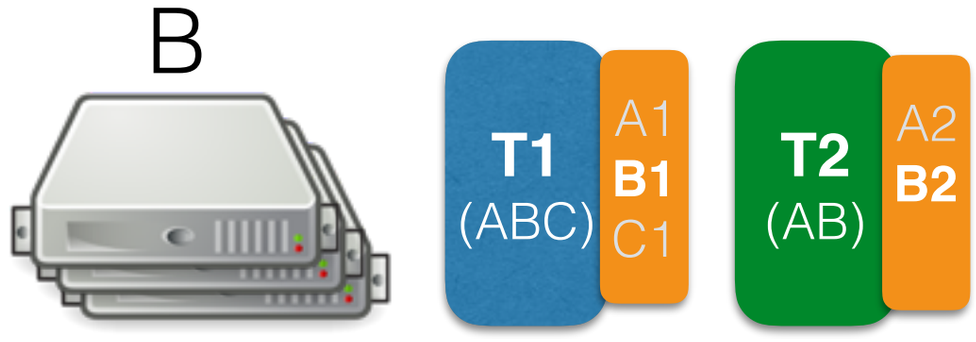


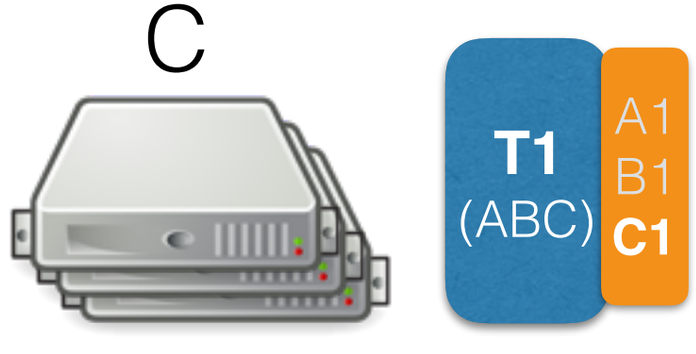
The Failure Coordinator

Failure Coordinator

Received A2?

