Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering

[Extended Version]*

Jialin Li Ellis Michael Naveen Kr. Sharma Adriana Szekeres Dan R. K. Ports
University of Washington
{lijl, emichael, naveenks, aaasz, drkp}@cs.washington.edu

Technical Report UW-CSE-16-09-02

Abstract

Distributed applications use replication, implemented by protocols like Paxos, to ensure data availability and transparently mask server failures. This paper presents a new approach to achieving replication in the data center without the performance cost of traditional methods. Our work carefully divides replication responsibility between the network and protocol layers. The network orders requests but *does not* ensure reliable delivery – using a new primitive we call *ordered unreliable multicast* (OUM). Implementing this primitive can be achieved with near-zero-cost in the data center. Our new replication protocol, *Network-Ordered Paxos* (NOPaxos), exploits network ordering to provide strongly consistent replication without coordination. The resulting system not only outperforms both latency- and throughput-optimized protocols on their respective metrics, but also yields throughput within 2% and latency within 16 μs of an unreplicated system – providing replication without the performance cost.

1 Introduction

Server failures are a fact of life for data center applications. To guarantee that critical services always remain available, today's applications rely on fault-tolerance techniques like state machine replication. These systems use application-level consensus protocols such as Paxos to ensure the consistency of replicas' states. Unfortunately, these protocols require expensive coordination on every request, imposing a substantial latency penalty and limiting system scalability. This paper demonstrates that replication in the data center need not impose such a cost by introducing a new replication protocol with performance within 2% of an unreplicated system.

It is well known that the communication model fundamentally affects the difficulty of consensus. Completely asynchronous and unordered networks require the full complexity of Paxos; if a network could provide a totally ordered atomic broadcast primitive, ensuring replica consistency would become a trivial matter. Yet this idea has yielded few gains in practice since traditional ordered-multicast systems are themselves equivalent to consensus; they simply move the same coordination expense to a different layer.

We show that a new division of responsibility between the network and the application can eliminate nearly all replication overhead. Our key insight is that the communication layer should provide a new *ordered unreliable multicast* (OUM) primitive – where all receivers are guaranteed to process multicast messages in the same order, but messages may be lost. This model is weak enough to be implemented efficiently, yet strong enough to dramatically reduce the costs of a replication protocol.

The ordered unreliable multicast model enables our new replication protocol, *Network-Ordered Paxos*. In normal cases, NOPaxos avoids coordination entirely by relying on the network to deliver messages in the same order. It requires application-level coordination only to handle dropped packets, a fundamentally simpler problem than ordering requests. The resulting protocol is simple, achieves near-optimal throughput and latency, and remains robust to network-level failures.

We describe several ways to build the OUM communications layer, all of which offer net performance benefits when combined with NOPaxos. In particular, we achieve an essentially zero-overhead implementation by *relying on the network fabric itself to sequence requests*, using software-defined networking technologies and the advanced packet processing capabilities of next-generation data center network hardware [10, 50, 60]. We achieve similar throughput benefits (albeit with a smaller latency improvement) using an endpoint-based implementation that requires no specialized hardware or network design.

By relying on the OUM primitive, NOPaxos avoids all coordination except in rare cases, eliminating nearly all the performance overhead of traditional replication protocols. It provides *throughput within 2% and latency within 16 μs of an unreplicated system*, demonstrating

*This document is an extended version of the paper by the same title that appeared in OSDI 2016 [40]. An overview of the additional content is provided in §1.1.

that there need not be a tradeoff between enforcing strong consistency and providing maximum performance.

This paper makes four specific contributions:

1. We define the *ordered unreliable multicast* model for data center networks and argue that it strikes an effective balance between providing semantics strong enough to be useful to application-level protocols yet weak enough to be implemented efficiently.

2. We demonstrate how to implement this network model in the data center by presenting three implementations: (1) an implementation in P4 [9] for programmable switches, (2) a middlebox-style prototype using a Cavium Octeon network processor, and (3) a software-based implementation that requires no specialized hardware but imposes slightly higher latency.

