

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

Jialin Li  
University of Washington  
lijl@cs.washington.edu

Ellis Michael  
University of Washington  
emichael@cs.washington.edu

Dan R. K. Ports  
University of Washington  
drkp@cs.washington.edu

## ABSTRACT

Distributed storage systems aim to provide strong consistency and isolation guarantees on an architecture that is partitioned across multiple shards for scalability and replicated for fault tolerance. Traditionally, achieving all of these goals has required an expensive combination of atomic commitment and replication protocols – introducing extensive coordination overhead. Our system, Eris, takes a different approach. It moves a core piece of concurrency control functionality, which we term *multi-sequencing*, into the datacenter network itself. This network primitive takes on the responsibility for consistently ordering transactions, and a new lightweight transaction protocol ensures atomicity.

The end result is that Eris avoids both replication and transaction coordination overhead: we show that it can process a large class of distributed transactions *in a single round-trip* from the client to the storage system *without any explicit coordination* between shards or replicas in the normal case. It provides atomicity, consistency, and fault tolerance with less than 10% overhead – achieving throughput  $3.6\text{--}35\times$  higher and latency 72–80% lower than a conventional design on standard benchmarks.

## CCS CONCEPTS

• **Information systems** → **Database transaction processing**; **Distributed database transactions**; • **Networks** → **In-network processing**; *Data center networks*; • **Computer systems organization** → **Reliability**;

## KEYWORDS

distributed transactions, in-network concurrency control, network multi-sequencing

### ACM Reference Format:

Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017*, 17 pages. <https://doi.org/10.1145/3132747.3132751>

## 1 INTRODUCTION

Distributed storage systems today face a tension between transactional semantics and performance. To meet the demands of large-scale applications, these storage systems must be partitioned for scalability and replicated for availability. Supporting strong consistency and strict serializability would give the system the same semantics as a single system executing each transaction in isolation – freeing programmers from the need to reason about consistency and concurrency. Unfortunately, doing so is often at odds with the performance requirements of modern applications, which demand not just high scalability but also tight latency bounds. Interactive applications now require contacting hundreds or thousands of individual storage services on each request, potentially leaving individual transactions with sub-millisecond latency budgets [23, 49].

The conventional wisdom is that transaction processing systems cannot meet these performance requirements due to coordination costs. A traditional architecture calls for each transaction to be carefully orchestrated through a dizzying array of coordination protocols – e.g., Paxos for replication, two-phase commit for atomicity, and two-phase locking for isolation – each adding its own overhead. As we show in Section 8, this can increase latency and reduce throughput by an order of magnitude or more.

This paper challenges that conventional wisdom with Eris,<sup>1</sup> a new system for high-performance distributed transaction processing. Eris is optimized for high throughput and low latency in the datacenter environment. Eris executes an important class of transactions *with no coordination overhead*

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SOSP '17, October 28, 2017, Shanghai, China*

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-5085-3/17/10...\$15.00

<https://doi.org/10.1145/3132747.3132751>

<sup>1</sup>Eris takes its name from the ancient Greek goddess of discord, i.e., *lack of coordination*.

*whatsoever* – neither from concurrency control, atomic commitment, nor replication – and fully generic transactions with minimal overhead. It is able to execute a variety of workloads, including TPC-C [61], with less than 10% overhead compared to a non-transactional, unreplicated system.

The Eris architecture divides the responsibility for transaction isolation, fault tolerance, and atomic coordination in a new way. Eris isolates the core problem of transaction sequencing using *independent transactions* [21, 56], then optimizes their processing with a new network-integrated protocol. An independent transaction represents an atomic execution of a single, one-shot code block across multiple shards [21]. This abstraction is a useful one in itself – many workloads can be expressed solely using independent transactions [36] – as well as a building block for more complex operations.

The main contribution of Eris is a new protocol that can establish a linearizable order of execution for independent transactions and consensus on transaction commit without explicit coordination. Eris uses the datacenter network itself as a concurrency control mechanism for assigning transaction order. We define and implement a new network-level abstraction, *multi-sequencing*, which ensures that messages are delivered to all replicas of each shard in a consistent order and detects lost messages. Eris augments this network-level abstraction with an application-level protocol that ensures reliable delivery. In the normal case, this protocol is capable of committing independent transactions in a *single round trip* from clients to server, *without* requiring servers to communicate with each other.

Eris builds on recent work that uses network-level sequencing to order requests in replicated systems [22, 43, 54]. Sequencing transactions in a partitioned system (i.e., multi-sequencing) is substantially more challenging than ordering operations to a *single* replica group, as servers in different shards do not see the same set of operations, yet must ensure that they execute cross-shard transactions in a consistent order. Eris addresses this with a new concept, the *multi-stamp*, which provides enough information to sequence transactions, and can be implemented readily in an in-network sequencer.

While independent transactions are useful, they do not capture all possible operations. We show that independent transactions can be used as a building block to execute fully general transactions. Eris uses *preliminary transactions* to gather read dependencies, then commits them with a single *conclusory* independent transaction. Although doing so imposes locking overhead, by leveraging the high performance of the underlying independent transaction primitive, it continues to outperform conventional approaches that must handle replication and coordination separately.

We evaluate Eris experimentally and demonstrate that it provides throughput 3.6–35× higher and latency 72–80%

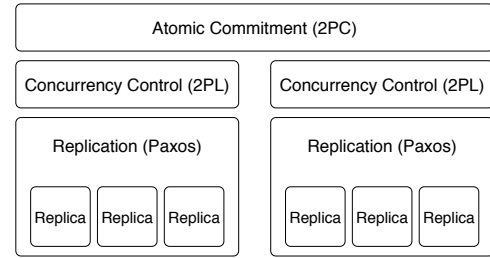


Figure 1: Standard layered architecture for a partitioned, replicated storage system

lower than a conventional design (two-phase commit with Paxos and locking). Because Eris can execute most transactions in a single round trip without communication between servers, it achieves performance within 3% of a non-transactional, unreplicated system on the TPC-C benchmark, demonstrating that strong transactionality, consistency, and fault tolerance guarantees can be achieved without a performance penalty.

## 2 BACKGROUND

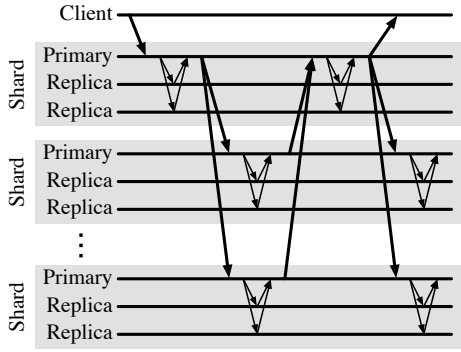
We consider storage systems that are partitioned for scalability, and replicated for fault tolerance. Data is partitioned among different *shards*, each consisting of multiple replicas with copies of all shard data. Clients (e.g., front-end servers) submit transactions to be processed. We limit ourselves here to systems where all nodes are located in the same datacenter.

A storage system should provide several guarantees. Every transaction should be applied to all shards it affects, or none at all (atomicity). The execution should be identical to each transaction being executed in sequence (strict serializable isolation). And these guarantees should hold even though some of the nodes in each shard can fail (fault tolerance).

### 2.1 The Status Quo: Extensive Coordination

Existing systems generally achieve these goals using a layered approach, as shown in Figure 1. A replication protocol (e.g., Paxos [40]) provides fault tolerance within each shard. Across shards, an atomic commitment protocol (e.g., two-phase commit) provides atomicity and is combined with a concurrency control protocol (e.g., two-phase locking or optimistic concurrency control). Though the specific protocols differ, many systems use this structure [2, 3, 16, 19, 21, 29, 38, 47].

A consequence is that coordinating a single transaction commit requires multiple rounds of coordination. As an example, Figure 2 shows the protocol exchange required to commit a transaction in a conventional layered architecture like Google’s Spanner [19]. Each phase of the two-phase commit protocol requires synchronously executing a replication protocol to make the transaction coordination decision persistent. Moreover, two-phase locking requires that locks



**Figure 2: Coordination required to commit a single transaction with traditional two-phase commit and replication**

be held between prepare and commit operations, blocking conflicting transactions. This combination seriously impacts system performance.

### 3 ERIS DESIGN PRINCIPLES

Eris takes a different approach to transaction coordination that allows it to achieve higher performance. It is based on the following three principles:

*Principle 1: Separating Ordering from Execution.* Traditional coordination protocols establish the serializable order of transactions concurrently with executing those transactions, e.g., as a result of the locks that are acquired during execution. Eris explicitly separates the task of establishing the serial order of transactions from their execution, allowing it to use an optimized protocol for transaction ordering.

To make this possible, the Eris protocol relies on a specialized transaction model: independent transactions [21]. Independent transactions apply concurrent changes atomically at multiple shards, but forbid cross-shard data dependencies (we make this definition precise in Section 4.1). Independent transactions have the key property that *executing them sequentially at each shard in global sequence order guarantees serializability*. That is, establishing a global serial order allows transaction execution to proceed without further coordination.

