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

[Extended Version]*

Jialin Li Ellis Michael Dan R. K. Ports
University of Washington
{lijl, emichael, drkp}@cs.washington.edu

UW Technical Report UW-CSE-17-10-01

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 3.6–35× higher and latency 72–80% lower than a conventional design on standard benchmarks.

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,[1] 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 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

*This document is an extended version of the paper by the same title that appeared in SOSP 2017 [42]. A summary of the additional content is provided in Section 1.1.

[1] Eris takes its name from the ancient Greek goddess of discord, i.e., *lack of coordination*.

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% 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.

1.1 Additional Technical Report Content

This technical report contains the following additional material over our SOSP submission:

- The full details of the independent transaction processing layer's view change protocol (Section 6.4), epoch change protocol (Section 6.5), and synchronization protocol (Section 6.6).
- A proof of the independent transaction processing layer's correctness (Section 6.7).
- A TLA+ specification of the Eris protocol (Appendix A).
- A P4 implementation of the Eris multi-sequencer (Appendix B).

Additional materials are highlighted with a gray bar in the margin.

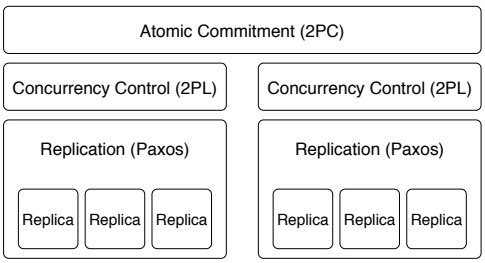

Figure 1: Standard layered architecture for a partitioned, replicated storage system

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 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,

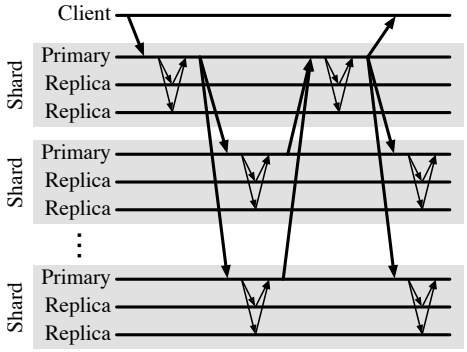

Figure 2: Coordination required to commit a single transaction with traditional two-phase commit and replication

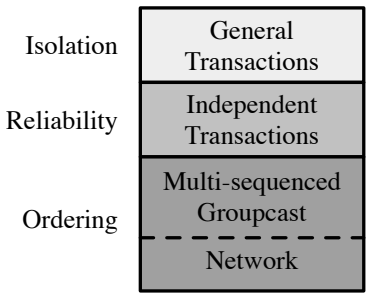

Figure 3: The layers of Eris and the guarantees they provide

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.

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 predefined 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 $\langle$*group-id*, *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 in Appendix B. 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

	Throughput (packets/second)	Latency (μs)
Middlebox	6.19M (σ = 3.16K)	13.64 (σ = 0.42)
Endhost	1.61M (σ = 19.98K)	24.60 (σ = 1.02)

Table 1: Performance of endhost and middlebox sequencers

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 μs per transaction), and system throughput may be limited 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[2] (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 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* mod *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. 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].

6.2 Normal Case

In the normal case, clients submit independent transactions via the multi-sequencing layer, and each replica that receives

[2]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.

Replica:

- *replica-id* = ⟨*shard-num*, *replica-num*⟩
- *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 ⟨*epoch-num*, *shard-num*, *sequence-num*⟩, 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

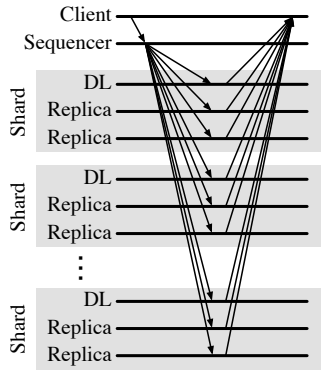

Figure 5: Communication pattern of Eris in the normal case, where the independent transaction is sent via multi-sequenced groupcast to multiple shards, each consisting of 3 replicas (one replica in each shard is a Designated Learner)

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 ⟨REPLY, *txn-index*, *view-num*, *result*⟩, 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.

In the normal case, transactions are received from the multi-sequencing layer and added to the log via this protocol. However, replicas can also learn about new transactions or NO-OPs through the dropped message, view or epoch change, or synchronization protocols below. If this causes them to add a new transaction to the log before it is received from the multi-sequencing layer, the replica ignores the corresponding message or DROP-NOTIFICATION when it is later received.

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 ⟨FIND-TXN, *txn-id*⟩ 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 ⟨TXN-REQUEST, *txn-id*⟩ to *all replicas in all shards*. If the FC already found or dropped the transaction, it replies with ⟨TXN-FOUND, *txn*⟩ or ⟨TXN-DROPPED, *txn-id*⟩, respectively.
3. When a replica receives TXN-REQUEST, if it has received a transaction matching *txn-id*, it replies with ⟨HAS-TXN, *txn*⟩. Otherwise, it adds *txn-id* to *temp-drops* and replies with ⟨TEMP-DROPPED-TXN, *view-num*, *txn-id*⟩.

   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 ⟨TXN-FOUND, *txn*⟩ 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 ⟨TXN-DROPPED, *txn-id*⟩ 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].

1. When a replica suspects the DL to have failed, it changes its *status* to `ViewChange` and increments its *view-num*. While in the `ViewChange` state, it does not accept any messages except view change and epoch change messages. It then sends ⟨VIEW-CHANGE, *view-num*, *log*, *temp-drops*, *perm-drops*, *un-drops*⟩ to the DL for the new view. It also sends ⟨VIEW-CHANGE-REQ, *view-num*⟩ to the other replicas.

2. When a replica receives a VIEW-CHANGE-REQ for a *view-num* greater than its own, it updates its *view-num*, sets its *status* to `ViewChange`, and sends the VIEW-CHANGE as above.

3. When the DL for the new view receives VIEW-CHANGE messages from a majority of replicas (including itself), it updates its *view-num*, and then merges its own *log*, *temp-drops*, *perm-drops*, and *un-drops* with those it received in the VIEW-CHANGE messages. The merging operation for *temp-drops*, *perm-drops*, and *un-drops* is a simple set union.

   The merged *log* is computed by taking the longest log received, and replacing any slots which have matching *txn-id*s in *perm-drops* with NO-OPs.

   If, at this point, the new *log* has any transactions matching *txn-id*s in *temp-drops* without corresponding *txn-id*s in *un-drops*, the DL must wait for the FC to come to a decision about those *txn-id*s, retrying—asking the FC and sending any HAS-TXNs—if necessary.

   It then sets its *status* to `Normal` and sends ⟨START-VIEW, *view-num*, *log*, *temp-drops*, *perm-drops*, *un-drops*⟩ to the other replicas in the shard.

4. The non-DL replicas, upon receiving a START-VIEW for a view greater their own (or equal to their own if their *status* is `ViewChange`), adopt the new *log*, *view-num*, *temp-drops*, *perm-drops*, and *un-drops* and set their *status* to `Normal` as well.

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.

Note that several messages in the view change protocol, as well as the epoch change protocol below, are presented as containing replicas' full *log*s and other state. This is only for simplicity of exposition. To avoid transferring considerable amounts of state, in a real deployment these messages would contain only metadata, and the recipients can then pull any missing data from the sender – a standard optimization.

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.

1. Whenever a replica receives a NEW-EPOCH notification from the network layer (indicating a sequencer failover) it sends ⟨EPOCH-CHANGE-REQ, *epoch-num*⟩ to the FC.

2. Whenever the FC receives a EPOCH-CHANGE-REQ with an *epoch-num* greater than its own, the FC sets its *epoch-num* to the new value and sends out ⟨EPOCH-CHANGE, *epoch-num*⟩ to all replicas.

3. When a replica receives a EPOCH-CHANGE for a later epoch, it updates its *epoch-num* and sets its *status* to `EpochChange`. While in this state, it does not accept any messages except epoch change messages. It then sends ⟨EPOCH-CHANGE-ACK, *epoch-num*, *last-norm-epoch*, *view-num*, *log*⟩ back to the FC, where *last-norm-epoch* was the last epoch in which the replica had *status* `Normal`.

4. When the FC receives EPOCH-CHANGE-ACK messages from a simple majority of replicas from all shards, it first merges all logs from all shards to create a combined log. From each shard, the FC only uses *log*s where the associated *last-norm-epoch* is the latest epoch the FC started, if they exist; otherwise, it uses the *log* from the saved START-EPOCH message for that shard (see below).

   The FC then determines if there are any gaps for any shard in the combined log and decides that the missing transactions should be permanently dropped.

   When this is done, for each shard it sends out ⟨START-EPOCH, *epoch-num*, *new-view-num*, *log*⟩ to all of the shard's replicas, where *new-view-num* is the highest *view-num* it received from any replica in that shard and *log* contains all of that shard's transactions from the combined log (with NO-OPs for any transactions the FC previously dropped).

   The FC saves these START-EPOCH messages until a majority of replicas from each shard acknowledge the new epoch, in case it needs to resend them or use them for subsequent epoch changes.

5. When a replica receives a START-EPOCH for an epoch higher than its own (or equal to its own if its *status* is `EpochChange`), it adopts the new *epoch-num*, *view-num*, and *log* and clears its *temp-drops*, *perm-drops*, and *un-drops*. It then sets its *status* to `Normal`, executes any new transactions in its *log*, and begins listening for multi-sequenced groupcast messages in the new epoch.

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.

1. The DL sends ⟨SYNC-PREPARE, *view-num*, *log*, *perm-drops*, *un-drops*⟩ to the other replicas.

2. When a replica receives a SYNC-PREPARE from the DL with a *view-num* matching its own, it first sets its *perm-drops* and *un-drops*, respectively, to be the union of its own and the DL's.

   The replica then merges the DL's *log* into its own, adding any new transactions and NO-OPs, replacing transactions with NO-OPs as necessary (i.e., replacing those transactions now matching entries in *perm-drops*).

   Next, the replica discards any entries in its *temp-drops* matching transactions in the DL's *log*.[3]

   Finally, the replica replies to the DL with ⟨SYNC-REPLY, *view-num*, *syncpoint*⟩ where *syncpoint* is the index of the latest entry in the *log* the DL sent.

3. After receiving SYNC-REPLYs with *syncpoint* corresponding to its previously sent SYNC-PREPARE and *view-num* matching its own from a majority of the replicas in its shard (when counting itself towards that majority), the DL broadcasts ⟨SYNC-COMMIT, *view-num*, *syncpoint*⟩.

4. When a replica receives a SYNC-COMMIT with a *view-num* matching its own, if it previously received the corresponding SYNC-PREPARE from the DL, it can safely execute the transactions in its *log* up to *syncpoint*.