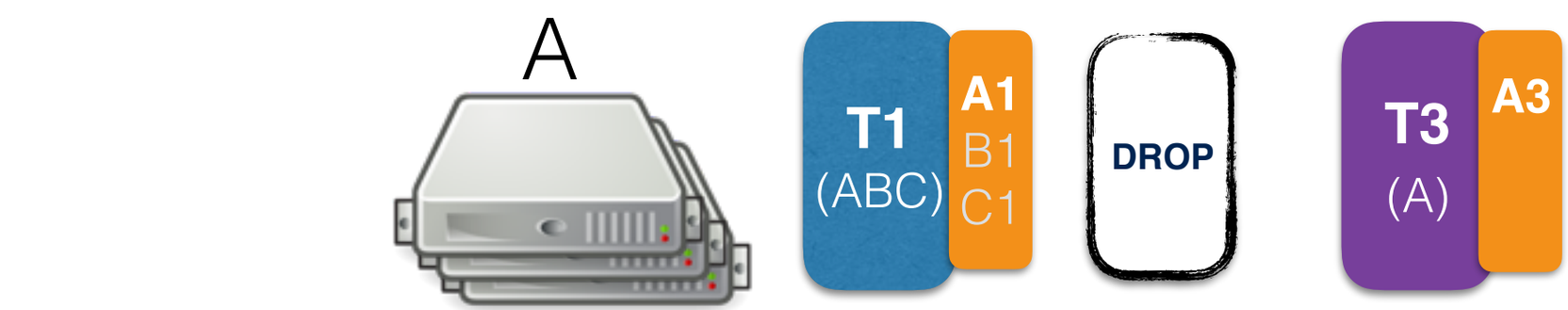


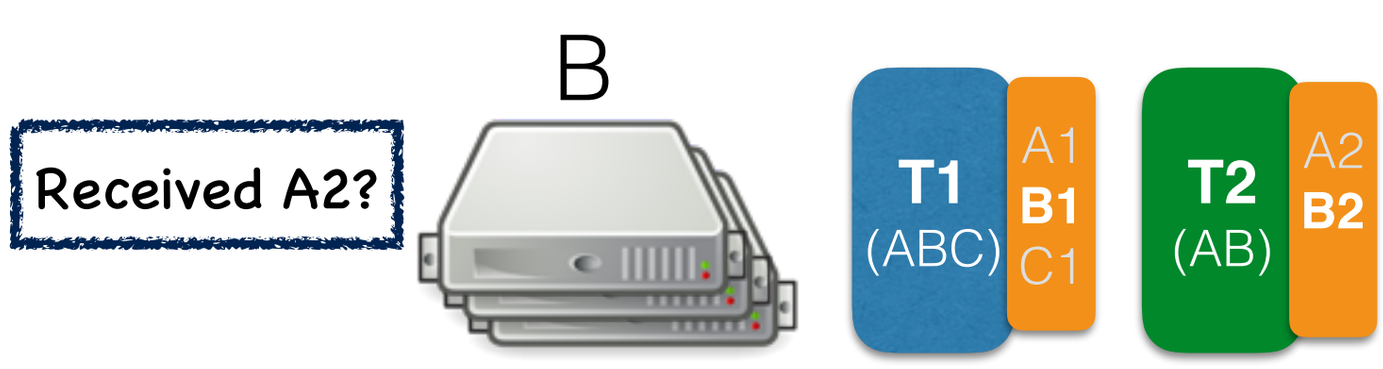


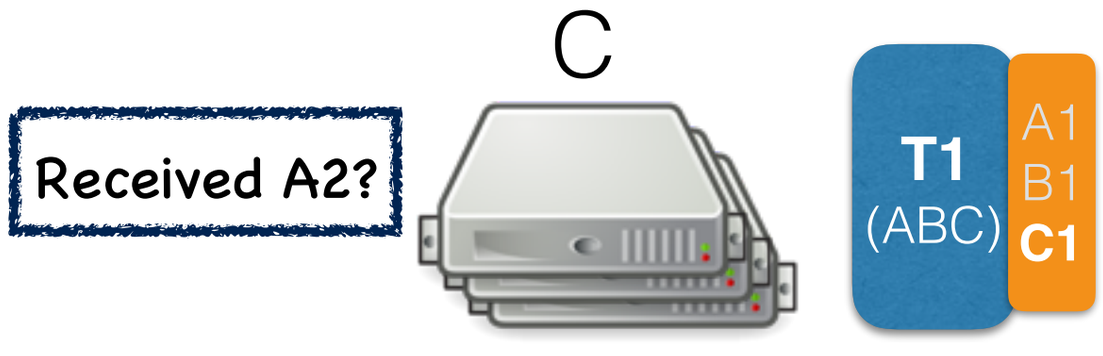


The Failure Coordinator

Failure Coordinator

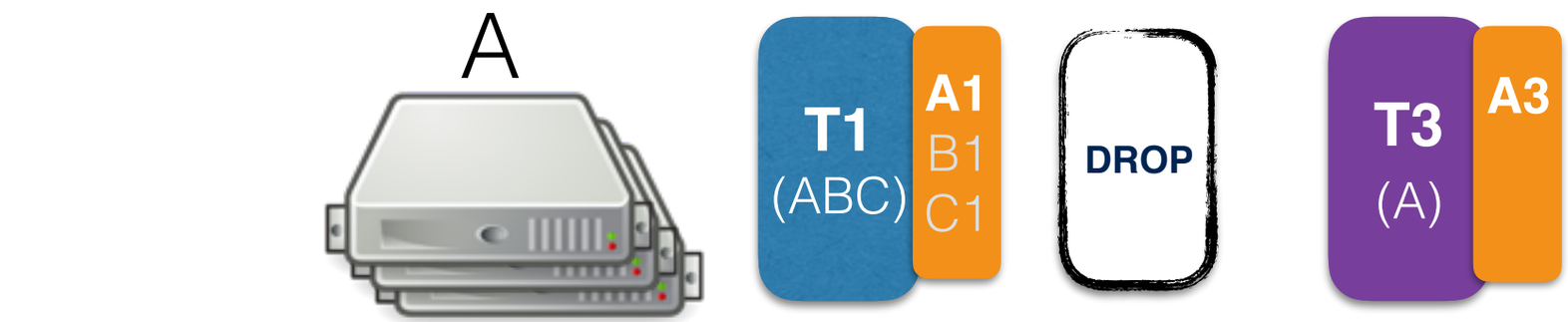


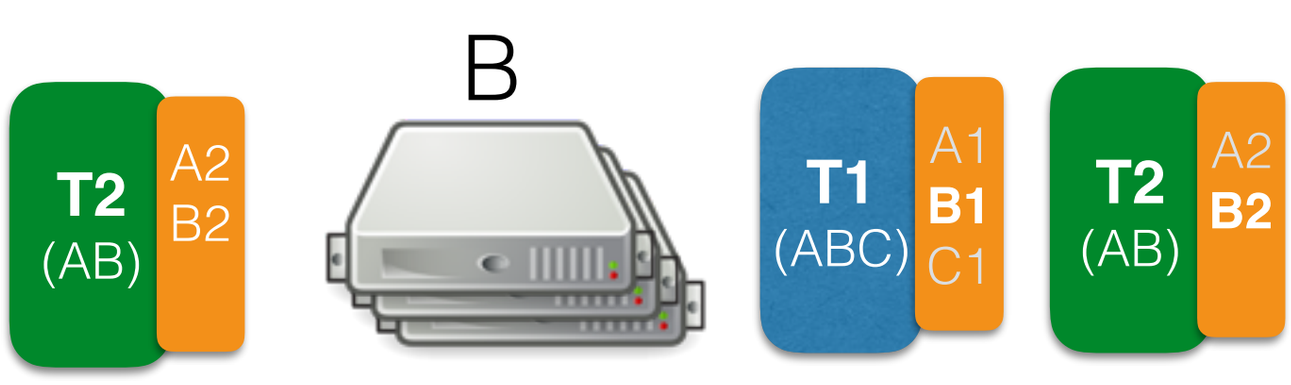


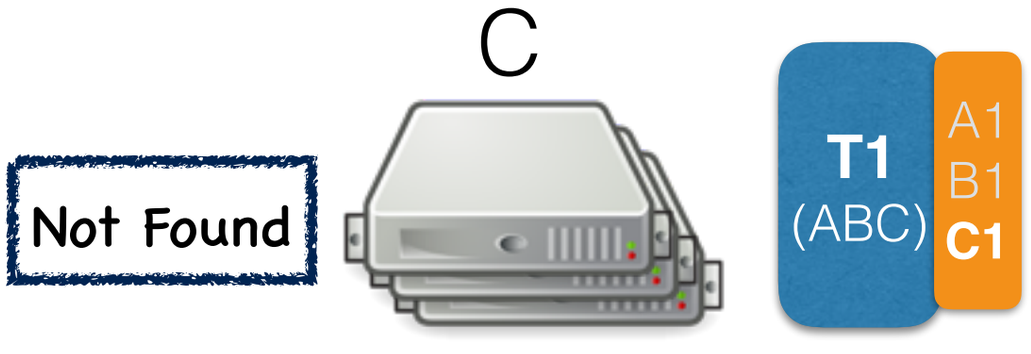


The Failure Coordinator

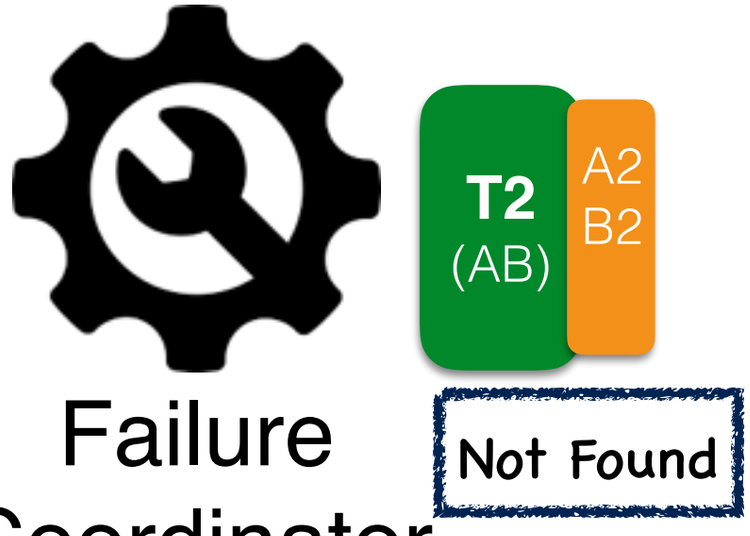
Failure Coordinator

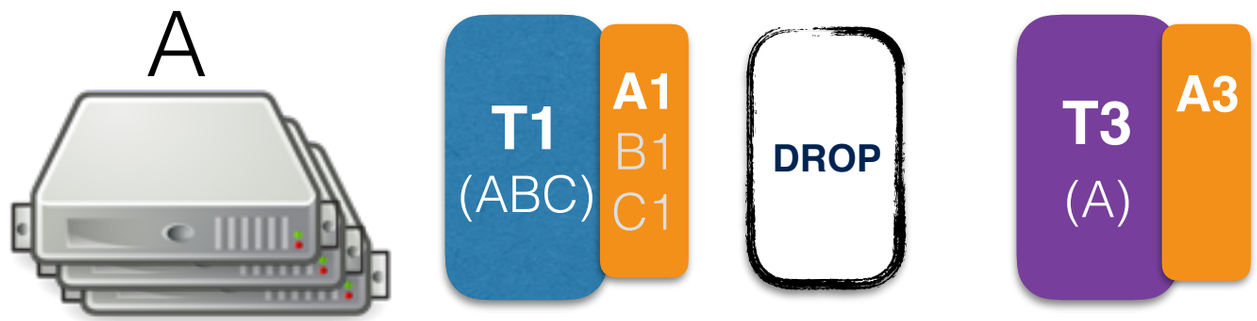


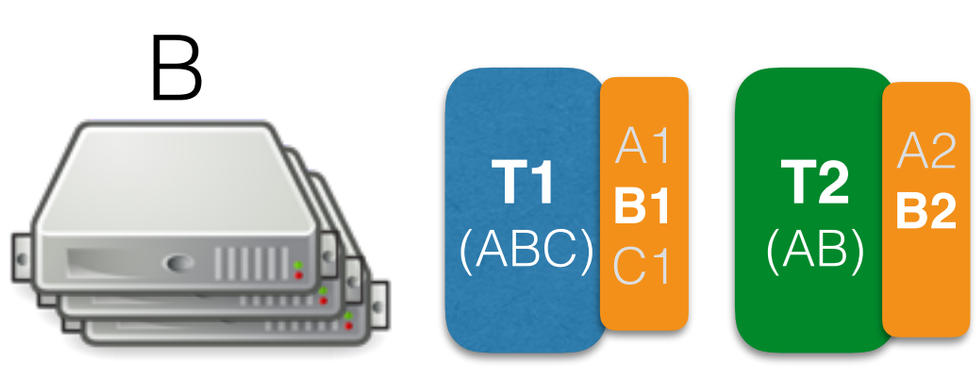


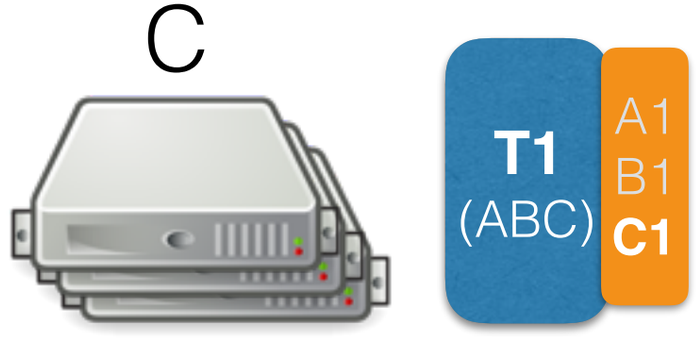


The Failure Coordinator

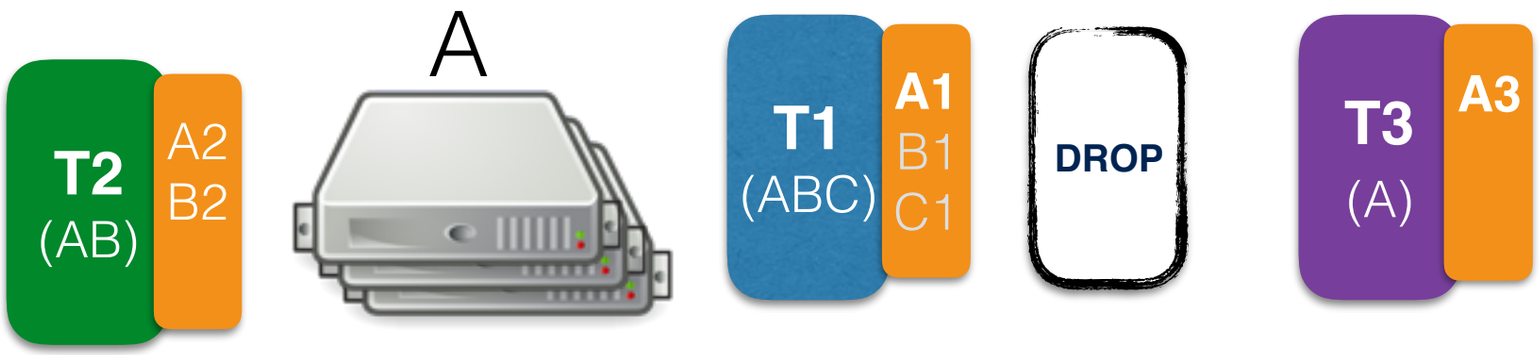




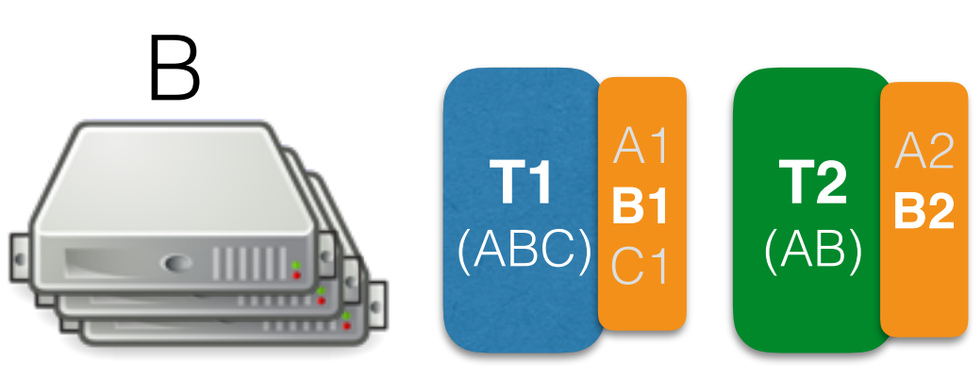


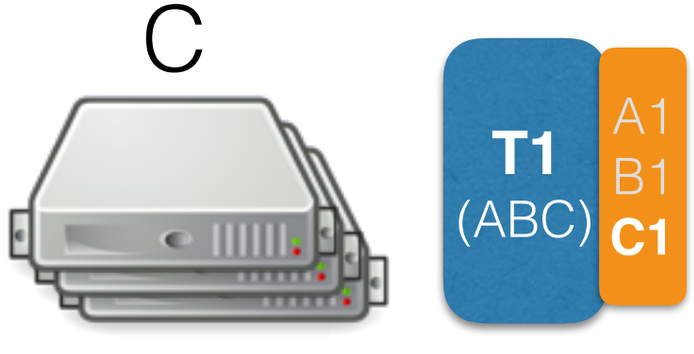


The Failure Coordinator



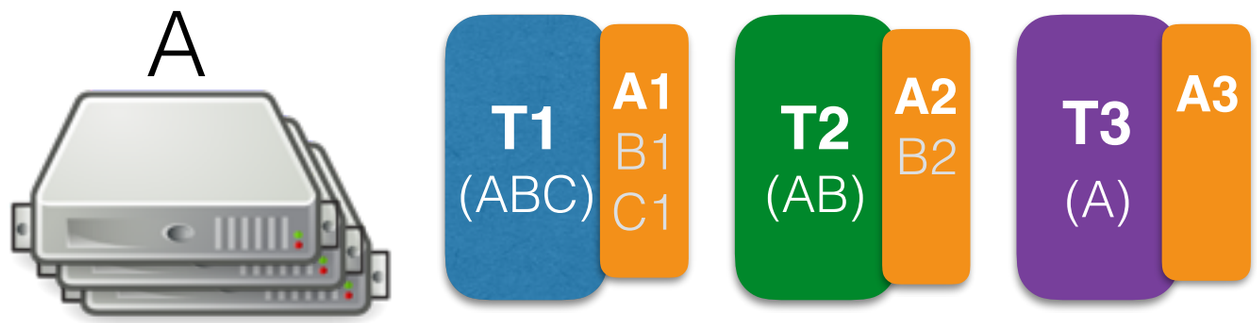
Failure Coordinator