3. We introduce NOPaxos, an algorithm which provides state machine replication on an ordered, unreliable network. Because NOPaxos relies on the OUM primitive, it avoids the need to coordinate on every incoming request to ensure a total ordering of requests. Instead, it uses application-level coordination only when requests are lost in the network or after certain failures of server or network components.

4. We evaluate NOPaxos on a testbed using our OpenFlow/Cavium prototype and demonstrate that it outperforms classic leader-based Paxos by 54% in latency and 4.7× in throughput. It simultaneously provides 42% better latency and 24% better throughput than latency- and throughput-optimized protocols respectively, circumventing a classic tradeoff.

1.1 Additional Technical Report Content

This technical report includes the following additional content over the paper published in the proceedings of OSDI [40]:

- the details of the NOPaxos synchronization protocol (§5.2.4),
- a complete proof of correctness for NOPaxos (Appendix A),
- a TLA+ specification of the NOPaxos protocol (Appendix B),
- the P4 source code that implements the switch sequencer (Appendix C), and
- measurements of the latency induced by network serialization (§4.1, Figure 4) and by the Cavium network processor sequencer prototype (§4.2.2, Figure 5).

2 Separating Ordering from Reliable Delivery in State Machine Replication

We consider the problem of *state machine replication* [57]. Replication, used throughout data center applications, keeps key services consistent and available despite the inevitability of failures. For example, Google's Chubby [11] 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 [8, 16, 55].

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 [34, 35] or Viewstamped Replication [43, 49] – to achieve agreement on operation order. Most deployments of Paxos-based replicated systems use the Multi-Paxos optimization [35] (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) [6, 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.

We note that achieving reliable delivery despite the range of possible failures is a formidable task, and the end-to-end principle suggests that it is best left to the application [15, 56]. However, ordering without a guarantee of reliability permits a straightforward, efficient implementation: assigning sequence numbers to messages and then discarding those that arrive out of sequence number order. We show in §3 that this approach can be implemented at almost no cost in data center network hardware.

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.

Prior work has considered an asynchronous network that provides ordering and reliability in the common case but does not guarantee either. Fast Paxos [37] and related systems [30, 46, 51] provide agreement in one fewer message delay when requests usually arrive at replicas in the same order, but they require more replicas and/or larger quorum sizes. Speculative Paxos [54] takes this further by having replicas speculatively execute operations without coordination, eliminating another message delay and a throughput bottleneck at the cost of significantly reduced performance (including application-level rollback) when the network violates its best-effort ordering property. Our approach avoids these problems by strengthening network semantics. Table 1 summarizes the properties of these protocols.

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 [51,59], that data center networks can be engineered to support a multicast primitive that has this property [54], and that it is possible to use this fact to design protocols that are more efficient in the common case [30, 37, 54]. 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 §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 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 [54]. 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

[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.

	Paxos [34, 35, 49]	**Fast Paxos** [37]	**Paxos+batching**	**Speculative Paxos** [54]	**NOPaxos**
Network ordering	No	Best-effort	No	Best-effort	Yes
Latency	4	3	4+	2	2
Messages at bottleneck	$2n$	$2n$	$2+\frac{2n}{b}$	2	2
Quorum size	$>n/2$	$>2n/3$	$>n/2$	$>3n/4$	$>n/2$
Reordering/Dropped packet penalty	low	medium	low	high	low

Table 1: Comparison of NOPaxos to prior systems.

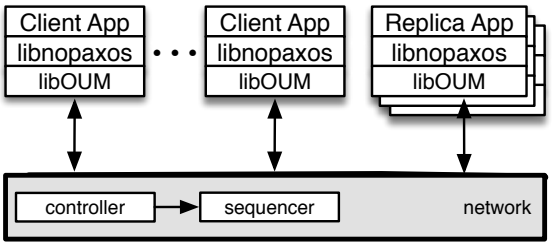

Figure 1: Architecture of NOPaxos.

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.

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 §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.

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* (§4.1). Second, we must implement a sequencer capable of high throughput and low latency. We present three such implementations in §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 (§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 [54]; 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 [1,23,44,47] extend this basic design to support large numbers of physical machines using many commodity switches and often provide full bisection bandwidth. Figure 3 shows the testbed we use, implementing a fat-tree network [1].