*Principle 2: Rapid Ordering with In-Network Concurrency Control.* How quickly can we establish a global order of independent transactions? Existing systems require each of the participating shards in a transaction to coordinate with each other in order to ensure that transactions are processed at each affected shard in a consistent order. This requires at least one round of communication before the transaction can be executed, impeding system performance.

Eris establishes a global order of transactions with minimal latency by using the network itself to sequence requests. Recent work has shown that network-level processing elements

can be used to assign a sequence number to each message destined for a replica group, making it possible to detect messages that are dropped or delivered out of order [43]. Eris takes this approach further, using the network to sequence multiple streams of operations destined for different shards. The key primitive, multi-sequencing, *atomically* applies a sequence number for each destination of a message, establishing a global order of messages and ensuring that any recipient can detect lost or reordered messages. Eris uses this to build a transaction processing protocol where coordination is not required unless messages are lost.

*Principle 3: Unifying Replication and Transaction Coordination.* Traditional layered designs use separate protocols for atomic commitment of transactions across shards and for replication of operations within an individual shard. While this separation provides modularity, it has been recently observed that it leads to redundant coordination between the two layers [66]. Protocols that integrate cross-shard coordination and intra-shard replication into a unified protocol have been able to achieve higher throughput and lower latency [38, 48, 66].

This approach integrates particularly well with Eris’s in-network concurrency control. Because requests are sequenced by the network, each individual replica in a shard can independently process requests in the same order. As a result, in the common case Eris can execute independent transactions in a single round trip, without requiring *either* cross-shard or intra-shard coordination.

### 4 ERIS ARCHITECTURE

Eris divides the responsibility for different guarantees in a new way, enabling it to execute many transactions without coordination. The protocol itself is divided into three layers, as shown in Figure 3:

- (1) The *in-network concurrency control layer* (Section 5) uses a new network primitive to establish a **consistent ordering** of transactions, both within and across shards, but does not guarantee reliable message delivery.
- (2) The *independent transaction layer* (Section 6) adds **reliability** and **atomicity** to the ordered operations, ensuring that each transaction is eventually executed at all non-faulty replicas within each shard (or fails entirely). This combination of ordering and reliability is sufficient to guarantee linearizability for an important class of transactions.
- (3) The *general transaction layer* (Section 7) provides **isolation** for fully general transactions, by building them out of independent transactions and relying on the linearizable execution provided by other layers.

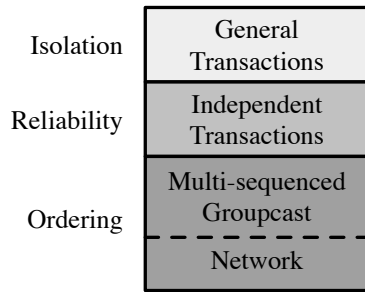


Figure 3: The layers of Eris and the guarantees they provide

#### 4.1 Transaction Model

Transactions in Eris come in two flavors. The core transaction sequencing layer handles *independent transactions*. These can be used directly by many applications, and doing so offers higher performance. Eris also supports *general transactions*.

**Independent transactions** are one-shot operations (i.e., stored procedures) that are executed atomically across a set of participants. That is, the transaction consists of a piece of code to be executed on a subset of shards. These stored procedures cannot interact with the client, nor can different shards communicate during their execution. Each shard must independently come to the same “commit” or “abort” decision without coordination—e.g., by always committing. This definition of independent transactions is taken from Granola [21]; essentially the same definition was previously proposed by the authors of H-Store [56] under the name “strongly two-phase”.

Like the H-Store architecture, in our implementation of Eris, the underlying data store executes independent transactions sequentially on each participant, without concurrency. This allows it to avoid the overhead of lock- and latch-based synchronization, which collectively amount to as much as 30% of the execution cost of traditional DBMS designs [30]. This architecture restricts transaction execution to a single thread. Multicore systems can operate one logical partition per core, at the cost of potentially increasing the number of distributed transactions.

Although the independent transaction model is restrictive, it captures many common classes of transactions. Any read-only transaction can be expressed as an independent transaction; Eris’s semantics make it a consistent snapshot read. Any one-round distributed read/write transaction that always commits (e.g., unconditionally incrementing a set of values) is an independent transaction. Finally, data replicated across different shards (as is common for frequently accessed data) can be updated consistently with an independent transaction. Prior work has shown that many applications consist largely or entirely of independent transactions [20, 21, 56]. As one example, TPC-C [61], an industry standard benchmark designed to represent transaction processing workloads, can be

expressed entirely using independent transactions, despite its complexity [56].

**General transactions** provide a standard interactive transaction model. Clients begin a transaction, then execute a sequence of reads and writes at different shards; each may depend on the results of previous operations. Finally, the client decides whether to `Commit` or `Abort` the transaction. These can be used to implement any transaction processing workload.

## 5 IN-NETWORK CONCURRENCY CONTROL

Traditional transaction processing systems are network-oblivious, relying on application-level protocols for everything from sequencing operations to ensuring that messages are delivered to the right participants. Recent work has demonstrated that it is possible to accelerate coordination for replicated systems by using advanced processing capabilities in the network layer to build sequencing primitives [43, 54]. However, large-scale transaction processing presents major new challenges.

Using a dedicated sequencing component is not, in itself, a new idea. Sequencers have previously been used to accelerate consensus [4, 34] and transaction processing systems [9, 28, 53, 62], and have been implemented in software [4, 53], using RDMA NICs [35], and using in-network processing components [43]. In particular, NOPaxos [43] showed that it is possible to build a network-level device that assigns globally consistent, consecutive sequence numbers to all packets destined for a replica group. Sequence numbers allow receivers to reject messages that arrive out of order, and to detect dropped messages (as gaps in the sequence numbers). These, in turn, enable an optimized replication protocol where replicas only need to coordinate when messages are lost or reordered in the network.

Can the same be done for transaction processing? In this paper, we show that existing network-layer mechanisms (including NOPaxos’s OUM) are not suited for this purpose. They establish an order over a set of messages to a single destination group, while coordination-free transaction execution requires a *consistent* ordering across messages delivered to *many* destination shards. Eris’s contribution is an in-network concurrency control primitive that establishes such an ordering and allows receivers to detect dropped messages, along with a strategy to realize this primitive efficiently in programmable switch architectures.

Part of achieving this ordering is making the set of transaction participants explicit to the network layer. Traditionally, clients send transactions to multiple groups by sending separate multicast messages to each group (or, often, separate unicast messages to each member of each group). This makes it impossible to guarantee a meaningful order at the network

level: without knowing which separate messages correspond to the same logical operation, one cannot guarantee a consistent ordering across different transaction participants. To address this semantic gap, we introduce two new concepts:

- *Groupcast* – an extended multicast primitive delivers messages to a client-specified *set* of multicast groups.
- *Multi-sequenced groupcast* – a specialized groupcast that guarantees messages are delivered to all groupcast recipients in a globally consistent order. The multi-sequenced groupcast primitive does not guarantee reliable delivery, but it does guarantee that recipients can detect dropped messages.

An important goal of this design is to minimize the logic required for the network. This simplifies implementation and increases overall system reliability; end-to-end guarantees are enforced in the application. The primitives that we identify are sophisticated enough to enable the Eris transaction processing algorithm and thus dramatically increase system performance, but simple enough to be readily and efficiently implemented in a variety of network devices.

### 5.1 Why Multi-Sequencing?

Our work extends the OUM model [43] to the multi-group environment of transaction processing. This requires messages to be sequenced atomically for *multiple* replica groups with the same guarantees. To illustrate the need for such an ordering mechanism, and the challenges in achieving one, we consider two straw-man proposals:

1) *Total Global Sequencing*. Consider first applying the OUM approach directly to the entire storage system, using a single sequencer. All transactions are sent through this sequencer, which assigns each a sequence number, then forwards them to all replicas of all shards in the system. Because of the single global sequence number, this design is capable of ensuring both ordering (no two receivers process messages in different orders) and drop detection (recipients are notified of any dropped message). However, it requires every server to receive every message involving *any* shard in the system, clearly impeding system performance.

Note that it is not possible to adapt this design so that messages are delivered only to replicas in the affected shards while still maintaining ordering and drop detection. With a global sequence number, a receiver seeing message  $n$  followed by message  $n + 2$  cannot distinguish the case where it was intended to receive message  $n + 1$  from the case in which message  $n + 1$  was not sent to its shard.

2) *Multiple Independent Sequencing*. Alternatively, consider employing the OUM approach by treating each shard as a *separate* OUM group. Messages sent to a shard are sequenced independently of other shards and then delivered

to all replicas in the shard. Unlike total global sequencing, with this approach messages are only delivered to the shards that need to process them. Moreover, replicas in a shard can detect dropped messages within a shard. However, ordering and detection are not guaranteed *across* different shards. If transactions  $T_1$  and  $T_2$  are each sent to both shards  $A$  and  $B$ , it is possible that the sequencer for shard  $A$  processes  $T_1$  before  $T_2$  while the sequencer for shard  $B$  processes  $T_2$  before  $T_1$ . It is also possible that a transaction processed by  $A$ 's sequencer is dropped in the network before ever reaching  $B$ 's sequencer, or vice versa. These anomalies could result in violations of system correctness.