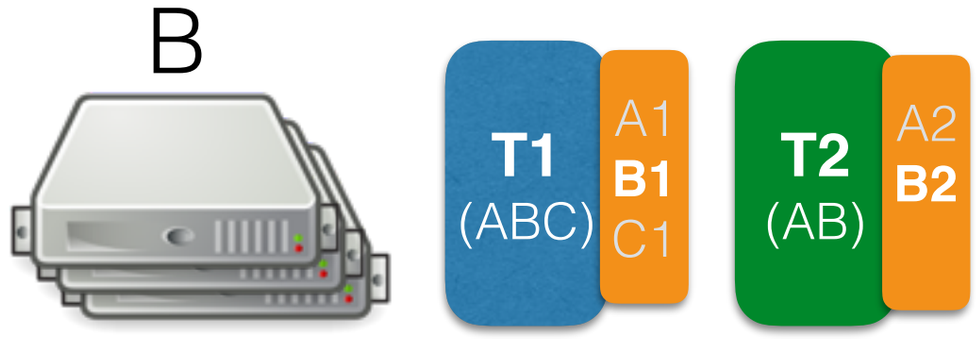


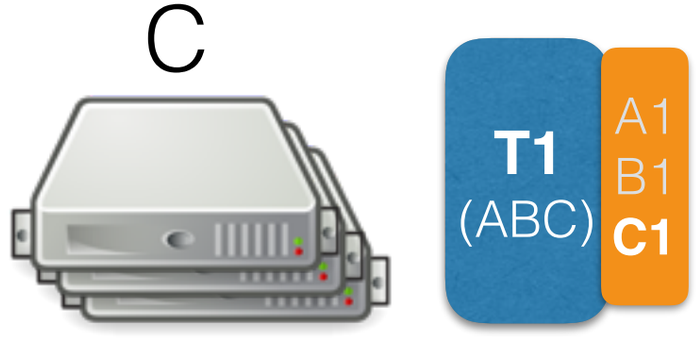


The Failure Coordinator

Failure Coordinator

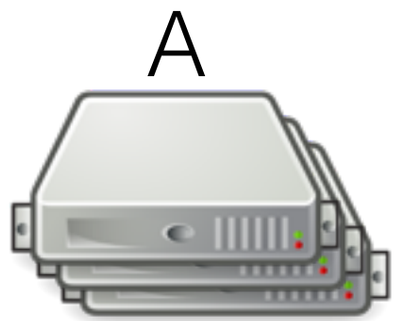


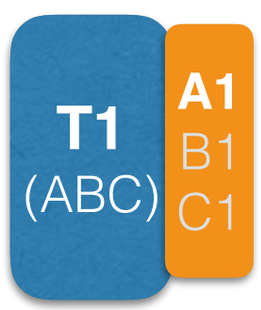


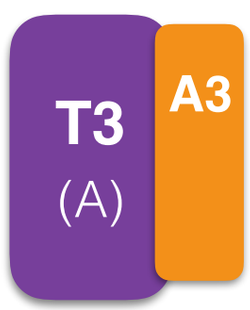


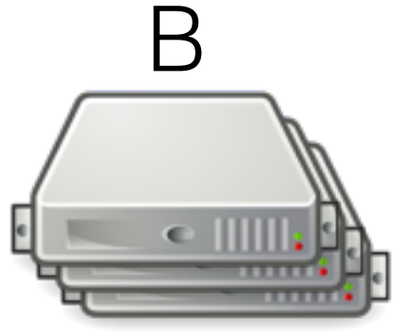
The Failure Coordinator

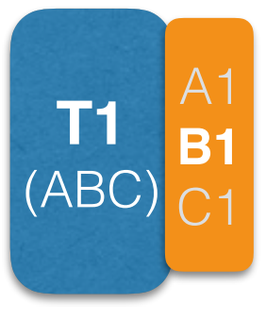
Failure Coordinator

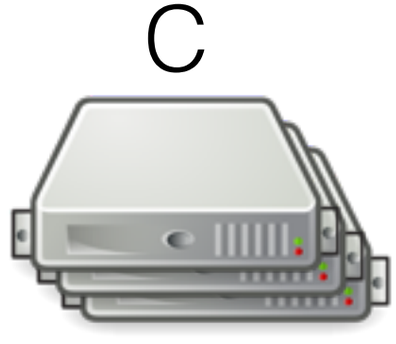


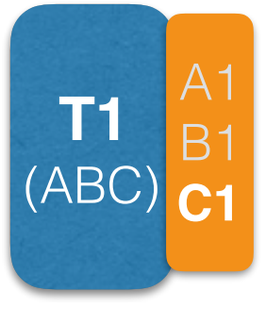






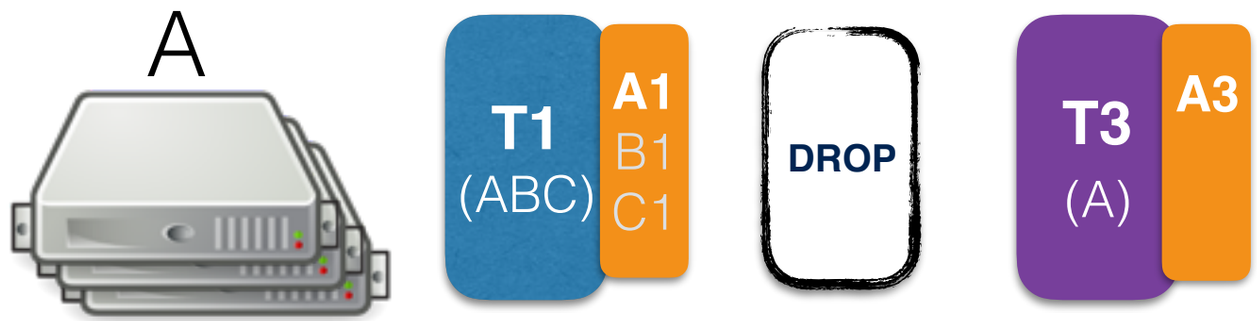


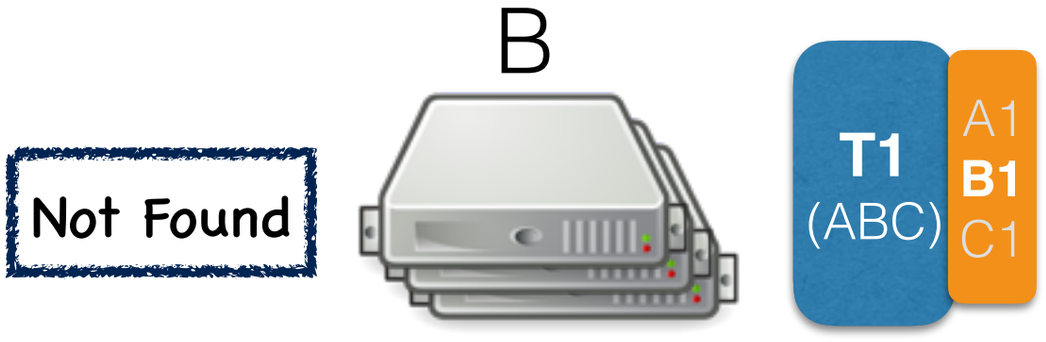


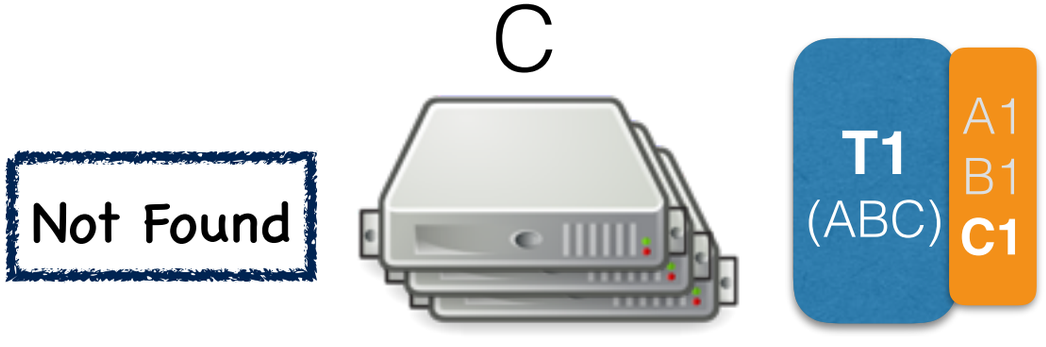


The Failure Coordinator

Failure Coordinator





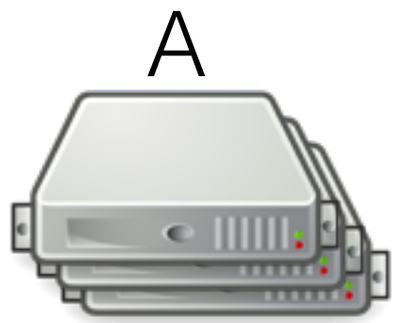


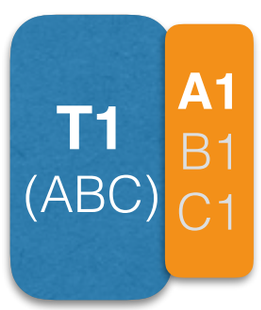
The Failure Coordinator

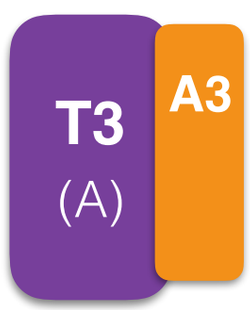