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 [45], 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 (§4.2.1). 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. For load balancing, different OUM groups are assigned different sequencers, e.g., using hash-based partitioning.

Figure 3 shows an example of network serialization forwarding paths in a 12-switch, 3-level fat tree network. Sequencers are implemented as network processors (§4.2.2) connected to root switches. Messages from a client machine are first forwarded upward to the designated sequencer – here, attached to the leftmost root switch – then distributed downward to all recipients.

Network serialization could create longer paths than traditional IP multicast because all traffic must be routed to the sequencer, but this effect is minimal in practice. We quantified this latency penalty using packet-level network simulation. The simulated network contained 2,560 end-hosts and 119 switches configured in a 3-level fat tree network, with background traffic modeled on Microsoft data centers [4]. Each client sent multicast messages to a random group of 5 receivers. Figure 4 shows the distribution of latency required for each message to be received

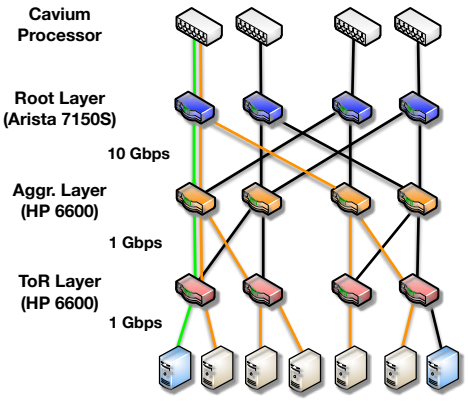

Figure 3: Testbed network topology. Green lines indicate the upward path from a client to the sequencer, and orange lines indicate the downward path from the sequencer to receivers.

by a quorum of receivers. In 88% of cases, network serialization added *no additional latency* for the message to be received by a quorum of 3 receivers; the 99th-percentile was less than 5 μs of added latency. This minimal increase in latency is due to the fact that the sequencer is a least-common-ancestor switch of the replica group, and most packets have to traverse that switch anyway to reach a majority of the group.

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 §4.3.

Our sequencer design is general; we discuss three possible implementations here. The most efficient one targets upcoming programmable network switches, using the switch itself as the sequencer, incurring no latency cost (§4.2.1). As this hardware is not yet available, we describe

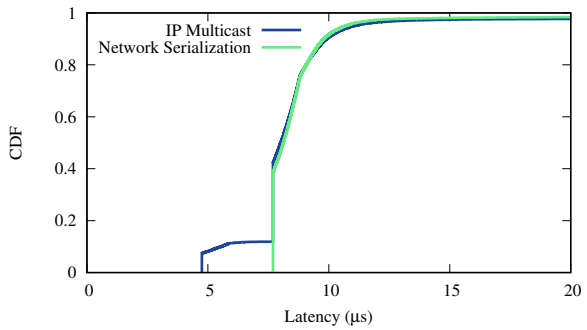

Figure 4: Network simulation showing latency difference between IP multicast and network serialization.

a prototype that uses a network processor to implement a middlebox-like sequencer (§4.2.2). Finally, we discuss using an end-host as a sequencer (§4.2.3).

4.2.1 In-Switch Sequencing

Ideally, switches themselves could serve as sequencers. The benefit of doing so is latency: packets could be sequenced by one of the switches through which they already transit, rather than having to be redirected to a dedicated device. Moreover, switching hardware is highly optimized for low-latency packet processing, unlike end-hosts.

Using a switch as a sequencer is made possible by the increasing ability of data center switches to perform flexible, per-packet computations. An emerging class of switch architectures – such as Reconfigurable Match Tables [10], Intel's FlexPipe [50], and Cavium's XPliant [60] – allow the switch's behavior to be controlled on a per-packet granularity, supporting the parsing and matching of arbitrary packet fields, rewriting of packet contents, and maintaining of small amounts of state between packets. Exposed through high-level languages like P4 [9], this increased flexibility lets us consider network switches as not simply forwarding elements, but as devices with computational ability.