What is needed in order to ensure a correct, consistent ordering is a way to ensure that messages delivered to multiple multicast groups *are sequenced atomically across all recipient groups*. Our design below achieves this goal in two parts. Groupcast provides a way for applications to direct messages to *multiple* multicast groups, and multi-sequencing ensures atomic sequencing across all destination groups. This is achieved using a new technique, the *multi-stamp*.

### 5.2 Groupcast and Multi-sequenced Groupcast

We begin by defining the properties of the groupcast and multi-sequenced groupcast primitives.

*Groupcast*. Traditional multicast sends messages to a pre-defined group of recipients, e.g., an IGMP group. Communication in a partitioned, replicated transaction processing system does not fit this communication model well. Transactions must be delivered to *multiple* groups of replicas, one for each shard affected by the transaction; which groups are involved varies depending on the transaction particulars.

We instead propose the *groupcast* primitive, where a message is sent to *multiple* multicast groups. The set of destinations is specified. In our design, this is achieved by sending the message to a special groupcast IP address. Using SDN rules, packets matching this destination IP address are processed specially. An additional header located between the IP and UDP headers specifies a list of destination groups; the packet is delivered to each member of each group.

*Multi-sequenced groupcast*. Multi-sequencing extends the groupcast primitive with additional ordering guarantees. Namely, it provides the following properties:

- **Unreliability**. There is no guarantee that any message will ever be delivered to its recipient.
- **Partial Ordering**. The set of all multi-sequenced groupcast messages are partially ordered—with the restriction that any two messages with a destination group in common are comparable. Furthermore, if  $m_1 \prec m_2$ ,

and a receiver delivers both  $m_1$  and  $m_2$ , then it delivers  $m_1$  before  $m_2$ .

- **Drop Detection.** Let  $R(m)$  be the set of recipients of message  $m$ . For any message  $m$ , either: (1) every receiver  $r \in R(m)$  delivers either  $m$  or a DROP-NOTIFICATION for  $m$ , or (2) no receiver  $r \in R(m)$  delivers  $m$  or a DROP-NOTIFICATION for  $m$ .

Multi-sequencing can thus establish an ordering relationship between messages with *different sets of receivers*. This is an important distinction with OUM, which only supports ordering *within a single multicast group*. Multi-sequencing requires an ordering relationship between any two messages that have some receiver in common, i.e.,  $R(m_1) \cap R(m_2) \neq \emptyset$ .

### 5.3 Multi-Sequencing Design

Multi-sequenced groupcast is implemented using a centralized, network-level *sequencer*. One sequencer is designated for the system at any time; it can be replaced when it fails. Depending on implementation (Section 5.4), the sequencer can be either an end-host, a middlebox, or a sufficiently powerful switch. All multi-sequenced groupcast packets are routed through this sequencer, which modifies them to reflect their position in a global sequence. Receivers then ensure that they only process messages in sequence number order.

The challenge for multi-sequenced groupcast is how the sequencer should modify packets. As described above, affixing a single sequence number creates a global sequence, making it possible to meet the ordering requirement but not the drop detection requirement. In order to satisfy both requirements, we introduce a new concept, the *multi-stamp*.

A multi-stamp is a set of  $\langle \text{group-id}, \text{sequence-num} \rangle$  pairs, one for each destination group of the message. To apply multi-stamps, a sequencer maintains a separate counter for each destination group it supports. Upon receiving a packet, it parses the groupcast header, identifies the appropriate counters, increments each of them *atomically*, and writes the set of counters into the packet header as a multi-stamp.

Including the full set of counters for each destination group in the multi-stamp serves two purposes. First, each receiver can ensure the ordering and drop detection properties. It checks the appropriate sequence number for its group; if the value is lower than that of the last delivered packet, this indicates an out-of-order packet, and it is dropped. If the sequence number is higher than the next expected packet, this indicates a potentially dropped packet, so the application (i.e., Eris) is notified. Second, a receiver can request a missing packet by its sequence number, even from other groups.

*Fault tolerance and epochs.* Multi-sequencing requires the sequencer to keep state: the latest sequence number for each destination group. Of course, sequencers can fail. Rather than

trying to keep sequencer state persistent – which would require synchronous replication of the sequencer and complex agreement protocols – we instead have the sequencer keep only soft state, and expose sequencer failures to the application.

To handle sequencer failures, we introduce a global *epoch number* for the system. This number is maintained by the sequencer, and added to the groupcast header along with the multi-stamp. Responsibility for sequencer failover lies with the SDN controller. When it suspects the sequencer of having failed (e.g., after a timeout), it selects a new sequencer, increments the epoch number, and installs that epoch number in the new sequencer. Notice that delivery in lexicographic, epoch number major, multi-stamp minor order satisfies the partial ordering multi-sequencing requirement.

When a receiver receives a multi-sequenced groupcast message with a higher epoch number than it has seen before, it delivers a NEW-EPOCH notification to the application (i.e., Eris). This notifies the application that some packets may have been lost; the application is responsible for reaching agreement on which packets from the previous epoch were successfully delivered before processing messages from the next epoch.

As in OUM, the SDN must install strictly increasing epoch numbers to successive sequencers [43]. For fault tolerance, we replicate the controller using standard means, a common practice [32, 37]. Alternatively, a new sequencer could set its epoch number using the latest physical clock value, provided that clocks are sufficiently well synchronized to remain monotonic in this context.

### 5.4 Implementation and Scalability

Our implementation of the Eris network layer includes the in-network sequencer, an SDN controller, and an end-host library to interface with applications like the Eris transaction protocol. The SDN controller, implemented using POX [50], manages groupcast membership and installs rules that route groupcast traffic through the sequencer. The end-host library provides an API for sending and receiving multi-sequenced groupcast messages. In particular, it monitors the appropriate sequence numbers on incoming multi-stamped messages and sends the application DROP-NOTIFICATION or NEW-EPOCH notifications as necessary.

The sequencer itself can be implemented in several ways. We have built software-based prototypes that run on a conventional end-host and a middlebox implemented using a network processor, and evaluated their performance as shown in Table 1. However, the highest-performance option is to implement multi-sequenced groupcast functionality directly in a switch. This is made possible by programmable network dataplane architectures that support per-packet processing.

*In-switch designs.* For maximum performance, we envision multi-sequencing and groupcast being implemented directly in network switches. Programmable network hardware architectures such as Reconfigurable Match Tables [13], Intel FlexPipe [52], Cavium XPliant [65], and Barefoot Tofino [6] provide the necessary processing capability.

We have implemented multi-sequenced groupcast in the P4 language [12], supporting compilation to several of these future architectures. The complete P4 source code is available [42]. The hardware required to evaluate this approach is not yet commercially available, though such products are expected within the next year. We can, however, analyze the resource usage of our design to understand the feasibility and potential scalability of in-network concurrency control.

Consider a Reconfigurable Match Table (RMT) [13] architecture. This architecture provides a pipeline of stages that match on header fields and perform actions. It also provides stateful memory, one register of which can store each per-shard counter. This design allows line-rate processing at terabit speed, if the necessary functionality can be expressed in the packet processing pipeline. The barrier to scalability, then, is the number of shards to which a single multi-sequenced groupcast packet can be addressed. Two resource constraints govern this limit. The first is how many stateful counters can be incremented on each packet. The RMT proposal specifies 32 stages, each with 4–6 register-attached ALUs per stage, supporting 128–192 destinations per packet. Second, the packet header vector containing fields used for matching and action is limited to 512 bytes. Assuming 32-bit shard IDs and counter values, this allows 116 simultaneous destinations after accounting for IP and UDP headers. For very large systems where transactions may span more than 100 shards, it may be necessary to use special-case handling for global (all-shard) messages.

*Middlebox prototype.* As sufficiently capable switches are not yet available, we implement a multi-stamping sequencer on a Cavium Octeon II CN6880 network processor. This device contains 32 MIPS64 cores and provides low-latency access to four 10 Gb/s Ethernet interfaces. We use the middlebox implementation in our evaluation (Section 8). Although it uses neither a heavily optimized implementation nor especially powerful hardware (the CN6880 was released in 2010), it can process 6.19M multi-sequenced packets per second, close to the maximum capacity of its 10 Gb/s link (7M packets/sec).