Failure Coordinator

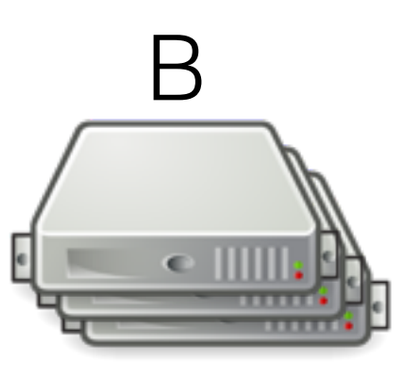
Not Found

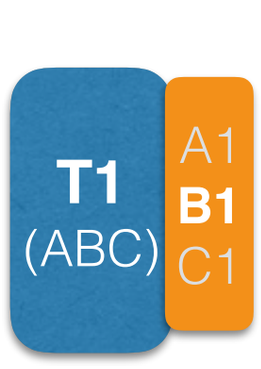
Not Found

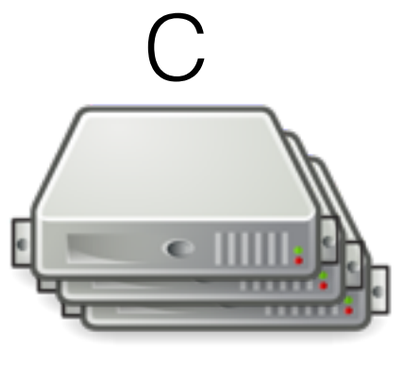


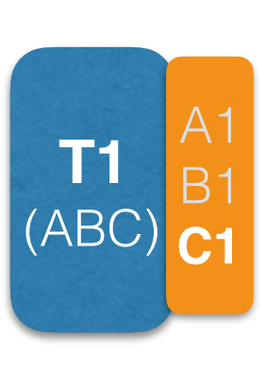






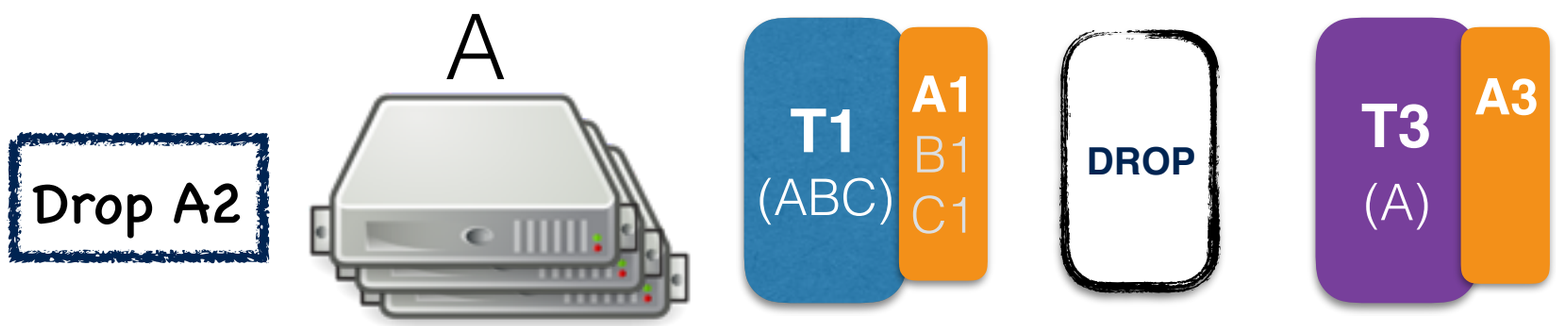


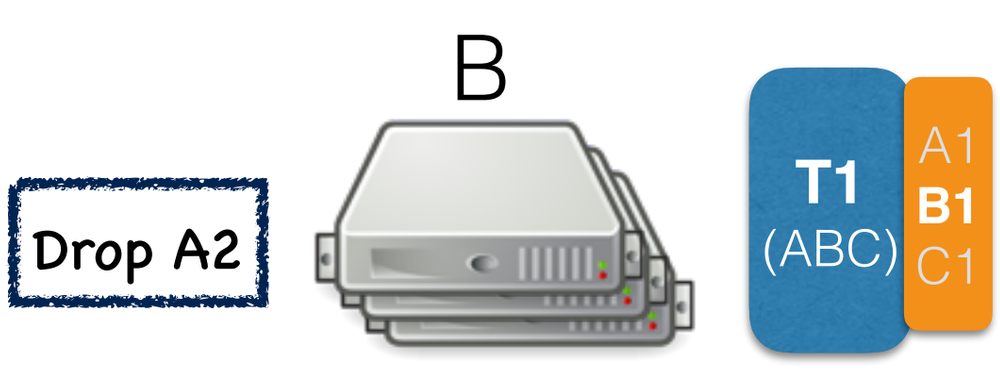


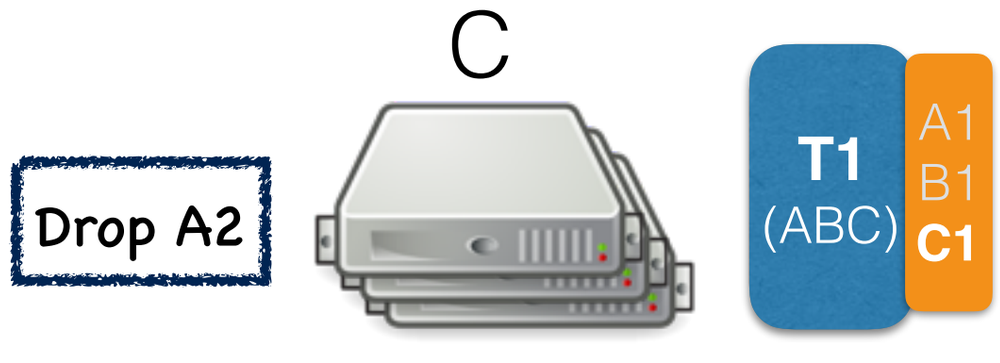


The Failure Coordinator

Failure Coordinator

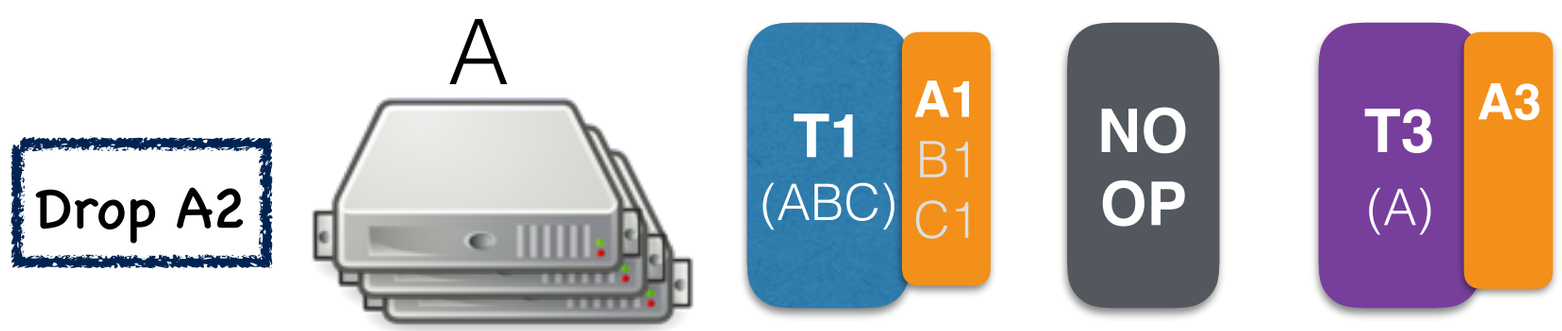


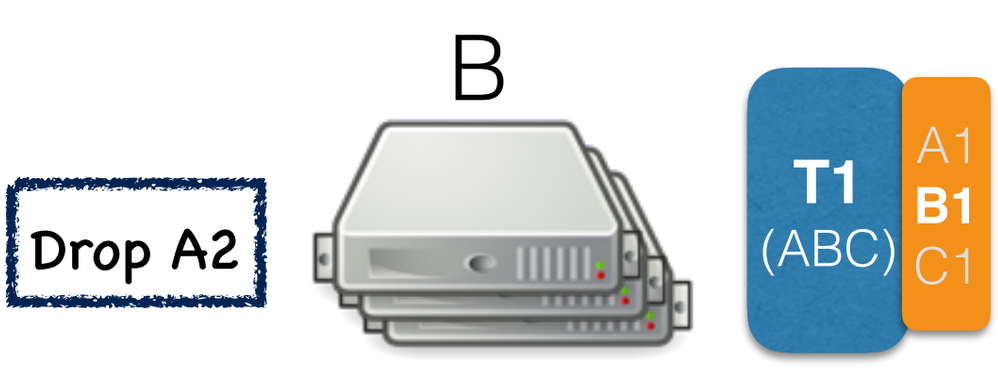


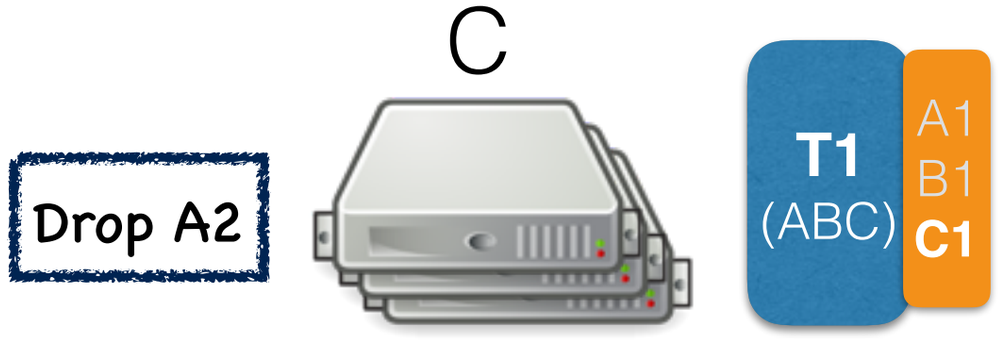


The Failure Coordinator

Failure Coordinator

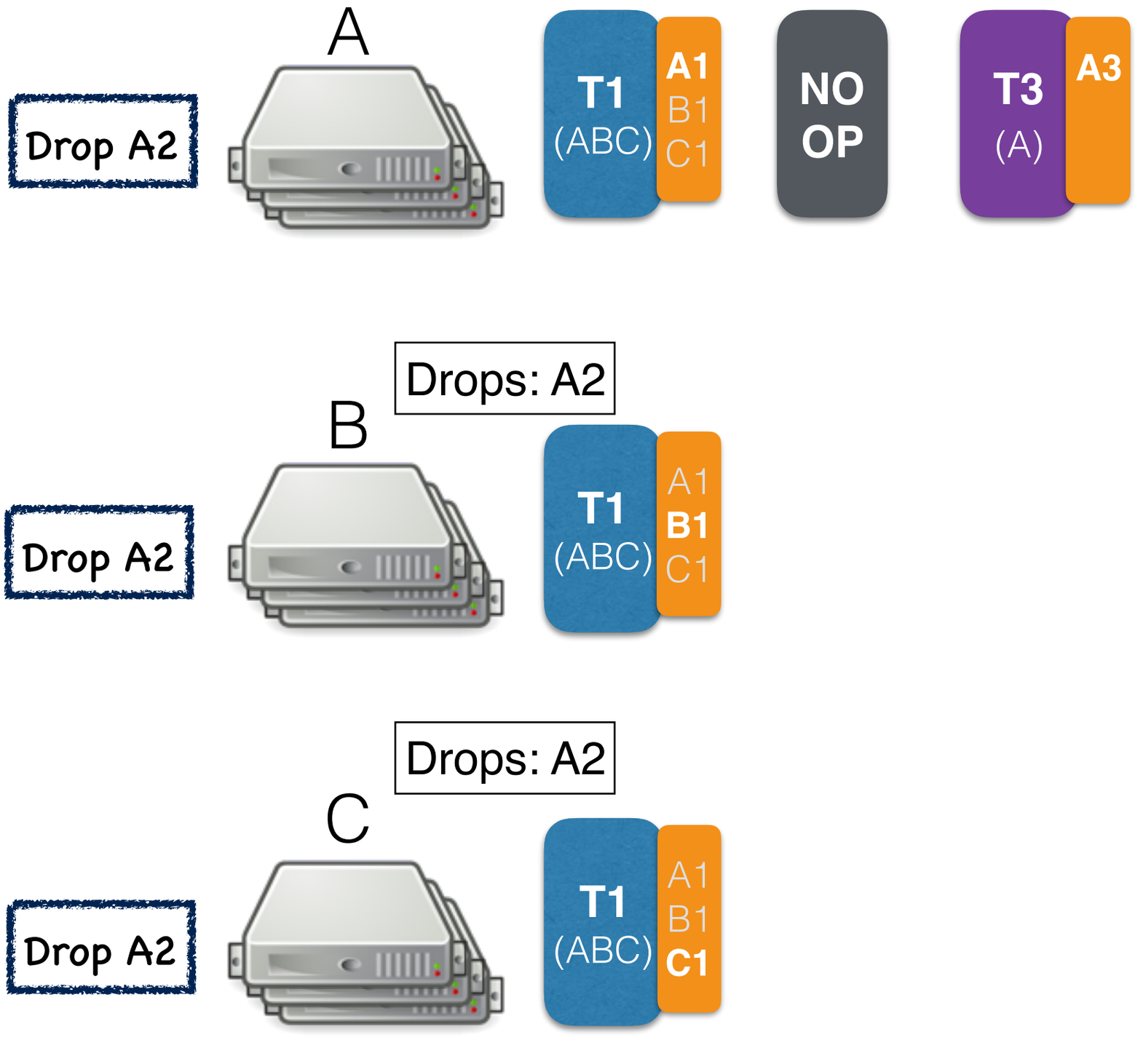






The Failure Coordinator

Failure Coordinator



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**

Can we process **non-independent transactions** efficiently?