We implemented our switch sequencing functionality in the P4 language, which allows it to be compiled and deployed to upcoming programmable switches as well as software switches. Our implementation uses the reconfigurable parser capabilities of these switches to define a custom packet header that includes the OUM sequence and session numbers. It uses stateful memory (register arrays) to store the current sequence number for every OUM group and increments it on each packet. Complete NOPaxos P4 code is available in Appendix C.

Programmable switches capable of this processing are not yet commercially available, although we expect them to be within the next year. Therefore, we cannot evaluate their performance, but there is reason to believe they can execute this processing with no measurable increase in latency. As evidence, Intel's FlexPipe chips (now available, e.g., in the Arista 7150 switch) can modify packets to include the egress timestamp with zero latency cost [2, 50].

We note that a network switch provides orders-of-magnitude lower latency and greater reliability [22] than an end-host. Today's fastest cut-through switches can consistently process packets in approximately 300 ns [2], while a typical Linux server has median latency in the 10–100 μs range and 99.9th-percentile latency over 5 *ms* [41]. This trend seems unlikely to change: even with high-performance server operating systems [3,53], NIC latency remains an important factor [21]. At the same time, the limited computational model of the switch requires a careful partitioning of functionality between the network and application. The OUM model offers such a design.

4.2.2 Hardware Middlebox Prototype Sequencing

Because available switches do not provide the necessary flexibility to run P4 programs, we implemented a prototype using existing OpenFlow switches and a network processor.

This prototype is part of the testbed that we use to evaluate our OUM model and its uses for distributed protocols. This testbed simulates the 12-switch, 3-layer fat-tree network configuration depicted in Figure 3. We implemented it on three physical switches by using VLANs and appropriate OpenFlow forwarding rules to emulate separate virtual switches: two HP 6600 switches implement the ToR and aggregation tiers, and one Arista 7050S switch implements the core tier.

We implemented the sequencer as a form of middlebox using a Cavium Octeon II CN68XX network processor. This device contains 32 MIPS64 cores and supports 10 Gb/s Ethernet I/O. Users can customize network functionality by loading C binaries that match, route, drop or modify packets going through the processor. Onboard DRAM maintains per-group state. We attached the middlebox to the root switches and installed OpenFlow rules to redirect OUM packets to the middlebox.

This implementation does not provide latency as low as the switch-based sequencer; routing traffic through the network processor adds latency. As shown in Figure 5, we measured this latency to be 8 μs in the median case and 16 μs in the 99th percentile. This remains considerably lower than implementing packet processing in an end-host.

4.2.3 End-host Sequencing

Finally, we also implemented the sequencing functionality on a conventional server. While this incurs higher latency, it allows the OUM abstraction to be implemented without any specialized hardware. Nevertheless, using a dedicated host for network-level sequencing can still provide throughput, if not latency, benefits as we demonstrate in §6. We implemented a simple Linux program that uses

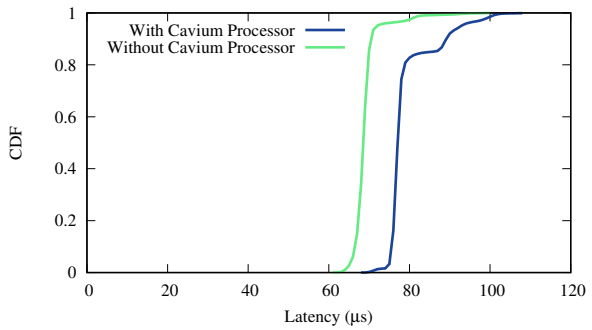

Figure 5: Latency induced by the Cavium network processor emulating switch sequencing.

raw sockets to access packet headers.

4.2.4 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 [44]. 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 [22], and rerouting can be completed within a few milliseconds [44], 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 $\langle$*session-number*, *sequence-number*$\rangle$ 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 (§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 [27, 32]. We note that our replication protocol, NOPaxos (§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, and we discuss how they affect NOPaxos's performance in §6.

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 [43].

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 $\langle$*leader-num*, *session-num*$\rangle$. 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 regarding 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.

[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* = $\langle$*leader-num*, *session-num*$\rangle$ — the view number, a tuple of the current leader number and OUM session number, partially ordered, initially $\langle 0, 0 \rangle$
- *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 6: Local state of NOPaxos replicas.