*End-host sequencing.* An alternate design option is to implement the sequencing functionality on an end host. This provides a more convenient deployment option for environments where the network infrastructure cannot be modified. The tradeoff is this imposes higher latency (approximately 10  $\mu$ s per transaction), and system throughput may be limited

	Throughput (packets/second)	Latency ( $\mu$ s)
<b>Middlebox</b>	6.19M ( $\sigma = 3.16$ K)	13.64 ( $\sigma = 0.42$ )
<b>Endhost</b>	1.61M ( $\sigma = 19.98$ K)	24.60 ( $\sigma = 1.02$ )

Table 1: Performance of endhost and middlebox sequencers

by sequencer capacity. Our straightforward implementation of the multi-sequencer in user space on Linux can sequence up to 1.61M requests per second on a 24-core Xeon E5-2680 machine, sufficient for smaller deployments. Low-level optimizations and new hardware such as RDMA NICs can likely improve this capacity [35].

## 6 PROCESSING INDEPENDENT TRANSACTIONS

Eris’s independent transaction processing layer provides single-copy linearizable<sup>2</sup> (or strict serializable) semantics for independent transactions. Independent transactions have the property that executing them one-at-a-time at each shard guarantees strict serializable behavior, provided they are executed in a consistent order. Network multi-sequencing establishes just such an order over transactions. However, it does not guarantee reliable delivery. Thus, for correctness Eris must build reliable delivery semantics at the application layer and ensure that replicas agree on *which* transactions to commit, not their order. In the normal case, Eris is able to execute independent transactions using only a single round trip from the client to all replicas.

### 6.1 Overview

Eris uses a quorum-based protocol to maintain safety always – even when servers and the underlying network behave asynchronously – and availability even when up to  $f$  out of  $2f + 1$  replicas in any shard fail by crashing. Eris clients send independent transactions directly to the replicas in the affected shards using multi-sequenced groupcast and wait for replies from a majority quorum from each shard. There is a single *Designated Learner (DL)* replica in each shard. Only this replica actually executes transactions synchronously; the other replicas simply log them and execute them later. As a result, Eris requires that clients wait for a response from the DL before considering a quorum complete. Using a DL serves two purposes. First, it allows single-round-trip execution without the need for speculation and rollback: only the DL executes the request, and, unless it fails and is replaced, it is involved in every transaction committed by the shard. (NOPaxos [43] uses the same principle.) Second, only the DL in each shard sends the transaction result to the client; the

<sup>2</sup>Linearizability is the strongest practical correctness condition for concurrent objects [31]. It is equivalent to strict serializability for transactions; because independent transactions are one-shot operations on each shard, we use the term “linearizability” here.

others only send an acknowledgment, avoiding unnecessary network congestion at the client.

Eris must be resilient to replica failures (in particular, DL failures) and network anomalies. In our multi-sequencing abstraction, these anomalies consist of DROP-NOTIFICATIONS (when a multi-sequenced groupcast transaction is dropped or reordered in the network) and NEW-EPOCH notifications (when a sequencer has been replaced). In Eris, failure of the DL is handled entirely within the shard by a protocol similar in spirit to standard leader change protocols [40, 43, 51]. DROP-NOTIFICATIONS and NEW-EPOCH notifications, however, require coordination across shards. For DROP-NOTIFICATIONS, all participant shards for the dropped transaction must reach the same decision about whether or not to discard the message. For NEW-EPOCH notifications, the shards must ensure that they transition to the new epoch *in a consistent state*.

To manage the complexity of these two failure cases, we introduce a novel element to the Eris architecture: the Failure Coordinator (FC). The FC is a service that coordinates with the replicas to recover consistently from packet drops and sequencer failures. The FC must be replicated using standard means [39, 43, 51] to remain available. However, the overhead of replication and coordination is not an issue: Eris invokes the FC and incurs its overhead only in rare failure cases, not in the normal path.

The state maintained by replicas is summarized in Figure 4. Two important pieces of state are the *view-num* and *epoch-num*, which track the current DL and multi-sequencing epoch. Specifically, the DL for *view-num*  $v$  is replica number  $v \bmod N$ , where  $N$  is the number of replicas in the shard. Eris replicas and the FC tag all messages with their current *epoch-num* and do not accept messages from previous epochs (except during epoch change). If a replica ever receives a message from a later epoch, it must use the FC to transition to the new epoch before continuing.

Eris consists of five sub-protocols: the normal case protocol, the protocol to handle dropped messages, the protocol to change the DL within a shard, the protocol to change epochs, and the protocol to periodically synchronize replicas' states and allow all replicas to safely execute transactions. Below we present all five. Some details are elided for brevity; full details are in an accompanying technical report [42]. Throughout these protocols, messages that are sent but not acknowledged with the proper reply are retried. In particular, clients repeatedly retry transactions until they receive the correct responses; at-most-once semantics are guaranteed using the standard technique of maintaining a table of the most recent transaction from each client [45].

#### Replica:

- *replica-id* =  $\langle \text{shard-num}, \text{replica-num} \rangle$
- *status* — one of Normal, ViewChange, EpochChange
- *view-num* — indicates which replica within the shard is believed to be the DL
- *epoch-num* — indicates which sequencer the replica is currently accepting transactions from
- *log* — independent transactions and NO-OPs in sequential order
- *temp-drops* — set of tuples of the form  $\langle \text{epoch-num}, \text{shard-num}, \text{sequence-num} \rangle$ , indicating which transactions the replica has tentatively agreed to disregard
- *perm-drops* — indicates which transactions the FC has committed as permanently dropped
- *un-drops* — indicates which transactions the FC has committed for processing

Figure 4: Local state of Eris replicas used for independent transaction processing

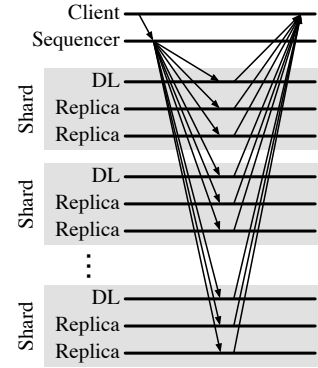


Figure 5: Communication pattern of Eris in the normal case, where the independent transaction is sent via multi-sequenced groupcast to multiple shards, each consisting of 3 replicas (one replica in each shard is a Designated Learner)

## 6.2 Normal Case

In the normal case, clients submit independent transactions via the multi-sequencing layer, and each replica that receives the message in order simply responds to the client; the DL executes the transaction and includes the result. Thus, transactions are processed in a single round trip. Figure 5 illustrates this process.

- (1) First, the client sends the transaction to all replica groups for all participant shards, using multi-sequenced groupcast.
- (2) The replicas receive the transaction, place it in their logs, and reply with  $\langle \text{REPLY}, \text{txn-index}, \text{view-num}, \text{result} \rangle$ , where *txn-index* is the index of the transaction in the replica's *log*. Only the DL for the view actually executes the transaction and includes its *result*; the other replicas simply log the transaction.



- (3) The client waits for a *view-consistent* quorum reply from each shard.

Here, a *view-consistent quorum reply* from a shard is a `REPLY` from a majority of the shard’s replicas, with matching *txn-index*, *view-num*, and *epoch-num*, including one from the DL.

Note that a replica cannot process a transaction if it has a matching transaction identifier in its *perm-drops* or *temp-drops*; if there is a matching identifier in its *perm-drops*, it inserts a `NO-OP` into its *log* in the transaction’s place and continues—otherwise the replica must wait. A matching identifier in its *perm-drops* or *temp-drops* indicates that the FC considers the transaction definitively or potentially failed, as discussed below.

### 6.3 Dropped Messages

Replicas receive `DROP-NOTIFICATIONS` from the multi-sequencing layer when they miss a message intended for their shard because of a network anomaly. Here, atomicity requires that either every participant shard learn and execute the missing transaction (as in Section 6.2), or that none execute it. This process is coordinated by the FC, which contacts the other nodes in the system in an attempt to recover the missing transaction. If *any* node has a copy of the missing transaction, the FC sends it to the other replicas. Otherwise, the FC uses a round of agreement to ensure that all replicas agree to drop the transaction and move on.

- (1) When a replica in a shard detects that it missed some transaction, it sends  $\langle \text{FIND-TXN}, \text{txn-id} \rangle$  to the FC, where *txn-id* is a triple of the replica’s *shard-num*, its current *epoch-num*, and its shard’s sequence number for the message.
- (2) The FC receives this `FIND-TXN` and (assuming that it hasn’t already found or dropped the missing transaction) broadcasts  $\langle \text{TXN-REQUEST}, \text{txn-id} \rangle$  to *all replicas in all shards*.
- (3) When a replica receives `TXN-REQUEST`, if it has received a transaction matching *txn-id*, it replies with  $\langle \text{HAS-TXN}, \text{txn} \rangle$ . Otherwise, it adds *txn-id* to *temp-drops* and replies with  $\langle \text{TEMP-DROPPED-TXN}, \text{view-num}, \text{txn-id} \rangle$ .

Once a replica sends `TEMP-DROPPED-TXN`, it cedes control of that transaction’s fate to the FC: even if it later receives the transaction, it cannot process it until it has learned whether the FC has found or permanently dropped the transaction.

