Fault-Tolerant Distributed Protocols in Ordered Networks

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

Abstract

Protocols for fault-tolerant distributed systems typically assume a completely asynchronous network, in which messages can be dropped or reordered arbitrarily. Guaranteeing the safety of client-server based distributed storage systems in this context involves guaranteeing two distinct properties: *order* of state updates and their *durability*. This work shows that in single datacenters, separating these two concerns and guaranteeing ordering in the network using the capabilities of upcoming programmable switches can avoid most of the latency and throughput costs of replication and consistency. This work studies two related problems—state machine replication and distributed transaction processing—and presents two protocols—NOPaxos and Eris—which leverage different network ordering properties to deliver drastically improved normal case performance.

1 Introduction

Server failures are commonplace in today's datacenters, and today's datacenter applications are built on top of fault-tolerant distributed protocols in order to mitigate the detrimental effects of failures to availability. In the best case, these protocols provide strong consistency guarantees; however, strong consistency often comes at the cost of performance, even in the context of a single data center. Even relatively simple state machine replication protocols such as Paxos have both latency and throughput penalties as compared to an unreplicated, single machine.

These costs are not fundamental. This work describes how to mitigate the overhead of fault-tolerance and consistency within a single datacenter through protocol-level and network-level techniques. In particular, the datacenter has a highly structured network and allows for the deployment of emerging programmable network hardware, hardware which can be used to do small amounts of processing in the fabric of the network itself and overcome the limitations of a strict asynchronous network model. This work focuses on two related problems: state machine replication and distributed transaction processing.

In both cases, this work argues for a new separation of concerns between the network and application wherein a network-level primitive is responsible for guaranteeing *order* of operations, while the application is responsible for guaranteeing *reliability* and other properties. In the case of state machine replication, we present the *ordered unreliable multicast* primitive, which provides the twin guarantees of order and drop detection in order to support applications with minimal communication. In the case of distributed transaction processing, we show how OUM must be modified to support multiple recipient groups while still providing the same guarantees; the resulting primitive we call the *multi-sequenced groupcast*. We show how both of these primitives can be realized in the network.

Then, using the OUM and multi-sequenced groupcast primitives, we show how to build Network-Ordered Paxos (NOPaxos) and Eris, respectively. These are protocols for state machine replication and distributed transaction processing which avoid all coordination except in rare cases—circumventing classic sources of overhead. NOPaxos and Eris both assume a client-server model, and both rely on the network to handle the out-cast problem and use the client to handle the in-cast problem (ensuring reliability).

While NOPaxos' interface is completely generic, Eris has a particular model of transactions—in fact, it has two. At one level, Eris supports *independent transaction* [19,62]; these are atomic, one-shot code blocks which run locally on nodes themselves and do not depend on the global state of the system. Independent transactions are exactly the transactions which can be processed in a single round-trip from the client. On top of its independent transaction processing layer, Eris also supports more general transactions composed of multiple independent transactions. There can be dependencies among these component transactions, and Eris uses two-phase locking to provide isolation to the whole. Clients in Eris act as their own transaction managers, and Eris provides a method for dealing with failure of clients.

2 State Machine Replication

We first consider the problem of *state machine replication* [59]. Replication, used throughout data center applications, keeps key services consistent and available despite the inevitability of failures. For example, Google's Chubby [12] and Apache ZooKeeper [25] use replication to build a highly available lock service that is widely used to coordinate access to shared resources and configuration information. It is also used in many storage services to prevent system outages or data loss [9, 17, 57].

Correctness for state machine replication requires a system to behave as a linearizable [24] entity. Assuming that the application code at the replicas is deterministic, establishing a single totally ordered set of operations ensures that all replicas remain in a consistent state. We divide this into two separate properties:

1. **Ordering**: If some replica processes request a before b, no replica processes b before a.
2. **Reliable Delivery**: Every request submitted by a client is either processed by all replicas or none.

Our research examines the question: *Can the responsibility for either of these properties be moved from the application layer into the network*?

State of the art. Traditional state machine replication uses consensus protocol – e.g., Paxos [36, 37] or Viewstamped Replication [42, 52] – to achieve agreement on operation order. Most deployments of Paxos-based replicated systems use the Multi-Paxos optimization [37] (equivalent to Viewstamped Replication), where one replica is the designated leader and assigns an order to requests. Its normal operation proceeds in four phases: clients submit requests to the leader; the leader assigns a sequence number and notifies the other replicas; a majority of other replicas acknowledge; and the leader executes the request and notifies the client.

These protocols are designed for an *asynchronous network*, where there are no guarantees that packets will be received in a timely manner, in any particular order, or even delivered at all. As a result, the application-level protocol assumes responsibility for both ordering and reliability.

The case for ordering without reliable delivery. If the network itself provided stronger guarantees, the full complexity of Paxos-style replication would be unnecessary. At one extreme, an atomic broadcast primitive (i.e., a *virtually synchronous* model) [7, 28] ensures *both* reliable delivery and consistent ordering, which makes replication trivial. Unfortunately, implementing atomic broadcast is a problem equivalent to consensus [14] and incurs the same costs, merely in a different layer.

This paper envisions a middle ground: an *ordered but unreliable* network. We show that a new division of responsibility – providing ordering in the network layer but leaving reliability to the replication protocol – leads to a more efficient whole. What makes this possible is that an ordered unreliable multicast primitive can be implemented efficiently and easily in the network, yet fundamentally simplifies the task of the replication layer.

At the same time, providing an ordering guarantee simplifies the replication layer dramatically. Rather than agree on *which* request should be executed next, it needs to ensure only all-or-nothing delivery of each message. We show that this enables a simpler replication protocol that can execute operations without inter-replica coordination in the common case when messages are not lost, yet can recover quickly from lost messages.

3 Ordered Unreliable Multicast

We have argued for a separation of concerns between ordering and reliable delivery. Towards this end, we seek to design an *ordered* but *unreliable* network. In this section, we precisely define the properties that this network provides, and show how it can be realized efficiently using in-network processing.

We are not the first to argue for a network with ordered delivery semantics. Prior work has observed that some networks often deliver requests to replicas in the same order [54, 67], that data center networks can be engineered to support a multicast primitive that has this property [56], and that it is possible to use this fact to design protocols that are more efficient in the common case [31, 39, 56]. We contribute by demonstrating that it is possible to build a network with ordering *guarantees* rather than probabilistic or best-effort properties. As we show in Section 5, doing so can support simpler and more efficient protocols.

Figure 1 shows the architecture of an OUM/NOPaxos deployment. All components reside in a single data center. OUM is implemented by components in the network along with a library, libOUM, that runs on senders and receivers. NOPaxos is a replication system that uses libOUM; clients use libOUM to send messages, and replicas use libOUM to receive clients' messages.

3.1 Ordered Unreliable Multicast Properties

We begin by describing the basic primitive provided by our networking layer: *ordered unreliable multicast*. More specifically, our model is an *asynchronous, unreliable network* that supports *ordered multicast* with *multicast drop detection*. These properties are defined as follows:

- **Asynchrony:** There is no bound on the latency of message delivery.
- **Unreliability:** The network does not guarantee that any message will ever be delivered to any recipient.
- **Ordered Multicast**: The network supports a multicast operation such that if two messages, m and m', are multicast to a set of processes, R, then all processes in R that receive m and m' receive them in the same order.
- **Multicast Drop Detection**: If some message, m, is multicast to some set of processes, R, then either: (1) every process in R receives m or a notification that there was a dropped message before receiving the next multicast, or (2) no process in R receives m or

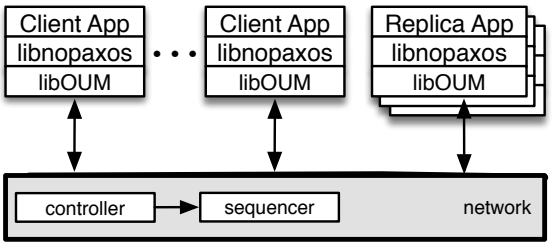

Figure 1: Architecture of NOPaxos.

a dropped message notification for m.[1]

The asynchrony and unreliability properties are standard in network design. Ordered multicast is not: existing multicast mechanisms do not exhibit this property, although Mostly-Ordered Multicast provides it on a best-effort basis [56]. Importantly, our model requires that any pair of multicast messages successfully sent to the same group are *always* delivered in the same order to all receivers – unless one of the messages is not received. In this case, however, the receiver is notified.

3.2 OUM Sessions and the libOUM API

Our OUM primitive is implemented using a combination of a network-layer sequencer and a communication library called libOUM. libOUM's API is a refinement of the OUM model described above. An OUM *group* is a set of receivers and is identified by an IP address. We explain group membership changes in Section 5.2.5.

libOUM introduces an additional concept, *sessions*. For each OUM group, there are one or more sessions, which are intervals during which the OUM guarantees hold. Conceptually, the stream of messages being sent to a particular group is divided into consecutive OUM sessions. From the beginning of an OUM session to the time it terminates, all OUM guarantees apply. However, OUM sessions are not guaranteed to terminate at the same point in the message stream for each multicast receiver: an arbitrary number of messages at the end of an OUM session could be dropped without notification, and this number might differ for each multicast receiver. Thus, each multicast recipient receives a prefix of the messages assigned to each OUM session, where some messages are replaced with drop notifications.

Sessions are generally long-lived. However, rare, exceptional network events (sequencer failures) can terminate them. In this case, the application is notified of session termination and then must ensure that it is in a consistent state with the other receivers before listening for the next session. In this respect, OUM sessions resemble TCP connections: they guarantee ordering within their lifetime, but failures may cause them to end.

[1] This second case can be thought of as a *sender omission*, whereas the first case can be thought of as a *receiver omission*, with the added drop notification guarantee.

libOUM Sender Interface

- `send(addr destination, byte[] message)` — send a message to the given OUM group

libOUM Receiver Interface

- `getMessage()` — returns the next message, a DROP-NOTIFICATION, or a SESSION-TERMINATED error
- `listen(int sessionNum, int messageNum)` — resets libOUM to begin listening in OUM session *sessionNum* for message *messageNum*

Figure 2: The libOUM interface.

Applications access OUM sessions via the libOUM interface (Figure 2). The receiver interface provides a `getMessage()` function, which returns either a message or a DROP-NOTIFICATION during an OUM session. When an OUM session terminates, `getMessage()` returns a special value, SESSION-TERMINATED, until the user of libOUM starts the next OUM session. To begin listening to the next OUM session and receiving its messages and DROP-NOTIFICATIONs, the receiver calls `listen(int newSessionNum, 0)`. To start an OUM session at a particular position in the message stream, the receiver can call `listen(int sessionNum, int messageNum)`. Users of libOUM must ensure that all OUM receivers begin listening to the new session in a consistent state.

4 OUM Design and Implementation

We implement OUM in the context of a single data center network. The basic design is straightforward: the network routes all packets destined for a given OUM group through a single *sequencer*, a low-latency device that serves one purpose: to add a sequence number to each packet before forwarding it to its destination. Since all packets have been marked with a sequence number, the libOUM library can ensure ordering by discarding messages that are received out of order and detect and report dropped messages by noticing gaps in the sequence number.

Achieving this design poses three challenges. First, the network must serialize all requests through the sequencer; we use software-defined networking (SDN) to provide this *network serialization* (Section 4.1). Second, we must implement a sequencer capable of high throughput and low latency. We present three such implementations in Section 4.2: a zero-additional-latency implementation for programmable data center switches, a middlebox-like prototype using a network processor, and a pure-software implementation. Finally, the system must remain robust to failures of network components, including the sequencer (Section 4.3).

4.1 Network Serialization

The first aspect of our design is *network serialization*, where all OUM packets for a particular group are routed through a sequencer on the common path. Network serialization was previously used to implement a best-effort multicast [56]; we adapt that design here.

Our design targets a data center that uses software-defined networking, as is common today. Data center networks are engineered networks constructed with a particular topology – generally some variant of a multi-rooted tree. A traditional design calls for a three-level tree topology where many top-of-rack switches, each connecting to a few dozen server, are interconnected via aggregation switches that themselves connect through core switches. More sophisticated topologies, such as fat-tree or Clos networks [2,23,43,50] extend this basic design to support large numbers of physical machines using many commodity switches and often provide full bisection bandwidth.

Software-defined networking additionally allows the data center network to be managed by a central controller. This controller can install custom forwarding, filtering, and rewriting rules in switches. The current generation of SDN switches, e.g., OpenFlow [47], allow these rules to be installed at a per-flow granularity, matching on a fixed set of packet headers.

To implement network serialization, we assign each OUM group a distinct address in the data center network that senders can use to address messages to the group. The SDN controller installs forwarding rules for this address that route messages through the sequencer, then to group members.

To do this, the controller must select a sequencer for each group. In the most efficient design, switches themselves are used as sequencers. In this case, the controller selects a switch that is a common ancestor of all destination nodes in the tree hierarchy to avoid increasing path lengths, e.g., a root switch or an aggregation switch if all receivers are in the same subtree.

4.2 Implementing the Sequencer

The sequencer plays a simple but critical role: assigning a sequence number to each message destined for a particular OUM group, and writing that sequence number into the packet header. This establishes a total order over packets and is the key element that elevates our design from a best-effort ordering property to an ordering guarantee. Even if packets are dropped (e.g., due to congestion or link failures) or reordered (e.g., due to multipath effects) in the network, receivers can use the sequence numbers to ensure that they process packets in order and deliver drop notifications for missing packets.

Sequencers maintain one counter per OUM group. For every packet destined for that group, they increment the counter and write it into a designated field in the packet header. The counter must be incremented by 1 on each packet (as opposed to a timestamp, which monotonically increases but may have gaps). This counter lets libOUM return DROP-NOTIFICATIONs when it notices gaps in the sequence numbers of incoming messages. Sequencers also maintain and write into each packet the OUM *session number* that is used to handle sequencer failures; we describe its use in Section 4.3.