Outline. NOPaxos consists of four subprotocols:

- *Normal Operations* (§5.2.1): NOPaxos processes client REQUESTs in a single round-trip in the normal case.
- *Gap Agreement* (§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* (§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* (§5.2.4): Periodically, the leader synchronizes the logs of all replicas.

Figure 6 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 $\langle$REQUEST, *op*, *request-id*$\rangle$ 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 $\langle$REPLY, *view-id*, *log-slot-num*, *request-id*, *result*$\rangle$, 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 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 [43]. 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

[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.

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. 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.

1. The leader broadcasts a ⟨SYNC-PREPARE, *session-msg-num*, *log*⟩ message.
2. When a replica receives a ⟨SYNC-PREPARE, *session-msg-num*, *log*⟩ message from the leader of its current view, it adds any new REQUESTs to its log and adds any new NO-OPs, replacing REQUESTs if necessary. If the new *session-msg-num* is larger than its current one, it updates it and calls `listen(session-num, session-msg-num)` in libOUM. The replica then processes new client REQUESTs as in §5.2.3. Finally, it sends a ⟨SYNC-REPLY, *sync-point*⟩ to the leader, where *sync-point* is index of the last entry in the *log* the replica received from the leader.
3. Upon receiving SYNC-REPLY for *sync-point* from f different replicas, the leader broadcasts a ⟨SYNC-COMMIT, *sync-point*⟩ and updates its own *sync-point*.
4. When the replica receives a ⟨SYNC-COMMIT, *sync-point*⟩ from the leader with *sync-point* greater than its own:
   - If the replica already received the corresponding SYNC-PREPARE message from the leader, it updates its *sync-point* and can now safely execute all REQUESTs up to the *sync-point*.
   - If the replica did not already receive the SYNC-PREPARE messages, it requests it from the leader, processes it as above, and then updates its *sync-point*.

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 [43]). 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.

[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.

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 [38]). 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. §6.3 demonstrates that NOPaxos achieves the same throughput regardless of the number of replicas.

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 [34, 49], 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 B 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 pro-

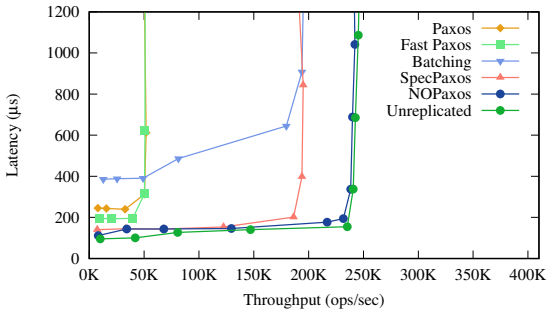

Figure 7: Latency vs. throughput comparison for testbed deployment of NOPaxos and other protocols.

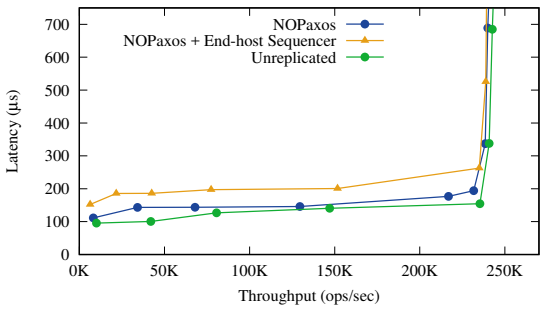

Figure 8: Comparison of running NOPaxos with the prototype Cavium sequencer and an end-host sequencer.

cessing 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 Evaluation

We implemented the NOPaxos protocol in approximately 5,000 lines of C++ code. We ran our experiments using the 3-level fat-tree network testbed shown in Figure 3. All clients and replicas ran on servers with 2.5 GHz Intel Xeon E5-2680 processors and 64GB of RAM. All experiments used five replicas (thereby tolerating two replica failures).