- (4) The FC waits for either a *quorum* of `TEMP-DROPPED-TXNs` from every shard or a *single* `HAS-TXN`, whichever comes first. As before, each quorum must be view-consistent and include the DL of the view.  
If the FC first receives the `HAS-TXN` and hasn’t previously dropped the transaction, it saves it and sends

$\langle \text{TXN-FOUND}, \text{txn} \rangle$  to all participants in the transaction.

If it first receives the necessary `TEMP-DROPPED-TXNs` (or receives `HAS-TXN`, having previously dropped the transaction), it decides that the transaction matching *txn-id* is permanently dropped and sends  $\langle \text{TXN-DROPPED}, \text{txn-id} \rangle$  to all replicas.

- (5) When a replica hears back from the FC, if it receives a `TXN-FOUND`, it adds the transaction to its *un-drops*, adding the transaction to its *log* and replying to the client. If it receives a `TXN-DROPPED`, it adds the *txn-id* to *perm-drops*, adding a `NO-OP` to its *log* if necessary. In either case, the replica can then proceed to execute subsequent transactions.

As an optimization, before executing this procedure, a replica that receives a `DROP-NOTIFICATION` first contacts the other replicas in its shard. If one of them received the missing message, it can respond with it, allowing the first replica to process the transaction as normal. If successful, this allows a replica to recover from a dropped message without involving the FC. In our experience, this optimization is important, as message losses that affect all replicas in a shard are rare.

### 6.4 Designated Learner Failure

Because only the DL executes transactions, and the ability to make progress is dependent on each shard having a DL, Eris has a view change protocol to replace the DL if it fails. To ensure the system remains correct, the new DL must learn about all transactions committed in previous views. It must also learn about any `TEMP-DROPPED-TXNs` sent by a majority in previous views, and refrain from processing these transactions until learning their outcome from the FC.

The view change is achieved using a protocol similar to Viewstamped Replication [45, 51]. We provide only a brief overview due to space constraints; the details appear in [42].

Any replica that suspects the DL to have failed stops processing requests and sends a `VIEW-CHANGE` message to the DL of the next view, including its current state. The DL of the new view waits for such messages from a majority, then assembles the state for the new view using the longest *log* it received and the union of the *temp-drops*, *perm-drops*, and *un-drops* it received, overwriting any transactions in the new *log* with `NO-OPS` if they match *txn-ids* in *perm-drops*. If the new *log* has any transactions matching *txn-ids* in *temp-drops* without corresponding *txn-ids* in *un-drops*, the DL must wait for the FC to come to a decision about those *txn-ids*, retrying—asking the FC and sending any necessary `HAS-TXNs`—if necessary. The DL then sends a `START-VIEW` message with the new state to the other replicas, which adopt the new state and transition to the new view.

A view change (or epoch change) could result in the DL from the old view having executed transactions which are eventually dropped, potentially requiring application-level rollback. Eris handles this possibility with application state transfer.

## 6.5 Epoch Change

Eris also needs to be able to handle epoch changes in the multi-sequencing layer, i.e., sequencer failures. As with dropped messages, the FC manages this process. It ensures all replicas across all shards start in the new epoch in *consistent states*, i.e., that replicas learn about transactions committed in previous epochs and that no replica knows about a transaction which the other participants in the transaction do not know about.

When a replica receives a NEW-EPOCH notification from the network layer, it stops processing requests and sends the FC an EPOCH-CHANGE-REQ message. The FC then requests from every replica the replica's current state (or sufficient metadata) along with a promise not to process operations for any lower-numbered epoch. The FC waits for such promises from a simple majority of replicas in every shard. It then decides the initial state for every shard in the new epoch using only those responses from the most recent epoch the FC started. For each shard, it takes the highest *view-num* it received and a *log* that contains all of the transactions the FC received which have that shard as a participant (with transactions the FC previously dropped being replaced by NO-OPS). It sends that state as a START-EPOCH message to all replicas, which adopt the new *view-num* and *log*, execute any new transactions, and clear their *temp-drops*, *perm-drops*, and *un-drops*. Having reached a consistent state, they are now able to process messages from the new sequencer. The FC retains and retransmits these START-EPOCH messages until a majority of replicas from each shard acknowledge the new epoch. See [42] for the full protocol details.

## 6.6 Synchronization

During the normal processing of independent transactions (Section 6.2), only the DL of each shard executes independent transactions synchronously; other replicas simply log transactions. In order to prevent the application states of those replicas from becoming too out of date, Eris utilizes a synchronization protocol exactly as in NOPaxos [43]. Periodically, the DL of each shard synchronizes its *log* with the other replicas and informs them that it is safe to execute the independent transactions therein. Again, see [42] for the full details.

## 6.7 Correctness

Eris guarantees linearizable execution of independent transactions and liveness during periods of stability. A full proof and

a model-checked TLA+ specification appear in [42]. Here, we give a sketch of the main safety argument.

*Definitions.* Transaction  $t$  is committed at a shard in a log slot if that shard sent a view-consistent quorum of REPLYs for  $t$ . A *drop promise* for *txn-id*  $\tau$  is committed if a view-consistent quorum sent TEMP-DROPPED-TXNs for  $\tau$ . We say a *log* (consisting of transactions and NO-OPS) is *stable* if its transactions are a prefix of the transactions of the *logs* of all replicas in the same shard in later epochs and views.

We first consider the behavior of a single shard:

*Invariant:* If a transaction is committed in a log slot, then it will be in that slot in the *log* of any replica in the same shard in a later view or epoch. Similarly, if a drop promise for a *txn-id* is committed, then any replica in the same shard starting a later view in the same epoch will have the *txn-id* in its *temp-drops*.

This guarantees that at any time, the DL of a shard has a record of all previously committed transactions and current drop promises. The protocol ensures this invariant by starting a new view or epoch with the union of the states from a majority of replicas, at least one of which must have participated in any previously committed transaction or drop promise.

Because replicas add transactions to their *logs* in sequential order, once a transaction in the DL's *log* is committed, any earlier transactions in its *log* must also have been present at a majority and will therefore survive into later views and epochs. Further, no lower-numbered transactions not in its *log* can survive into later views and epochs: any NO-OPS in the *log* correspond to transactions dropped by the FC after a committed drop promise. Future DLs will find out about this drop promise during view change, and the FC will not start the next epoch with that transaction. Therefore, the DL's *log* is stable, meaning the behavior of a single shard is indistinguishable from a single, correct node.

Next, we consider the aggregate behavior of multiple shards. Because of the prior invariant and the fact that the FC only drops a transaction after a drop promise from every shard, the following invariant holds (a full proof appears in [42]):

*Invariant:* If transaction  $t$  was committed by shard  $s$ , then for all other participant shards  $s'$ : if  $s'$  has committed  $t' \succ t$ , then  $s'$  has committed  $t$ . Here,  $\succ$  is the partial order assigned by the multi-sequencing layer.

That is, transactions are executed atomically across shards in a way that respects the multi-sequencing order. Since multi-sequencing guarantees a partial order where any potentially conflicting transactions are comparable, any execution respecting this order will be free of conflict cycles and thus serializable. Further, that order respects the real-time ordering

of transactions as received by successive sequencers, making Eris’s transaction execution linearizable.

## 7 BUILDING GENERAL TRANSACTIONS

Many – but not all – important operations are expressible as independent transactions. One type of exception is a conditional update that depends on data stored on another shard, e.g., a banking transaction which moves funds from one account to another only if there are sufficient funds. In many cases, these operations can be avoided through careful partitioning (or even state duplication), e.g., by placing both accounts on one shard [21]. However, to support all workloads, we extend Eris to support *general transactions*, which can have cross-shard dependencies.

Eris runs general transactions by dividing them into multiple independent transactions. General transaction execution is thus a layer running atop independent transaction execution. This simplifies design: the general transaction implementation can rely on the fact that the independent transaction processing layer is correct and provides linearizable execution. That is, it can assume that a single, correct machine is processing independent transactions sequentially.

Supporting strong isolation in the presence of these more general transactions requires an additional concurrency control mechanism. Eris uses strict two-phase locking. Shards maintain read and write locks for every data item, used only when there are outstanding general transactions. While a lock is held, any independent or general transactions that affect the corresponding data item wait until it is released.

### 7.1 General Transaction Protocol

We first consider general transactions whose full read/write sets are known *a priori*. These transactions are committed in two phases. In the first phase, the client sends a *preliminary transaction*, which executes the reads and acquires all read and write locks. In the second phase, the client sends a *conclusory transaction*, which either `Commit`s or `Abort`s the general transaction. A `Commit` installs the transaction’s modifications; in both cases, the transaction’s locks are released.

Eris can also execute transactions whose read and write sets are not known at start time, i.e., they are state dependent. To this end, Eris employs *reconnaissance queries* precisely as in Calvin [59]. That is, before sending the preliminary component of a general transaction, the client sends single-message, non-transactional reads to determine the full read/write sets. The preliminary transaction checks that the values returned by reconnaissance queries are still valid. If any have been changed, the general transaction will be aborted. Otherwise, the conclusory transaction can proceed as above.