Can we process **non-independent transactions** efficiently?

Yes, by dividing them into multiple independent transactions

(See the paper!)

Outline

1. Introduction
2. In-Network Concurrency Control
3. Transaction Model
4. Eris Protocol
5. Evaluation

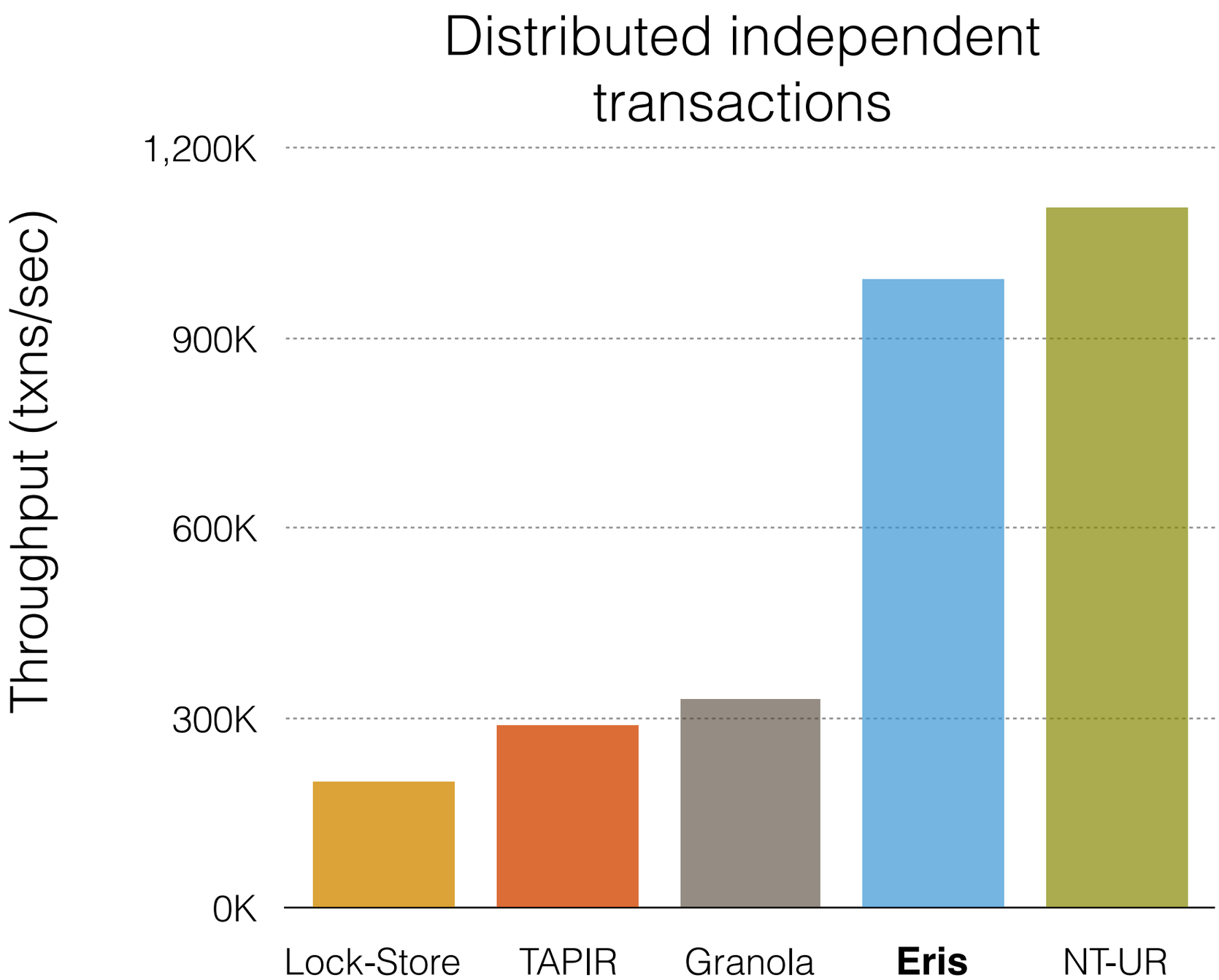
Evaluation Setup

- 3-level fat-tree topology testbed
- 15 shards, 3 replicas per shard
- 2.5 GHz Intel Xeon E5-2680 servers
- Middlebox sequencer implementation using Cavium Octeon CN6880
- YCSB+T and TPC-C workloads