6.7 Correctness

As we prove below, Eris guarantees linearizable execution of independent transactions. Additionally, Appendix A contains a TLA+ specification of the Eris independent transaction processing protocol which was model-checked against the high-level invariants in the proof.

6.7.1 Linearizability of Independent Transactions

First, we need to introduce terms to describe the two types of "promises" shards in Eris make.

Definition (Committed Transactions and Drop Promises)**.** A shard *commits* transaction *t* in slot *s* if a majority of its replicas send REPLYs for *t* with matching *view-num*s, *epoch-num*s, and *txn-index*s, where one of the REPLYs came from the DL of the view. A shard commits a *drop promise* for transaction *t* if a majority of its replicas send TEMP-DROPPED-TXNs for a *txn-id* corresponding to *t* with matching *view-num*s, where one of the TEMP-DROPPED-TXNs came from the DL of the view.

Next, we introduce the notion of *log stability*, which describes logs which have attained a state of "permanence."

Definition (Log Stability)**.** A replica's *log* (consisting of transactions and NO-OPs) is *stable* if the transactions in its *log* are a prefix of the transactions in the *log*s of all replicas in the same shard in later views or epochs which have status `Normal`.

Finally, we introduce the notation used throughout the proof.

Definition. Let $DL(s, v)$ denote the DL of shard s for view v.

Definition. Let $P(t)$ denote the set of participant shards for transaction t (where each shard is itself a set of replicas).

[3] Discarding these entries in *temp-drops* is safe because the replica knows the FC will never receive a TEMP-DROPPED-TXN from the DL in this view and thus will never use the replica's previously sent TEMP-DROPPED-TXN to decide to drop that transaction.

We begin the proof of linearizability with a few basic facts about the Eris protocol, which follow directly from the details of the protocol itself.

Lemma 1. *Replicas never decrement their view-num or epoch-num once the view or epoch has started (i.e., once the DL or FC decides to start the view or epoch).*

Lemma 2. *The FC never sends a* TXN-FOUND *for a transaction it previously dropped by sending a* TXN-DROPPED *for a matching txn-id, and vice-versa.*

Lemma 3. *If replica r has status* `Normal` *and has transaction t in its log with a matching txn-id in its temp-drops, then $t \in$ r.un-drops.*

Lemma 4. *If the FC sends a* TXN-DROPPED *for txn-id τ, all shards have committed a drop promise for τ.*

Lemma 5. *Replicas in the same shard which start an epoch (by receiving the corresponding* START-EPOCH *message and setting their status to* `Normal`*) do so with the exact same log.*

Next, we will show that an inductive property, I, is invariant over any execution of the independent transaction processing protocol, and that I itself implies linearizability. I is the conjunction of the following sub-properties:

I_1: Replicas' *log*s are always in strict multi-stamp order. The transactions in replicas' *log*s are in multi-stamp order, and during each epoch, for each position in the *log* following the *log* the replica began the epoch with, that position either contains a NO-OP or the corresponding transaction in that epoch for the replica's shard.

I_2: If replica r from shard s in epoch e has transaction t from epoch e in its *log* but does not have transaction $t' \prec t$ also from epoch e for shard s in its *log*, then $\exists \tau \in r.perm\text{-}drops$ such that τ matches t.

I_3: If transaction t was committed by shard s, then $\forall s' \in P(t)$, if s' has committed $t' \succ t$, then s' has committed t.

I_4: If a drop promise for *txn-id* τ was committed at shard s in view v, epoch e, then $\forall r \in s$:

$$\begin{aligned}(&r.epoch\text{-}num = e \wedge (r.view\text{-}num > v \\ &\qquad \vee (r.view\text{-}num = v \wedge DL(s,v) = r))) \\ &\implies \tau \in r.temp\text{-}drops\end{aligned}$$

I_5: During a single view and epoch, while a majority of replicas in a shard still recognize that view, the DL's log only grows; entries are not overwritten.

I_6: If transaction t is committed in or present at log index l at a view-consistent majority of replicas in shard s, all in view v and epoch e, then $\forall r \in s$:

$$\begin{aligned}&(r.view\text{-}num > v \vee r.epoch\text{-}num > e \\ &\vee (r.view\text{-}num = v \wedge r.epoch\text{-}num = e \wedge DL(s,v) = r)) \\ &\implies r.log[l] = t\end{aligned}$$

I_7: When the FC starts an epoch, it will not start any shard with a *log* containing transactions matching *txn-id*s it previously dropped.

I_8: The *log*s replicas start epochs with are stable.

Before we prove that I is invariant, we will show that I implies linearizability.

Lemma 6 (Stability of Learners' Logs)**.** *If transaction t is committed at shard s in view v, epoch e, then $DL(s,v).log$ was stable at the time it sent the corresponding* REPLY *for t.*

Proof. Consider the transactions in $DL(s,v).log$ when it sent the REPLY for t. These transactions either were or were not part of the *log* that began epoch e for shard s. Those that were, by I_8, will be in the *log*'s of all replicas in s starting later views and epochs. For transactions committed during epoch e, by I_1, every replica that sent one of the REPLYs for t must have had either the corresponding transaction or a NO-OP in its *log*. However, for a replica to have a NO-OP in its *log*, the FC must have received a drop promise for t from shard s; Lemma 1, I_4, Lemma 3, and Lemma 2 then lead to a contradiction. Therefore, all transactions in $DL(s,v).log$ when it sent the REPLY for t were in the *log*s of the other replicas that helped commit t, when they sent their REPLYs.

The FC cannot later receive a drop promise from s and drop any of these previous transactions in view v, since these transactions are in $DL(s,v).log$. Therefore, the next view that starts at a majority must start with these transactions in the same indices in replicas' *log*s, so by I_6 these transactions will be present in the *log*s of replicas in *all* later views or epochs.

I_1 and I_8 guarantee that no replica in a later epoch will have transactions from epochs prior to e not in $DL(s,v).log$ when it sent the reply to t.

I_2 implies that for all $t' \prec t$ where t' is some other transaction from epoch e with shard s as a participant *not* in $DL(s,v).log$ when it replied to t, there exists $\tau \in DL(s,v).perm\text{-}drops$ such that τ matches transaction t', indicating that the FC dropped these transactions. Shard s must have committed drop promises for these *txn-id*s for the FC to have dropped them. Therefore, any replica starting a later view in epoch e will have those *txn-id*s in its *temp-drops* by I_4. Lemma 3 guarantees that in order for such a replica to be available for processing new transactions and have any of these previously dropped transactions in its *log*, it must

have received a TXN-FOUND for those transactions, which Lemma 2 guarantees never happens.

Finally, any replica starting a later epoch will do so with a *log* sent by the FC, which by I_7 will not contain any transactions dropped in epoch e.

Therefore, any replica in a later view or epoch with status `Normal` will have in its *log* all of the transactions in the DL's *log* at the time it sent the corresponding REPLY for t and will not have any transactions $t' \prec t$ not in the DL's *log*, and all of the transactions will be in the same order as in the DL's *log* by I_1. □

Theorem 1. *Eris guarantees linearizable execution of independent transactions.*

Proof. Lemma 6 and I_5 imply that if two clients both receive view-consistent replies from the same shard for transactions t, t', where $t \succ t'$, then the DL of the view-consistent reply for t' previously executed t—with the exact same state that the DL of the view-consistent reply for t had. Since, by Lemma 1, shards move through successive views and epochs and never return to previous views and epochs, the behavior of a single shard is indistinguishable from a single, correct process from the point of view of the clients.

Furthermore, I_3 implies that for any execution of Eris, the execution of transactions by shards happens in an order which respects the order assigned by the multi-sequenced groupcast layer and that if one shard commits a transaction, all of the other participants of the transaction will commit that transaction before committing any later transactions. Transactions are partially ordered by the multi-sequenced groupcast layer, and that order guarantees that any potentially conflicting transactions are comparable. Therefore, any execution of transactions by Eris shards which respects this order will be free of conflict cycles and thus serializable. Because the multi-sequenced groupcast order will respect the real-time ordering of transactions as received by successive sequencers, execution of independent transactions in Eris is also linearizable. □

Now, we show by induction that I holds throughout any execution of Eris.

Theorem 2. *I is invariant throughout any execution of Eris.*

Proof. First, we note that all sub-properties of I trivially hold in the initial state of the system. Next, we will show that each sub-property of I holds for any given step in a distributed execution assuming that all properties held for previous steps.

I_1: A replica only inserts transactions or NO-OPs into its *log* when it receives a transaction from the multi-sequenced groupcast layer or when it receives a TXN-FOUND or a TXN-DROPPED from the FC (having been waiting for that transaction because it received a DROP-NOTIFICATION). In both cases, the replica is inserting into its *log* in the exact order prescribed by the multi-sequenced groupcast layer.

A replica can replace a transaction in its *log* if it receives TXN-DROPPED from the FC, but the invariant still holds.

The only other times a replica's *log* changes are when it starts a view or epoch. When it starts a view, the *log* it starts the view with was previously the *log* of another replica (potentially with some transactions replaced by NO-OPs), so by induction the property holds. The FC ensures that replicas start epochs with *log*s in transaction order.

I_2: The only step a replica could take to invalidate this invariant is adding transaction t to its log without having previously added t' and without a *txn-id* matching t' in *perm-drops*. By I_1, the replica must have a NO-OP in place of t' in its *log*. A NO-OP is only added to a *log* during an epoch when the FC sends a TXN-DROPPED, but the recipient of a TXN-DROPPED also adds the *txn-id* to its *perm-drops*, contradicting the assumption that there is no matching *txn-id* for t' in *perm-drops*.

I_3: There are two steps which could invalidate this invariant. Either shard s' commits t' having not committed t after s committed t, or shard s commits t after s' committed t' having not committed t.

If t and t' are in different epochs, this would mean that s' started the new epoch (i.e., t''s epoch) having not committed t, even though s committed t in the old epoch. This cannot happen because the FC found all committed transactions before starting the new epoch (since it read from a majority from each shard, and by I_6 one *log* it received contained t) and ensured that the *log*s it used to start the epoch were consistent.

If t and t' are in the same epoch, by I_1 and I_2 we know that some replica in shard s' at one point had a *txn-id* matching t in its *perm-drops*, implying the FC must have sent a TXN-DROPPED for that *txn-id*. Lemma 4 implies that shard s must have committed a drop promise for a *txn-id* matching t. That promise could not have happened after s committed t since t would have been in the DL's *log* by I_6, forcing it to send a HAS-TXN instead. However, I_4 then implies that the DL of s replying to t when s committed t must have had a *txn-id* matching t in its *temp-drops*, and Lemma 3 then implies that the FC must have sent a TXN-FOUND for t, contradicting Lemma 2.