Our sequencer design is general; we offer three possible deployment options:

1. The most efficient design targets upcoming programmable network switches (e.g., Reconfigurable Match Tables [11], Intel's FlexPipe [53], and Cavium's XPliant [68]), using the switch itself as the sequencer, incurring minimal latency cost. The sequencer functionality can be implemented in high-level languages such as P4 [10] and deployed onto the switch itself. Example P4 code can be found in [41].
2. As this hardware is not yet available, we also suggest a design using a network processor to implement a middlebox-like sequencer. This is how our prototype sequencer was implemented.
3. Lastly, an end-host itself can be used as a sequencer, though at the cost of added latency.

Sequencer Scalability Since all OUM packets for a particular group go through the sequencer, a valid concern is whether the sequencer will become the performance bottleneck. Switches and network processors are designed to process packets at line rate and thus will not become the bottleneck for a single OUM group (group receivers are already limited by the link bandwidth). Previous work [29] has demonstrated that an end-host sequencer using RDMA can process close to 100 million requests per second, many more than any single OUM group can process. We note that different OUM groups need not share a sequencer, and therefore deployment of multiple OUM groups can scale horizontally.

4.3 Fault Tolerance

Designating a sequencer and placing it on the common path for all messages to a particular group introduces an obvious challenge: what if it fails or becomes unreachable? If link failures or failures of other switches render the sequencer unreachable, local rerouting mechanisms may be able to identify an alternate path [43]. However, if the sequencer itself fails, or local rerouting is not possible, replacing the sequencer becomes necessary.

In our design, the network controller monitors the sequencer's availability. If it fails or no longer has a path to all OUM group members, the controller selects a different switch. It reconfigures the network to use this new sequencer by updating routes in other switches. During

the reconfiguration period, multicast messages may not be delivered. However, failures of root switches happen infrequently [21], and rerouting can be completed within a few milliseconds [43], so this should not significantly affect system availability.

We must also ensure that the ordering guarantee of multicast messages is robust to sequencer failures. This requires the continuous, ordered assignment of sequence numbers even when the network controller fails over to a new sequencer.

To address this, we introduce a unique, monotonically increasing *session number*, incremented each time sequencer failover occurs. When the controller detects a sequencer failure, it updates the forwarding rules and contacts the new sequencer to set its local session number to the appropriate value. As a result, the total order of messages follows the lexicographical order of the ⟨*session-number*, *sequence-number*⟩ tuple, and clients can still discard packets received out of order.

Once libOUM receives a message with a session number higher than the receiver is listening for, it realizes that a new sequencer is active and stops delivering messages from the old session. However, libOUM does not know if it missed any packets from the old sequencer. As a result, it cannot deliver DROP-NOTIFICATIONs during a session change. Instead, it delivers a SESSION-TERMINATED notification, exposing this uncertainty to the application. NOPaxos, for example, resolves this by executing a view change (Section 5.2.3) so that replicas agree on exactly which requests were received in the old session.

The network controller must ensure that session numbers for any given group monotonically increase, even across controller failures. Many design options are available, for example using timestamps as session numbers, or recording session numbers in stable or replicated storage. Our implementation uses a Paxos-replicated controller group, since SDN controller replication is already common in practice [26, 33]. We note that our replication protocol, NOPaxos (Section 5), is completely decoupled from controller replication, and the controller updates only on sequencer failures, not for every NOPaxos request.

5 NOPaxos

NOPaxos, or Network-Ordered Paxos, is a new replication protocol which leverages the Ordered Unreliable Multicast sessions provided by the network layer.

5.1 Model

NOPaxos replicas communicate over an asynchronous network that provides OUM sessions (via libOUM). NOPaxos requires the network to provide ordered but unreliable delivery of multicast messages within a session. In the normal case, these messages are delivered sequentially and are not dropped; however, it remains robust to dropped packets (presented as DROP-NOTIFICATION through libOUM). NOPaxos is also robust to SESSION-TERMINATED notifications that occur if the sequencer fails. These network anomalies do not affect NOPaxos's safety guarantees; [41] discusses how they affect NOPaxos's performance.

NOPaxos assumes a crash failure model. It uses $2f+1$ replicas, where f replicas are allowed to fail. However, in the presence of more than f failures, the system still guarantees safety. Furthermore, NOPaxos guarantees safety even in an asynchronous network with no bound on message latency (provided the OUM guarantees continue to hold).

NOPaxos provides linearizability of client requests. It provides at-most-once semantics using the standard mechanism of maintaining a table of the most recent request from each client [42].

5.2 Protocol

Overview. NOPaxos is built on top of the guarantees of the OUM network primitive. During a single OUM session, REQUESTs broadcast to the replicas are totally ordered but can be dropped. As a result, the replicas have to agree only on which REQUESTs to execute and which to permanently ignore, a simpler task than agreeing on the order of requests. Conceptually, this is equivalent to running multiple rounds of *binary consensus*. However, NOPaxos must explicitly run this consensus only when DROP-NOTIFICATIONs are received. To switch OUM sessions (in the case of sequencer failure), the replicas must agree on the contents of their shared log before they start listening to the new session.

To these ends, NOPaxos uses a view-based approach: each view has a single OUM *session-num* and a single replica acting as *leader*. The leader executes requests and drives the agreement to skip a dropped request. That is, it decides which of the sequencer's REQUESTs to ignore and treat as NO-OPs. The view ID is a tuple ⟨*leader-num*, *session-num*⟩. Here, *leader-num* is incremented each time a new leader is needed; the current leader of any view is *leader-num* $\pmod n$; and *session-num* is the latest session ID from libOUM. View IDs in NOPaxos are partially ordered.[2] However, the IDs of all views that successfully start will be comparable.

In the normal case, the replicas receive a REQUEST from libOUM. The replicas then reply to the client, the leader replying with the result of the REQUEST, so the client's REQUEST is processed in only a single round-trip. NOPaxos uses a single round-trip in the normal case because, like many speculative protocols, the client checks the durability of requests. However, unlike most speculative protocols, NOPaxos clients have a guarantee regard-

[2] That is, $v_1 \leq v_2$ iff both v_1's *leader-num* and *session-num* are less than or equal to v_2's.

Replica:

- *replica-num* — replica number
- *status* — one of `Normal` or `ViewChange`
- *view-id* = ⟨*leader-num*, *session-num*⟩ — the view number, a tuple of the current leader number and OUM session number, partially ordered, initially ⟨0, 0⟩
- *session-msg-num* — the number of messages (REQUESTs or DROP-NOTIFICATIONs) received in this OUM session
- *log* — client REQUESTs and NO-OPs in sequential order
- *sync-point* — the latest synchronization point

Figure 3: Local state of NOPaxos replicas.

ing ordering of operations; they need only check that the operation was received.

When replicas receive a DROP-NOTIFICATION from libOUM, they first try to recover the missing REQUEST from each other. Failing that, the leader initiates a round of agreement to commit a NO-OP into the corresponding slot in the log. Finally, NOPaxos uses a view change protocol to handle leader failures and OUM session termination while maintaining consistency.

Outline. NOPaxos consists of four subprotocols:

- *Normal Operations* (Section 5.2.1): NOPaxos processes client REQUESTs in a single round-trip in the normal case.
- *Gap Agreement* (Section 5.2.2): NOPaxos ensures correctness in the face of DROP-NOTIFICATIONs by having the replicas reach agreement on which sequence numbers should be permanently dropped.
- *View Change* (Section 5.2.3): NOPaxos ensures correctness in the face of leader failures or OUM session termination using a variation of a standard view change protocol.
- *Synchronization* (Section 5.2.4): Periodically, the leader synchronizes the logs of all replicas.

Figure 3 illustrates the state maintained at each NOPaxos replica. Replicas tag all messages sent to each other with their current *view-id*, and while in the Normal Operations, Gap Agreement, and Synchronization subprotocols, *replicas ignore all messages from different views*. Only in the View Change protocol do replicas with different *view-id*s communicate.

5.2.1 Normal Operations

In the normal case when replicas receive REQUESTs instead of DROP-NOTIFICATIONs, client requests are committed and executed in a single phase. Clients broadcast ⟨REQUEST, *op*, *request-id*⟩ to all replicas through libOUM, where *op* is the operation they want to execute, and *request-id* is a unique id used to match requests and their responses.

When each replica receives the client's REQUEST, it increments *session-msg-num* and appends *op* to the log. If the replica is the leader of the current view, it executes the *op* (or looks up the previous result if it is a duplicate of a completed request). Each replica then replies to the client with ⟨REPLY, *view-id*, *log-slot-num*, *request-id*, *result*⟩, where *log-slot-num* is the index of *op* in the log. If the replica is the leader, it includes the *result* of the operation; NULL otherwise.

The client waits for REPLYs to the REQUEST with matching *view-id*s and *log-slot-num*s from $f+1$ replicas, where one of those replicas is the leader of the view. This indicates that the request will remain persistent even across view changes. If the client does not receive the required REPLYs within a timeout, it retries the request.

5.2.2 Gap Agreement

NOPaxos replicas always process operations in order. When a replica receives a DROP-NOTIFICATION from libOUM (and increments its *session-msg-num*), it must either recover the contents of the missing request or prevent it from succeeding before moving on to subsequent requests. Non-leader replicas do this by contacting the leader for a copy of the request. If the leader itself receives a DROP-NOTIFICATION, it coordinates to commit a NO-OP operation in place of that request:

1. If the leader receives a DROP-NOTIFICATION, it inserts a NO-OP into its *log* and sends a ⟨GAP-COMMIT, *log-slot*⟩ to the other replicas, where *log-slot* is the slot into which the NO-OP was inserted.
2. When a non-leader replica receives the GAP-COMMIT and has filled all log slots up to the one specified by the leader,[3] it inserts a NO-OP into its *log* at the specified location[4] (possibly overwriting a REQUEST) and replies to the leader with a ⟨GAP-COMMIT-REP, *log-slot*⟩.
3. The leader waits for f GAP-COMMIT-REPs (retrying if necessary).

Clients need not be notified explicitly when a NO-OP has been committed in place of one of their requests. They simply retry their request after failing to receive a quorum of responses. Note that the retried operation will be considered a new request and will have a new slot in the replicas' logs. Replicas identify duplicate client requests by checking if they have processed another request with

[3] It is not strictly necessary that all previous log slots are filled. However, care must be taken to maintain consistency between replicas' *log*s and the OUM session

[4] If the replica had not already filled *log-slot* in its log or received a DROP-NOTIFICATION for that slot when it inserted the NO-OP, it ignores the next REQUEST or DROP-NOTIFICATION from libOUM (and increments *session-msg-num*), maintaining consistency between its position in the OUM session and its log.

the same *client-id* and *request-id*, as is commonly done in other protocols.

This protocol ensures correctness because clients do not consider an operation completed until they receive a response from the leader, so the leader can propose a NO-OP regardless of whether the other replicas received the REQUEST. However, before proceeding to the next sequence number, the leader must ensure that a majority of replicas have learned of its decision to commit a NO-OP. When combined with the view change protocol, this ensures that the decision persists even if the leader fails.

As an optimization, the leader can first try to contact the other replicas to obtain a copy of the REQUEST and initiate the gap commit protocol only if no replicas respond before a timeout. While not necessary for correctness, this reduces the number of NO-OPs.

5.2.3 View Change

During each view, a NOPaxos group has a particular leader and OUM session number. NOPaxos must perform view changes to ensure progress in two cases: (1) when the leader is suspected of having failed (e.g, by failing to respond to pings), or (2) when a replica detects the end of an OUM session. To successfully replace the leader or move to a new OUM session, NOPaxos runs a view change protocol. This protocol ensures that all successful operations from the old view are carried over into the new view and that all replicas start the new view in a consistent state.

NOPaxos's view change protocol resembles that used in Viewstamped Replication [42]. The principal difference is that NOPaxos views serve two purposes, and so NOPaxos view IDs are therefore a tuple of ⟨*leader-num*, *session-num*⟩ rather than a simple integer. A view change can increment either one. However, NOPaxos ensures that each replica's *leader-num* and *session-num* never go backwards. This maintains a total order over all views that successfully start.

1. A replica initiates a view change when: (1) it suspects that the leader in its current view has failed; (2) it receives a SESSION-TERMINATED notification from libOUM; or (3) it receives a VIEW-CHANGE or VIEW-CHANGE-REQ message from another replica with a higher *leader-num* or *session-num*. In all cases, the replica appropriately increments the *leader-num* and/or *session-num* in its *view-id* and sets its *status* to `ViewChange`. If the replica incremented its *session-num*, it resets its *session-msg-num* to 0.

   It then sends ⟨VIEW-CHANGE-REQ, *view-id*⟩ to the other replicas and ⟨VIEW-CHANGE, *view-id*, v', *session-msg-num*, *log*⟩ to the leader of the new view, where v' is the view ID of the last view in which its *status* was `Normal`. While in `ViewChange` status, the replica ignores all replica-to-replica messages (except START-VIEW, VIEW-CHANGE, and VIEW-CHANGE-REQ).

   If the replica ever times out waiting for the view change to complete, it simply rebroadcasts the VIEW-CHANGE and VIEW-CHANGE-REQ messages.

2. When the leader for the new view receives $f+1$ VIEW-CHANGE messages (including one from itself) with matching *view-id*s, it performs the following steps:

   - The leader merges the *log*s *from the most recent (largest) view* in which the replicas had *status* `Normal`.[5] For each slot in the log, the merged result is a NO-OP if any log has a NO-OP. Otherwise, the result is a REQUEST if at least one has a REQUEST. It then updates its *log* to the merged one.
   - The leader sets its *view-id* to the one from the VIEW-CHANGE messages and its *session-msg-num* to the highest out of all the messages used to form the merged log.
   - It then sends ⟨START-VIEW, *view-id*, *session-msg-num*, *log*⟩ to all replicas (including itself).