### 7.2 Handling Client Failures

Eris clients are their own transaction managers. Because clients can fail, Eris must be able to abort a general transaction started by a failed client to allow the system to maintain progress. In general, solving this problem is the domain of complex cooperative termination protocols [8]. Because Eris builds on the atomic execution of independent transactions, however, it permits a simple solution. When an Eris replica suspects that a client has failed because it has held locks for too long without sending the conclusory `Commit` or `Abort`, the replica can unilaterally abort the general transaction simply by *sending the `Abort` command as an independent transaction itself*, sequenced through the independent transaction layer. This ensures all participant shards reach the same `Commit/Abort` decision, even if the client concurrently attempts to send a `Commit`.

### 7.3 Discussion

Eris builds its general transaction layer atop its core independent transaction primitive. This modularity simplifies the design, particularly for handling client failures. This layered design is practical because Eris is able to commit independent transactions in a single round trip. Such an approach would not be practical in previous systems like Granola, where independent transactions still involve significant coordination overhead. As a result, Granola uses separate, specialized protocols for independent and general transactions, with complicated (and costly) procedures for transitioning between the two [21].

Furthermore, Eris’s use of in-network concurrency control prevents deadlocks, eliminating a large class of concurrency-induced `Aborts` and complex deadlock detection mechanisms: acquiring locks in a single, atomic step executed by a linearizable layer means cycles in the wait-for graph are not possible. Combined with the throughput and latency benefits of the independent transaction processing protocol, this allows Eris to better cope with high contention.

## 8 EVALUATION

We implemented the Eris protocol in approximately 7,500 lines of C++ code. Eris servers were deployed on 9 machines with 2.5 GHz Intel Xeon E5-2680 processors and 64GB of RAM running Ubuntu Linux 16.04. Load was generated using client machines deployed on an additional 10 servers with Xeon L5640 processors. All servers were interconnected using a 10 Gbps Ethernet network that emulates a three level fat-tree topology using three Arista 7050S-64 switches. Multi-sequencing was implemented with a middlebox prototype using a Cavium Octeon CN6880 network processor. All experiments used three replicas per shard (thereby tolerating one replica failure), and fifteen shards (unless otherwise noted).

We evaluated the performance of Eris against three other transactional systems: Granola [21], TAPIR [66] (a Fast Paxos [41]-based protocol), and a standard distributed transaction protocol – similar to Google’s Spanner [19] – that uses two phase commit, two phase locking, and Multi-Paxos (Lock-Store). As a baseline for ideal performance, we also compared against a nontransactional, unreplicated (NT-UR) system that provides neither consistency nor fault tolerance guarantees. It uses a single node per shard with no coordination, replication, or concurrency control; while this system uses fewer servers than Eris, its performance is the maximum expected of any system with the same number of shards. All systems were implemented in the same C++ framework as Eris, and all transactions used stored procedures.

### 8.1 Microbenchmarks with YCSB+T

To examine different aspects of Eris’s performance, we ran all systems against a series of tests using YCSB+T [24], a transactional extension of the popular YCSB key-value store benchmark [18]. YCSB+T wraps key-value store operations inside simple transactions such as read, insert, or read-modify-write. To test distributed transactions across multiple shards, we added multi-key read-modify-write transactions to YCSB+T.

We evaluated the latency, throughput (reported as committed transactions per second), and scalability of Eris using three workloads in the YCSB+T framework. The first was the standard single-shard read/write (SRW) workload which issued single-key reads and writes in a 1:1 ratio. Next, a custom multi-shard read-modify-write (MRMW) workload issued both single-key reads and updates to two randomly selected keys; these updates did not have cross-shard dependencies and were therefore independent transactions. Lastly, we ran a custom cross-shard read-modify-write (CRMW) workload that issued single-key reads and transactionally swapped the values of two random keys, requiring cross-shard updates (and therefore general transactions).

*Latency vs. Throughput.* The SRW workload tests ideal conditions for all systems: minimal contention and no distributed transactions. Figure 6 shows that Eris achieved a maximum throughput of 1.26M transactions/second. This is a  $2.5\times$  and  $4.5\times$  increase over Granola and Lock-Store, which incur Multi-Paxos replication overhead, and  $2.9\times$  higher than TAPIR, which must process additional commit and finalize messages for each transaction. Eris avoids this coordination overhead, and so achieved throughput within 10% of the theoretical maximum implied by the NT-UR system. By requiring only one round trip to commit independent transactions, Eris also achieved latency within 10% of the NT-UR system:  $99\ \mu\text{s}$ , 48–72% lower than the other systems. The throughput gap between Eris and the NT-UR baseline is largely due to the small amount of protocol logic that Eris must execute for

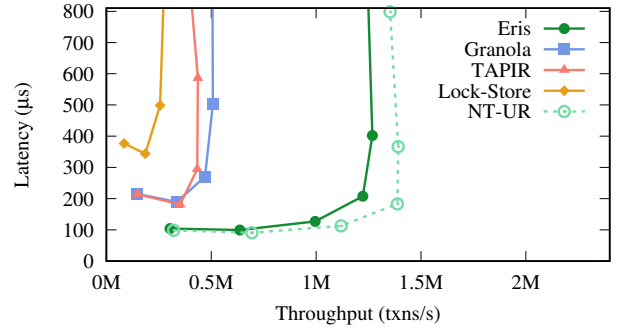


Figure 6: Throughput and latency of the YCSB+T SRW workload with uniform key-access

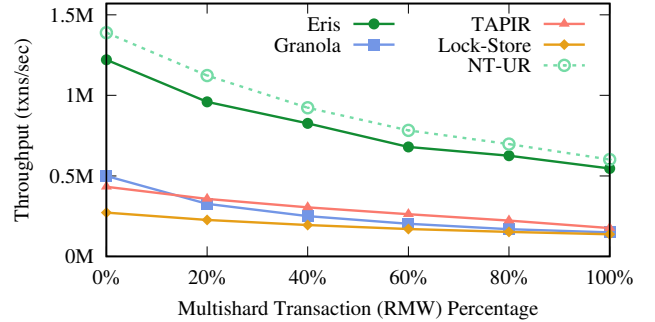


Figure 7: YCSB+T MRMW throughput with an increasing percentage of multi-shard transactions and uniform key-access

every transaction (e.g., multi-stamp parsing and manipulation, out-of-order packets processing and buffering, etc.), while Eris’s higher latency can be attributed to the overhead of our middlebox multi-sequencing implementation.

*Distributed Transactions.* Eris outperformed other systems by a greater margin on distributed transactional workloads. The MRMW experiment shown in Figure 7 gradually increased the percentage of multi-shard RMW independent transactions; contention levels remain low because keys were selected uniformly at random. Because Eris uses in-network concurrency control for coordination-free distributed transactions, it maintained throughput within 10% of the NT-UR system. (NT-UR throughput is also lower for distributed transactions as one two-shard operation is equivalent to two one-shard operations.) For more complex, many-shard transactions, see Section 8.2.

*Contention.* The benefits of Eris’s in-network concurrency control are particularly relevant for high-contention workloads, as Eris processes independent transactions without

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

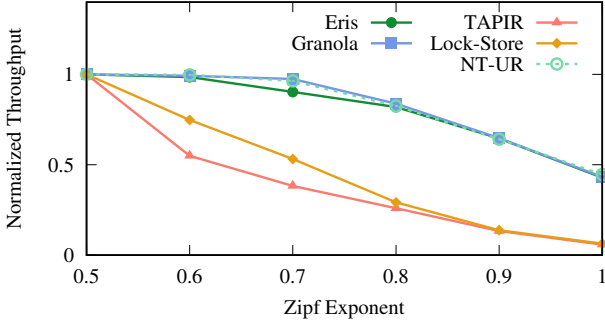


Figure 8: Maximum throughput of the YCSB+T MRMW workload using 20% distributed transactions and Zipf key-access distribution, normalized to throughput at 0.5

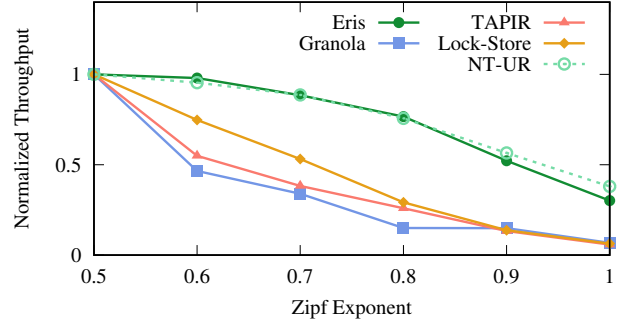


Figure 10: Normalized throughput of the YCSB+T CRMW (generalized transaction) workload using 20% distributed transactions and Zipf key distribution

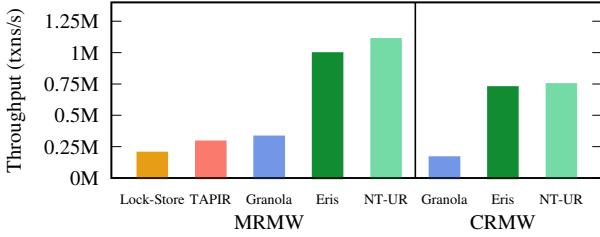


Figure 9: Throughput of the YCSB+T MRMW and CRMW workloads with 20% distributed transactions and Zipf key-access with exponent 0.5. Lock-Store and TAPIR are only shown once; both use the same coordination protocol for MRMW and CRMW and thus have the same performance on the two workloads.

locking or aborts. Figure 8 shows this using the MRMW workload with 20% distributed transactions and an increasingly skewed Zipf key-access distribution. Results are normalized, showing how relative performance is affected by contention. The throughput of TAPIR and Lock-Store fell significantly at high contention rates due to frequent lock conflicts and OCC aborts. Eris retained a throughput close to the NT-UR system in all circumstances. Granola uses timestamps to order independent transactions without locking, and thus also avoids throughput collapse. In absolute terms, Eris outperformed Lock-Store by  $35.0\times$  and TAPIR by  $25.6\times$  on the most skewed workload.