I_4: The only step that could invalidate this invariant is a view change, but because replicas never return to earlier views by Lemma 1, by induction one of the VIEW-CHANGE messages must have contained the *txn-id* in *temp-drops*, so the new DL would start the new view with the *txn-id* in *temp-drops*.

I_5: In order for an *log* entry to be overwritten, the FC must send a TXN-DROPPED message with a corresponding *txn-id*, but once the FC starts an epoch, it no longer handles transactions from older epochs, and replicas ignore

messages from older epochs. Furthermore, before the FC drops a transaction, it secures a drop promise from all shards. While a DL is recognized by a majority, these drop promises cannot be sent by later views. I_4 then guarantees that the DL of the current active view has a record of all drop promises made in the current epoch in its *temp-drops*. By Lemma 3, any transactions in the DL's *log* matching *txn-id*s in its *temp-drops* must also be in its *un-drops*, implying that the FC sent a TXN-FOUND for those transactions. Lemma 2 guarantees that the FC will never drop these transactions.

I_6: The FC could not have decided to drop t before it was committed or present at a view-consistent majority, since it would have needed a drop promise from s, and I_4, Lemma 3, and Lemma 2 then lead to a contradiction. Therefore, the FC cannot have yet decided to drop the transaction, since by induction it could not have gotten a drop promise from shard s after t was committed or present at a view-consistent majority (because the DLs of successive views have t in their *log*s, and Lemma 1 guarantees that a majority of replicas cannot return to an earlier view).

Therefore, by I_1, the only steps that could invalidate this invariant are view change and epoch change.

The transaction would be "visible" to any new DL starting a view (having received the *log*s of a majority, one of which guaranteed by Lemma 1 and induction to have t to have t in the log index l), and since the transaction couldn't have been dropped, the new view that starts starts with t in log index l.

Furthermore, the FC only starts a new epoch after receiving acknowledgments from a majority of replicas from shard s. If it does not use the *log*s from these replicas, the saved START-EPOCH message that it does use must have been used to start an epoch greater than e (since at least one acknowledgment must have come from a replica which had *status* `Normal` during epoch e because a majority of replicas entered epoch e). By induction, that *log* will have t in log index l. In either case, the new epoch will start with t in log index l.

I_7: When the FC starts an epoch, for each shard, it either uses *log*s from the most previous epoch that it started or it uses the *log* from its saved START-EPOCH message, both of which can't have previously dropped transactions from epochs prior to the most recent one by induction. Furthermore, the FC ensures that dropped transactions from the most previous epoch do not appear in the *log*s it forms to start the new epoch.

I_8: Once a replica begins an epoch, it ignores transactions from earlier epochs. In fact it ignores all messages from previous epochs so cannot receive a message from the FC dropping a transaction in the *log* it began the epoch with (since the FC only drops transactions from its current epoch).

Furthermore, during any view change, the new DL will receive *log*s with all transactions from previous epochs by induction and Lemma 5.

Finally, during an epoch change, the FC either uses the *log*s from the most recent epoch or from its saved START-EPOCH message, both of which contain all of the transactions from the *log* that began the epoch in question and no earlier transactions not in that *log* by induction.

This completes the proof that I is invariant. □

6.7.2 *Liveness During Stable Periods*

Guaranteeing the safety of the Eris independent transaction processing protocol is critically important, but a proof of safety is useless without some guarantees about the liveness of the system. While it is obvious that Eris is a non-trivial protocol and has the potential to make progress and commit transactions from the initial state of the system, we would also like to show that it is impossible to reach a state of *deadlock*. Specifically, we would like to show that no matter what state Eris has reached in a specific execution, as long as the following conditions hold for a sufficient amount of time Eris can commit new transactions:

- No more than a minority of replicas in any shard has failed.
- There is a single, active sequencer which has not failed.
- Messages are not dropped nor reordered in the network.
- There is a bound on message delays and the differences in processor speeds (i.e., the systems behaves synchronously).
- The FC remains active and available.

These conditions are by no means tight; Eris could still make progress even when some of these conditions are not attained. However, if progress is guaranteed under these conditions, it provides us with some assurance that deadlock is not reachable.

The progress argument is rather straightforward. First, we know that as long as client requests are sent, the epoch corresponding to the single, active sequencer will eventually be detected, and a replica will eventually forward the EPOCH-CHANGE-REQ message to the FC. Since the FC need only contact a simple majority of replicas from each shard to start a new epoch, we know that it will eventually start the new epoch.

Next, since a majority of replicas in each shard will not fail for a sufficient period of time, the number of view changes that can begin after this period of stability starts is bounded. Once a "correct" process is proposed as the DL of a shard

(and all other "correct" processes receive the VIEW-CHANGE-REQ and then vote to begin the view), no process will try to start a new view since the DL will not fail. This assumes that processes have access to a perfect failure detector (i.e., that processes "know" the bound on message delay and processor speed differentials); however, the process of arriving at stable views in each shard can be adapted to work with eventually perfect failure detectors. In that case, there is a bound on the number of times a process may erroneously suspect a non-failed process of having failed.

Then, since there will eventually be stable views in each shard and a single, stable epoch, the FC will eventually be able to resolve any outstanding FIND-TXNs, since doing so only requires contacting a view-consistent majority of replicas from each shard, all in the same epoch as the FC.

Finally, when all outstanding FIND-TXNs have been resolved, any independent transaction that is then sent to Eris will be sequenced and sent to all participant shards in order. A view-consistent majority of replicas in these shards will receive the transaction (instead of a DROP-NOTIFICATION), and be able to process it and reply since they don't have any outstanding dropped transactions being resolved by the FC. Therefore, the transaction will be committed by all participant shards, and the client library will return a result.

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 `Abort`s 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× and 4.5× increase over Granola and Lock-Store, which incur Multi-Paxos replication overhead, and 2.9× 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 μs, 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 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.

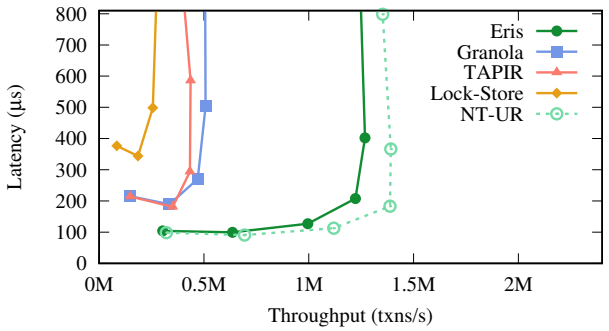

Figure 6: Throughput and latency of the YCSB+T SRW workload with uniform key-access

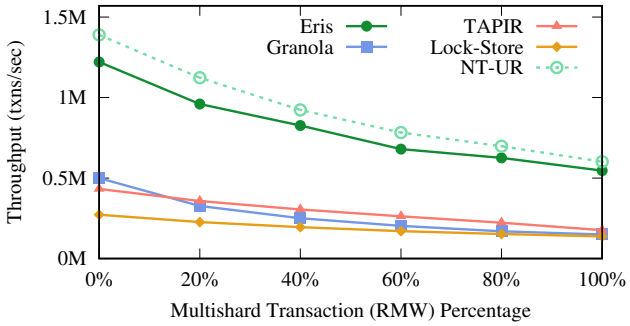

Figure 7: YCSB+T MRMW throughput with an increasing percentage of multi-shard transactions and uniform key-access

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 trans-

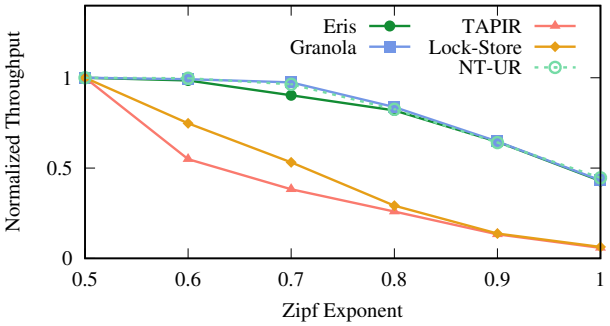

Figure 8: Maximum throughput of the YCSB+T MRMW workload using 20% distributed transactions and Zipf key-access distribution, normalized to throughput at 0.5

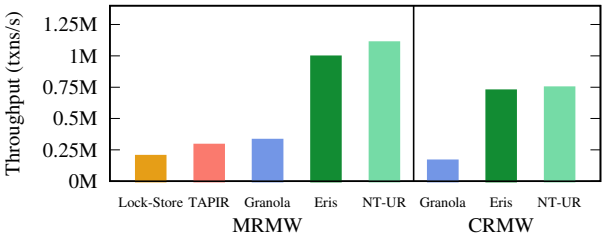

Figure 9: Throughput of the YCSB+T MRMW and CRMW workloads with 20% distributed transactions and Zipf key-access with exponent 0.5. Lock-Store and TAPIR are only shown once; both use the same coordination protocol for MRMW and CRMW and thus have the same performance on the two workloads.

actions, 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 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× and TAPIR by 25.6× 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.

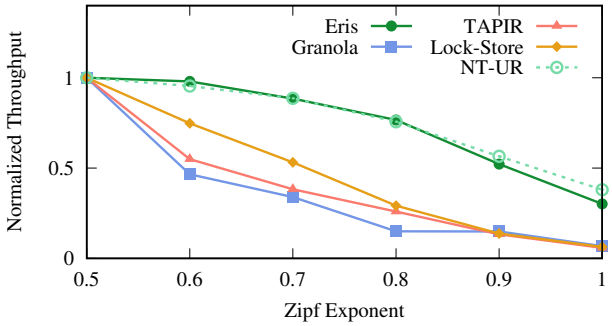

Figure 10: Normalized throughput of the YCSB+T CRMW (generalized transaction) workload using 20% distributed transactions and Zipf key distribution

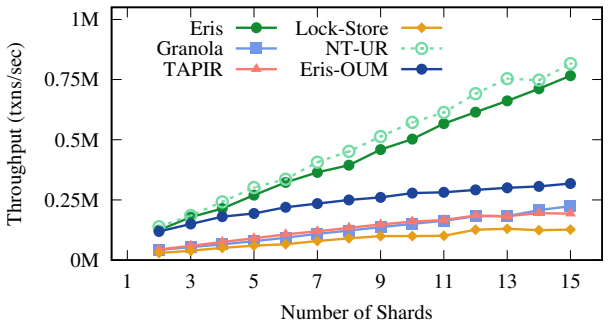

Figure 11: Throughput scalability of the YCSB+T MRMW workload with 20% distributed transactions and 0.5 Zipf exponent

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. 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.