3. When a replica receives a START-VIEW message with a *view-id* greater than or equal to its current *view-id*, it first updates its *view-id*, *log*, and *session-msg-num* to the new values. It then calls `listen(session-num, session-msg-num)` in libOUM. The replica sends REPLYs to clients for all new REQUESTs added to its log (executing them if the replica is the new leader). Finally, the replica sets its *status* to `Normal` and begins receiving messages from libOUM again.[6]

5.2.4 Synchronization

During any view, only the leader executes operations and provides results. Thus, all successful client REQUESTs are committed on a *stable log* at the leader, which contains only persistent client REQUESTs. In contrast, non-leader replicas might have *speculative* operations throughout their logs. If the leader crashes, the view change protocol ensures that the new leader first recreates the stable log of successful operations. However, it must then execute all operations before it can process new ones. While this protocol is correct, it is clearly inefficient.

Therefore, as an optimization, NOPaxos periodically executes a synchronization protocol in the background.

[5] While *view-id*s are only partially ordered, because individual replicas' *view-id*s only increase and views require a quorum of replicas to start, all views that successfully start *are* comparable – so identifying the view with the highest number is in fact meaningful. For a full proof of this fact, see Appendix A.

[6] Replicas also send an acknowledgment to the leader's START-VIEW message, and the leader periodically resends the START-VIEW to those replicas from whom it has yet to receive an acknowledgment.

This protocol ensures that all other replicas learn which operations have successfully completed and which the leader has replaced with NO-OPs. That is, synchronization ensures that all replicas' logs are stable up to their *sync-point* and that they can safely execute all REQUESTs up to this point in the background.

For brevity, we omit the details of the Synchronization protocol here; they can be found in [41].

5.2.5 Recovery and Reconfiguration

While the NOPaxos protocol as presented above assumes a crash failure model and a fixed replica group, it can also facilitate recovery and reconfiguration using adaptations of standard mechanisms (e.g. Viewstamped Replication [42]). While the recovery mechanism is a direct equivalent of the Viewstamped Replication protocol, the reconfiguration protocol additionally requires a membership change in the OUM group. The OUM membership is changed by contacting the controller and having it install new forwarding rules for the new members, as well as a new *session-num* in the sequencer (terminating the old session). The protocol then ensures all members of the new configuration start in a consistent state.

5.3 Benefits of NOPaxos

NOPaxos achieves the theoretical minimum latency and maximum throughput: it can execute operations in *one round-trip* from the client to the replicas and does not require replicas to coordinate on each request. By relying on the network to stamp requests with sequence numbers, it requires replies only from a *simple majority* of replicas and uses a *cheaper* and *rollback-free* mechanism to correctly account for network anomalies.

The OUM session guarantees mean that the replicas already agree on the ordering of all operations. As a consequence, clients need not wait for a superquorum of replicas to reply, as in Fast Paxos and Speculative Paxos (and as is required by any protocol that provides fewer message delays than Paxos in an asynchronous, unordered network [40]). In NOPaxos, a simple majority of replicas suffices to guarantee the durability of a REQUEST in the replicas' shared log.

Additionally, the OUM guarantees enable NOPaxos to avoid expensive mechanisms needed to detect when replicas are not in the same state, such as using hashing to detect conflicting logs from replicas. To keep the replicas' logs consistent, the leader need only coordinate with the other replicas when it receives DROP-NOTIFICATIONs. Committing a NO-OP takes but a single round-trip and requires no expensive reconciliation protocol.

NOPaxos also avoids rollback, which is usually necessary in speculative protocols. It does so not by coordinating on every operation, as in non-speculative protocols, but by having only the leader execute operations. Non-leader replicas do not execute requests during normal operations (except, as an optimization, when the synchronization protocol indicates it is safe to do so), so they need not rollback. The leader executes operations speculatively, without coordinating with the other replicas, but clients do not accept a leader's response unless it is supported by matching responses from f other replicas. The only rare case when a replica will execute an operation that is not eventually committed is if a functioning leader is incorrectly replaced through a view change, losing some operations it executed. Because this case is rare, it is reasonable to handle it by having the ousted leader transfer application state from another replica, rather than application-level rollback.

Finally, unlike many replication protocols, NOPaxos replicas send and receive a constant number of messages for each REQUEST in the normal case, irrespective of the total number of replicas. This means that NOPaxos can be deployed with an increasing number of replicas without the typical performance degradation, allowing for greater fault-tolerance.

5.4 Correctness

NOPaxos guarantees *linearizability*: that operations submitted by multiple concurrent clients appear to be executed by a single, correct machine. In a sense, correctness in NOPaxos is a much simpler property than in other systems, such as Paxos and Viewstamped Replication [36, 52], because the replicas need not agree on the order of the REQUESTs they execute. Since the REQUEST order is already provided by the guarantees of OUM sessions, the replicas must only agree on *which* REQUESTs to execute and which REQUESTs to drop.

Below, we sketch the proof of correctness for the NOPaxos protocol. For a full, detailed proof, see Appendix A. Additionally, see Appendix C for a TLA+ specification of the NOPaxos protocol.

Definitions. We say that a REQUEST or NO-OP is *committed* in a log slot if it is processed by $f+1$ replicas with matching *view-id*s, including the leader of that view. We say that a REQUEST is *successful* if it is committed and the client receives the $f+1$ suitable REPLYs. We say a log is *stable* in view v if it will be a prefix of the log of every replica in views higher than v.

Sketch of Proof. During a view, a leader's log grows monotonically (i.e., entries are only appended and never overwritten). Also, leaders execute only the first of duplicate REQUESTs. Therefore, to prove linearizability it is sufficient to show that: (1) every successful operation was appended to a stable log at the leader and that the resulting log is also stable, and (2) replicas always start a view listening to the correct *session-msg-num* in an OUM session (i.e., the message corresponding to the number of

REQUESTs or NO-OPs committed in that OUM session).

First, note that any REQUEST or NO-OP that is committed in a log slot will stay in that log slot for all future views: it takes $f+1$ replicas to commit a view and $f+1$ replicas to complete a view change, so, by quorum intersection, at least one replica initiating the view change will have received the REQUEST or NO-OP. Also, because it takes the leader to commit a REQUEST or NO-OP and its log grows monotonically, only a single REQUEST or NO-OP is ever committed in the same slot during a view. Therefore, any log consisting of only committed REQUESTs and NO-OPs is stable.

Next, every view that starts (i.e., $f+1$ replicas receive the START-VIEW and enter `Normal` status) trivially starts with a log containing only committed REQUESTs and NO-OPs. Replicas send REPLYs to a REQUEST only after all log slots before the REQUEST's slot have been filled with REQUESTs or NO-OPs; further, a replica inserts a NO-OP only if the leader already inserted that NO-OP. Therefore, if a REQUEST is committed, all previous REQUESTs and NO-OPs in the leader's log were already committed.

This means that any REQUEST that is successful in a view must have been appended to a stable log at the leader, and the resulting log must also be stable, showing (1). To see that (2) is true, notice that the last entry in the combined *log* formed during a view change and the *session-msg-num* are taken from the same replica(s) and therefore must be consistent.

NOPaxos also guarantees *liveness* given a sufficient amount of time during which the following properties hold: the network over which the replicas communicate is fair-lossy; there is some bound on the relative processing speeds of replicas; there is a quorum of replicas that stays up; there is a replica that stays up that no replica suspects of having failed; all replicas correctly suspect crashed nodes of having failed; no replica receives a DROP-NOTIFICATION or SESSION-TERMINATED from libOUM; and clients' REQUESTs eventually get delivered through libOUM.

6 Distributed Transaction Processing

Next, we consider storage systems that are partitioned for scalability and replicated for fault-tolerance. Data is partitioned among different *shards*, and each shard contains multiple replicas that hold copies of the data associated with that shard. Clients (which are generally higher-level application servers) submit transactions to be processed. We limit ourselves here to systems where all nodes are located in the same datacenter, although some of the ideas of our protocol, Eris, could also be applied to geodistributed systems.

A transactional storage system should provide several guarantees:

- **atomicity**: every transaction is applied to all shards it affects (or none at all)
- strict serializable **isolation**: the execution is identical to each transaction being executed in sequence on a single machine
- **fault-tolerance**: these guarantees continue to hold even when some of the nodes in any shard might fail

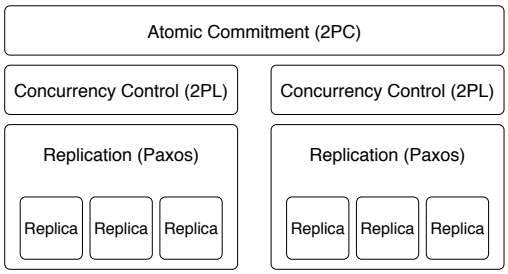

Figure 4: Standard layered architecture for a partitioned replicated storage system

Existing systems generally achieve these goals using a layered approach, as shown in Figure 4. A replication protocol (e.g., Paxos [37]) is used to provide 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 may differ, a layered architecture like this is used by many existing systems [1, 4, 15, 17, 19, 22, 35, 46].

A consequence of this architecture is that coordinating a single transaction commit requires multiple rounds of coordination. As an example, Figure 5 shows the protocol exchange required to commit a transaction in a conventional layered architecture like Google's Spanner [17]. Each of the phases of the two-phase commit protocol requires synchronously executing a replication protocol to ensure that the transaction coordination decision is replicated to each replica within a shard. Moreover, two-phase locking requires that locks are held during the period between prepare and commit operations, preventing conflicting transactions from executing. This combination seriously impacts the performance of these systems.

6.1 Eris Architecture

Eris divides the responsibility for different guarantees in a new way, enabling it to execute many transactions without coordination. Rather than use separate protocols for ordering transactions across shards and operations within a shard, Eris uses a unified protocol that is responsible for executing transactions in the correct order at each replica of all participating shards. The protocol itself is divided into three layers, as shown in Figure 6:

1. The *network multi-sequenced groupcast layer* (Sec-

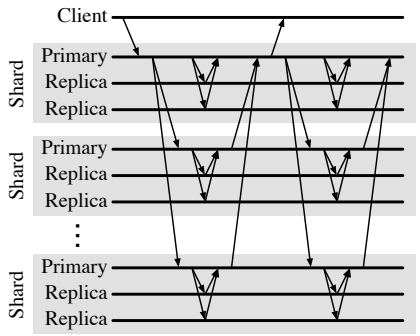

Figure 5: Coordination required to commit a single transaction with traditional two-phase commit and replication

tion 7) 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 8.1) adds **reliability** and **atomicity** to the ordered operations, ensuring that each transaction is either executed at all shards or none. This combination of ordering and reliability is sufficient to guarantee linearizability for an important class of transactions.

3. The *general transaction layer* (Section 8.2) provides **isolation** for concurrent execution of fully general transactions, building them atop the linearizable ordering of independent transactions.

The base Eris protocol is responsible for establishing a linearizable order of transactions, the core problem in distributed transaction processing. This part of the protocol processes *independent transactions* [19], a restricted transaction model where transactions execute single-shot computation atomically across shards, but have no inter-shard data dependencies (we make this definition precise in the following section). Atop the independent transaction processing layer, Eris can handle fully general transactions by translating them into multiple independent transactions.

Eris establishes a linearizable order of independent transactions using a network-integrated protocol. Following lessons learned from OUM and NOPaxos, this protocol is divided into two parts. A network-level component, the *multi-sequencer*, is responsible for *ordering*, but not *reliability*. That is, it assigns a global order to transactions, but does not guarantee that every shard, or every replica in a shard, receives all the transactions it should execute. The Eris application-level protocol is responsible for *reliability*: it detects when transactions have been delivered out of sequence by the messaging layer or lost entirely, and ensures that every transaction is eventually executed at every participant (or fails entirely).

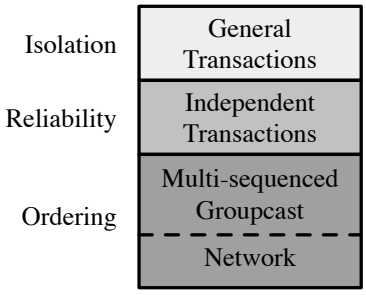

Figure 6: The layers of Eris and the guarantees they provide

6.2 Independent Transactions

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. However, 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 the execution. Each shard must independently come to the same `Commit` or `Abort` decision without any coordination (e.g. by always committing). This definition of independent transactions is taken from Granola [19]; essentially the same definition was previously proposed by the authors of H-Store [62] under the name "strongly two-phase".

Although the independent transaction model is restrictive, it captures a common class 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, if data is replicated across different shards (as is common for frequently accessed data), it can be updated consistently with an independent transaction. Prior work has argued that many applications consist largely or entirely of independent transactions [18, 19, 62]. As one example, TPC-C [66], an industry standard benchmark designed to represent transaction processing workloads, can be expressed entirely using independent transactions, despite its complexity [62].

General transactions provide a standard interactive transaction model. Clients begin a transaction, then execute a sequence operations (reads and writes) at different shards; each operation 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. As discussed in Section 8.2, Eris supports general transactions, albeit with lower performance than independent transac-

tions.

7 Multi-Sequenced Groupcast

We seek to provide ordering guarantees in Eris at the network level as in NOPaxos. However, straightforward application of OUM will not work. To this end, we present the multi-sequenced groupcast primitive. A groupcast in Eris is an extended multicast primitive that allows a message to be delivered to a client-specified set of multicast groups (i.e., a set of Eris shards), while multi-sequenced groupcast provides similar guarantees to OUM in the groupcast setting. In fact, the only guarantee which differs is that the ordering property states that all messages sent through multi-sequenced groupcast are totally ordered and received in the corresponding order (no matter which groups are involved).

7.1 Implementing Multi-sequenced Groupcast Design

At a high level, our design for multi-sequenced groupcast follows the same strategy as using network-level sequencing to implement OUM. One *sequencer* is designated for the system at any time; depending on implementation, this 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 mentioned above, appending a single sequence number as in OUM 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 mechanism, the *multi-stamp*.

