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.
STATE MACHINE REPLICAION

Operation A

Operation B

Operation C
STATE MACHINE REPLICATION
STATE MACHINE REPLICATION
PAXOS FOR STATE MACHINE REPLICACTION

Client

Leader Replica

Replica

Replica

request  |  prepare  |  prepareok  |  reply
Paxos for State Machine Replication

- Client
- Leader Replica
- Replica

Throughput Bottleneck

- Request
- Prepare
- Prepareok
- Reply
Paxos for State Machine Replication

Throughput Bottleneck

Latency Penalty
Network properties determine replication complexity

- Paxos protocol on every operation
- High performance cost
Asynchronous Network

- Paxos protocol on every operation
- High performance cost

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

Reliability
Ordering

Paxos
NETWORK PROPERTIES DETERMINE REPLICATION COMPLEXITY

Asynchronous Network

• Paxos protocol on every operation
• High performance cost

Reliability
Ordering

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

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

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

Asynchronous Network

Paxos

Weak

Network Guarantee

Strong

Reliability

Ordering
Can we build a network model that:

• provides **performance benefits**
• can be implemented more **efficiently**
Spec Paxos 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

Senders

Counter: 0

Receivers
Ordered Unreliable Multicast

Senders

Receivers

Counter: 2
Ordered Unreliable Multicast

Senders

Receivers

Counter: 4
Ordered Unreliable Multicast

Counter: 4

Senders

Receivers
Ordered Unreliable

Ordered Multicast:
no coordination required to determine order of messages
Ordered Unreliable

Ordered Multicast:
no coordination required to determine order of messages

Counter: 4

Drop Detection:
coordination only required when messages are dropped

Senders

Receivers
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

Client

Replica (leader)

Replica

Replica

request

OUM
NORMAL OPERATION

Client

Replica (leader)

Replica

Replica

request

reply

OUM

Execute
NORMAL OPERATION

Client

Replica (leader)

Replica

Replica

request

reply

waits for replies from majority including leader’s

OUM

Execute
NORMAL OPERATION

Client

Replica (leader)

Replica

Replica

request

OUM

reply

Execute

waits for replies from majority including leader's

no coordination
NORMAL OPERATION

1 Round Trip Time

request

reply

Client

Replica (leader)

Replica

Replica

OUM

no coordination

waits for replies from majority including leader’s
**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

<table>
<thead>
<tr>
<th>Latency (us)</th>
<th>Throughput (ops/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>better ↓</td>
<td>better →</td>
</tr>
</tbody>
</table>
NOPaxos achieves better throughput and latency

Latency (us)

better ↓

Throughput (ops/sec)

better →
NOPaxos achieves better throughput and latency

Throughput (ops/sec)

Latency (us)

1000

750

500

250

NOPaxos

Paxos

Fast Paxos

4.7X throughput and more than 40% reduction in latency
NOPaxos achieves better throughput and latency

Throughput (ops/sec)

Latency (us)

Fast Paxos

Paxos

NOPaxos

Paxos + Batching

4.7X throughput and more than 40% reduction in latency

better ↓

better →
NOPaxos achieves better throughput and latency.

25% higher throughput and 6X lower latency.
NOPaxos is resilient to network anomalies

Throughput (ops/sec) vs. Packet Drop Rate

- **NOPaxos**
- **Paxos**
- **SpecPaxos**
NOPaxos is resilient to network anomalies
**NOPaxos is resilient to network anomalies**

Throughput (ops/sec)

- **NOPaxos**: Above 260,000
- **Speculative Paxos**: Drops to 24% of maximum throughput
- **Paxos**: Below 65,000

Packet Drop Rate

- **0.001%**: Above 260,000
- **0.01%**: Above 195,000
- **0.1%**: Above 130,000
- **1%**: Above 65,000
NOPaxos attains throughput within 2% of an unreplicated system
NOPaxos attains throughput within 2% of an unreuplicated 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.
NOPaxos attains throughput within 2% 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
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

Replica

Replica

Replica
TRADITIONAL LAYERED APPROACH

Atomic Commitment (2PC)

Concurrency Control (2PL)

Concurrency Control (2PL)

Replication (Paxos)

Replication (Paxos)

Reliability (within shard)

Ordering (within shard)
**Traditional Layered Approach**

- Atomic Commitment (2PC)
- Concurrency Control (2PL)
- Replication (Paxos)
- Replication (Paxos)
- Isolation
- Reliability (within shard)
- Ordering (within shard)
TRADITIONAL LAYERED APPROACH

Atomic Commitment (2PC)
Concurrency Control (2PL)
Replication (Paxos)