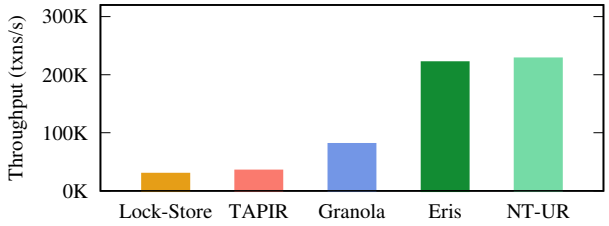

Figure 12: Maximum new-order transaction throughput with 10% distributed transactions on TPC-C workload

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].[4] We 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× and 6.38× greater throughput than Lock-Store and TAPIR respectively. It is also 2.75× 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 ≈ 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 packet loss is TAPIR, which experiences replica state divergence that forces the more expensive consensus slow path.

[4] Our results are not intended to be a fully conforming implementation of the TPC-C specification, which imposes many other requirements.

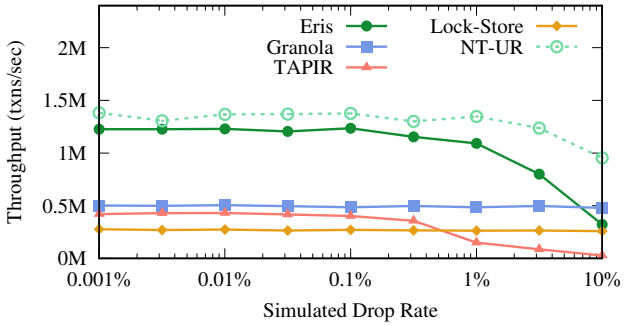

Figure 13: Maximum throughput of the YCSB+T SRW as the simulated packet drop rate increases

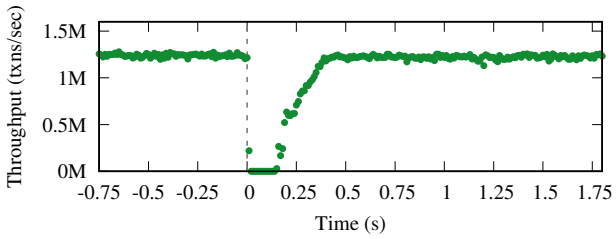

Figure 14: Throughput of the YCSB+T SRW workload during a sequencer failover and epoch change that begins at $t = 0$

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 4.5–35× higher throughput and 72–80% lower latency than standard designs; even for general transactions it provides a 3.6× 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. Linearizabiliy: 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.

[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. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Beijing, China, Oct. 2017. ACM.

[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 datacenter 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. Fritchie, 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`.

[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.

A Eris Specification

MODULE *Eris*

Specifies the *Eris* protocol.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

Constants

It's not strictly necessary that all *shards* have the same number of replicas

CONSTANTS *NumShards*, *NumReplicasPerShard*
ASSUME $NumShards \in Nat \land NumReplicasPerShard \in Nat$
$Shards \triangleq (1 \,..\, NumShards)$
$Replicas \triangleq (1 \,..\, (NumShards * NumReplicasPerShard))$

Message sequencers

CONSTANT *NumSequencers* Normally infinite, assumed finite for model checking
$Sequencers \triangleq (1 \,..\, NumSequencers)$

CONSTANT *NoOp*

Replica Statuses

CONSTANTS *StNormal*, *StViewChange*, *StEpochChange*

Message Types

CONSTANTS *MClientRequest*,
MStampedClientRequest,
MRequestReply,
MFindTxn,
MTxnRequest,
MHasTxn,
MTempDroppedTxn,
MTxnFound,
MTxnDropped,
MViewChange,
MViewChangeReq,
MStartView,
MEpochChangeReq,
MEpochChange,
MEpochChangeAck,
MStartEpoch

Message Schemas

ClientRequest (Client to Sequencer)
[*mtype* $\mapsto$ *MClientRequest*,
shards $\mapsto S \in$ SUBSET *Shards*]

MStampedClientRequest (Sequencer to *Replicas*)
[*mtype* $\mapsto$ *MStampedClientRequest*,
shards $\mapsto S \in$ SUBSET *Shards*,

$stamp \mapsto [\ s \in S \mapsto (1\ ..\ )\ ],$
$epochNum \mapsto e \in (1\ ..\ )\ ]$

RequestReply (Replicas to Client)
$[\ mtype \mapsto MRequestReply,$
$sender \mapsto r \in Replicas,$
$txnIndex \mapsto i \in (1\ ..\ )$
$request \mapsto v \in [MStampedClientRequest] \cup \{NoOp\},$
$viewNum \mapsto v \in (1\ ..\ ),$
$epochNum \mapsto e \in (1\ ..\ )\ ]$

$txnIDs \triangleq [\ shard \mapsto s \in Shards,$
$epoch \mapsto e \in (1\ ..\ ),\ msg \mapsto m \in (1\ ..\ )]$

FindTxn (Replicas to *Fcor*)
$[\ mtype \mapsto FindTxn,$
$shard \mapsto s \in Shards,$
$msgNum \mapsto m \in (1\ ..\ ),$
$epochNum \mapsto e \in (1\ ..\ )\ ]$

MTxnRequest (*Fcor* to *Replicas*)
$[\ mtype \mapsto MTxnRequest,$
$txnID \mapsto t \in txnIDs,$
$dest \mapsto r \in Replicas\ ]$

MHasTxn (Replicas to *Fcor*)
$[\ mtype \mapsto MHasTxn,$
$txn \mapsto t \in [MStampedClientRequest]\ ]$

MTempDroppedTxn (Replicas to *Fcor*)
$[\ mtype \mapsto MTempDroppedTxn,$
$viewNum \mapsto v \in (1\ ..\ ),$
$sender \mapsto r \in Replicas,$
$txnID \mapsto t \in txnIDs\ ]$

MTxnFound (*Fcor* to *Replicas*)
$[\ mtype \mapsto MTxnFound,$
$txn \mapsto t \in [MStampedClientRequest],$
$dest \mapsto r \in Replicas\ ]$

MTxnDropped (*Fcor* to *Replicas*)
$[\ mtype \mapsto MTxnDropped,$
$txnID \mapsto t \in txnIDs,\ dest \mapsto r \in Replicas\ ]$

MViewChangeReq (Replica to *Replicas*)
$[\ mtype \mapsto MViewChange,$
$viewNum \mapsto v \in (1\ ..\ ),$
$epochNum \mapsto e \in (1\ ..\ ),$
$dest \mapsto r \in Replicas\ ]$

MViewChange (Replica to *Replicas*)
$[\ mtype \mapsto MViewChange,$
$viewNum \mapsto v \in (1\ ..\ ),$
$epochNum \mapsto e \in (1\ ..\ ),$
$log \mapsto l \in (1\ ..\ ) \times ([MStampedClientRequest] \cup \{NoOp\}),$

$tempDrops \mapsto t \in \text{SUBSET } txnIDs$,
$permDrops \mapsto t \in \text{SUBSET } txnIDs$,
$unDrops \mapsto t \in \text{SUBSET } txnIDs$,
$epochMsgNum \mapsto t \in txnIDs$,
$sender \mapsto r \in Replicas$,
$dest \mapsto r \in Replicas$]

MStartView (Replica to *Replicas*)
[$mtype \mapsto MStartView$,
$viewNum \mapsto v \in (1 \ldots)$,
$epochNum \mapsto e \in (1 \ldots)$,
$log \mapsto l \in (1 \ldots) \times ([MStampedClientRequest] \cup \{NoOp\})$,
$tempDrops \mapsto t \in \text{SUBSET } txnIDs$,
$permDrops \mapsto t \in \text{SUBSET } txnIDs$,
$unDrops \mapsto t \in \text{SUBSET } txnIDs$,
$epochMsgNum \mapsto t \in txnIDs$,
$dest \mapsto r \in Replicas$]

MEpochChangeReq (Replica to *Fcor*)
[$mtype \mapsto MEpochChangeReq$,
$epochNum \mapsto e \in (1 \ldots)$]

MEpochChange (*Fcor* to *Replicas*)
[$mtype \mapsto MEpochChange$,
$epochNum \mapsto e \in (1 \ldots)$,
$lastNormEpoch \mapsto e \in (1 \ldots)$,
$dest \mapsto r \in Replicas$]

MEpochChangeAck (Replicas to *Fcor*)
[$mtype \mapsto MEpochChangeAck$,
$epochNum \mapsto e \in (1 \ldots)$,
$viewNum \mapsto v \in (1 \ldots)$,
$log \mapsto l \in (1 \ldots) \times ([MStampedClientRequest] \cup \{NoOp\})$,
$epochMsgNum \mapsto m \in (1 \ldots)$, (not necessary but useful)
$sender \mapsto r \in Replicas$]

MStartEpoch (*Fcor* to *Replicas*)
[$mtype \mapsto MStartEpoch$,
$epochNum \mapsto e \in (1 \ldots)$,
$viewNum \mapsto viewNum$,
$log \mapsto l \in (1 \ldots) \times ([MStampedClientRequest] \cup \{NoOp\})$,
$dest \mapsto r \in Replicas$]

Variables

Network State

VARIABLE *messages* Set of all messages sent

$networkVars \triangleq \langle messages \rangle$
$InitNetworkState \triangleq messages = \{\}$

Sequencer State

VARIABLE $seqCounters$

$$sequencerVars \triangleq \langle seqCounters \rangle$$
$$InitSequencerState \triangleq seqCounters = [s \in Sequencers \mapsto [h \in Shards \mapsto 1]]$$

Replica State

VARIABLES $vReplicaStatus$, $vLog$, $vEpochMsgNum$, $vViewNum$, $vEpochNum$, $vTempDrops$, $vPermDrops$, $vUnDrops$, $vViewChanges$, $vLastNormEpoch$

$$replicaVars \triangleq \langle vReplicaStatus, vLog, vEpochMsgNum, vViewNum, vEpochNum, vTempDrops, vPermDrops, vUnDrops, vViewChanges, vLastNormEpoch \rangle$$

$$\begin{aligned} InitReplicaState \triangleq\ & \wedge vReplicaStatus = [r \in Replicas \mapsto StNormal] \\ & \wedge vLog = [r \in Replicas \mapsto \langle\rangle] \\ & \wedge vEpochMsgNum = [r \in Replicas \mapsto 1] \\ & \wedge vViewNum = [r \in Replicas \mapsto 1] \\ & \wedge vEpochNum = [r \in Replicas \mapsto 1] \\ & \wedge vTempDrops = [r \in Replicas \mapsto \{\}] \\ & \wedge vPermDrops = [r \in Replicas \mapsto \{\}] \\ & \wedge vUnDrops = [r \in Replicas \mapsto \{\}] \\ & \wedge vViewChanges = [r \in Replicas \mapsto \{\}] \\ & \wedge vLastNormEpoch = [r \in Replicas \mapsto 1] \end{aligned}$$