A multi-stamp consists of a sequence of $\langle$*group-id*, *sequence-num*$\rangle$ pairs, one for each destination group of the message. In order to apply multi-stamps, a sequencer must maintain 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, then writes the sequence of counters into the packet header as a multi-stamp.

Fault Tolerance and Epochs. Multi-sequenced groupcast requires the sequencer to keep state: the latest sequence number for each destination group. Of course, sequencers can fail. As in OUM, we expose these failures to the application; here, we term the monotonic ID assigned to sequencers an *epoch number*.

When a receiver receives a multi-sequenced groupcast message with a higher epoch number than it has processed previously, it delivers NEW-EPOCH notification to the application code (i.e., Eris). This notifies the application that some number of packets might have been dropped; the application is responsible for determining which packets from the previous epoch have been successfully delivered and transitioning the system to receiving multi-sequenced groupcast messages in the new epoch *in a consistent state*.

8 Eris

We now describe how to use the underlying multi-sequenced groupcast primitive to build the independent transaction processing and general transaction layers of Eris.

8.1 Processing Independent Transactions

The multi-sequenced groupcast provided by the network establishes an order over all transactions, but does not guarantee reliable delivery. The Eris independent transaction protocol builds reliable delivery semantics at the application layer in order to provide linearizable execution of independent transactions. Like NOPaxos, Eris replicas can process transactions in a single round trip from the client.

Eris uses a quorum-based protocol: in order to maintain availability even when a minority of replicas in a shard fails, clients in Eris only wait for replies from a simple majority from each shard in a transaction. However, Eris has a single *designated learner* in each shard, rather than the "leader" in NOPaxos. Only this replica actually executes transactions; the other replicas simply log the requests they receive. As a result, Eris requires that clients wait for a response from the designated learner before considering a quorum complete; however, the designated learner can be replaced if it fails.

Eris must be resilient to replica failures (in particular, the failure of designated learners) as well as network anomalies. In our multi-sequencing abstraction, these anomalies consist of DROP-NOTIFICATIONs (indicating that a multi-sequenced groupcast transaction was dropped or reordered in the network) and NEW-EPOCH notifications (indicating that the current sequencer has been replaced). In Eris, failure of the designated learner is handled entirely within the shard by a protocol similar to NOPaxos'. DROP-NOTIFICATIONs and NEW-EPOCH notifications, however, require coordination across *all shards*. In the case of DROP-NOTIFICATIONs, all participant shards for the dropped transaction must reach the same decision about whether or not to discard the message. And in the case of 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 replicated ser-

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 designated learner
- *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 7: Local state of Eris replicas used for independent transaction processing

vice[7] which communicates with the replicas to recover from DROP-NOTIFICATIONs and NEW-EPOCH notifications. The FC is replicated using standard means [36, 41, 52]. Using another replicated service as part of failure recovery might raise the question: doesn't this just push the need for coordination – and its associated overhead – to a different place? In a sense, the answer to that question is "yes." However, what Eris achieves with the introduction of the FC is moving the overhead of replication and atomic commitment out of the normal path and into the failure case. The system then incurs this overhead – both in terms of throughput and latency penalties – in rare cases.

The state maintained by replicas in summarized in Figure 7. Two important pieces of state are the *view-num* and *epoch-num*, which track the current designated learner[8] and multi-sequencing epoch. The former is necessary so that the clients and FC can communicate with the *current* designated learner, and the latter is used to ensure that replicas achieve a consistent state after a sequencer failure before processing requests from the next sequencer. Eris replicas and the FC tag all intra-Eris messages with their current *epoch-num* and do not accept messages from previous epochs. If a replica ever receives a message from a later epoch, it must use the FC to transition to the new epoch before continuing.

Below we go into more detail about all four subprotocols of Eris's independent transaction processing layer. Throughout these protocols, messages that are sent but not acknowledged with the proper reply are retried.

[7] Having the FC as a separate entity is not strictly necessary. Its job could be done by the replicas themselves. However, this would involve schemes such as global leader election across the entire system and result in unnecessary complexity as well as significant load on the replicas.

[8] The designated learner corresponding to *view-num* v is the replica with *replica-num* $\equiv v \pmod N$, where N is the number of replicas in the shard.

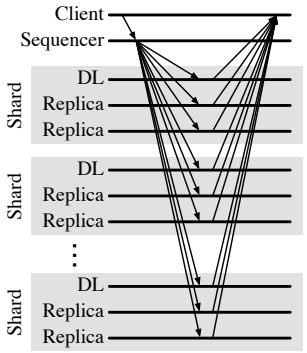

Figure 8: Communication pattern of Eris in the normal case, where the independent transaction is sent via multi-sequenced groupcast to multiple shards, each consisting of 3 replicas (one replica in each shard is a Designated Learner)

8.1.1 Normal Case

In the normal case, when a replica receives an independent transaction from the multi-sequencing layer, rather than a DROP-NOTIFICATION, independent transactions are processed in a single round trip from the client to the replicas, in a way similar to NOPaxos. Figure 8 illustrates this process. The only difference is that in Eris, the client must wait for a reply from every participant shard. We omit the details of the protocol here; they can be found in Appendix B.

8.1.2 Dropped Messages

When a replica receives a DROP-NOTIFICATION, indicating that it missed some message intended for its shard because of a network anomaly, it must either learn the missing transaction, or ensure that it does not succeed at any other replicas. This process is coordinated through the FC, which contacts the other nodes in the system in an attempt to recover the missing transaction. If this fails, it 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 independent 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*.
3. When a replica receives TXN-REQUEST, if it has received a transaction matching *txn-id* already, it replies with ⟨HAS-TXN, *txn*⟩. Otherwise, it adds *txn-id* to *temp-drops* and replies with ⟨TEMP-DROPPED-TXN, *view-num*, *txn-id*⟩.
4. The FC waits for either a *single* HAS-TXN or *a quorum*

of TEMP-DROPPED-TXNs from every shard, whichever comes first. As in Section 8.1.1, receiving a quorum from a shard requires a TEMP-DROPPED-TXN response from a majority of the shard's replicas, where all have matching *view-num*s and where one is from the designated learner for the view.

(a) If the FC first receives the HAS-TXN, it saves it and sends ⟨TXN-FOUND, *txn*⟩ to all participants in the transaction.

(b) If the FC first receives the necessary TEMP-DROPPED-TXNs, 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 coordinator, it does the following:

(a) If it receives a TXN-FOUND, it adds the transaction to its *un-drops*, adding the transaction to the log and replying to the client if the replica was blocked waiting for this sequence number.

(b) If it receives a TXN-DROPPED, it adds the *txn-id* to *perm-drops*, adding a NO-OP to its *log* if it was blocked waiting for this sequence number (or possibly replacing an earlier transaction with a NO-OP in the *log* for non-designated-learner replicas).

8.1.3 Designated Learner Failure

Because only the designated learner executes transactions in Eris and the ability to make progress is dependent on each shard having a designated learner, Eris must have a way to replace a designated learner if it fails. To ensure the system remains correct, the new designated learner must learn about all transactions committed in previous views and about the TEMP-DROPPED-TXNs sent by a majority in the previous views.

This protocol is straightforward and similar to NOPaxos. Again, we relegate the full details to Appendix B.

8.1.4 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. The FC is responsible for ensuring that all replicas across all shards start in the new epoch in *consistent states* (i.e., 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 multi-sequenced groupcast layer (indicating a sequencer failover) it sends ⟨EPOCH-CHANGE-REQ, *epoch-num*⟩ to the FC.

2. Whenever the FC receives a EPOCH-CHANGE-REQ from a replica (one from the FC's current epoch), the FC increments its *epoch-num* 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*, *view-num*, *log*⟩ back to the FC.

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. It then determines if there are any gaps 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 in case it needs to resend them.

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 optionally clears its *temp-drops*, *perm-drops*, and *un-drops*. It then sets its *status* to `Normal` and begins listening for multi-sequenced groupcast messages in the new epoch.

8.1.5 Correctness

Eris provides linearizability of independent transactions. We briefly sketch a proof of correctness below:

First of all, we consider a single epoch and a single view in a shard in which there is a single designated learner (recognized by a majority of replicas). During the view, the designated learner only adds items (transactions and NO-OPs) to its *log*; it never replaces them because the FC waits for a reply from the designated learner before committing a NO-OP for any *txn-id*. Furthermore, it adds them in the exact order provided by the multi-sequencing layer, where exactly one transaction or NO-OP is added to its *log* for each transaction or NO-OP received.

Because the FC waits for promises from *all* shards before committing a NO-OP for any *txn-id*, we know that if any designated learner in any shard commits a NO-OP in place of a transaction, then all will.

When a view change happens, the new learner is guaranteed to learn about any commitments (either successfully processed transactions or tentative drop promises

made to the FC) made during the previous view. Because the FC is replicated, the new designated learner can reliably query the FC if any decision about whether a NO-OP was committed for a transaction is unknown, and that decision will be the same one made across all shards.

Finally, when an epoch change happens, because the FC reads from a majority of replicas from each shard, it is guaranteed to learn about any transaction which for which there was a successful reply to a client. Furthermore, the FC already knows for which transactions NO-OPs were committed. It then starts replicas in the new epoch in a consistent state by ensuring that no shard has a transaction that the other participant shards do not have.

Taken together, what this means is that, from the clients' perspective, there exists some subset of the independent transaction multi-sequenced groupcast to the replicas, and each shard processes all of the transactions in this subset transactions in which it was a participant, in the exact order give by the multi-sequencing layer (i.e., lexicographic, epoch number major order). Therefore, the independent transaction processing layer of Eris is linearizable.

8.2 Processing General Transactions

The base Eris protocol supports independent transactions. As already mentioned, a large class of important operations are expressible as independent transactions. However, not all operations are expressible as independent transactions. One example is a conditional update that depends on information stored on another shard, e.g., a banking transaction which moves funds from one account to another only if the source account has sufficient funds (assuming the two accounts are stored on different shards). In many cases, these types of operations can be avoided through careful partitioning [19]. However, to allow Eris to support all workloads, we describe an extension to the protocol to support fully general transactions.

Eris can support general transactions by building them up using multiple independent transactions as primitives. This approach is well-suited for Eris because it is able to commit independent transactions in a single round-trip. Other systems, such as Granola, require separate, specialized protocols for independent and general transactions because processing independent transaction in Granola involves significant coordination overhead [19].

Supporting strong isolation in the presence of these more general transactions requires concurrency control. Eris uses strict two-phase locking for this purpose. Shards maintain read and write locks for every data item; these are only used when processing general transactions. While a lock is held, independent or general transactions that affect the same data item are blocked.

General transactions are executed in two phases. In the first phase, the client sends *reconnaissance* transactions. These are independent transactions that execute the transaction's reads and acquire both read and write locks for the transaction. For transactions where the read set is known in advance (i.e., there are no data-dependent reads) only one round of reconnaissance transactions is required. In the second phase, the client sends a final independent transaction that either `Aborts` the transaction or `Commits` it; a `Commit` installs the transaction's modifications, and in both cases the transaction's locks are released.

Because the independent transaction protocol provides strong guarantees about ordering and reliability, this locking is sufficient to provide the same strong guarantees for general transactions, as well as strong isolation. Importantly, unrelated independent transactions can still be executed on a shard while some locks are held on that shard, as long as those transactions' operations do not conflict with the locks being held (those that do conflict must wait until the locks are released).

Eris uses wait-queue semantics for locking. When a transaction attempts to acquire locks which are already held, the transaction cannot be aborted because all shards must reach the same `Commit`/`Abort` decision for each transaction. Additionally, locks are granted in FIFO order to guarantee that transactions are executed in a deterministic order across all shards.

Eris uses clients as their own transaction managers. Because clients can fail, Eris must be able to abort a general transaction started by a failed client in order to allow the system to continue to make progress. In general, solving this problem is the domain of complex cooperative termination protocols [5]. Because Eris builds on the atomic execution of independent transactions, it admits a simple solution. When an Eris replica suspects that a client has failed because it has held locks for too long without sending a `Commit` or `Abort`, the replica can unilaterally decide to abort the general transaction *by sending the `Abort` command as an independent transaction itself*, sequenced through the sequencer and independent transaction processing layer. This ensures that all participant shards will reach the same `Commit`/`Abort` decision, even if the client did not fail and concurrently attempts to `Commit` the general transaction.

9 Related Work

Our work draws on techniques from distributed protocol design as well as network-level processing mechanisms.

Consensus protocols Many protocols have been proposed for the equivalent problems of consensus, state machine replication, and atomic broadcast. Most closely related is a line of work on achieving better performance when requests *typically* arrive at replicas in the same order, including Fast Paxos [39], Speculative Paxos [56], and Optimistic Atomic Broadcast [31, 54, 55]; Zyzzyva [34] applies a similar idea in the context of Byzantine fault tolerant replication. These protocols can reduce consen-

sus latency in a manner similar to NOPaxos. However, because requests are not *guaranteed* to arrive in the same order, they incur extra complexity and require supermajority quorum sizes to complete a request (either 2/3 or 3/4 of replicas rather than a simple majority). This difference is fundamental: the possibility of conflicting orders requires either an extra message round or a larger quorum size [40].

Another line of work aims to reduce latency and improve throughput by avoiding coordination for operations that are commutative or otherwise need not be ordered [13, 38, 48, 69]; this requires application support to identify commutative operations. NOPaxos avoids coordination for *all* operations.

Ordered Unreliable Multicast is related to a long line of work on totally ordered broadcast primitives, usually in the context of group communication systems [7, 8]. Years ago, a great debate raged in the SOSP community about the effectiveness and limits of this causal and totally ordered communication support (CATOCS) [6, 16]. Our work draws inspiration from both sides of this debate, but occupies a new point in the design space by splitting the responsibility between an ordered but unreliable communications layer and an application-level reliability layer. In particular, the choice to leave reliability to the application is inspired by the end-to-end argument [16, 58].