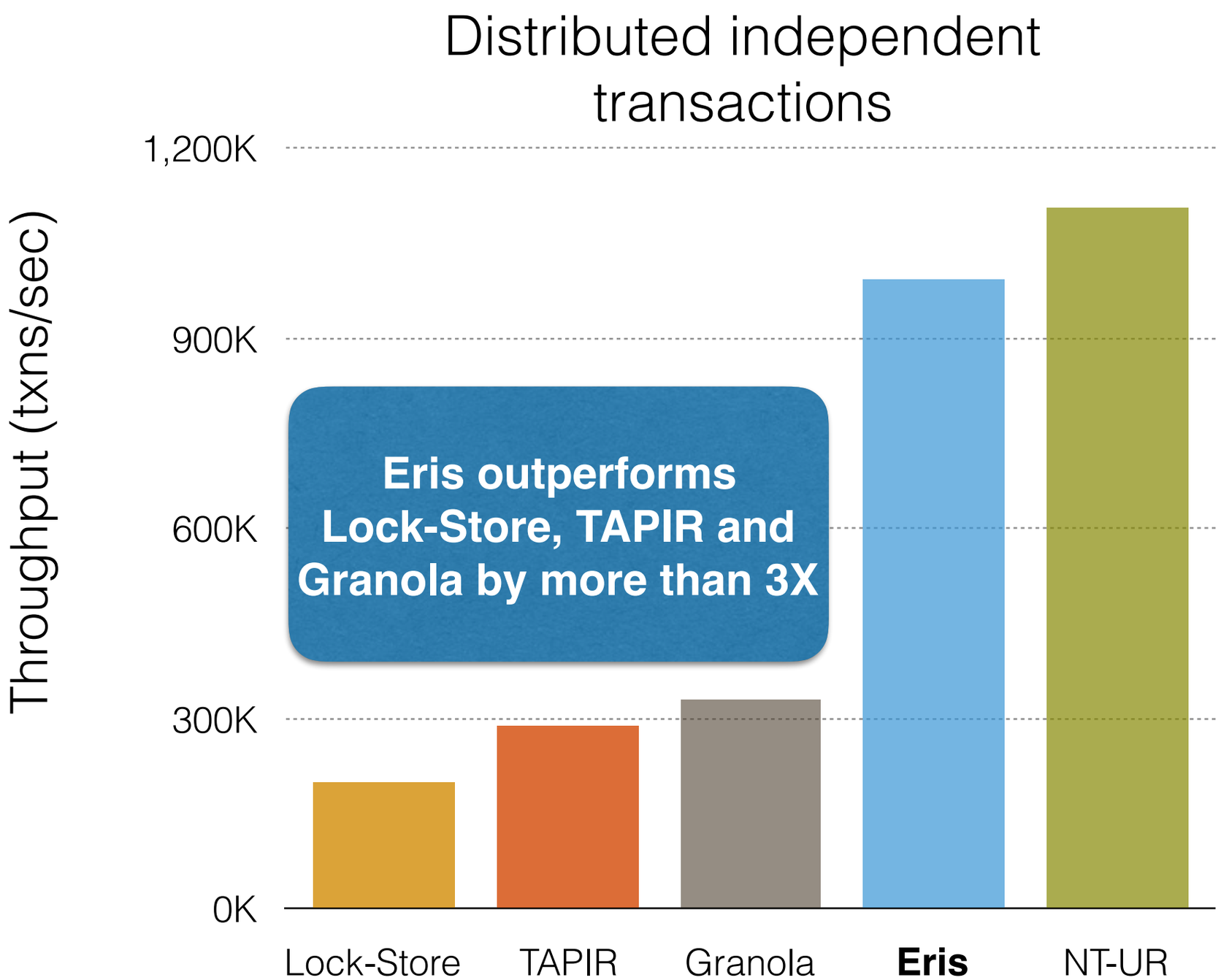
Comparison Systems

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

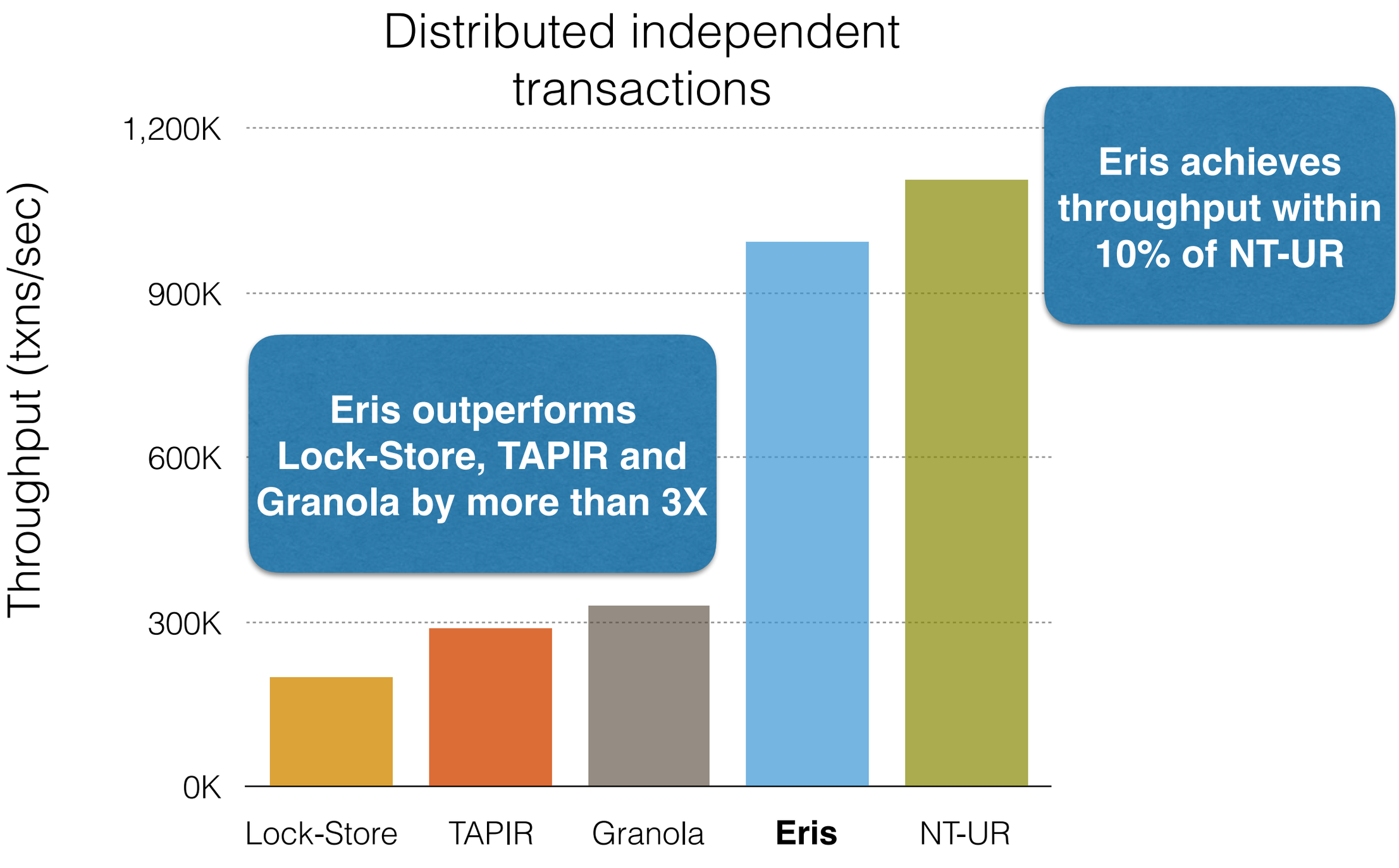
Eris performs well on **independent** transactions



Eris performs well on **independent** transactions



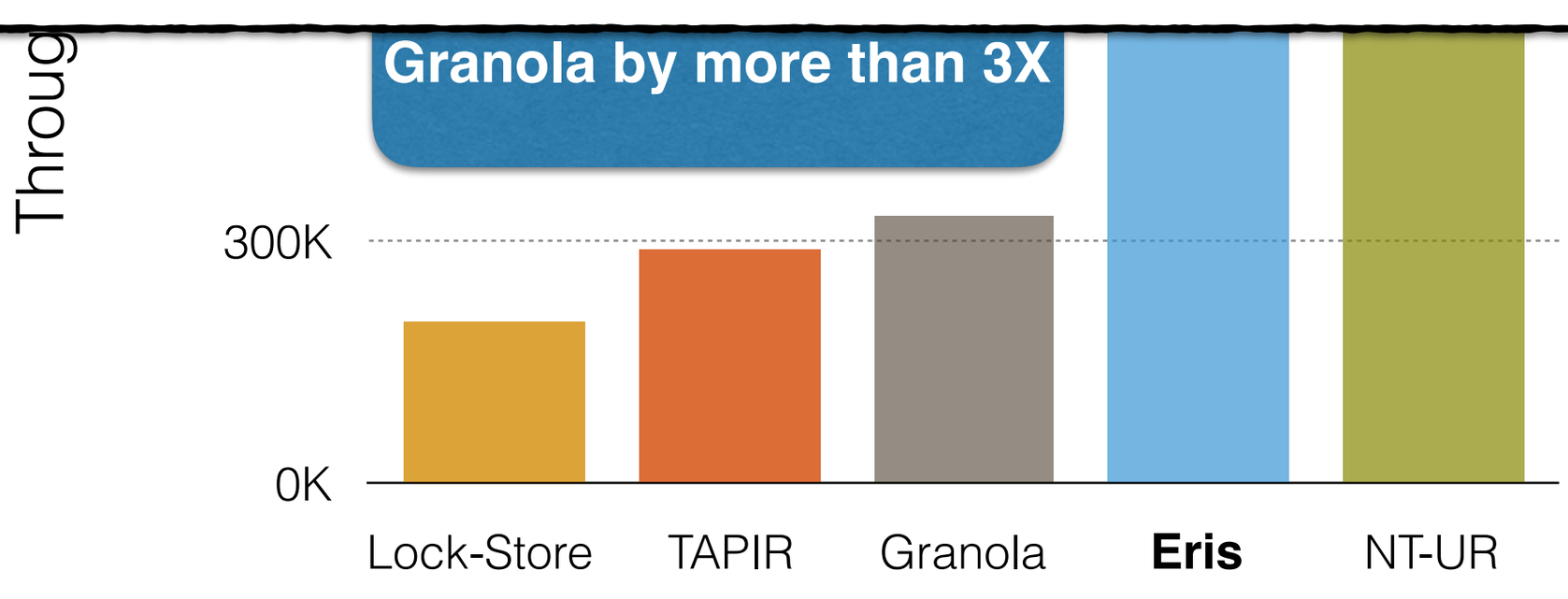
Eris performs well on **independent** transactions



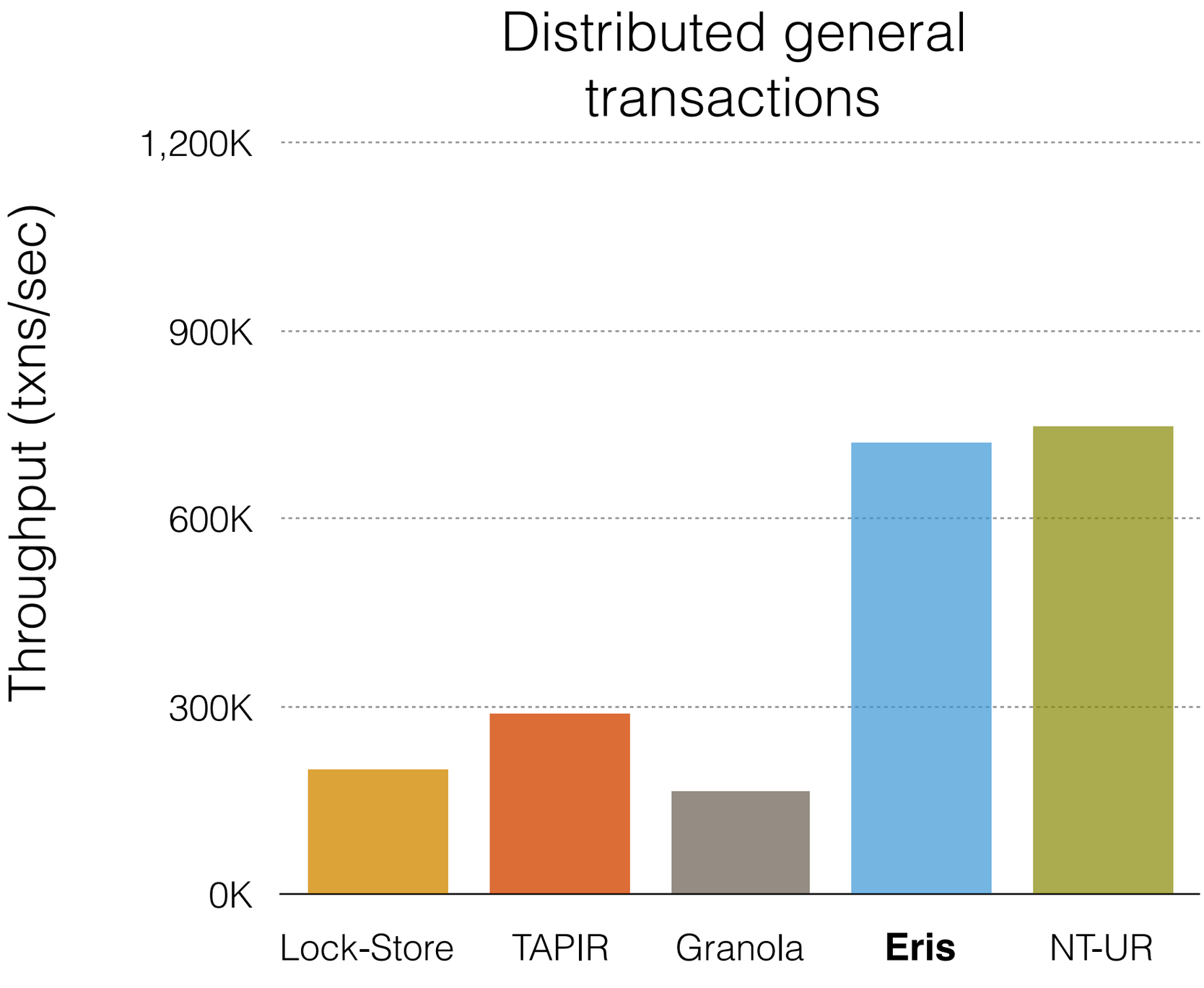
Eris performs well on **independent** transactions