Reliability (across shards)
Isolation
Reliability (within shard)
Ordering (across shard)
Ordering (within shard)
A NEW WAY TO DIVIDE RESPONSIBILITIES

- Isolation
- Reliability (across shards)
- Reliability (within shard)
- Ordering (across shard)
- Ordering (within shard)

Eris
- General Transaction Protocol
- Independent Transaction Protocol
- Multi-sequencing
A NEW WAY TO DIVIDE RESPONSIBILITIES

Eris
- General Transaction Protocol
- Independent Transaction Protocol
- Multi-sequencing

Application
- Ordering (across shard)
- Ordering (within shard)

Reliability
- (across shards)
- (within shard)

Isolation

Network
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
Receivers

A

B

C

T1 (ABC)

T2 (AB)

T1 (ABC)

T2 (AB)

T1 (ABC)
Receivers

A

B

C

T1 (ABC)  T2 (AB)

T1 (ABC)  T2 (AB)

T1 (ABC)
A

B

C

Receivers

DROP

T2
(AB)

T1
(ABC)

T2
(AB)

T1
(ABC)
**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

Observers
Sequencer

Counter: A0 B0 C0

T1
(ABC)

Receivers

A

B

C
Sequencer

Counter:
A1  B1  C1

(A BC)

Receivers
Sequencer

Counter: A1 B1 C1

Receivers
 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
Sequencer

Counter:
A1  B1  C1

Receivers

A

T1
(ABC)

A1
B1
C1

B

T1
(ABC)

A1
B1
C1

C

T1
(ABC)

A1
B1
C1
Sequencer

Counter: A2 B2 C1

A

B

C

Receivers
Sequencer

Counter:
A2  B2  C1

Receivers

A

B

C

T1 (ABC)  A1  B1  C1
T2 (AB)  A2  B2

T1 (ABC)  A1  B1  C1
T2 (AB)  A2  B2

T1 (ABC)  A1  B1  C1
Receivers

Sequencer

Counter: A2 B2 C1
Sequencer

Counter:
A3  B2  C1

Receivers

A

B

C

T3
(A)

T1
(ABC)

A1
B1
C1

T1
(ABC)

A1
B1
C1

T2
(AB)

A2
B2

T1
(ABC)

A1
B1
C1
Sequencer

Counter: A3 B2 C1

A

B

C

Receivers

T3 (A)

T1 (ABC) A1 B1 C1

T1 (ABC)

A2 B2

T1 (ABC) A1 B1 C1
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

Receivers

A

B

C
Sequencer

Counter: A3  B2  C1

Receivers

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

DROP

T3 (A) A3
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>600</td>
</tr>
<tr>
<td>Bob</td>
<td>350</td>
</tr>
<tr>
<td>Charlie</td>
<td>400</td>
</tr>
</tbody>
</table>

START TRANSACTION

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

COMMIT

START TRANSACTION

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

COMMIT
INDEPENDENT TRANSACTION

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>600</td>
</tr>
<tr>
<td>Bob</td>
<td>350</td>
</tr>
<tr>
<td>Charlie</td>
<td>400</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>600</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>450</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charlie</td>
<td>500</td>
</tr>
</tbody>
</table>
**INDEPENDENT TRANSACTION**

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>600</td>
</tr>
<tr>
<td>Bob</td>
<td>450</td>
</tr>
<tr>
<td>Charlie</td>
<td>500</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

Name | Salary
--- | ---
Alice | 600
Bob | 450
Charlie | 500
**INDEPENDENT TRANSACTION**

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>600</td>
</tr>
<tr>
<td>Bob</td>
<td>450</td>
</tr>
<tr>
<td>Charlie</td>
<td>500</td>
</tr>
</tbody>
</table>

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

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

```
START TRANSACTION
UPDATE tb t1
SET t1.Salary = t1.Salary + 100
WHERE t1.Salary < 500
COMMIT
```
Many applications consist *entirely* of independent transactions

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>600</td>
</tr>
<tr>
<td>Bob</td>
<td>450</td>
</tr>
<tr>
<td>Charlie</td>
<td>500</td>
</tr>
</tbody>
</table>
**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
Replica

Shard 2
Learner
Replica
Replica
Replica

Shard 3
Learner
Replica
Replica
Replica
NORMAL CASE

Client

Sequencer

Shard 1
  Learner
  Replica
  Replica
  Replica

Shard 2
  Learner
  Replica
  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
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 → Shard 1 Replica → Shard 2 Learner → Shard 2 Replica → Shard 3 Learner → Shard 3 Replica

1 round trip
NORMAL CASE

1 round trip

no coordination
HOW TO HANDLE DROPPED MESSAGES?
How to handle dropped messages?

A

T1 (ABC) A1 B1
T3 (A) A3

B

T1 (ABC) A1 B1

C

T1 (ABC) A1 B1
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?