**General Transactions.** To consider workloads that contain non-independent transactions, we compared the MRMW and CRMW workloads, both using 20% distributed transactions. Figure 9 shows that Eris suffers only a modest 28% throughput drop when processing general transactions relative to independent ones. Much of this difference is fundamental to the workload: NT-UR throughput also drops for the CRMW workload because data must be exchanged between shards.

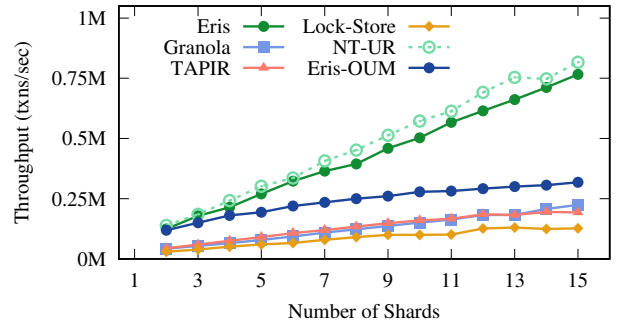


Figure 11: Throughput scalability of the YCSB+T MRMW workload with 20% distributed transactions and 0.5 Zipf exponent

By contrast, Granola’s throughput drops by more than 50% on the CRMW workload because it switches to a less efficient locking mode. This difference becomes extreme under high contention (Figure 10). Eris benefits from fast independent transactions that reduce the contention window and in-network sequencing that enables it to avoid deadlock.

**Scalability.** Eris scales nearly perfectly as the number of shards increases (Figure 11). Much of this benefit is from multi-sequencing, which establishes a consistent partial order of messages. To demonstrate this, we also ran Eris on a globally sequenced network (Eris-OUM), one of the straw-man designs from Section 5. This scheme scales poorly, as every server receives every message involving any shard.

## 8.2 Application-Level Performance: TPC-C

For a more complex workload, we used the well-known TPC-C benchmark, which simulates order processing [61].<sup>3</sup> We

<sup>3</sup>Our results are not intended to be a fully conforming implementation of the TPC-C specification, which imposes many other requirements.

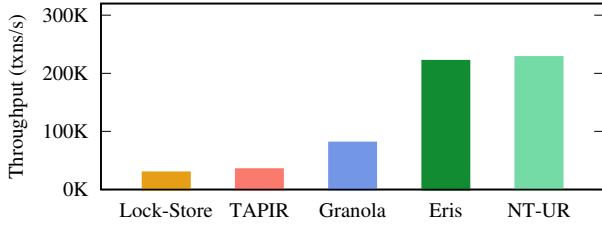


Figure 12: Maximum new-order transaction throughput with 10% distributed transactions on TPC-C workload

used 15 warehouses, with 10% of transactions issued to multiple participants. We report new-order transactions per second, the standard metric for this workload. We adopted the data partitioning scheme from H-Store [56] which allows expressing all TPC-C transactions as independent transactions. For systems that do not support independent transactions, we enabled locking and undo logging. All systems store the entire database in memory and run transactions as stored procedures. As is common, we used closed-loop clients with no wait time.

As Figure 12 demonstrates, Eris achieved a throughput of 221K new order transactions per second. This is  $7.6\times$  and  $6.38\times$  greater throughput than Lock-Store and TAPIR respectively. It is also  $2.75\times$  higher than Granola, even though both are optimized for lock-free independent transactions, because Eris’s protocol avoids the need for timestamp coordination and intra-shard replication. Finally, Eris obtained throughput within 3% of the NT-UR system, which runs TPC-C operations directly (and unsafely) on each shard without replication, coordination, or concurrency control.

### 8.3 Network Resilience

The prior experiments considered a normal-case network. We artificially injected failures to examine Eris’s resilience to poor network conditions.

*Dropped Messages.* Eris relies on in-network sequencing for its high performance, but must invoke the FC when packets are lost. In Figure 13, we randomly dropped an increasing fraction of packets. Even at a high packet drop rate (1%), Eris’s throughput fell only by  $\approx 10\%$ , showing that it avoids the dramatic performance degradation seen in many speculative protocols [54]. Eris replicas immediately detect dropped messages via sequence numbers, and in most cases recover the dropped message from other replicas in the shard, without invoking the FC. At a packet drop rate of 10%, Eris’s throughput degrades more and drops below Granola’s. However, our Eris implementation is designed for normal datacenter network conditions and could be further optimized to handle higher drop rates. The other system significantly affected by

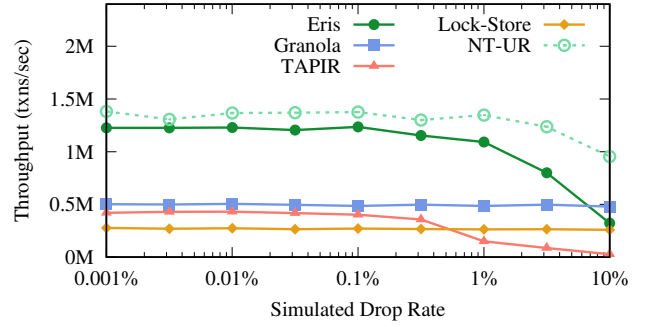


Figure 13: Maximum throughput of the YCSB+T SRW as the simulated packet drop rate increases

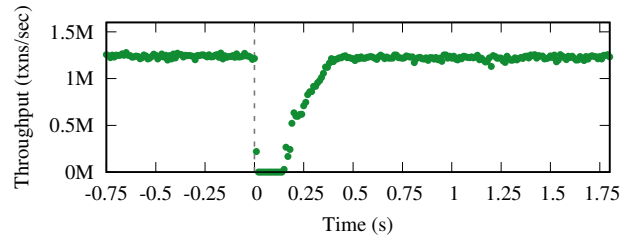


Figure 14: Throughput of the YCSB+T SRW workload during a sequencer failover and epoch change that begins at  $t = 0$

packet loss is TAPIR, which experiences replica state divergence that forces the more expensive consensus slow path.

*Sequencer Failover.* When the network sequencer fails, the network controller must reroute to a new sequencer, and all replicas must coordinate with the FC as part of the epoch change protocol. To evaluate this cost, we triggered a failure in the middle of a YCSB+T SRW workload. Figure 14 shows that Eris resumed normal operation after 130 ms and maximum throughput after 300 ms. Most of the delay is caused by the controller re-establishing network connectivity and could be avoided with a faster rerouting protocol [46].

## 9 RELATED WORK

Eris builds on prior work in co-designing distributed algorithms with network primitives, and transaction processing.

*Network Co-Design.* A promising new direction in distributed systems research takes advantage of the increased capabilities of datacenter network devices. Eris is inspired by Ordered Unreliable Multicast [43]. OUM can efficiently sequence requests in a single replica group; extending this to transactions requires a more sophisticated sequencing mechanism (multi-sequencing), a more complex failure recovery protocol, and the independent transaction concept.



The implementation of multi-sequencing is motivated by recent programmable switch architectures [6, 14, 52, 65]. These have been used to implement a variety of ordering primitives, ranging from the best-effort Mostly-Ordered Multicast [54] to complete in-switch implementations of Paxos [22]. These represent a new take on group communication primitives, a classic distributed systems problem [10, 11].

Many systems have used centralized sequencers to implement group communication primitives [4, 34] and transaction processing systems [9, 62]. These systems provide different ordering and fault-tolerance semantics for the sequencer. Eris’s sequencer design is unique in that it both sequences transactions atomically for multiple destination groups and supports an in-network implementation without persistent in-switch state. vCorfu’s [62] materialized stream abstraction is similar in spirit to Eris’s multi-sequencing. However, multi-sequencing is implemented in-network, and vCorfu itself uses a variant of chain replication that takes at least four round trips to commit a transaction.