Failure Coordinator State

VARIABLES $fFound$, $fDropped$, $fTempDrops$, $fStatus$, $fEpochNum$, $fEpochChanges$, $fLastNormEpoch$

$$fcorVars \triangleq \langle fFound, fDropped, fTempDrops, fStatus, fEpochNum, fEpochChanges, fLastNormEpoch \rangle$$

$$
\begin{aligned}
&InitFcorState \triangleq \\
&\quad \land fFound = \{\} \\
&\quad \land fDropped = \{\} \\
&\quad \land fTempDrops = \{\} \\
&\quad \land fStatus = StNormal \\
&\quad \land fEpochNum = 1 \\
&\quad \land fEpochChanges = \{\} \\
&\quad \land fLastNormEpoch = 1
\end{aligned}
$$

Set of all vars

$$vars \triangleq \langle networkVars,\ sequencerVars,\ replicaVars \rangle$$

Initial state

$$
\begin{aligned}
Init \triangleq\ & \land InitNetworkState \\
& \land InitSequencerState \\
& \land InitReplicaState \\
& \land InitFcorState
\end{aligned}
$$

Helpers

$$Max(s) \triangleq \text{CHOOSE } x \in s : \forall\, y \in s : x \geq y$$

$$Min(s) \triangleq \text{CHOOSE } x \in s : \forall\, y \in s : x \leq y$$

$$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$$

Add a message to the network

$$Send(ms) \triangleq messages' = messages \cup ms$$

$$
\begin{aligned}
Shard(r) \triangleq\ & ((r \div NumReplicasPerShard) - \\
& (\text{IF } r\%NumReplicasPerShard = 0 \text{ THEN } 1 \text{ ELSE } 0) \\
& + 1)
\end{aligned}
$$

$$ShardReplicas(s) \triangleq \{r \in Replicas : Shard(r) = s\}$$

$$
\begin{aligned}
Learner(shard,\ viewNum) \triangleq\ & (((viewNum - 1)\%NumReplicasPerShard) + \\
& ((shard - 1) * NumReplicasPerShard) + \\
& 1)
\end{aligned}
$$

$$tID(shard,\ epoch,\ msg) \triangleq [shard \mapsto shard,\ epoch \mapsto epoch,\ msg \mapsto msg]$$

Returns whether or not a *txnID* matching *txn* is in S

$$
\begin{aligned}
&txnMatches(txn,\ S) \triangleq \\
&\quad \land txn \neq NoOp \\
&\quad \land \exists\, s \in txn.shards : tID(s,\ txn.epochNum,\ txn.stamp[s]) \in S
\end{aligned}
$$

Returns whether or not a *txn* matching *txnID* is in S

$$
\begin{array}{l}
txnIDMatches(txnID,\ S) \triangleq \\
\quad \land \exists\, txn \in S : \land\ txn \neq NoOp \\
\qquad\qquad\qquad\ \land\ txn.epochNum = txnID.epoch \\
\qquad\qquad\qquad\ \land\ txnID.shard \in txn.shards \\
\qquad\qquad\qquad\ \land\ txn.stamp[txnID.shard] = txnID.msg
\end{array}
$$

Returns if $txn1$ has a later timestamp than $txn2$

$$
\begin{array}{l}
txnLater(txn1,\ txn2) \triangleq\ \lor\ txn1.epochNum > txn2.epochNum \\
\qquad\qquad\qquad\qquad\qquad\ \lor\ \land\ txn1.epochNum = txn2.epochNum \\
\qquad\qquad\qquad\qquad\qquad\qquad \land \exists\, s\ \in txn1.shards : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ s \in txn2.shards \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ txn1.stamp[s] > txn2.stamp[s]
\end{array}
$$

Main Spec

$$
\begin{array}{l}
RRs(s) \triangleq \{m \in messages : \land\ m.mtype = MRequestReply \\
\qquad\qquad\qquad\qquad\qquad\ \land\ Shard(m.sender) = s\}
\end{array}
$$

$$
RRsSlot(s,\ i) \triangleq \{m \in RRs(s) : m.txnIndex\ = i\}
$$

$$
RRsTxn(txn,\ s) \triangleq \{m \in RRs(s) : m.request = txn\}
$$

$$
RRsTxnSlot(txn,\ s,\ i) \triangleq \{m \in RRsTxn(txn,\ s) : m.txnIndex = i\}
$$

$$
\begin{array}{l}
CommittedInView(txn,\ s,\ i,\ v) \triangleq \\
\quad \land \exists\, M \in \text{SUBSET}\ \{m \in RRsTxnSlot(txn,\ s,\ i) : m.viewNum = v\} : \\
\qquad \text{From a majority} \\
\qquad \land\ 2 * Cardinality(M) > NumReplicasPerShard \\
\qquad \text{Matching } viewNums,\ epochNums,\ txnIndexs \\
\qquad \land \exists\, m1 \in M : \forall\, m2 \in M : m1.epochNum = m2.epochNum \\
\qquad \text{One from the learner} \\
\qquad \land \exists\, m \in M : m.sender = Learner(s,\ v)
\end{array}
$$

$$
\begin{array}{l}
CommittedInSlot(txn,\ s,\ i) \triangleq \\
\quad \exists\, v \in \{m.viewNum : m \in RRsTxnSlot(txn,\ s,\ i)\} : \\
\qquad CommittedInView(txn,\ s,\ i,\ v)
\end{array}
$$

$$
\begin{array}{l}
CommittedAtShard(txn,\ s) \triangleq \\
\quad \exists\, i \in \{m.txnIndex : m \in RRsTxn(txn,\ s)\} : \\
\qquad CommittedInSlot(txn,\ s,\ i)
\end{array}
$$

$$
\begin{array}{l}
MinCommittedView(txn,\ s,\ i) \triangleq \\
\quad Min(\{v \in \{m.viewNum : m \in RRsTxnSlot(txn,\ s,\ i)\} : \\
\qquad\quad CommittedInView(txn,\ s,\ i,\ v)\})
\end{array}
$$

A transaction being committed in a view implies that the designated learner in that view and ALL replicas in later views have that transaction in the

correct place in their logs.

$$
\begin{array}{l}
Memory(txn,\ s,\ i,\ v) \triangleq \\
\quad \forall\, r \in ShardReplicas(s) : \\
\quad (\ \wedge\ vReplicaStatus[r] = StNormal \\
\qquad \wedge\ \vee\ vViewNum[r] > v \\
\qquad\quad \vee\ \wedge\ vViewNum[r] = v \\
\qquad\qquad \wedge\ r = Learner(s,\ v)) \Rightarrow\ \wedge\ i \in \text{DOMAIN}\ vLog[r] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ vLog[r][i] = txn
\end{array}
$$

A transaction being committed at one shard implies that for all other participants to that transaction, if that shard has committed a later transaction, it has also committed that transaction

$$
\begin{array}{l}
GlobalOrder(txn,\ s,\ i) \triangleq \\
\quad \forall\, s2 \in txn.shards : \\
\quad \forall\, txn2 \in (\{m.request : m \in RRs(s2)\} \setminus \{NoOp\}) : \\
\quad (\ \wedge\ txnLater(txn2,\ txn) \\
\qquad \wedge\ CommittedAtShard(txn2,\ s2)) \Rightarrow CommittedAtShard(txn,\ s2)
\end{array}
$$

A transaction being committed at one shard in a slot implies that for every lower slot in that shard, there is some committed transaction, or the *Learner* had a *NoOp* in its *log*, and all replicas in later views have *NoOps* in their logs for that slot in all later views

$$
\begin{array}{l}
Gapless(txn,\ s,\ i,\ v) \triangleq \\
\text{LET} \\
\quad hadNoOp(ip) \triangleq \exists\, m \in RRsTxnSlot(NoOp,\ s,\ ip) : \\
\qquad \wedge\ m.viewNum = v \\
\qquad \wedge\ m.sender = Learner(s,\ v) \\
\text{IN} \\
\quad \forall\, i2 \in (1\ ..\ i-1) : \\
\quad \vee\ \wedge\ hadNoOp(i2) \\
\qquad \wedge \forall\, r \in ShardReplicas(s) : \\
\qquad\quad (\ \wedge\ vReplicaStatus[r] = StNormal \\
\qquad\qquad \wedge\ \vee\ vViewNum[r] > v \\
\qquad\qquad\quad \vee\ \wedge\ vViewNum[r] = v \\
\qquad\qquad\qquad \wedge\ r = Learner(s,\ v)) \Rightarrow vLog[r][i2] = NoOp \\
\quad \vee\ \wedge \neg hadNoOp(i2) \\
\qquad \wedge \exists\, txn2 \in \{m.request : m \in RRsSlot(s,\ i2)\} : \\
\qquad\quad CommittedInSlot(txn2,\ s,\ i2)
\end{array}
$$

Below is the main safety lemma. It does imply linearizability, but there are some other facts necessary to see that:

1) The *viewNums* of *Replicas* which are in *StNormal* state grow monotonically.
2) Messages are never removed from the network (so $CommittedInView(txn,\ s,\ i,\ v)$ at time t implies $Committed(txn,\ s,\ i,\ v)$ for all times $> t$.
3) The *Memory* property (along with monotonicity of *viewNums*) implies that no two transactions are ever committed in the same slot in a shard.