Network-level processing NOPaxos and Eris take advantage of flexible network processing to implement the OUM and multi-sequenced groupcast primitives. Many designs have been proposed for flexible processing, including fully flexible, software-based designs like Click [32] and others based on network processors [61] or FPGA platforms [51]. At the other extreme, existing software defined networking mechanisms like OpenFlow [47] can easily achieve line-rate performance in commodity hardware implementations but lack the flexibility to implement our multicast primitive. We use the P4 language [10], which supports several high-performance hardware designs like Reconfigurable Match Tables [11].

Transaction Algorithms. Eris draws on a vast literature on distributed storage systems with a wide range of consistency levels and transaction support; we do not attempt to detail them all here. Many recent systems have provided for weaker semantics (causal or eventual consistency and limited or no transactions) as a result of the coordination overhead of strongly consistent distributed transactions [3, 44, 45]. Those that do provide strongly consistent transactional storage do so using a variety of coordination mechanisms [1, 46, 70].

Eris's transaction model is based on independent transactions. Independent transactions were defined as part of the H-Store [27, 30, 62] and Granola [19] projects. Granola provided an application-level protocol for sequencing independent transactions; Eris's network-based multi-sequenced groupcast outperforms it. Although H-Store originally proposed optimizing for independent (or "strongly two-phase") transactions [62], to our knowledge, the proposed protocol was never successfully completed; subsequent work on H-Store abandoned the idea in favor of a different design [27, 30]. Deterministic transaction ordering in Calvin [63–65] takes a similar approach, albeit with a different transaction model, by sequencing transactions through a coordination service before executing them, so that concurrent transactions will acquire locks in the same order across replicas.

Eris builds more complex general transactions out of multiple independent transactions. This is similar in spirit to Lynx [71] and Rococo [49], which divide complex operations into simple pieces using transaction chopping [60].

Most partitioned replicated storage systems 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 [69] and MDCC [35], which are also unified protocols (though the latter only provides weak isolation).

10 Conclusions

We have presented two protocols for fault-tolerant distributed storage—one targeting single-group state machine replication, the other targeting distributed transactions. These protocols—NOPaxos and Eris—each rely on network-level primitives—ordered unreliable multicast and multi-sequenced groupcast—which we have shown can be implemented most efficiently using upcoming programmable network hardware.

NOPaxos and Eris avoid traditional protocol inefficiencies associated with coordination in all but rare failure cases. In the normal case, NOPaxos and Eris can both commit operations (independent transactions in the case of Eris) in a single round trip from the client, with each replica only sending and receiving a single message.

NOPaxos and Eris both provide strong safety guarantees. Both provide linearizability of operations, even in the face of concurrent failures. Additionally, both provide liveness during times of favorable network conditions.

References

[1] 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.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of ACM 2008*, New York, NY, USA, Aug. 2008.

[3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3), 2013.

[4] 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.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery In Database Systems*. Addison-Wesley, Boston, MA, USA, Feb. 1987.

[6] K. Birman. A response to cheriton and skeen's criticism of causal and totally ordered communication. *ACM SIGOPS Operating Systems Review*, 28(1):11–21, Jan. 1994.

[7] 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.

[8] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, Jan. 1987.

[9] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, USA, Apr. 2011. USENIX.

[10] 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. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[11] 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*, pages 99–110. ACM, 2013.

[12] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, Nov. 2006.

[13] L. Camargos, R. Schmidt, and F. Pedone. Multi-coordinated Paxos. Technical report, University of Lugano Faculty of Informatics, 2007/02, Jan. 2007.

[14] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[15] 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.

[16] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, NC, USA, Dec. 1993. ACM.

[17] 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.

[18] J. Cowling. *Low-Overhead Distributed Transaction Coordination*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2012.

[19] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[21] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of ACM SIGCOMM 2011*, Toronto, ON, Canada, Aug. 2011.

[22] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proc. of SOSP*, 2011.

[23] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM 2009*, Barcelona, Spain, Aug. 2009.

[24] 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.

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, USA, June 2010.

[26] 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.

[27] 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.

[28] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[29] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[30] 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. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[31] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC '99)*, Bratislava, Slovakia, Sept. 1999.

[32] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[33] 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.

[34] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, Oct. 2007.

[35] 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.

[36] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[37] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.

[38] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, Mar. 2005.

[39] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

[40] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, Oct. 2006.

[41] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering [extended version]. Technical Report UW-CSE-16-09-02, University of Washington CSE, Seattle, WA, USA, Nov. 2016.

[42] 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.

[43] 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.

[44] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, Oct. 2011. ACM.

[45] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, Apr. 2013. USENIX.

[46] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. E. Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. of VLDB*, 2013.

[47] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Apr. 2008.

[48] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, 2013.

[49] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, Oct. 2014. USENIX.

[50] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of ACM SIGCOMM 2009*, Barcelona, Spain, Aug. 2009.

[51] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the netFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 1–9, New York, NY, USA, 2008. ACM.

[52] 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.

[53] R. Ozdag. Intel® Ethernet switch FM6000 series-software defined networking.

[54] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*, Andros, Greece, Sept. 1998.

[55] F. Pedone and A. Schiper. Optimistic atomic broadcast: A pragmatic viewpoint. *Theor. Comput. Sci.*, 291(1):79–101, Jan. 2003.

[56] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proc. of NSDI*, 2015.

[57] J. Rao, E. J. Shekita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proc. of VLDB*, 4(4):243–254, Apr. 2011.

[58] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[59] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[60] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3):325–363, Sept. 1995.

[61] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct. 2001. ACM.

[62] 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.

[63] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(10), 2010.

[64] 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.

[65] 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.

[66] Transaction Processing Performance Council. TPC Benchmark C. http://www.tpc.org/tpcc/, Feb. 2010.

[67] P. Urbán, X. Défago, and A. Schiper. Chasing the FLP impossibility result in a LAN: or, how robust can a fault tolerant server be? In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS '01)*, New Orleans, LA USA, Oct. 2001.

[68] XPliant Ethernet switch product family. www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[69] 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 Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, Monterey, California, 2015.

[70] 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.

[71] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, USA, Nov. 2013. ACM.

A Proof of NOPaxos's Correctness

A.1 Comparable Views

First, we need to show that the leader's selection of candidate log in Section 5.2.3 is well defined. The protocol states that the leader chooses the logs from the largest view in which replicas have status `Normal`. Since views are partially ordered, the largest view might not exist. Therefore, we need to show that all views in which a replica could have had normal status are comparable.

Lemma 1. *The local view-id of every replica grows monotonically over time (neither the leader-num nor session-num ever decreases).*

Proof. Notice that the only times a replica ever updates its *view-id* are: (1) when the replica detects a leader failure or receives SESSION-TERMINATED from libOUM and increments the corresponding *view-id* entry, (2) when the replica receives a VIEW-CHANGE-REQ and joins its *view-id* with the requested *view-id*, and (3) when the replica accepts a START-VIEW with *view-id* greater than or equal to its own. In each of these cases, the replica's *view-id* only grows. □

Theorem 1 (Comparable Views)**.** *All views that start (i.e. the leader for the view sends out* START-VIEW *messages) have comparable view-ids.*

Proof. In order for a view to start, the leader of that view must have received $f+1$ VIEW-CHANGE messages from different replicas. For every two views, having *view-id*s v_1 and v_2, that start, by quorum intersection there must have been at least one replica that send a VIEW-CHANGE messages for both v_1 and v_2. This implies that at some point, that replica must have adopted v_1 as its *view-id*, and at some point, it must have adopted v_2. So, by Lemma 1, v_1 and v_2 are comparable. □

Notice that Lemma 1 and Theorem 1 also imply:

Lemma 2. *The view-id adopted by a quorum (and therefore the view-id of any view capable of responding to a client) monotonically grows over time.*

A.2 Safety

With that detail out of the way, we move on to proving the safety of NOPaxos by building on the guarantees of the ordered unreliable multicast and libOUM. At a high level, we want to show that the system is linearizable [24].

Theorem 2 (NOPaxos Safety)**.** *NOPaxos guarantees linearizability of client ops and returned results (i.e. calls into and returns from the NOPaxos library at the clients).*

Before prove this, we need to define some properties of logs and REQUESTs. Note that throughout this section, we consider all of the REQUESTs delivered by libOUM in different sessions or positions in sessions to be different, even "duplicates" due to packet duplication or client retry.

Definition. We say that a REQUEST or NO-OP is *committed* in a log slot if it is processed by $f+1$ replicas in that slot with matching *view-id*s, including the leader of that view.

Definition. We say that a REQUEST is *successful* if the client receives the $f+1$ suitable REPLYs.

One obvious fact is that all successful REQUESTs are also committed.

Definition. We say a log is *stable* if it is a prefix of the log of every replica in views higher than the current one.

Now, we have all the definitions we need to write the main safety lemma.

Lemma 3 (Log Stability)**.** *Every successful* REQUEST *was previously appended onto a stable log at the leader, and the resulting log is also stable.*

Before proving log stability, we will show that it and the fact that during a view, the leader's log grows monotonically imply NOPaxos safety.

Proof of NOPaxos Safety (Theorem 2). Lemma 3 and the fact that during a view, the leader's log grows monotonically taken together imply that from the clients' perspective the behavior of the NOPaxos replicas is indistinguishable from the behavior of a single, correct machine that processes REQUESTs sequentially.

This means that any execution of NOPaxos is equivalent to some serial execution of REQUESTs. Furthermore, because clients retry sending REQUESTs until the REQUEST is successful and because NOPaxos does deduplication of any client retries or duplicates returned by libOUM, we know that a NOPaxos execution is actually equivalent to a serial execution of unique client *op*s (i.e. calls into the client NOPaxos library).

So far, we have proven serializability. To prove linearizability, all that remains is to show that NOPaxos is equivalent to a serial execution that respects the *real-time ordering* of sent *op*s and returned *result*s. This is straightforward. When a client gets the $f+1$ suitable REPLYs for a REQUEST, this means that the corresponding REQUEST was already added to a stable log (and after that point in time, all duplicate REQUESTs will be detected and ignored). Moreover, the only way a REQUEST ends up being executed by some leader is that a client previously sent the REQUEST. Therefore, the *op*s sent to and *result*s returned by the NOPaxos client libraries are linearizable. □

Now, let's prove log stability! First, we need some basic facts about logs, REQUESTs, and NO-OPs.

Lemma 4. *All replicas that begin a view begin the view with the exact same log.*

Proof. Replicas start the first view with identically empty logs.

The only way to begin a view other than the first is to receive the START-VIEW message from the leader, and when replicas do that, they adopt the log in that message as their own. Since there is only one leader for a view, and the leader only sends out one log for the view to start with, replicas must start the view with the same log. □

Lemma 5. *If a* NO-OP *is in any replica's log in some slot in some view, then no* REQUEST *was ever committed in that slot in that view.*

Proof. By Lemma 4, all replicas begin a view with the same log. In order for a NO-OP to be placed into a replica's log, the leader must have sent the GAP-COMMIT message for that log slot. This implies that the leader could not have received and processed a REQUEST for that log slot. Therefore, no REQUEST could have been committed. □

Lemma 6. *For any two replicas in the same view, no slot in their logs contains different* REQUEST*s.*

Proof. By Lemma 4, all replicas begin a view with the same log. Moreover, the replicas begin the view listening to the same position in the same OUM session and only update their listening position in a view if they adopt the leader's *session-msg-num* during synchronization. If this happens, however, they adopted the leader's log as well and are therefore still listening to a position in the OUM session that is consistent with the size of their log (i.e. the position is the same as if the replica had never received the SYNC-PREPARE messages and had received the same number of additional messages from libOUM as the number of entries it added to its log when it received the SYNC-PREPARE).

The messages replicas get from libOUM are the same sequence of client REQUEST, where some REQUESTs are replaced with DROP-NOTIFICATIONs (and perhaps terminated at different points). Replicas only add REQUESTs to their log in the order provided by libOUM, by getting the REQUEST from a replica that already put the REQUEST into its log in that slot, or by getting a SYNC-PREPARE from the leader and adopting the leader's log.

Furthermore, when replicas insert NO-OPs into their logs, they are either replacing a REQUEST already in their log, inserting a NO-OP as a result of a DROP-NOTIFICATION, or appending the NO-OP onto the end of their log and ignoring the next message from libOUM. This ensures that during a view, replicas' logs grow by exactly one REQUEST or NO-OP for every REQUEST or DROP-NOTIFICATION they receive from libOUM.

Therefore, the only way two replicas in the same view could have different REQUESTs in their logs is that libOUM must have returned different REQUESTs to two replicas for the same position in the OUM session, contradicting the guarantees of libOUM. □

Lemma 7. *Any* REQUEST *or* NO-OP *that is committed into some slot in some view will be in the same slot in all replica's logs in all subsequent views.*

Proof. If a REQUEST or NO-OP is committed in a view, v, that means that it was placed into $f+1$ replica's logs.

Consider the next view that starts. That view must have started with $f+1$ VIEW-CHANGE messages, and by quorum intersection at least one replica must have the committed REQUEST or NO-OP. Moreover, since this is the subsequent view, only the logs from view v will be considered when the leader creates the combined log to start the view with. If it was a REQUEST that was committed, by Lemma 5 no log used to create the combined log contains a NO-OP in that slot. Therefore, the REQUEST/NO-OP will be placed into the same slot in the log that starts the next view.

Since only logs from view v or greater will ever be considered for creating the combined log that starts any subsequent view, by induction, the REQUEST/NO-OP will be in the same slot in all subsequent views. □

Now, we have all the facts we need to prove log stability and finish the proof of safety.

Proof of Log Stability (Lemma 3). Before the leader ever adds a NO-OP to its log, it ensures that NO-OP is committed by waiting for f GAP-COMMIT-REPs. Replicas will only send a REPLY for a REQUEST after they have filled all previous slots in their log. Moreover, if some REQUEST, r, in the leader's log is committed, this means that all previous REQUESTs added to that leader's log during that view must also be committed. This is because the at least f replicas in the view also having r in their logs must have filled the log slots below r. They couldn't have filled the slots with NO-OPs without a GAP-COMMIT from the leader (which would contradict there being a REQUEST instead of a NO-OP in the leader's log). Also, by Lemma 6, the REQUESTs in those slots must be the same as r.