Distributed independent transactions

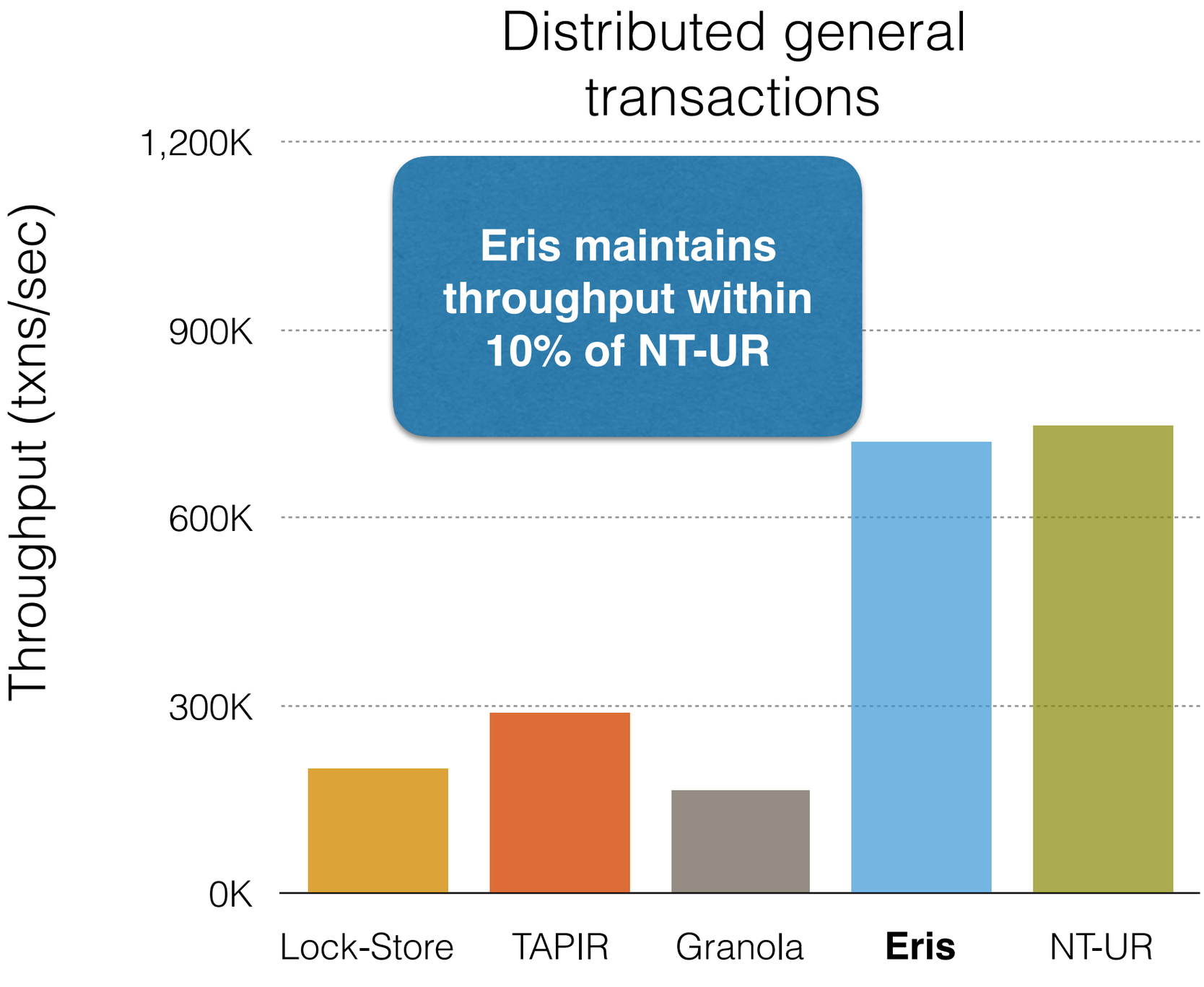
More than **70% reduction** in latency compared to Lock-Store, and **within 10%** latency of NT-UR



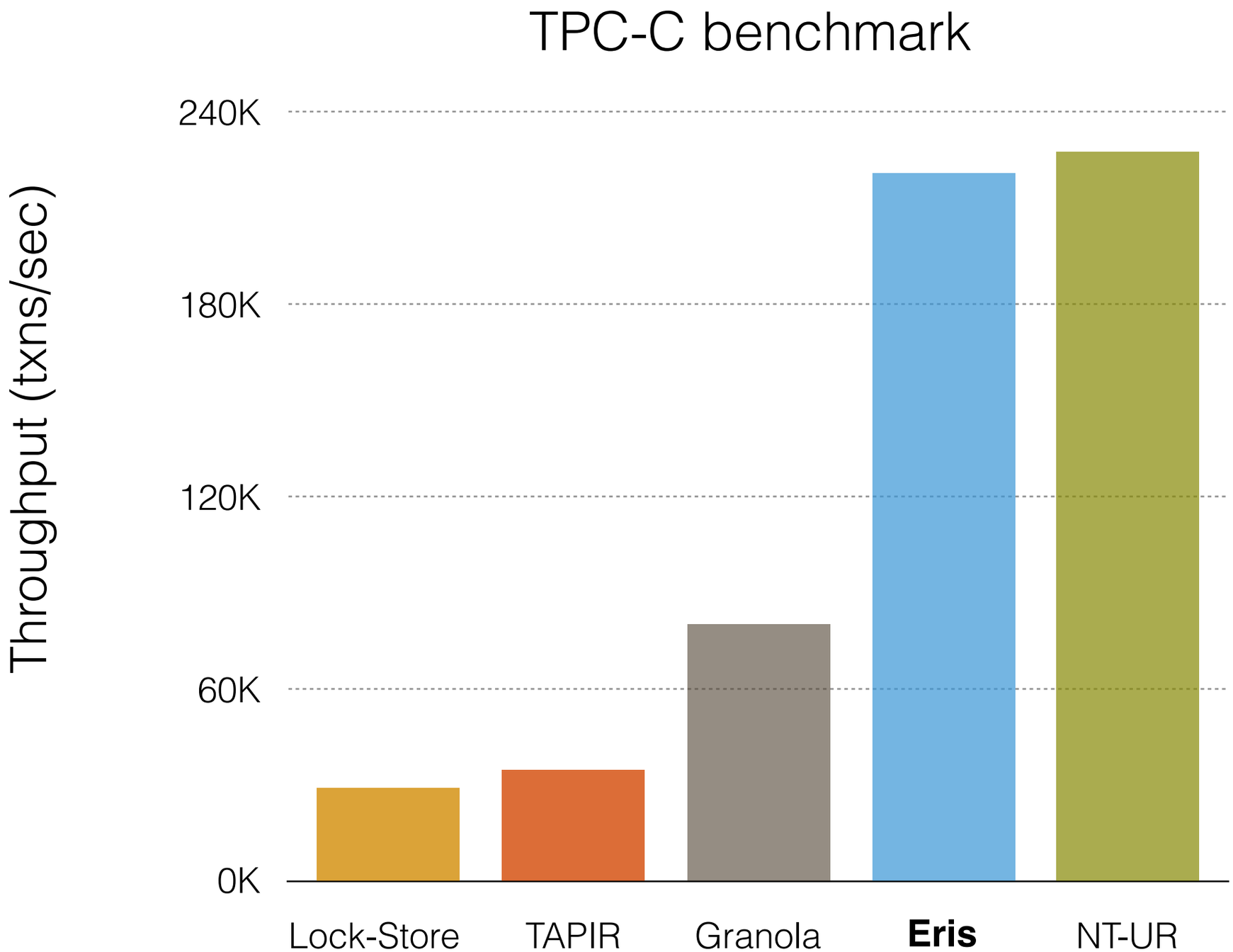
Eris also performs well on **general** transactions



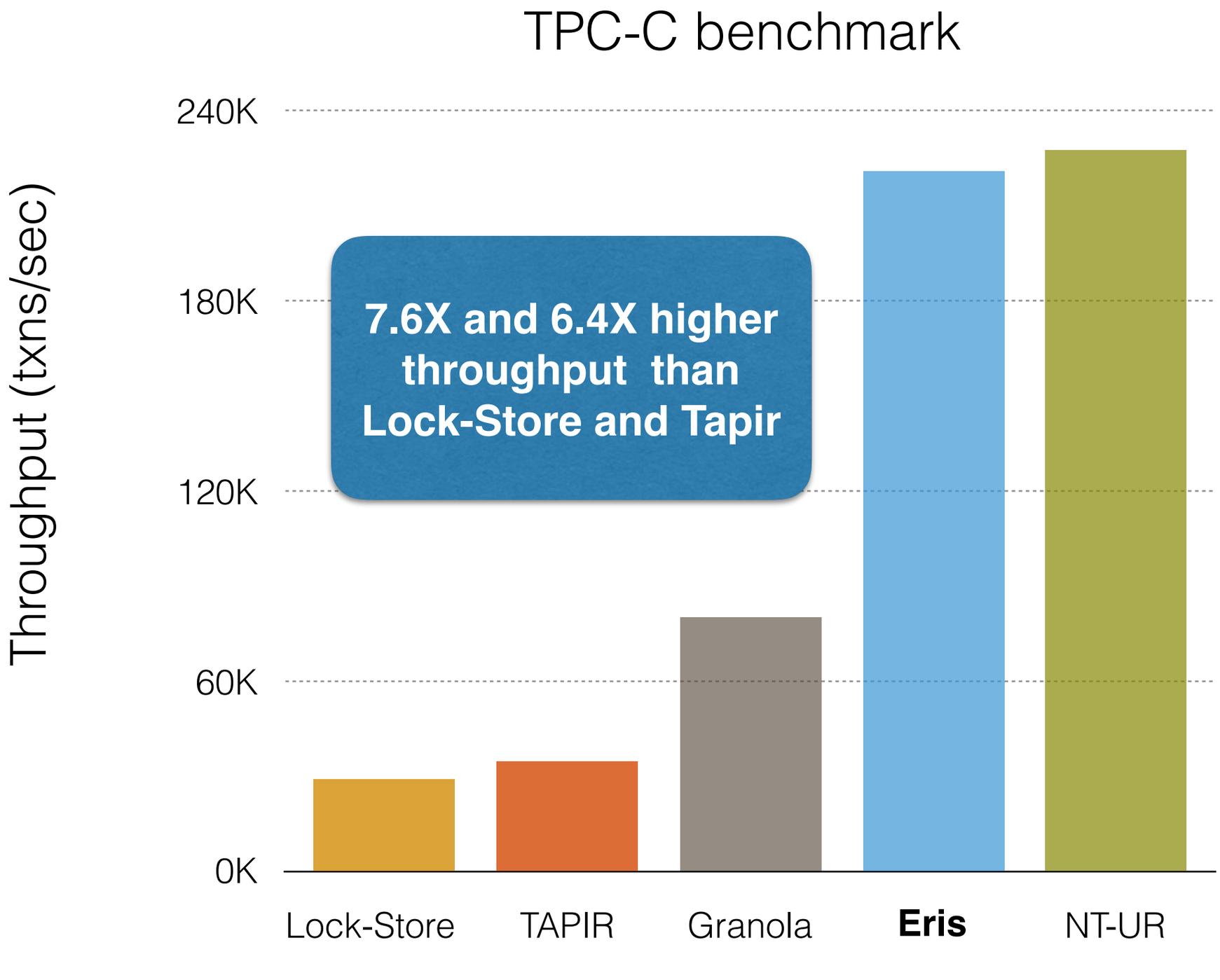
Eris also performs well on **general** transactions



Eris excels at complex transactional application.