$Safety \triangleq$
$\forall\, s \in Shards :$
$\forall\, txn \in \{m.request : m \in RRs(s)\} :$
$\forall\, i \in \{m.txnIndex : m \in RRsTxn(txn,\, s)\} :$
$CommittedInSlot(txn,\, s,\, i) \Rightarrow$
$\land\ Memory(txn,\, s,\, i,\, MinCommittedView(txn,\, s,\, i))$
$\land\ txn \neq NoOp \Rightarrow GlobalOrder(txn,\, s,\, i)$
$\land\ Gapless(txn,\, s,\, i,\, MinCommittedView(txn,\, s,\, i))$

Normal Case Actions and Handlers

Send a request

$ClientSendsRequest \triangleq \exists\, S \in \text{SUBSET } (Shards) :$
$\land\ Cardinality(S) > 0$
$\land\ Send(\{[mtype \mapsto MClientRequest,$
$\quad shards \mapsto S]\})$
$\land\ \text{UNCHANGED } \langle sequencerVars,\, replicaVars,\, fcorVars\rangle$

Sequencer s receives *MClientRequest*, m

$HandleClientRequest(s,\, m) \triangleq$
$\land\ Send(\{[mtype \mapsto MStampedClientRequest,$
$\quad shards \mapsto m.shards,$
$\quad stamp \mapsto [S \in m.shards \mapsto seqCounters[s][S]],$
$\quad epochNum \mapsto s]\})$
$\land\ seqCounters' = [seqCounters \text{ EXCEPT } ![s] =$
$\quad [h \in Shards \mapsto$
$\quad \text{IF } h \in m.shards \text{ THEN } @[h] + 1 \text{ ELSE } @[h]]]$
$\land\ \text{UNCHANGED } \langle replicaVars,\, fcorVars\rangle$

Replica r receives *MMarkedClientRequest*, m

$HandleStampedClientRequest(r,\, m) \triangleq$
LET
$tempDropped \triangleq txnMatches(m,\, vTempDrops[r]) \land m \notin vUnDrops[r]$
$dropped \triangleq txnMatches(m,\, vPermDrops[r])$
IN
Normal case
$\land\ \lor\ \land\ vReplicaStatus[r] = StNormal$
$\quad \land\ m.epochNum = vEpochNum[r]$
$\quad \land\ m.stamp[Shard(r)] = vEpochMsgNum[r]$
Check if dropped
$\quad \land\ \neg(tempDropped \lor dropped)$
Add to *log* and respond
$\quad \land\ vLog' = [vLog \text{ EXCEPT } ![r] = Append(@,\, m)]$
$\quad \land\ vEpochMsgNum' = [vEpochMsgNum \text{ EXCEPT } ![r] = @ + 1]$

```
        For model-checking purposes, reply to all transactions in log
        in the current view
      ∧ Send({[mtype     ↦ MRequestReply,
               sender    ↦ r,
               txnIndex  ↦ i,
               request   ↦ vLog'[r][i],
               viewNum   ↦ vViewNum[r],
               epochNum ↦ vEpochNum[r]] : i ∈ (1 .. Len(vLog'[r]))})
    Gap
    ∨ ∧ vReplicaStatus[r] = StNormal
      ∧ m.epochNum = vEpochNum[r]
      ∧ m.stamp[Shard(r)] > vEpochMsgNum[r]
      ∧ Send({[mtype     ↦ MFindTxn,
               shard     ↦ Shard(r),
               msgNum    ↦ vEpochMsgNum[r],
               epochNum ↦ vEpochNum[r]]})
      ∧ UNCHANGED ⟨vLog, vEpochMsgNum⟩
    New epoch
    ∨ ∧ m.epochNum > vEpochNum[r]
      ∧ Send({[mtype     ↦ MEpochChangeReq,
               epochNum ↦ m.epochNum]})
      ∧ UNCHANGED ⟨vLog, vEpochMsgNum⟩
  ∧ UNCHANGED ⟨sequencerVars, fcorVars, vReplicaStatus, vViewNum, vEpochNum,
               vTempDrops, vPermDrops, vUnDrops, vViewChanges,
               vLastNormEpoch⟩
```

Other Replica Actions Handlers

Gap Commit Handlers

```
  Replica r receives MTxnRequest, m
HandleTxnRequest(r, m) ≜
LET
  txns ≜ IF ∧ vReplicaStatus[r] = StViewChange
            ∧ Learner(Shard(r), vViewNum[r]) = r THEN
          UNION {Range(mp.log) : mp ∈ vViewChanges[r]}
        ELSE
          Range(vLog[r])

  hasTxn ≜ txnIDMatches(m.txnID, txns)
  txn ≜ CHOOSE txn ∈ txns :
    ∧ txn ≠ NoOp
    ∧ m.txnID.shard ∈ txn.shards
    ∧ txn.stamp[m.txnID.shard] = m.txnID.msg
```

```
    ∧ txn.epochNum = m.txnID.epoch
IN
  ∧ vReplicaStatus[r] ∈ {StNormal, StViewChange}
  ∧ m.txnID.epoch = vEpochNum[r]
  ∧ ∨ ∧ hasTxn
      ∧ Send({[mtype ↦ MHasTxn,
               txn   ↦ txn]})
      ∧ UNCHANGED vTempDrops
    ∨ ∧ ¬hasTxn
      ∧ vReplicaStatus[r] = StNormal
      ∧ vTempDrops' = [vTempDrops EXCEPT ![r] = {m.txnID} ∪ @]
      ∧ Send({[mtype    ↦ MTempDroppedTxn,
               viewNum  ↦ vViewNum[r],
               sender   ↦ r,
               txnID    ↦ m.txnID]})
  ∧ UNCHANGED ⟨sequencerVars, fcorVars, vReplicaStatus, vLog, vEpochMsgNum,
               vViewNum, vEpochNum, vPermDrops, vUnDrops, vViewChanges,
               vLastNormEpoch⟩
```

Replica r receives *HandleMTxnFound*, m

```
HandleTxnFound(r, m) ≜
LET
  canAddNext ≜ ∧ m.txn.stamp[Shard(r)] = vEpochMsgNum[r]
               ∧ vReplicaStatus[r] = StNormal
IN
  ∧ vReplicaStatus[r] ∈ {StNormal, StViewChange}
  ∧ m.txn.epochNum = vEpochNum[r]
  Add txn to unDrops
  ∧ vUnDrops' = [vUnDrops EXCEPT ![r] = {m.txn} ∪ @]
  Add to log if caught up
  ∧ ∨ ∧ canAddNext
      ∧ vLog' = [vLog EXCEPT ![r] = Append(@, m.txn)]
      ∧ vEpochMsgNum' = [vEpochMsgNum EXCEPT ![r] = @ + 1]
    ∨ ∧ ¬canAddNext
      ∧ UNCHANGED ⟨vLog, vEpochMsgNum⟩
  ∧ UNCHANGED ⟨sequencerVars, fcorVars, vReplicaStatus, vViewNum,
               vEpochNum, vTempDrops, vPermDrops, vViewChanges,
               vLastNormEpoch, networkVars⟩
```

Replica r receives *MTxnDropped*, m

```
HandleTxnDropped(r, m) ≜
LET
  isNext ≜ ∧ m.txnID.shard = Shard(r)
           ∧ m.txnID.msg = vEpochMsgNum[r]
```

```
IN
  ∧ vReplicaStatus[r] ∈ {StNormal, StViewChange}
  ∧ m.txnID.epoch = vEpochNum[r]
   Add txnID to permDrops
  ∧ vPermDrops' = [vPermDrops EXCEPT ![r] = {m.txnID} ∪ @]
     If this is the next expected transaction, append a NoOp
  ∧ ∨ ∧ isNext
      ∧ vReplicaStatus[r] = StNormal
      ∧ vLog' = [vLog EXCEPT ![r] = Append(@, NoOp)]
      ∧ vEpochMsgNum' = [vEpochMsgNum EXCEPT ![r] = @ + 1]
     Otherwise, replace matching log transactions (should be ≤ 1) with NoOps
    ∨ ∧ ¬isNext
      ∧ vReplicaStatus[r] = StNormal
      ∧ vLog' = [vLog EXCEPT ![r] =
          [i ∈ (1 .. Len(@)) ↦ IF txnMatches(@[i], vPermDrops'[r]) THEN
                                  NoOp
                                ELSE
                                  @[i]]]
      ∧ UNCHANGED ⟨vEpochMsgNum⟩
     If catching up during view change, simply continue
    ∨ ∧ vReplicaStatus[r] ≠ StNormal
      ∧ UNCHANGED ⟨vLog, vEpochMsgNum⟩
  ∧ UNCHANGED ⟨sequencerVars, networkVars, fcorVars, vReplicaStatus, vViewNum,
               vEpochNum, vTempDrops, vUnDrops, vViewChanges,
               vLastNormEpoch⟩
```

View Change Action and Handlers

```
 Replica r suspects the designated learner has failed
StartLeaderChange(r) ≜
  ∧ vReplicaStatus[r] ∈ {StNormal, StViewChange}
  ∧ Send({[mtype   ↦ MViewChangeReq,
           dest    ↦ d,
           epochNum ↦ vEpochNum[r],
           viewNum  ↦ vViewNum[r] + 1] : d ∈ ShardReplicas(Shard(r))})
  ∧ UNCHANGED ⟨replicaVars, sequencerVars, fcorVars⟩

 Replica r receives MViewChangeReq, m
HandleViewChangeReq(r, m) ≜
LET
  vChangeMessage ≜ [mtype    ↦ MViewChange,
                    viewNum  ↦ m.viewNum,
                    epochNum ↦ vEpochNum[r],
                    log      ↦ vLog[r],
```

$$
\begin{array}{ll}
tempDrops & \mapsto vTempDrops[r], \\
permDrops & \mapsto vPermDrops[r], \\
unDrops & \mapsto vUnDrops[r], \\
epochMsgNum & \mapsto vEpochMsgNum[r], \\
sender & \mapsto r, \\
dest & \mapsto Learner(Shard(r),\ m.viewNum)]
\end{array}
$$

$isNewLearner \triangleq Learner(Shard(r),\ m.viewNum) = r$

IN

$\land\ vReplicaStatus[r] \in \{StNormal,\ StViewChange\}$

$\land\ m.epochNum = vEpochNum[r]$

$\land\ m.viewNum > vViewNum[r]$ It's important for the way I check for a quorum that each replica only send out a single *msg* per new view (*i.e.*, so the messages don't both get counted)

$\land\ vReplicaStatus' = [vReplicaStatus \text{ EXCEPT } ![r] = StViewChange]$

$\land\ vViewNum' = [vViewNum \text{ EXCEPT } ![r] = m.viewNum]$

$\land\ \lor\ \land\ isNewLearner$

$\quad\ \land\ vViewChanges' = [vViewChanges \text{ EXCEPT } ![r] = \{vChangeMessage\}]$

$\quad\ \land \text{ UNCHANGED } \langle networkVars\rangle$

$\quad\ \lor\ \land \neg isNewLearner$

$\quad\ \land\ vViewChanges' = [vViewChanges \text{ EXCEPT } ![r] = \{\}]$

$\quad\ \land\ Send(\{vChangeMessage\})$

$\land \text{ UNCHANGED } \langle sequencerVars,\ fcorVars,\ vLog,\ vEpochMsgNum,\ vEpochNum,$
$vTempDrops,\ vPermDrops,\ vUnDrops,\ vLastNormEpoch\rangle$

Replica r receives *MViewChange*, m

$HandleViewChange(r,\ m) \triangleq$

LET

$newTempDrops \triangleq vTempDrops[r] \cup ($
$\quad \text{UNION } \{mp.tempDrops : mp \in vViewChanges'[r]\})$

$newPermDrops \triangleq vPermDrops[r] \cup ($
$\quad \text{UNION } \{mp.permDrops : mp \in vViewChanges'[r]\})$

$newUnDrops \triangleq vUnDrops[r] \cup ($
$\quad \text{UNION } \{mp.unDrops : mp \in vViewChanges'[r]\})$