A

T1 (ABC)

A1

B1

DROP

T3 (A)

A3

B

T1 (ABC)

A1

B1

T2 (AB)

A2

B2

C

T1 (ABC)

A1

B1
THE FAILURE COORDINATOR

Failure Coordinator

Received A2?

A

T1 (ABC)

T1 (ABC)

A2

B1

B2

C

T1 (ABC)

T1 (ABC)

A2

B1

A3

T3 (A)

DROP
THE FAILURE COORDINATOR

Failure Coordinator

Received A2?

Received A2?
THE FAILURE COORDINATOR

Failure Coordinator

A

T1
(ABC)

A
1
B1

DROP

T3
(A)

B

T2
(AB)

A2

B2

T1
(ABC)

A
1
B1

T2
(AB)

A2

B2

C

T1
(ABC)

A
1
B1

Not Found
THE FAILURE COORDINATOR

Failure Coordinator

Not Found
THE FAILURE COORDINATOR

Failure Coordinator

Diagram showing the failure coordinator with different servers and connections.
THE FAILURE COORDINATOR

Failure Coordinator
**The Failure Coordinator**

Failure Coordinator

Received A2?

Received A2?
**THE FAILURE COORDINATOR**

Failure Coordinator

Diagram showing the state of nodes A, B, and C, with 'Not Found' labels and T1 (ABC) and T3 (A) tasks.
THE FAILURE COORDINATOR

Failure Coordinator

Not Found

Not Found
**The Failure Coordinator**

![Diagram of Failure Coordinator](image)

- **A**
  - T1 (ABC)
  - Drop A2

- **B**
  - T1 (ABC)
  - Drop A2

- **C**
  - T1 (ABC)
  - Drop A2

- **T3 (A)**
  - A3

*Failure Coordinator*
The Failure Coordinator

Failure Coordinator
THE FAILURE COORDINATOR

Failure Coordinator

A
- Drop A2
- T1 (ABC) A 1 1
- NO OP
- T3 (A) A 1

B
- Drop A2
- T1 (ABC) A 1 1

C
- Drop A2
- T1 (ABC) A 1 1

Drops: A2
**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

Distributed independent transactions

Throughput (txns/sec)

- Lock-Store
- TAPIR
- Granola
- Eris
- NT-UR

- 1,200K
- 900K
- 600K
- 300K
- 0K
Eris performs well on independent transactions

Throughput (txns/sec)

Distributed independent transactions

Eris outperforms Lock-Store, TAPIR and Granola by more than 3X
**Eris Performs Well on Independent Transactions**

Eris achieves throughput within 10% of NT-UR.

Eris outperforms Lock-Store, TAPIR and Granola by more than 3X.
**Eris Performs Well on Independent Transactions**

Throughput (txns/sec)

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

- Eris achieves throughput within 10% of NT-UR
- Eris outperforms Lock-Store, TAPIR, and Granola by more than 3X
- More than 70% reduction in latency compared to Lock-Store, and within 10% latency of NT-UR
Eris also performs well on general transactions

<table>
<thead>
<tr>
<th>Throughput (txns/sec)</th>
<th>Lock-Store</th>
<th>TAPIR</th>
<th>Granola</th>
<th>Eris</th>
<th>NT-UR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed general transactions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Eris performs significantly better than the other systems, achieving throughput of over 1,200K transactions per second.
**ERIS ALSO PERFORMS WELL ON GENERAL TRANSACTIONS**

Eris maintains throughput within 10% of NT-UR.
ERIS EXCELS AT COMPLEX TRANSACTIONAL APPLICATIONS

TPC-C benchmark

Throughput (txns/sec)

- Lock-Store
- TAPIR
- Granola
- Eris
- NT-UR
Eris excels at complex transactional applications

TPC-C benchmark

Throughput (txns/sec)

7.6X and 6.4X higher throughput than Lock-Store and Tapur
**Eris excels at complex transactional applications**

TPC-C benchmark

- 7.6X and 6.4X higher throughput than Lock-Store and Tapir
- within 3% throughput of NT-UR
Eris is resilient to network anomalies

The diagram shows the throughput (txns/sec) against packet drop rate for different systems. Eris, Granola, Lock-Store, and TAPIR are compared. Eris maintains a higher throughput even as the packet drop rate increases, indicating its resilience to network anomalies.
Erisk is resilient to network anomalies.

![Graph showing throughput vs. packet drop rate for different systems.]

- **Eris**
- **Granola**
- **Lock-Store**
- **NT-UR**
- **TAPIR**

The graph illustrates the throughput (txns/sec) across different packet drop rates (0K to 1,800K) for each system.
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?