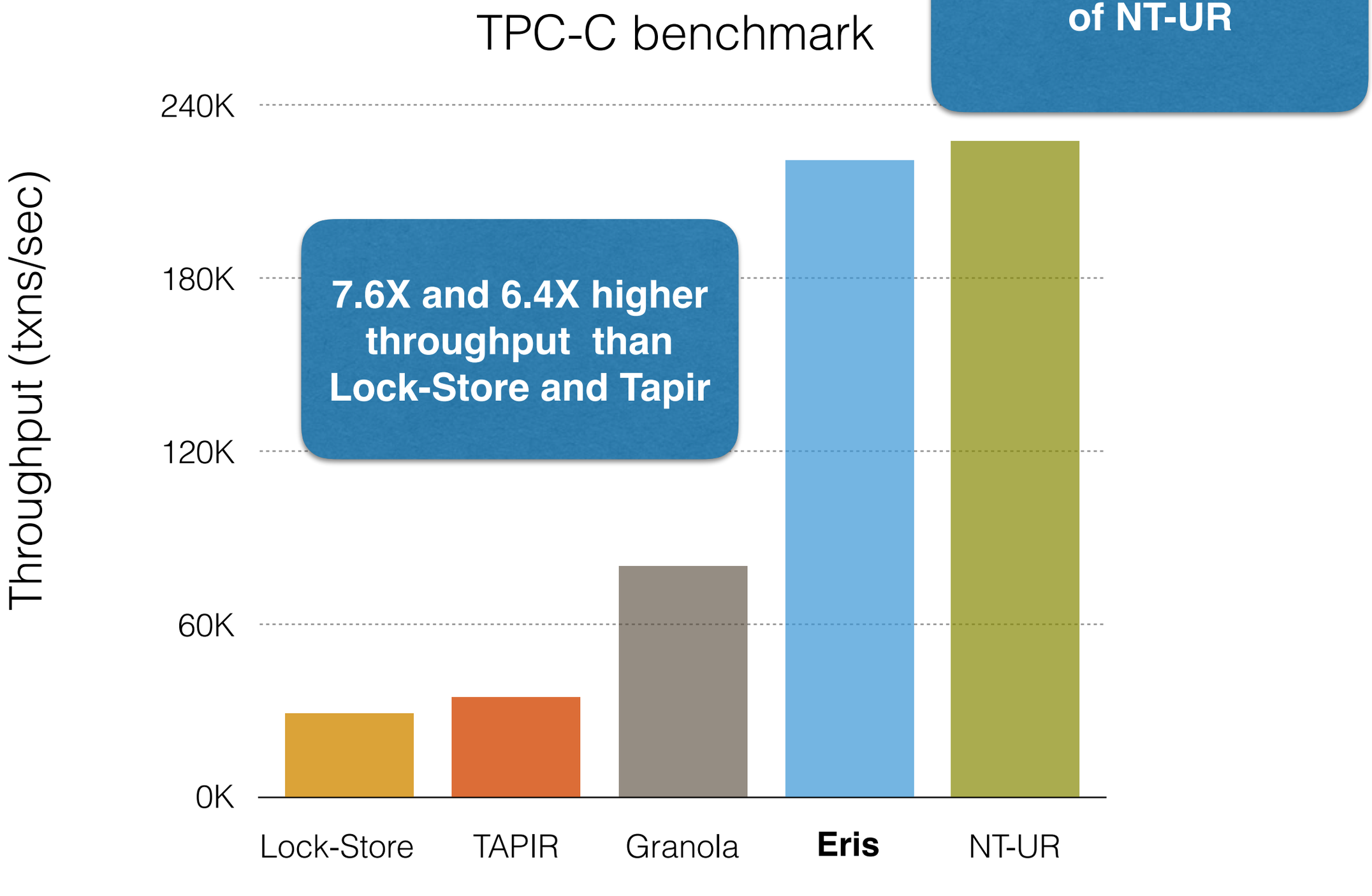


Eris excels at complex transactional application.

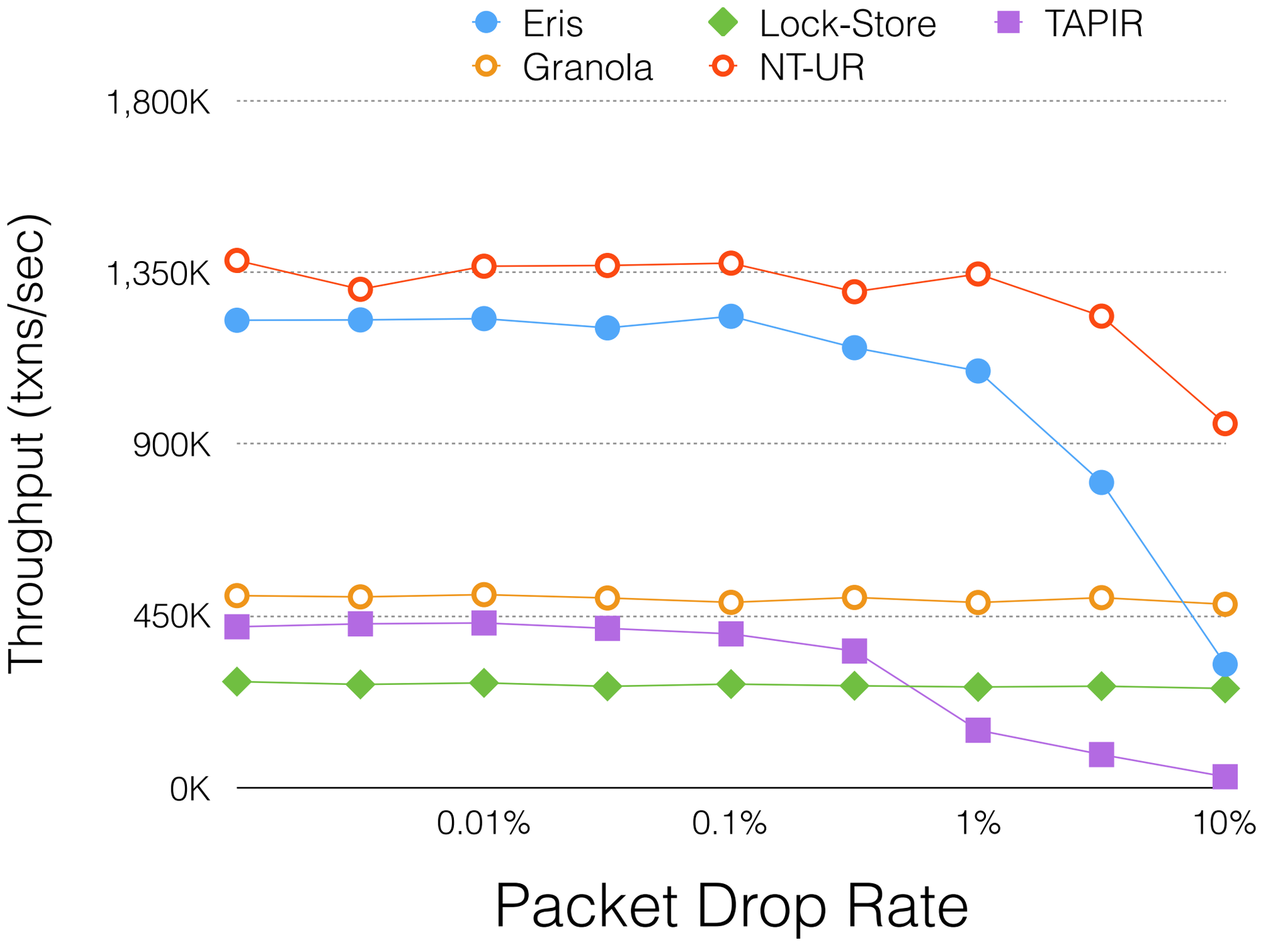


Eris excels at complex transactional applicat

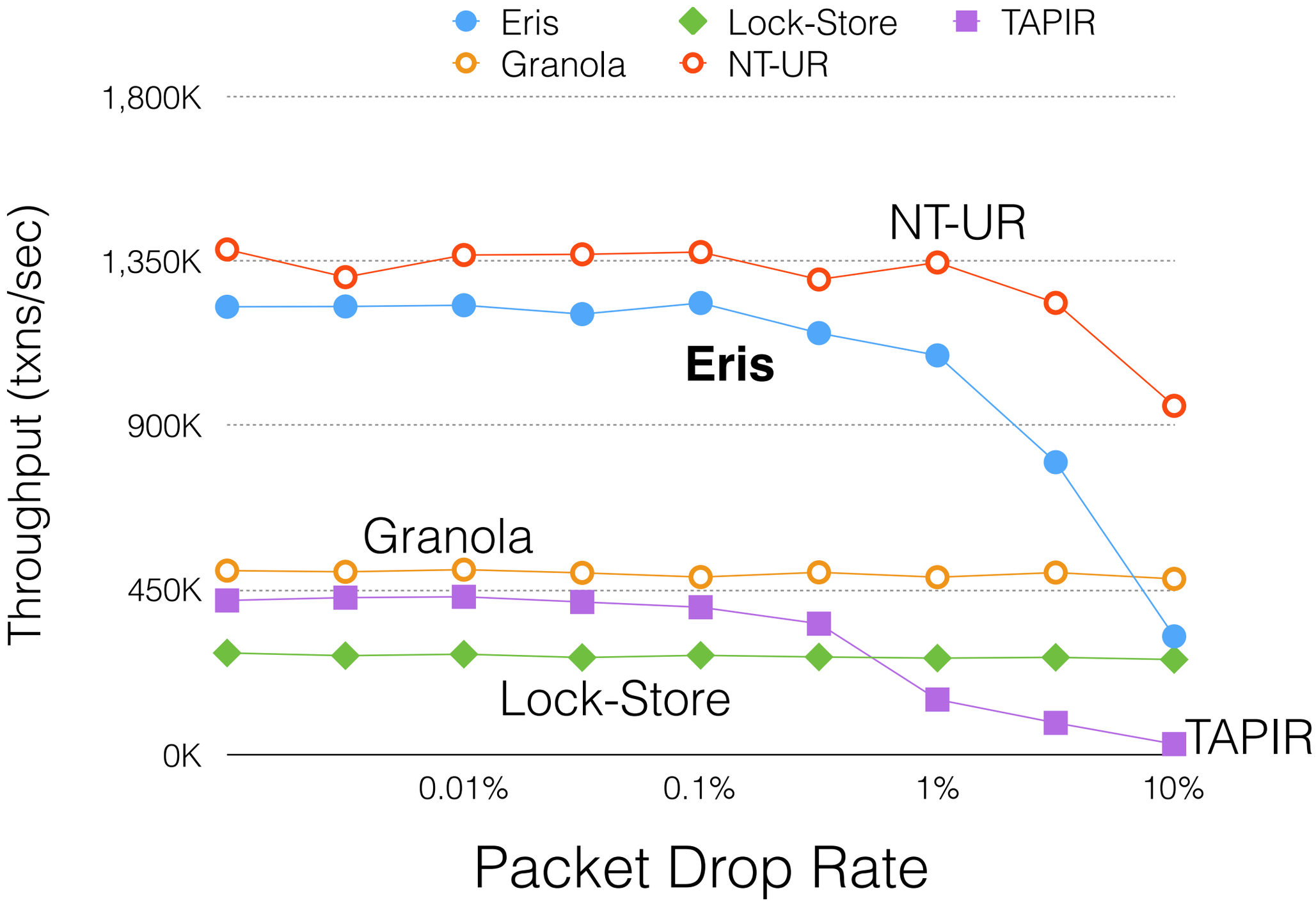
within 3% throughput of NT-UR



Eris is resilient to network anomalies



Eris is resilient to network anomalies



Related Work

Co-designing distributed systems with the network

- NOPaxos [OSDI ‘16], Speculative Paxos [NSDI ‘15], NetPaxos [SOSR ‘15]

Sequencers for transaction processing

- Hyder [CIDR ‘11], vCorfu [NSDI ‘17], Calvin [SIGMOD ‘12]

Independent and other restricted transaction models

- H-Store [VLDB ‘07], Granola [ATC ‘12], Calvin [SIGMOD ‘12]

Conclusion

- 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**