$logs \triangleq \{mp.log : mp \in vViewChanges'[r]\}$

$longestLog \triangleq \text{CHOOSE } log \in logs : \forall\, log2 \in logs : Len(log) \geq Len(log2)$

$newLog \triangleq [i \in (1\ ..\ Len(longestLog)) \mapsto$
$\quad \text{IF } txnMatches(longestLog[i],\ newPermDrops) \text{ THEN}$
$\quad\quad NoOp$
$\quad \text{ELSE}$
$\quad\quad longestLog[i]]$

$newEpochMsgNum \triangleq (\text{CHOOSE } mp \in vViewChanges'[r] :$
$\quad\quad Len(mp.log) = Len(longestLog)).epochMsgNum$

$canStartView \triangleq$
$\land\ 2 * Cardinality(vViewChanges'[r]) > NumReplicasPerShard$
$\land\ \forall\ t \in Range(newLog) :$
$txnMatches(t,\ newTempDrops) \Rightarrow t \in newUnDrops$
IN
$\land\ vReplicaStatus[r] = StViewChange$
$\land\ m.epochNum = vEpochNum[r]$
Add the message to the *log*
$\land\ vViewNum[r] = m.viewNum$ Must be equal for consistency of *vViewChanges*
$\land\ vViewChanges' = [vViewChanges \text{ EXCEPT } ![r] = \{m\} \cup @]$
If there's a quorum, start the new view
$\land\ \lor\ \land\ canStartView$
$\land\ vLog' = [vLog \text{ EXCEPT } ![r] = newLog]$
$\land\ vEpochMsgNum' = [vEpochMsgNum \text{ EXCEPT } ![r] = newEpochMsgNum]$
$\land\ vTempDrops' = [vTempDrops \text{ EXCEPT } ![r] = newTempDrops]$
$\land\ vPermDrops' = [vPermDrops \text{ EXCEPT } ![r] = newPermDrops]$
$\land\ vUnDrops' = [vUnDrops \text{ EXCEPT } ![r] = newUnDrops]$
$\land\ vReplicaStatus' = [vReplicaStatus \text{ EXCEPT } ![r] = StNormal]$
$\land\ Send(\{[mtype \mapsto MStartView,$
$viewNum \mapsto vViewNum[r],$
$epochNum \mapsto vEpochNum[r],$
$log \mapsto newLog,$
$tempDrops \mapsto newTempDrops,$
$permDrops \mapsto newPermDrops,$
$unDrops \mapsto newUnDrops,$
$epochMsgNum \mapsto newEpochMsgNum,$
$dest \mapsto rp] : rp \in (ShardReplicas(Shard(r)) \setminus \{r\})\})$
$\lor\ \land\ \neg canStartView$
$\land\ \text{UNCHANGED } \langle networkVars,\ vLog,\ vEpochMsgNum,\ vTempDrops,\ vPermDrops,$
$vUnDrops,\ vReplicaStatus\rangle$
$\land\ \text{UNCHANGED } \langle sequencerVars,\ fcorVars,\ vViewNum,\ vEpochNum,$
$vLastNormEpoch\rangle$

Replica r receives a *MStartView*, m
$HandleStartView(r,\ m) \triangleq$
$\land\ vReplicaStatus[r] \in \{StNormal,\ StViewChange\}$
$\land\ m.epochNum = vEpochNum[r]$
$\land\ \lor\ m.viewNum > vViewNum[r]$
$\lor\ m.viewNum = vViewNum[r] \land vReplicaStatus[r] = StViewChange$
$\land\ vViewNum' = [vViewNum \text{ EXCEPT } ![r] = m.viewNum]$
$\land\ vReplicaStatus' = [vReplicaStatus \text{ EXCEPT } ![r] = StNormal]$
$\land\ vLog' = [vLog \text{ EXCEPT } ![r] = m.log]$
$\land\ vTempDrops' = [vTempDrops \text{ EXCEPT } ![r] = m.tempDrops]$
$\land\ vPermDrops' = [vPermDrops \text{ EXCEPT } ![r] = m.permDrops]$

$\land\ vUnDrops' = [vUnDrops \text{ EXCEPT } ![r] = m.unDrops]$
$\land\ vEpochMsgNum' = [vEpochMsgNum \text{ EXCEPT } ![r] = m.epochMsgNum]$
$\land$ UNCHANGED $\langle sequencerVars,\ networkVars,\ fcorVars,\ vEpochNum,\ vViewChanges,$
$vLastNormEpoch\rangle$

Epoch Change Handlers

Replica r receives a *MEpochChange*, m

$HandleEpochChange(r,\ m) \triangleq$
$\land\ m.epochNum > vEpochNum[r]$
Force replicas to go through epochs that start one at a time (this could be done slightly differently)
$\land\ m.lastNormEpoch = vLastNormEpoch[r]$
$\land\ vReplicaStatus' = [vReplicaStatus \text{ EXCEPT } ![r] = StEpochChange]$
$\land\ vEpochNum' = [vEpochNum \text{ EXCEPT } ![r] = m.epochNum]$
$\land\ Send(\{[mtype \mapsto MEpochChangeAck,$
$epochNum \mapsto m.epochNum,$
$viewNum \mapsto vViewNum[r],$
$log \mapsto vLog[r],$
$epochMsgNum \mapsto vEpochMsgNum[r],$
$sender \mapsto r]\})$
$\land$ UNCHANGED $\langle sequencerVars,\ fcorVars,\ vLog,\ vEpochMsgNum,\ vViewNum,$
$vTempDrops,\ vPermDrops,\ vUnDrops,\ vViewChanges,$
$vLastNormEpoch\rangle$

Replica r receives a *MStartEpoch*, m

$HandleStartEpoch(r,\ m) \triangleq$
$\land\ \lor\ m.epochNum > vEpochNum[r]$
$\lor\ \land\ m.epochNum = vEpochNum[r]$
$\land\ vReplicaStatus[r] = StEpochChange$
$\land\ vReplicaStatus' = [vReplicaStatus \text{ EXCEPT } ![r] = StNormal]$
$\land\ vLog' = [vLog \text{ EXCEPT } ![r] = m.log]$
$\land\ vEpochMsgNum' = [vEpochMsgNum \text{ EXCEPT } ![r] = 1]$
$\land\ vViewNum' = [vViewNum \text{ EXCEPT } ![r] = m.viewNum]$
$\land\ vEpochNum' = [vEpochNum \text{ EXCEPT } ![r] = m.epochNum]$
$\land\ vTempDrops' = [vTempDrops \text{ EXCEPT } ![r] = \{\}]$
$\land\ vPermDrops' = [vPermDrops \text{ EXCEPT } ![r] = \{\}]$
$\land\ vUnDrops' = [vUnDrops \text{ EXCEPT } ![r] = \{\}]$
$\land\ vLastNormEpoch' = [vLastNormEpoch \text{ EXCEPT } ![r] = m.epochNum]$
$\land$ UNCHANGED $\langle sequencerVars,\ networkVars,\ fcorVars,\ vViewChanges\rangle$

Failure Coordinator Message Handlers

```
HandleFindTxn(m) ≜
LET
  txnID ≜ [shard ↦ m.shard, epoch ↦ m.epochNum, msg ↦ m.msgNum]
IN
  ∧ fStatus = StNormal
  ∧ m.epochNum = fEpochNum
  ∧ Send({[mtype ↦ MTxnRequest,
           txnID ↦ txnID,
           dest  ↦ r] : r ∈ Replicas})
  ∧ UNCHANGED ⟨sequencerVars, replicaVars, fcorVars⟩

HandleHasTxn(m) ≜
  ∧ fStatus = StNormal
  ∧ m.txn.epochNum = fEpochNum
  Don't "find" the transaction if it was already dropped
  ∧ ¬txnMatches(m.txn, fDropped)
  ∧ fFound' = {m.txn} ∪ fFound
  ∧ Send({[mtype ↦ MTxnFound,
           txn   ↦ m.txn,
           dest  ↦ r] : r ∈ {rp ∈ Replicas :
                             Shard(rp) ∈ m.txn.shards}})
  ∧ UNCHANGED ⟨sequencerVars, replicaVars, fDropped, fTempDrops, fStatus,
               fEpochNum, fEpochChanges, fLastNormEpoch⟩

HandleTempDroppedTxn(m) ≜
LET
  IsDropped(txnID) ≜ ∀ s ∈ Shards :
                       ∃ M ∈ SUBSET {mp ∈ fTempDrops' :
                                     ∧ mp.txnID = txnID
                                     ∧ Shard(mp.sender) = s} :
                         ∧ 2 * Cardinality(M) > NumReplicasPerShard
                         ∧ ∃ m1 ∈ M : ∀ m2 ∈ M : m1.viewNum = m2.viewNum
                         ∧ ∃ m1 ∈ M : m1.sender = Learner(s, m1.viewNum)
IN
  ∧ fStatus = StNormal
  ∧ m.txnID.epoch = fEpochNum
  Don't drop transactions already found
  ∧ ¬txnIDMatches(m.txnID, fFound)
  ∧ fTempDrops' = {m} ∪ fTempDrops
  ∧ ∨ ∧ IsDropped(m.txnID)
      ∧ fDropped' = {m.txnID} ∪ fDropped
      ∧ Send({[mtype ↦ MTxnDropped,
               txnID ↦ m.txnID,
               dest  ↦ r] : r ∈ {rp ∈ Replicas :
```

```
                                        Shard(rp) = m.txnID.shard}})
    ∨ ∧ ¬IsDropped(m.txnID)
      ∧ UNCHANGED ⟨networkVars, fDropped⟩
  ∧ UNCHANGED ⟨sequencerVars, replicaVars, fFound, fStatus, fEpochNum,
               fEpochChanges, fLastNormEpoch⟩

HandleEpochChangeReq(m) ≜
  ∧ m.epochNum > fEpochNum
  ∧ fStatus' = StEpochChange
  ∧ fEpochNum' = m.epochNum
  ∧ fEpochChanges' = {}
  ∧ Send({[mtype          ↦ MEpochChange,
           epochNum       ↦ m.epochNum,
           lastNormEpoch  ↦ fLastNormEpoch,
           dest           ↦ r] : r ∈ Replicas})
  ∧ UNCHANGED ⟨sequencerVars, replicaVars, fFound, fDropped, fTempDrops,
               fLastNormEpoch⟩

HandleEpochChangeAck(m) ≜
LET
  canStartEpoch(M) ≜ ∀ s ∈ Shards :
    2 * Cardinality({mp ∈ M : Shard(mp.sender) = s}) > NumReplicasPerShard

  newViewNum(s, M) ≜ Max({mp.viewNum : mp ∈
                          {mpp ∈ M : Shard(mpp.sender) = s}})

  simpleCombinedTxns(M) ≜ (UNION {Range(mp.log) : mp ∈ M}) \ {NoOp}

  combinedTxns(M) ≜ fFound ∪ {txn ∈ simpleCombinedTxns(M) :
    ∀ s ∈ txn.shards :
    tID(s, txn.epochNum, txn.stamp[s]) ∉ fDropped}

  logsFromShard(s, M) ≜ {mp.log : mp ∈
                         {mpp ∈ M : Shard(mpp.sender) = s}}

  lengthMaxLogFromShard(s, M) ≜ Max({Len(log) : log ∈ logsFromShard(s, M)})

  maxLogFromShard(s, M) ≜ CHOOSE log ∈ logsFromShard(s, M) :
    Len(log) = lengthMaxLogFromShard(s, M)

  prevEpochLastSlot(s, e, M) ≜
    LET
      log ≜ maxLogFromShard(s, M)
      hasNewMsg ≜ ∃ txn ∈ Range(log) : ∧ txn ≠ NoOp
                                       ∧ txn.epochNum = e
      newMsgIndex ≜ CHOOSE i ∈ DOMAIN log : ∧ log[i] ≠ NoOp
```

```
                                                    ∧ log[i].epochNum = e
    IN
      IF hasNewMsg THEN
        newMsgIndex − log[newMsgIndex].stamp[s]
      ELSE
        Len(log)

  maxTxnForShard(s, e, M) ≜ Max({txn.stamp[s] : txn ∈
    {txnp ∈ combinedTxns(M) : ∧ txnp.epochNum = e
                              ∧ s ∈ txnp.shards}} ∪ {0})

   e is last normal epoch
  newLog(s, e, M) ≜ (
     All of the messages from one of the logs from the old epochs
    SubSeq(maxLogFromShard(s, M), 1, prevEpochLastSlot(s, e, M)) ∘
     The messages for this shard which weren't dropped in the new epoch
    [i ∈ (1 .. maxTxnForShard(s, e, M)) ↦
      IF
        ∃ txn ∈ combinedTxns(M) :
          ∧ s ∈ txn.shards
          ∧ txn.stamp[s] = i
          ∧ txn.epochNum = e
      THEN
        CHOOSE txn ∈ combinedTxns(M) :
          ∧ s ∈ txn.shards
          ∧ txn.stamp[s] = i
          ∧ txn.epochNum = e
      ELSE
        NoOp
    ])
IN
  ∧ fStatus = StEpochChange
  ∧ m.epochNum = fEpochNum
  ∧ fEpochChanges' = {m} ∪ fEpochChanges
  ∧ ∨ ∧ ¬canStartEpoch(fEpochChanges')
      ∧ UNCHANGED ⟨networkVars, fStatus, fFound, fTempDrops, fDropped,
                   fLastNormEpoch⟩
    ∨ ∧ canStartEpoch(fEpochChanges')
      ∧ fStatus' = StNormal
      ∧ fFound' = {}
      ∧ fTempDrops' = {}
      ∧ fDropped' = {}
      ∧ fLastNormEpoch' = fEpochNum
      ∧ Send({[mtype      ↦ MStartEpoch,
               epochNum   ↦ fEpochNum,
```