Furthermore, if any REQUEST is successful in a view, that means that there must have been $f+1$ replicas, including the leader of the view, that began the view. By Lemma 4, this means that those replicas began the view with the same log, and therefore all of the REQUESTs and NO-OPs in that log are committed (in the new view) as well.

So, now we have that if some REQUEST is successful, all of the REQUESTs and NO-OPs in the leader's log before that REQUEST were already committed, so Lemma 7 finishes the proof. □

A.2.1 Synchronization

In addition to proving the linearizability of NOPaxos, we would also like to prove that the synchronization protocol (Section 5.2.4) ensures the correct property: that no replica will ever have to change its log up to its *sync-point*. Notice that because the leader's log only grows monotonically during a view, it is sufficient to prove the following:

Theorem 3 (Synchronization Safety)**.** *All replicas' logs are stable up to their sync-point.*

Proof. The only way the *sync-point* of some replica, *r*, ever increases is that *r* received the SYNC-COMMIT from the leader, which in turn must have received at least SYNC-REPLYs from f different replicas. Therefore, at least $f+1$ different replicas (the f who sent SYNC-REPLYs and the leader) must have adopted the leader's log up to *sync-point*. Furthermore, this log is the same as the *r*'s log because the only way *r* increases its *sync-point* is if it previously received the leader's log and adopted that log up to the *sync-point*.

Therefore, all of the REQUESTs and NO-OPs in a replica's log up to *sync-point* were at some point stored at $f+1$ replicas in the same view, including the leader of that view, and therefore committed. By Lemma 7, all of these REPLYs and NO-OPs will be in the same slot of all replica's logs in all subsequent views, so the prefix of the log up to *sync-point* is stable. □

A.3 Liveness

While it is nice to prove that NOPaxos will never do anything wrong, it would be even better to prove that it eventually does something right. Because the network the replicas communicate with each other over is asynchronous and because replicas can fail, it is impossible to prove that NOPaxos will always make progress [20]. Therefore, we prove that the replicas are able to successfully reply to clients' REQUEST given some conservative assumptions about the network and the failures that happen.

Theorem 4 (Liveness)**.** *As long as there is a sufficient amount of time during which*

(1) the network the replicas communicate over is fair-lossy,

(2) there is some bound on the relative processing speeds of replicas,

(3) there is a quorum of replicas that stays up,

(4) there is a replica that stays up which no replica suspects of having failed,

(5) all replicas correctly suspect crashed nodes of having failed,

(6) no replica receives a DROP-NOTIFICATION *or* SESSION-TERMINATED *from libOUM, and*

(7) clients' REQUEST *eventually get delivered through libOUM,*

then REQUESTs *which clients send to the system will eventually be successful.*

Proof. First, because there is some replica which doesn't crash and which no replica will suspect of having crashed and because no replica receives a SESSION-TERMINATED from libOUM, there are a finite number of view changes which can happen during this period. Once the "good" replica is elected leader, no other replica will ever increment the *leader-num* in its *view-id* again, and no replica will increment its *session-num* since no replica receives a SESSION-TERMINATED from libOUM.

Now, we also know that any view change that starts (i.e. some replica adopts that *view-id*) will eventually end, either by being supplanted with a view change into a larger view or by the new leader actually starting the view. This is because replicas with status `ViewChange` continually resend the VIEW-CHANGE-REQ messages to all other replicas, and as long as no view change for a larger view starts, those replicas will eventually receive the VIEW-CHANGE-REQ and send their own VIEW-CHANGE message to the leader. Since there is a quorum of replicas that is "up," the leader will eventually receive the $f+1$ VIEW-CHANGE messages it needs to start the view.

Furthermore, once a view starts, as long as the view isn't supplanted by some larger view, eventually all replicas that stay up will adopt the new view and start processing REQUESTs in it. This is because the leader resends the START-VIEW message to each replica until that replica acknowledges that it received the START-VIEW message.

We also know that during this period, the replicas will not get "stuck" in any view led by a crashed leader because the replicas will correctly suspect crashed nodes of having failed and will then initiate a view change.

This means that eventually, there will be some view which stays active (i.e. adopted by $f+1$ non-failed replicas, including a non-failed leader) during which progress will be made. Replicas will eventually be able to resolve any DROP-NOTIFICATION they might have received before this good period of time started. This happens because the leader will eventually be able to commit any NO-OPs that it wants to commit (since there are f replicas up in the current view which will respond to the leader's GAP-COMMIT), and any replica which needs to find out the fate of a certain log slot can ask the leader (which will reply with either a REQUEST or a NO-OP).

Therefore, the system will eventually get to a point where there is some view which stays active during which the replicas in the view only receive REQUESTs from libOUM. Once it reaches that point, even though the replicas might process the REQUESTs at different speeds, eventually every REQUEST delivered by libOUM will be suc-

cessful. Therefore because clients' REQUESTs eventually get delivered by libOUM, clients will eventually get the suitable $f + 1$ REPLYs for every REQUEST they send. □

The assumptions in Theorem 4 are quite strict and by no means necessary for NOPaxos to be able to make progress. In particular, as long is there is a single, active view with a leader that can communicate with the other replicas, NOPaxos will be able to deal with DROP-NOTIFICATIONs from libOUM. However, establishing exact necessary and sufficient conditions for liveness of distributed protocols is often complicated, and the conditions for liveness in Theorem 4 are the normal case operating conditions for a NOPaxos system running inside a data center.

B Eris Protocol Details

B.1 Normal Case

1. First, the client sends the transaction through the sequencer 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*, *epoch-num*, *result*⟩, where *txn-index* is the index of the transaction in the replica's *log*. Only the designated learner for the view actually executes the transaction and includes its *result*; the other replicas simply log the transaction and send `NULL` instead.
3. The client waits for a quorum of replies from each shard. A quorum reply from a shard must include REPLYs from a majority of the shard's replicas, all with matching *txn-index*, *view-num*, and *epoch-num*, and one of the REPLYs must be from the designated learner.

Note that if the replica has a *txn-id* matching the client's transaction in its *perm-drops* or *temp-drops* (but not *un-drops*), it cannot reply to the client. If it permanently dropped the transaction, it must instead commit a NO-OP in its place in the *log*. If it only tentatively dropped the transaction, it must wait for the FC's decision.

B.2 Designated Learner Failure

1. When a replica suspects the designated learner 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*⟩[9] to the designated learner 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 designated learner for the new view receives VIEW-CHANGE messages from a majority of replicas (possibly including itself), it updates its *view-num*, and then merges its own *log*,[10] *temp-drops*, *perm-drops*, and *un-drops* with those it received in the VIEW-CHANGE messages. The merged operation for *temp-drops*, *perm-drops*, and *un-drops* is a simple set union, while the merged *log* is the longest one it received, where the slots which have matching *txn-id*s in *perm-drops* are filled in with NO-OPs.

   It then sets its *status* to `Normal` and sends ⟨START-VIEW, *view-num*, *log*, *temp-drops*, *perm-drops*, *un-drops*⟩[9] to the other replicas in the shard.
4. The non-designated-learner 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*,[10] *view-num*, *temp-drops*, *perm-drops*, and *un-drops* and set their *status* to `Normal` as well.
5. If, at this point, the new designated learner has any *txn-id*s in its *temp-drops* for its shard which are not in *perm-drops* or *un-drops*, it must wait for the FC to come to a decision about those before continuing and starting to process transaction in the new view (retrying asking the FC if necessary).

[9] VIEW-CHANGE, START-VIEW, EPOCH-CHANGE-ACK, and START-EPOCH messages are presented here as containing replicas' full *log*s, etc. This is only for simplicity of exposition. In a real deployment, these messages contain only metadata, and the recipients can then pull missing data from the sender – a standard optimization.

[10] If the replica adds any transactions or NO-OPs to its *log*, it ignores the upcoming corresponding messages or DROP-NOTIFICATIONs coming from the multi-sequencing layer.

C TLA+ Specification of NOPaxos

MODULE *NOPaxos*

Specifies the *NOPaxos* protocol.

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

Constants

The set of replicas and an ordering of them

CONSTANTS *Replicas*, *ReplicaOrder*

ASSUME $IsFiniteSet(Replicas) \land ReplicaOrder \in Seq(Replicas)$

Message sequencers

CONSTANT *NumSequencers* Normally infinite, assumed finite for model checking

$Sequencers \triangleq (1 \,..\, NumSequencers)$

Set of possible values in a client request and a special null value

CONSTANTS *Values*, *NoOp*

Replica Statuses

CONSTANTS *StNormal*, *StViewChange*, *StGapCommit*

Message Types

CONSTANTS *MClientRequest*,	Sent by client to sequencer
MMarkedClientRequest,	Sent by sequencer to replicas
MRequestReply,	Sent by replicas to client
MSlotLookup,	Sent by followes to get the value of a slot
MSlotLookupRep,	Sent by the leader with a value/*NoOp*
MGapCommit,	Sent by the leader to commit a gap
MGapCommitRep,	Sent by the followers to *ACK* a gap commit
MViewChangeReq,	Sent when leader/sequencer failure detected
MViewChange,	Sent to *ACK* view change
MStartView,	Sent by new leader to start view
MSyncPrepare,	Sent by the leader to ensure *log* durability
MSyncRep,	Sent by followers as *ACK*
MSyncCommit	Sent by leaders to indicate stable *log*

Message Schemas

$ViewIDs \triangleq [\ leaderNum \mapsto n \in (1 \,..\,),\ sessNum \mapsto n \in (1 \,..\,)\ ]$

ClientRequest
[*mtype* $\mapsto$ *MClientRequest*,
value $\mapsto v \in$ *Values*]

MarkedClientRequest
[*mtype* $\mapsto$ *MMarkedClientRequest*,
dest $\mapsto r \in$ *Replicas*,
value $\mapsto v \in$ *Values*,
sessNum $\mapsto s \in$ *Sequencers*,
sessMsgNum $\mapsto n \in (1 \,..\,)$]

$$
\begin{array}{l}
\mathit{RequestReply} \\
\quad [\ \mathit{mtype} \mapsto \mathit{MRequestReply}, \\
\quad\ \mathit{sender} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{viewID} \mapsto v \in \mathit{ViewIDs}, \\
\quad\ \mathit{request} \mapsto v \in \mathit{Values} \cup \{\mathit{NoOp}\}, \\
\quad\ \mathit{logSlotNum} \mapsto n \in (1\,..\,)\ ] \\
\\
\mathit{SlotLookup} \\
\quad [\ \mathit{mtype} \mapsto \mathit{MSlotLookup}, \\
\quad\ \mathit{dest} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{sender} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{viewID} \mapsto v \in \mathit{ViewIDs}, \\
\quad\ \mathit{sessMsgNum} \mapsto n \in (1\,..\,)\ ] \\
\\
\mathit{GapCommit} \\
\quad [\ \mathit{mtype} \mapsto \mathit{MGapCommit}, \\
\quad\ \mathit{dest} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{viewID} \mapsto v \in \mathit{ViewIDs}, \\
\quad\ \mathit{slotNumber} \mapsto n \in (1\,..\,)\ ] \\
\\
\mathit{GapCommitRep} \\
\quad [\ \mathit{mtype} \mapsto \mathit{MGapCommitRep}, \\
\quad\ \mathit{dest} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{sender} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{viewID} \mapsto v \in \mathit{ViewIDs}, \\
\quad\ \mathit{slotNumber} \mapsto n \in (1\,..\,)\ ] \\
\\
\mathit{ViewChangeReq} \\
\quad [\ \mathit{mtype} \mapsto \mathit{MViewChangeReq}, \\
\quad\ \mathit{dest} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{viewID} \mapsto v \in \mathit{ViewIDs}\ ] \\
\\
\mathit{ViewChange} \\
\quad [\ \mathit{mtype} \mapsto \mathit{MViewChange}, \\
\quad\ \mathit{dest} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{sender} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{viewID} \mapsto v \in \mathit{ViewIDs}, \\
\quad\ \mathit{lastNormal} \mapsto v \in \mathit{ViewIDs}, \\
\quad\ \mathit{sessMsgNum} \mapsto n \in (1\,..\,), \\
\quad\ \mathit{log} \mapsto l \in (1\,..\,) \times (\mathit{Values} \cup \{\mathit{NoOp}\})\ ] \\
\\
\mathit{StartView} \\
\quad [\ \mathit{mtype} \mapsto \mathit{MStartView}, \\
\quad\ \mathit{dest} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{viewID} \mapsto v \in \mathit{ViewIDs}, \\
\quad\ \mathit{log} \mapsto l \in (1\,..\,) \times (\mathit{Values} \cup \{\mathit{NoOp}\}), \\
\quad\ \mathit{sessMsgNum} \mapsto n \in (1\,..\,)\ ] \\
\\
\mathit{SyncPrepare} \\
\quad [\ \mathit{mtype} \mapsto \mathit{MSyncPrepare}, \\
\quad\ \mathit{dest} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{sender} \mapsto r \in \mathit{Replicas}, \\
\quad\ \mathit{viewID} \mapsto v \in \mathit{ViewIDs},
\end{array}
$$

$sessMsgNum \mapsto n \in (1 \,..\,),$
$log \mapsto l \in (1 \,..\,) \times (Values \cup \{NoOp\})\ ]$

SyncRep
$[\ mtype \mapsto MSyncRep,$
$dest \mapsto r \in Replicas,$
$sender \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$
$logSlotNumber \mapsto n \in (1 \,..\,)\ ]$

SyncCommit
$[\ mtype \mapsto MSyncCommit,$
$dest \mapsto r \in Replicas,$
$sender \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$
$log \mapsto l \in (1 \,..\,) \times (Values \cup \{NoOp\}),$
$sessMsgNum \mapsto n \in (1 \,..\,)\ ]$