Other designs take advantage of other specialized hardware for faster application-level processing. CORFU [4, 5] uses a sequencer to assign an order to operations stored in a log built on clusters of flash drives. Loosely synchronized clocks [44] have been widely used for ordering [1, 19, 21, 27, 66]. FaRM [25, 26] and DrTM [17, 63] employ high-speed RDMA networks, transactional memory, and non-volatile RAM to accelerate distributed transactions. By accelerating network processing, they are able achieve higher levels of throughput than Eris or its baseline system. Integrating these technologies with Eris could offer even higher performance.

*Transaction Algorithms.* There is a vast literature on distributed storage systems with varying levels of transaction support; we do not attempt to detail them all here. Recent systems have explored various points in the design space for transactional partitioned replicated storage systems [2, 47, 67]. Most use a layered architecture with separate coordination mechanisms for cross-shard transactions and in-shard replication. Eris combines both in a single protocol. In this sense, it resembles TAPIR [66] and MDCC [38], which are also unified protocols (though the latter only provides weak isolation).

There is a long history of research on timestamp-ordering concurrency control mechanisms, which ensure serializability by either delaying or rejecting transactions that arrive out of timestamp order [7, 8, 15, 55, 57]. Long thought to be of limited value because of the overhead of tracking read and write timestamps for each data object [15], these techniques have seen renewed interest in response to trends in distributed and main-memory databases that make it more efficient to generate and store timestamps [1, 19, 64]. In particular, Google’s Spanner processes read-only transactions using a multiversion

timestamp-ordering protocol [19]. Eris can be viewed as a coarse-grained application of timestamp ordering, in that it processes transactions sequentially in their multi-stamp order.

Eris’s transaction model is based on independent transactions. Independent transactions were defined as part of the H-Store [33, 36, 56] and Granola [21] projects. Granola provided an application-level protocol for sequencing independent transactions. Although H-Store originally proposed optimizing for independent (or “strongly two-phase”) transactions [56], the proposed protocol was never completed and subsequent work abandoned the idea for a different design [33, 36]. Calvin [58–60] also uses a (different) restricted transaction model, and centralized transaction sequencer, but for a different purpose: so that concurrent transactions will acquire locks in the same order across multi-threaded replicas.

## 10 CONCLUSIONS

The Eris transaction processing system achieves high performance through a new division of responsibility between three parts. An in-network concurrency control primitive, multi-sequenced groupcast, establishes a consistent order of message delivery across shards, but does not ensure atomic or reliable delivery. The latter guarantees are provided by the Eris protocol, which makes sure that transactions are processed by all participant shards, or none at all. In combination, these allow linearizable execution of independent transactions, which make up a substantial part of many workloads. For other workloads, a general transaction layer builds arbitrary transactions out of multiple independent transactions.

The net result of this approach is that Eris can execute independent transactions *without any coordination*: in the normal case, transactions commit in a single round trip from clients to replicas, and servers do not need to coordinate with each other either within or across shards. For independent transactions, Eris achieves  $4.5\text{--}35\times$  higher throughput and  $72\text{--}80\%$  lower latency than standard designs; even for general transactions it provides a  $3.6\times$  performance improvement. In both cases, Eris achieves strongly consistent, fault-tolerant, transactional storage with overhead within 10% compared to a system that provides no such guarantees.

## ACKNOWLEDGMENTS

We thank Naveen Kr. Sharma for insights into programmable switch hardware and Adriana Szekeres for detailed feedback on the Eris protocol. We also thank Irene Zhang, the anonymous reviewers, and our shepherd Douglas B. Terry for their helpful feedback. This material is based upon work supported by the National Science Foundation under awards CNS-1518702 and CNS-1615102 and a Graduate Research Fellowship, and by gifts from Google and VMware.

## REFERENCES

- [1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, June 1995. ACM.
- [2] D. Agrawal, A. E. Abbadi, and K. Salem. A taxonomy of partitioned replicated cloud-based database systems. *IEEE Data Engineering Bulletin*, 38(1):4–9, Mar. 2015.
- [3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR '11)*, Asilomar, CA, USA, Jan. 2011. VLDB / ACM.
- [4] M. Balakrishnan, D. Malkhi, J. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Transactions on Computer Systems*, 31(4), Dec. 2013.
- [5] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, USA, Nov. 2013. ACM.
- [6] Barefoot Networks. Tofino. <https://www.barefootnetworks.com/technology/>.
- [7] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), June 1981.
- [8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery In Database Systems*. Addison-Wesley, Boston, MA, USA, Feb. 1987.
- [9] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A transactional record manager for shared flash. In *Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR '11)*, Asilomar, CA, USA, Jan. 2011. VLDB / ACM.
- [10] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, TX, USA, Oct. 1987. ACM.
- [11] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Jan. 1987.
- [12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.
- [13] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of ACM SIGCOMM 2013*, Hong Kong, China, Aug. 2013. ACM.
- [14] G. Brebner and W. Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, Jan. 2014.
- [15] M. J. Carey and M. R. Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases (VLDB '84)*, Singapore, Aug. 1984.
- [16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, Nov. 2006. USENIX.
- [17] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the 11th ACM SIGOPS EuroSys (EuroSys '16)*, London, United Kingdom, Apr. 2016. ACM.
- [18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, IN, USA, 2010. ACM.
- [19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, USA, Oct. 2012. USENIX.
- [20] J. Cowling. *Low-Overhead Distributed Transaction Coordination*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2012.
- [21] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012. USENIX.
- [22] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*, pages 5:1–5:7, New York, NY, USA, 2015. ACM.
- [23] J. Dean and L. A. Barosso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.
- [24] A. Dey, A. Fekete, R. Nambiar, and U. Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *Proceedings of the 30th IEEE International Conference on Data Engineering Workshops (ICDEW '14)*, Chicago, IL, USA, 2014. IEEE.
- [25] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, Apr. 2014. USENIX.
- [26] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, Oct. 2015. ACM.
- [27] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS '13)*, Braga, Portugal, Oct. 2013. IEEE.
- [28] D. G. Ferro and M. Yabandeh. A critique of snapshot isolation. In *Proceedings of the 7th ACM SIGOPS EuroSys (EuroSys '12)*, Bern, Switzerland, Apr. 2012. ACM.
- [29] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, Oct. 2011. ACM.
- [30] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, BC, Canada, June 2008. ACM.
- [31] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [32] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of ACM SIGCOMM 2013*, Hong Kong, China, Aug. 2013. ACM.



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

- [33] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Indianapolis, IN, USA, June 2010. ACM.
- [34] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems (ICDCS '91)*, Arlington, TX, USA, 1991. IEEE.
- [35] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC '16)*, Denver, CO, July 2016. USENIX.
- [36] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.
- [37] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, Oct. 2010. USENIX.
- [38] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Proceedings of the 8th ACM SIGOPS EuroSys (EuroSys '13)*, Prague, Czech Republic, Apr. 2013. ACM.
- [39] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
- [40] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.
- [41] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.
- [42] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control [extended version]. Technical Report UW-CSE-17-10-01, University of Washington CSE, Seattle, WA, USA, Oct. 2017.
- [43] J. Li, E. Michael, A. Szekeres, N. K. Sharma, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, Nov. 2016. USENIX.
- [44] B. Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC '91)*, Montreal, QC, Canada, Aug. 1991. ACM.
- [45] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.
- [46] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, Apr. 2013. USENIX.
- [47] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, July 2013.
- [48] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, Nov. 2016. USENIX.
- [49] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, Apr. 2013. USENIX.
- [50] NOX network control platform. The POX SDN controller. <https://github.com/noxrepo/pox>.
- [51] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC '88)*, Toronto, Ontario, Canada, Aug. 1988. ACM.
- [52] R. Ozdag. Intel® Ethernet switch FM6000 series-software defined networking.
- [53] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, Oct. 2010. USENIX.
- [54] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data-center networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, USA, May 2015. USENIX.
- [55] D. P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, Sept. 1978.
- [56] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, Vienna, Austria, Sept. 2007.
- [57] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.
- [58] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(10), 2010.
- [59] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, Scottsdale, AZ, USA, May 2012. ACM.
- [60] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast distributed transactions and strongly consistent replication for OLTP database systems. *ACM Transactions on Database Systems*, 39(2), May 2014.
- [61] Transaction Processing Performance Council. TPC Benchmark C. <http://www.tpc.org/tpcc/>, Feb. 2010.
- [62] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritch, S. Swanson, M. J. Freedman, and D. Malkhi. vCorfu: A cloud-scale object store on a shared log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, Mar. 2017. USENIX.
- [63] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, Oct. 2015. ACM.
- [64] S. Wolf, H. Mühe, A. Kemper, and T. Neumann. An evaluation of strict timestamp ordering concurrency control for main-memory database systems. In *IMDM Workshop*, 2013.
- [65] XPliant Ethernet switch product family. [www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html](http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html).
- [66] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, Oct. 2015. ACM.
- [67] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. When is operation ordering required in replicated transactional storage? *IEEE Data Engineering Bulletin*, 39(1):27–38, Mar. 2016.