$$
\begin{array}{ll}
viewNum & \mapsto newViewNum(Shard(r),\ fEpochChanges'), \\
log & \mapsto newLog(Shard(r),\ fLastNormEpoch, \\
 & \qquad\qquad fEpochChanges'), \\
dest & \mapsto r] : r \in Replicas\})
\end{array}
$$

$\land$ UNCHANGED $\langle sequencerVars,\ replicaVars,\ fEpochNum\rangle$

Main Transition Function

$$
\begin{array}{ll}
Next \triangleq & \text{Client Actions} \\
 & \lor\ ClientSendsRequest \\
 & \text{Normal Case Handlers} \\
 & \lor \exists\, m \in messages : \\
 & \quad \exists\, s \in Sequencers \\
 & \qquad\qquad : \land\ m.mtype = MClientRequest \\
 & \qquad\qquad\ \ \land\ HandleClientRequest(s,\ m) \\
 & \lor \exists\, m \in messages : \\
 & \quad \exists\, r \in Replicas : \land\ m.mtype = MStampedClientRequest \\
 & \qquad\qquad\ \ \land\ Shard(r) \in m.shards \\
 & \qquad\qquad\ \ \land\ HandleStampedClientRequest(r,\ m) \\
 & \text{Replica Actions} \\
 & \lor \exists\, r \in Replicas : StartLeaderChange(r) \\
 & \text{Other Replica Handlers} \\
 & \lor \exists\, m \in messages : \lor\ \land\ m.mtype = MTxnRequest \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleTxnRequest(m.dest,\ m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MTxnFound \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleTxnFound(m.dest,\ m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MTxnDropped \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleTxnDropped(m.dest,\ m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MViewChangeReq \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleViewChangeReq(m.dest,\ m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MViewChange \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleViewChange(m.dest,\ m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MStartView \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleStartView(m.dest,\ m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MEpochChange \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleEpochChange(m.dest,\ m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MStartEpoch \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleStartEpoch(m.dest,\ m) \\
 & \text{Failure coordinator handlers} \\
 & \lor \exists\, m \in messages : \lor\ \land\ m.mtype = MFindTxn \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleFindTxn(m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MHasTxn \\
 & \qquad\qquad\qquad\qquad\ \ \land\ HandleHasTxn(m) \\
 & \qquad\qquad\qquad\qquad \lor\ \land\ m.mtype = MTempDroppedTxn
\end{array}
$$

$$
\begin{array}{ll}
 & \land\ HandleTempDroppedTxn(m) \\
\lor & \land\ m.mtype = MEpochChangeReq \\
 & \land\ HandleEpochChangeReq(m) \\
\lor & \land\ m.mtype = MEpochChangeAck \\
 & \land\ HandleEpochChangeAck(m)
\end{array}
$$

B P4 Sequencer Implementation

headers.p4

```
header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

header_type ipv4_t {
    fields {
        version : 4;
        ihl : 4;
        diffserv : 8;
        totalLen : 16;
        identification : 16;
        flags : 3;
        fragOffset : 13;
        ttl : 8;
        protocol : 8;
        hdrChecksum : 16;
        srcAddr : 32;
        dstAddr: 32;
    }
}

header_type udp_eris_t {
    fields {
        srcPort : 16;
        dstPort : 16;
        length : 16;
        checksum : 16;

        /* These are eris specific fields appended to a standard UDP
         * packet. The sequence mask is bit representation of which
         * shards this request is touching. Currently, it supports upto
         * 16 shards, but can be easily extended using more fields. */
        sessionNumber : 16;
        sequenceMask  : 16;
    }
}

/* The sequence numbers appended by the switch. */
header_type eris_seq_t {
    fields {
        sequenceNumber : 32;
    }
}
```

parser.p4

```
#define ETHERTYPE_IPV4 0x0800
#define IP_PROTOCOL_ERIS 0x254

parser start {
    return parse_ethernet;
}

header ethernet_t ethernet;

parser parse_ethernet {
    extract(ethernet);
    return select(latest.etherType) {
        ETHERTYPE_IPV4 : parse_ipv4;
        default: ingress;
```

```
    }
}

header ipv4_t ipv4;

parser parse_ipv4 {
    extract(ipv4);
    return select(latest.protocol) {
        IP_PROTOCOL_ERIS : parse_eris;
        default: ingress;
    }
}

header udp_eris_t eris;

parser parse_eris {
    extract(eris);
    return ingress;
}

header eris_seq_t eris_seq[16];
```

eris.p4

```
#include "headers.p4"
#include "parser.p4"

#define MAX_SHARDS 16

/* This structure contains the data that moves through the packet
 * processing pipeline. */
header_type ingress_metadata_t {
    fields {
        sequenceNum   : 32;   // For transient calculations.
    }
}

metadata ingress_metadata_t ingress_data;

/* These stateful array of registers keep track of the sequence numbers
 * of for each shard. They are split over multiple stages to help scale.
 * A smart compiler could do this automatically.
 * sequence_numbers[i] = sequence number for shard i; */

register sequence_numbers {
    width : 32;
    instance_count : MAX_SHARDS;
}

/* This action block updates the current packet's session number and
 * sequence number corresponding to the replica group. */
action eris_update(shard) {

    // Reads the values from stateful memory into ingress metadata.
    register_read(ingress_data.sequenceNum, sequence_numbers, shard);

    // Increment the sequence number for this shard by 1.
    add_to_field(ingress_data.sequenceNum, 1);

    // Append the sequence number for this shard into the packet header stack.
    push(eris_seq, 1);
    modify_field(eris_seq[0].sequenceNumber, ingress_data.sequenceNum);

    // Write back the updated value into stateful memory.
    register_write(sequence_numbers, shard, ingress_data.sequenceNum);
}

action _drop() {
```

```
45      drop();
46  }
47
48  /* M+A tables that apply eris actions. */
49  table eris {
50      reads {
51          eris.sequenceMask : exact;
52      }
53      actions {
54          eris_update;
55          _drop;
56      }
57  }
58
59  /* Packet processing starts here. */
60  control ingress {
61      // Apply the eris sequence increment too all bits in the sequencer mask.
62
63      if (eris.sequenceMask & 0x0001) {
64          apply(eris);
65      }
66
67      if (eris.sequenceMask & 0x0002) {
68          apply(eris);
69      }
70
71      if (eris.sequenceMask & 0x0004) {
72          apply(eris);
73      }
74
75      if (eris.sequenceMask & 0x0008) {
76          apply(eris);
77      }
78
79      if (eris.sequenceMask & 0x0010) {
80          apply(eris);
81      }
82
83      if (eris.sequenceMask & 0x0020) {
84          apply(eris);
85      }
86
87      if (eris.sequenceMask & 0x0040) {
88          apply(eris);
89      }
90
91      if (eris.sequenceMask & 0x0080) {
92          apply(eris);
93      }
94
95      if (eris.sequenceMask & 0x0100) {
96          apply(eris);
97      }
98
99      if (eris.sequenceMask & 0x0200) {
100         apply(eris);
101     }
102
103     if (eris.sequenceMask & 0x0400) {
104         apply(eris);
105     }
106
107     if (eris.sequenceMask & 0x0800) {
108         apply(eris);
109     }
110
111     if (eris.sequenceMask & 0x1000) {
112         apply(eris);
```

```
113     }
114
115     if (eris.sequenceMask & 0x2000) {
116         apply(eris);
117     }
118
119     if (eris.sequenceMask & 0x4000) {
120         apply(eris);
121     }
122
123     if (eris.sequenceMask & 0x8000) {
124         apply(eris);
125     }
126
127     // The usual forwarding.
128     apply(forward);
129 }
```