Variables

Network State

VARIABLE *messages* Set of all messages sent

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

Sequencer State

VARIABLE *seqMsgNums*

$sequencerVars \triangleq \langle seqMsgNums \rangle$
$InitSequencerState \triangleq seqMsgNums = [s \in Sequencers \mapsto 1]$

Replica State

VARIABLES
vLog, Log of values and gaps
vSessMsgNum, The number of messages received in the *OUM* session
vReplicaStatus, One of *StNormal*, *StViewChange*, and *StGapCommit*
vViewID, Current *viewID* replicas recognize
vLastNormView, Last views in which replicas had status *StNormal*
vViewChanges, Used for logging view change votes
vCurrentGapSlot, Used for gap commit at leader
vGapCommitReps, Used for logging gap commit reps at leader
vSyncPoint, Synchronization point for replicas
vTentativeSync, Used by leader to mark current syncing point
vSyncReps Used for logging sync reps at leader

$replicaVars \triangleq \langle vLog,\ vViewID,\ vSessMsgNum,\ vLastNormView,\ vViewChanges,$
$vGapCommitReps,\ vCurrentGapSlot,\ vReplicaStatus,$
$vSyncPoint,\ vTentativeSync,\ vSyncReps \rangle$
$InitReplicaState \triangleq$

$$
\begin{array}{ll}
\land\ vLog & = [r \in Replicas \mapsto \langle\rangle] \\
\land\ vViewID & = [r \in Replicas \mapsto \\
 & \quad [sessNum \mapsto 1,\ leaderNum \mapsto 1]] \\
\land\ vLastNormView & = [r \in Replicas \mapsto \\
 & \quad [sessNum \mapsto 1,\ leaderNum \mapsto 1]] \\
\land\ vSessMsgNum & = [r \in Replicas \mapsto 1] \\
\land\ vViewChanges & = [r \in Replicas \mapsto \{\}] \\
\land\ vGapCommitReps & = [r \in Replicas \mapsto \{\}] \\
\land\ vCurrentGapSlot & = [r \in Replicas \mapsto 0] \\
\land\ vReplicaStatus & = [r \in Replicas \mapsto StNormal] \\
\land\ vSyncPoint & = [r \in Replicas \mapsto 0] \\
\land\ vTentativeSync & = [r \in Replicas \mapsto 0] \\
\land\ vSyncReps & = [r \in Replicas \mapsto \{\}]
\end{array}
$$

Set of all vars

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

Initial state

$$
\begin{array}{ll}
Init \triangleq & \land\ InitNetworkState \\
 & \land\ InitSequencerState \\
 & \land\ InitReplicaState
\end{array}
$$

Helpers

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

View ID Helpers

$$
\begin{array}{ll}
Leader(viewID) \triangleq & ReplicaOrder[(viewID.leaderNum \% Len(ReplicaOrder)) + \\
 & \qquad (\text{IF } viewID.leaderNum \geq Len(ReplicaOrder) \\
 & \qquad\quad \text{THEN } 1 \text{ ELSE } 0)] \\
ViewLe(v1,\ v2) \triangleq & \land\ v1.sessNum \leq v2.sessNum \\
 & \land\ v1.leaderNum \leq v2.leaderNum \\
ViewLt(v1,\ v2) \triangleq & ViewLe(v1,\ v2) \land v1 \neq v2
\end{array}
$$

Network Helpers

Add a message to the network

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

Log Manipulation Helpers

Combine logs, taking a *NoOp* for any slot that has a *NoOp* and a Value otherwise.

$$
\begin{array}{l}
CombineLogs(ls) \triangleq \\
\quad \text{LET} \\
\qquad combineSlot(xs) \triangleq \text{IF } NoOp \in xs \text{ THEN} \\
\qquad\qquad\qquad NoOp
\end{array}
$$

```
                                ELSE IF xs = {} THEN
                                 NoOp
                                ELSE
                                 CHOOSE x ∈ xs : x ≠ NoOp
    range ≜ Max({Len(l) : l ∈ ls})
  IN
    [i ∈ (1 .. range) ↦
      combineSlot({l[i] : l ∈ {k ∈ ls : i ≤ Len(k)}})]

 Insert x into log l at position i (which should be ≤ Len(l) + 1)
ReplaceItem(l, i, x) ≜
  [j ∈ 1 .. Max({Len(l), i}) ↦ IF j = i THEN x ELSE l[j]]

 Subroutine to send an MGapCommit message
SendGapCommit(r) ≜
  LET
    slot ≜ Len(vLog[r]) + 1
  IN
  ∧ Leader(vViewID[r]) = r
  ∧ vReplicaStatus[r]    = StNormal
  ∧ vReplicaStatus'      = [vReplicaStatus EXCEPT ![r] = StGapCommit]
  ∧ vGapCommitReps'      = [vGapCommitReps EXCEPT ![r] = {}]
  ∧ vCurrentGapSlot'     = [vCurrentGapSlot EXCEPT ![r] = slot]
  ∧ Send({[mtype      ↦ MGapCommit,
           dest       ↦ d,
           slotNumber ↦ slot,
           viewID     ↦ vViewID[r]] : d ∈ Replicas})
  ∧ UNCHANGED ⟨sequencerVars, vLog, vViewID, vSessMsgNum,
               vLastNormView, vViewChanges, vSyncPoint,
               vTentativeSync, vSyncReps⟩
```

Main Spec

```
Quorums ≜ {R ∈ SUBSET (Replicas) : Cardinality(R) * 2 > Cardinality(Replicas)}

 A request is committed if a quorum sent replies with matching view-id and log-slot-num, where
 one of the replies is from the leader. The following predicate is true iff value v is committed in
 slot i.
Committed(v, i) ≜
  ∃ M ∈ SUBSET ({m ∈ messages : ∧ m.mtype = MRequestReply
                                ∧ m.logSlotNum = i
                                ∧ m.request = v}) :
    Sent from a quorum
    ∧ {m.sender : m ∈ M} ∈ Quorums
    Matching view-id
    ∧ ∃ m1 ∈ M : ∀ m2 ∈ M : m1.viewID = m2.viewID
```

```
    One from the leader
    ∧ ∃ m ∈ M : m.sender = Leader(m.viewID)

We only provide the ordering layer here. This is an easier guarantee to provide than saying the execution is equivalent to a linear one. We don't currently model execution, and that's a much harder predicate to compute.
Linearizability ≜
  LET
    maxLogPosition ≜ Max({1} ∪
      {m.logSlotNum : m ∈ {m ∈ messages : m.mtype = MRequestReply}})
  IN  ¬(∃ v1, v2 ∈ Values ∪ {NoOp} :
          ∧ v1 ≠ v2
          ∧ ∃ i ∈ (1 .. maxLogPosition) :
            ∧ Committed(v1, i)
            ∧ Committed(v2, i)
        )

SyncSafety ≜ ∀ r ∈ Replicas :
               ∀ i ∈ 1 .. vSyncPoint[r] :
               Committed(vLog[r][i], i)
```

Message Handlers and Actions

Client action

Send a request for value v

```
ClientSendsRequest(v) ≜  ∧ Send({[mtype ↦ MClientRequest,
                                  value ↦ v]})
                         ∧ UNCHANGED ⟨sequencerVars, replicaVars⟩
```

Normal Case Handlers

Sequencer s receives *MClientRequest*, m

```
HandleClientRequest(m, s) ≜
  LET
    smn ≜ seqMsgNums[s]
  IN
   ∧ Send({[mtype      ↦ MMarkedClientRequest,
            dest       ↦ r,
            value      ↦ m.value,
            sessNum    ↦ s,
            sessMsgNum ↦ smn] : r ∈ Replicas})
   ∧ seqMsgNums' = [seqMsgNums EXCEPT ![s] = smn + 1]
   ∧ UNCHANGED replicaVars
```

Replica r receives *MMarkedClientRequest*, m

```
HandleMarkedClientRequest(r, m) ≜
  ∧ vReplicaStatus[r] = StNormal
```

```
    Normal case
 ∧ ∨ ∧ m.sessNum          = vViewID[r].sessNum
     ∧ m.sessMsgNum       = vSessMsgNum[r]
     ∧ vLog'              = [vLog EXCEPT ![r] = Append(vLog[r], m.value)]
     ∧ vSessMsgNum'       = [vSessMsgNum EXCEPT ![r] = vSessMsgNum[r] + 1]
     ∧ Send({[mtype       ↦ MRequestReply,
              request     ↦ m.value,
              viewID      ↦ vViewID[r],
              logSlotNum  ↦ Len(vLog'[r]),
              sender      ↦ r]})
     ∧ UNCHANGED ⟨sequencerVars,
                  vViewID, vLastNormView, vCurrentGapSlot, vGapCommitReps,
                  vViewChanges, vReplicaStatus, vSyncPoint,
                  vTentativeSync, vSyncReps⟩
    SESSION-TERMINATED Case
   ∨ ∧ m.sessNum > vViewID[r].sessNum
     ∧ LET
         newViewID ≜ [sessNum    ↦ m.sessNum,
                      leaderNum  ↦ vViewID[r].leaderNum]
       IN
       ∧ Send({[mtype  ↦ MViewChangeReq,
                dest   ↦ d,
                viewID ↦ newViewID] : d ∈ Replicas})
       ∧ UNCHANGED ⟨replicaVars, sequencerVars⟩
    DROP-NOTIFICATION Case
   ∨ ∧ m.sessNum      = vViewID[r].sessNum
     ∧ m.sessMsgNum > vSessMsgNum[r]
       If leader, commit a gap
     ∧ ∨ ∧ r = Leader(vViewID[r])
         ∧ SendGapCommit(r)
       Otherwise, ask the leader
       ∨ ∧ r ≠ Leader(vViewID[r])
         ∧ Send({[mtype       ↦ MSlotLookup,
                  viewID      ↦ vViewID[r],
                  dest        ↦ Leader(vViewID[r]),
                  sender      ↦ r,
                  sessMsgNum  ↦ vSessMsgNum[r]]})
         ∧ UNCHANGED ⟨replicaVars, sequencerVars⟩

 Gap Commit Handlers
 Replica r receives SlotLookup, m
HandleSlotLookup(r, m) ≜
  LET
    logSlotNum ≜ Len(vLog[r]) + 1 − (vSessMsgNum[r] − m.sessMsgNum)
  IN
```

```
  ∧ m.viewID              = vViewID[r]
  ∧ Leader(vViewID[r]) = r
  ∧ vReplicaStatus[r] = StNormal
  ∧ ∨ ∧ logSlotNum ≤ Len(vLog[r])
      ∧ Send({[mtype          ↦ MMarkedClientRequest,
               dest           ↦ m.sender,
               value          ↦ vLog[r][logSlotNum],
               sessNum        ↦ vViewID[r].sessNum,
               sessMsgNum ↦ m.sessMsgNum]})
      ∧ UNCHANGED ⟨replicaVars, sequencerVars⟩
    ∨ ∧ logSlotNum = Len(vLog[r]) + 1
      ∧ SendGapCommit(r)

 Replica r receives GapCommit, m
HandleGapCommit(r, m) ≜
  ∧ m.viewID              = vViewID[r]
  ∧ m.slotNumber          ≤ Len(vLog[r]) + 1
  ∧ ∨ vReplicaStatus[r]   = StNormal
    ∨ vReplicaStatus[r]   = StGapCommit
  ∧ vLog' = [vLog EXCEPT ![r] = ReplaceItem(vLog[r], m.slotNumber, NoOp)]
   Increment the msgNumber if necessary
  ∧ IF m.slotNumber > Len(vLog[r]) THEN
      vSessMsgNum' = [vSessMsgNum EXCEPT ![r] = vSessMsgNum[r] + 1]
    ELSE
      UNCHANGED vSessMsgNum
  ∧ Send({[mtype          ↦ MGapCommitRep,
           dest           ↦ Leader(vViewID[r]),
           sender         ↦ r,
           slotNumber ↦ m.slotNumber,
           viewID         ↦ vViewID[r]],
          [mtype          ↦ MRequestReply,
           request        ↦ NoOp,
           viewID         ↦ vViewID[r],
           logSlotNum ↦ m.slotNumber,
           sender         ↦ r]})
  ∧ UNCHANGED ⟨sequencerVars, vGapCommitReps, vViewID, vCurrentGapSlot,
               vReplicaStatus, vLastNormView, vViewChanges,
               vSyncPoint, vTentativeSync, vSyncReps⟩

 Replica r receives GapCommitRep, m
HandleGapCommitRep(r, m) ≜
  ∧ vReplicaStatus[r]     = StGapCommit
  ∧ m.viewID              = vViewID[r]
  ∧ Leader(vViewID[r]) = r
  ∧ m.slotNumber            = vCurrentGapSlot[r]
```

$\land\ vGapCommitReps' = [vGapCommitReps \text{ EXCEPT } ![r] = vGapCommitReps[r] \cup \{m\}]$

When there's enough, resume *StNormal* and process more messages

$\land$ LET $isViewPromise(M) \triangleq \land \{n.sender : n \in M\} \in Quorums$
$\land \exists\, n \in M : n.sender = r$
$gCRs \triangleq \{n \in vGapCommitReps'[r] :$
$\land\ n.mtype = MGapCommitRep$
$\land\ n.viewID = vViewID[r]$
$\land\ n.slotNumber = vCurrentGapSlot[r]\}$

IN

IF $isViewPromise(gCRs)$ THEN

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

ELSE

UNCHANGED $vReplicaStatus$

$\land$ UNCHANGED $\langle sequencerVars,\ networkVars,\ vLog,\ vViewID,\ vCurrentGapSlot,$
$vSessMsgNum,\ vLastNormView,\ vViewChanges,\ vSyncPoint,$
$vTentativeSync,\ vSyncReps\rangle$

Failure Cases

Replica r starts a *Leader* change

$StartLeaderChange(r) \triangleq$

LET

$newViewID \triangleq [sessNum \mapsto vViewID[r].sessNum,$
$leaderNum \mapsto vViewID[r].leaderNum + 1]$

IN

$\land\ Send(\{[mtype \mapsto MViewChangeReq,$
$dest \mapsto d,$
$viewID \mapsto newViewID] : d \in Replicas\})$

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

View Change Handlers

Replica r gets $MViewChangeReq$, m

$HandleViewChangeReq(r,\ m) \triangleq$

LET

$currentViewID \triangleq vViewID[r]$
$newSessNum \triangleq Max(\{currentViewID.sessNum,\ m.viewID.sessNum\})$
$newLeaderNum \triangleq Max(\{currentViewID.leaderNum,\ m.viewID.leaderNum\})$
$newViewID \triangleq [sessNum \mapsto newSessNum,\ leaderNum \mapsto newLeaderNum]$

IN

$\land\ currentViewID \neq newViewID$
$\land\ vReplicaStatus' = [vReplicaStatus \text{ EXCEPT } ![r] = StViewChange]$
$\land\ vViewID' = [vViewID \text{ EXCEPT } ![r] = newViewID]$
$\land\ vViewChanges' = [vViewChanges \text{ EXCEPT } ![r] = \{\}]$
$\land\ Send(\{[mtype \mapsto MViewChange,$
$dest \mapsto Leader(newViewID),$

```
                    sender      ↦ r,
                    viewID      ↦ newViewID,
                    lastNormal  ↦ vLastNormView[r],
                    sessMsgNum  ↦ vSessMsgNum[r],
                    log         ↦ vLog[r]]} ∪
                  Send the MViewChangeReqs in case this is an entirely new view
                  {[mtype   ↦ MViewChangeReq,
                    dest    ↦ d,
                    viewID  ↦ newViewID] : d ∈ Replicas})
  ∧ UNCHANGED ⟨vCurrentGapSlot, vGapCommitReps, vLog, vSessMsgNum,
               vLastNormView, sequencerVars, vSyncPoint,
               vTentativeSync, vSyncReps⟩

 Replica r receives MViewChange, m
HandleViewChange(r, m) ≜
  Add the message to the log
  ∧ vViewID[r]          = m.viewID
  ∧ vReplicaStatus[r]   = StViewChange
  ∧ Leader(vViewID[r]) = r
  ∧ vViewChanges' =
    [vViewChanges EXCEPT ![r] = vViewChanges[r] ∪ {m}]
  If there's enough, start the new view
  ∧ LET
      isViewPromise(M) ≜ ∧ {n.sender : n ∈ M} ∈ Quorums
                         ∧ ∃ n ∈ M : n.sender = r
      vCMs             ≜ {n ∈ vViewChanges'[r] :
                            ∧ n.mtype  = MViewChange
                            ∧ n.viewID = vViewID[r]}
      Create the state for the new view
      normalViews ≜ {n.lastNormal : n ∈ vCMs}
      lastNormal  ≜ (CHOOSE v ∈ normalViews : ∀ v2 ∈ normalViews :
                       ViewLe(v2, v))
      goodLogs    ≜ {n.log : n ∈
                       {o ∈ vCMs : o.lastNormal = lastNormal}}
      If updating seqNum, revert sessMsgNum to 0; otherwise use latest
      newMsgNum   ≜
        IF lastNormal.sessNum = vViewID[r].sessNum THEN
           Max({n.sessMsgNum : n ∈
                  {o ∈ vCMs : o.lastNormal = lastNormal}})
        ELSE
          0
    IN
      IF isViewPromise(vCMs) THEN
        Send({[mtype      ↦ MStartView,
               dest       ↦ d,
```

$viewID \mapsto vViewID[r],$
$log \mapsto CombineLogs(goodLogs),$
$sessMsgNum \mapsto newMsgNum] : d \in Replicas\})$

ELSE

UNCHANGED $networkVars$

$\land$ UNCHANGED $\langle vReplicaStatus, vViewID, vLog, vSessMsgNum, vCurrentGapSlot, vGapCommitReps, vLastNormView, sequencerVars, vSyncPoint, vTentativeSync, vSyncReps\rangle$

Replica r receives a *MStartView*, m

$HandleStartView(r, m) \triangleq$

Note how I handle this. There was actually a bug in prose description in the paper where the following guard was underspecified.

$\land \lor ViewLt(vViewID[r], m.viewID)$
$\lor vViewID[r] = m.viewID \land vReplicaStatus[r] = StViewChange$
$\land vLog' = [vLog \text{ EXCEPT } ![r] = m.log]$
$\land vSessMsgNum' = [vSessMsgNum \text{ EXCEPT } ![r] = m.sessMsgNum]$
$\land vReplicaStatus' = [vReplicaStatus \text{ EXCEPT } ![r] = StNormal]$
$\land vViewID' = [vViewID \text{ EXCEPT } ![r] = m.viewID]$
$\land vLastNormView' = [vLastNormView \text{ EXCEPT } ![r] = m.viewID]$

Send replies (in the new view) for all *log* items

$\land Send(\{[mtype \mapsto MRequestReply,$
$request \mapsto m.log[i],$
$viewID \mapsto m.viewID,$
$logSlotNum \mapsto i,$
$sender \mapsto r] : i \in (1 \,..\, Len(m.log))\})$
$\land$ UNCHANGED $\langle sequencerVars, vViewChanges, vCurrentGapSlot, vGapCommitReps, vSyncPoint, vTentativeSync, vSyncReps\rangle$

Synchronization handlers

Leader replica r starts synchronization

$StartSync(r) \triangleq$
$\land Leader(vViewID[r]) = r$
$\land vReplicaStatus[r] = StNormal$
$\land vSyncReps' = [vSyncReps \text{ EXCEPT } ![r] = \{\}]$
$\land vTentativeSync' = [vTentativeSync \text{ EXCEPT } ![r] = Len(vLog[r])]$
$\land Send(\{[mtype \mapsto MSyncPrepare,$
$sender \mapsto r,$
$dest \mapsto d,$
$viewID \mapsto vViewID[r],$
$sessMsgNum \mapsto vSessMsgNum[r],$
$log \mapsto vLog[r]] : d \in Replicas\})$
$\land$ UNCHANGED $\langle sequencerVars, vLog, vViewID, vSessMsgNum, vLastNormView, vCurrentGapSlot, vViewChanges, vReplicaStatus,$

$vGapCommitReps,\ vSyncPoint\rangle$

Replica r receives *MSyncPrepare*, m

$$
\begin{array}{l}
HandleSyncPrepare(r,\ m) \triangleq \\
\quad \text{LET} \\
\quad\quad newLog \triangleq m.log \circ SubSeq(vLog[r],\ Len(m.log)+1,\ Len(vLog[r])) \\
\quad\quad newMsgNum \triangleq vSessMsgNum[r] + (Len(newLog) - Len(vLog[r])) \\
\quad \text{IN} \\
\quad \land vReplicaStatus[r] = StNormal \\
\quad \land m.viewID = vViewID[r] \\
\quad \land m.sender = Leader(vViewID[r]) \\
\quad \land vLog' = [vLog \text{ EXCEPT } ![r] = newLog] \\
\quad \land vSessMsgNum' = [vSessMsgNum \text{ EXCEPT } ![r] = newMsgNum] \\
\quad \land Send(\{[mtype \mapsto MSyncRep, \\
\qquad\qquad sender \mapsto r, \\
\qquad\qquad dest \mapsto m.sender, \\
\qquad\qquad viewID \mapsto vViewID[r], \\
\qquad\qquad logSlotNumber \mapsto Len(m.log)]\} \cup \\
\qquad\quad \{[mtype \mapsto MRequestReply, \\
\qquad\qquad request \mapsto vLog'[r][i], \\
\qquad\qquad viewID \mapsto vViewID[r], \\
\qquad\qquad logSlotNum \mapsto i, \\
\qquad\qquad sender \mapsto r] : i \in 1 \,..\, Len(vLog'[r])\}) \\
\quad \land \text{UNCHANGED } \langle sequencerVars,\ vViewID,\ vLastNormView,\ vCurrentGapSlot, \\
\qquad\qquad vViewChanges,\ vReplicaStatus,\ vGapCommitReps, \\
\qquad\qquad vSyncPoint,\ vTentativeSync,\ vSyncReps\rangle
\end{array}
$$

Replica r receives *MSyncRep*, m

$$
\begin{array}{l}
HandleSyncRep(r,\ m) \triangleq \\
\quad \land m.viewID = vViewID[r] \\
\quad \land vReplicaStatus[r] = StNormal \\
\quad \land vSyncReps' = [vSyncReps \text{ EXCEPT } ![r] = vSyncReps[r] \cup \{m\}] \\
\quad \land \text{LET } isViewPromise(M) \triangleq \land \{n.sender : n \in M\} \in Quorums \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land \exists\, n \in M \ : n.sender = r \\
\qquad\qquad sRMs \triangleq \{n \in vSyncReps'[r] : \\
\qquad\qquad\qquad\qquad\quad \land n.mtype = MSyncRep \\
\qquad\qquad\qquad\qquad\quad \land n.viewID = vViewID[r] \\
\qquad\qquad\qquad\qquad\quad \land n.logSlotNumber = vTentativeSync[r]\} \\
\qquad\qquad committedLog \triangleq \text{IF } vTentativeSync[r] \geq 1 \text{ THEN} \\
\qquad\qquad\qquad\qquad\qquad\quad SubSeq(vLog[r],\ 1,\ vTentativeSync[r]) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE} \\
\qquad\qquad\qquad\qquad\qquad\quad \langle\rangle \\
\quad\quad \text{IN} \\
\quad\quad\quad \text{IF } isViewPromise(sRMs) \text{ THEN} \\
\quad\quad\quad\quad Send(\{[mtype \mapsto MSyncCommit,
\end{array}
$$

$$
\begin{array}{lll}
& sender & \mapsto r, \\
& dest & \mapsto d, \\
& viewID & \mapsto vViewID[r], \\
& log & \mapsto committedLog, \\
& sessMsgNum & \mapsto vSessMsgNum[r] - \\
& & \quad (Len(vLog[r]) - Len(committedLog))] : \\
d \in Replicas\})
\end{array}
$$

ELSE

UNCHANGED $networkVars$

$\land$ UNCHANGED $\langle sequencerVars,\ vLog,\ vViewID,\ vSessMsgNum,\ vLastNormView,$
$vCurrentGapSlot,\ vViewChanges,\ vReplicaStatus,$
$vGapCommitReps,\ vSyncPoint,\ vTentativeSync\rangle$

Replica r receives *MSyncCommit*, m

$HandleSyncCommit(r,\ m) \triangleq$

LET

$newLog \triangleq m.log \circ SubSeq(vLog[r],\ Len(m.log) + 1,\ Len(vLog[r]))$

$newMsgNum \triangleq vSessMsgNum[r] + (Len(newLog) - Len(vLog[r]))$

IN

$\land\ vReplicaStatus[r] = StNormal$

$\land\ m.viewID = vViewID[r]$

$\land\ m.sender = Leader(vViewID[r])$

$\land\ vLog' = [vLog \text{ EXCEPT } ![r] = newLog]$

$\land\ vSessMsgNum' = [vSessMsgNum \text{ EXCEPT } ![r] = newMsgNum]$

$\land\ vSyncPoint' = [vSyncPoint \text{ EXCEPT } ![r] = Len(m.log)]$

$\land$ UNCHANGED $\langle sequencerVars,\ networkVars,\ vViewID,\ vLastNormView,$
$vCurrentGapSlot,\ vViewChanges,\ vReplicaStatus,$
$vGapCommitReps,\ vTentativeSync,\ vSyncReps\rangle$

Main Transition Function

$Next \triangleq$ Handle Messages

$$
\begin{array}{lll}
\lor \exists\, m \in messages : & & \\
\quad \exists\, s \in Sequencers & & \\
\qquad : & \land\ m.mtype = MClientRequest & \\
& \land\ HandleClientRequest(m,\ s) & \\
\lor \exists\, m \in messages : & \land\ m.mtype = MMarkedClientRequest & \\
& \land\ HandleMarkedClientRequest(m.dest,\ m) & \\
\lor \exists\, m \in messages : & \land\ m.mtype = MViewChangeReq & \\
& \land\ HandleViewChangeReq(m.dest,\ m) & \\
\lor \exists\, m \in messages : & \land\ m.mtype = MViewChange & \\
& \land\ HandleViewChange(m.dest,\ m) & \\
\lor \exists\, m \in messages : & \land\ m.mtype = MStartView & \\
& \land\ HandleStartView(m.dest,\ m) & \\
\lor \exists\, m \in messages : & \land\ m.mtype = MSlotLookup &
\end{array}
$$

$$
\begin{array}{ll}
 & \land\ HandleSlotLookup(m.dest,\ m) \\
\lor\ \exists\, m \in messages : & \land\ m.mtype = MGapCommit \\
 & \land\ HandleGapCommit(m.dest,\ m) \\
\lor\ \exists\, m \in messages : & \land\ m.mtype = MGapCommitRep \\
 & \land\ HandleGapCommitRep(m.dest,\ m) \\
\lor\ \exists\, m \in messages : & \land\ m.mtype = MSyncPrepare \\
 & \land\ HandleSyncPrepare(m.dest,\ m) \\
\lor\ \exists\, m \in messages : & \land\ m.mtype = MSyncRep \\
 & \land\ HandleSyncRep(m.dest,\ m) \\
\lor\ \exists\, m \in messages : & \land\ m.mtype = MSyncCommit \\
 & \land\ HandleSyncCommit(m.dest,\ m)
\end{array}
$$

Client Actions

$$\lor\ \exists\, v \in Values : ClientSendsRequest(v)$$

Start synchronization

$$\lor\ \exists\, r \in Replicas : StartSync(r)$$

Failure case

$$\lor\ \exists\, r \in Replicas : StartLeaderChange(r)$$