To evaluate the performance of NOPaxos, we compared it to four other replication protocols: Paxos, Fast Paxos, Paxos with batching, and Speculative Paxos; we also evaluated it against an unreplicated system that provides no fault tolerance. Like NOPaxos, the clients in both Speculative Paxos and Fast Paxos multicast their requests to the replicas through a root serialization switch to minimize message reordering. Requests from NOPaxos clients, however, are also routed through the Cavium processor to be stamped with the sequencer's OUM session number and current request sequence number. For the batching variant of Paxos, we used a sliding-window technique where the system adaptively adjusts the batch size, keeping at least one batch in progress at all times; this approach reduces latency at low load while still providing throughput benefits at high load [13].

6.1 Latency vs. Throughput

To compare the latency and throughput of NOPaxos and the other four protocols, we ran each system with an increasing number of concurrent closed-loop clients. Figure 7 shows results of this experiment. NOPaxos achieves a much higher maximum throughput than Paxos and Fast Paxos (370% increases in both cases) without any additional latency. The leaders in both Paxos and Fast Paxos send and receive more messages than the other replicas, and the leaders' message processing quickly becomes the bottleneck of these systems. NOPaxos has no such inefficiency. NOPaxos also achieves higher throughput than Speculative Paxos (24% increase) because Speculative Paxos requires replicas to compute hashes of their logs for each client request.

Figure 7 also shows that NOPaxos has lower latency (111 μs) than Paxos (240 μs) and Fast Paxos (193 μs) because NOPaxos requires fewer message delays in the normal case. Speculative Paxos also has higher latency than NOPaxos because clients must wait for a superquorum of replica replies instead of NOPaxos's simple quorum.

Batching improves Paxos's throughput by reducing the number of messages sent by the leader. Paxos with batching is able to reach a maximum throughput equivalent to Speculative Paxos. However, batching also increases the latency of Paxos (385 μs at low load and 907 μs at maximum throughput). NOPaxos attains *both* higher throughput and lower latency than Paxos with batching.

NOPaxos is able to attain throughput within 2% of an unreplicated system and latency within 16 μs. However, we note that our middlebox prototype adds around 8 μs to NOPaxos's latency. We envision that implementing the sequencer in a switch could bring NOPaxos's latency even closer to the unreplicated system. This demonstrates that NOPaxos can achieve close to optimal performance while providing fault-tolerance and strong consistency.

We also evaluated the performance of NOPaxos when using an end-host as the sequencer instead of the network processor. Figure 8 shows that NOPaxos still achieves impressive throughput when using an end-host sequencer, though at a cost of 36% more latency due to the additional message delay required.

6.2 Resilience to Network Anomalies

To test the performance of NOPaxos in an unreliable network, we randomly dropped a fraction of all packets. Figure 9 shows the maximum throughput of the five protocols and the unreplicated system with an increasing packet drop rate. Paxos's and Fast Paxos's throughput

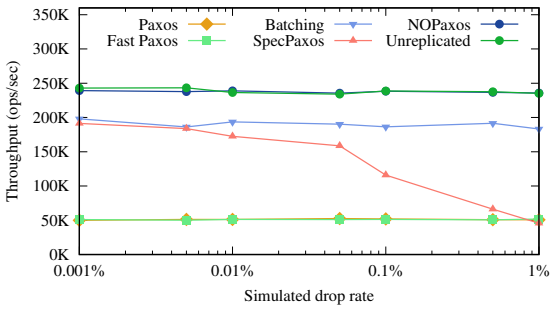

Figure 9: Maximum throughput with simulated packet dropping.

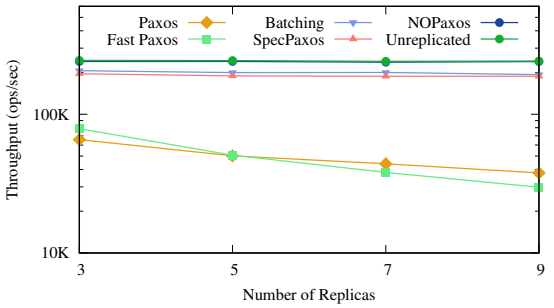

Figure 10: Maximum throughput with increasing number of replicas.

do not decrease significantly, while Paxos with batching shows a larger drop in throughput due to frequent state transfers. However, the throughput of Speculative Paxos drops substantially after 0.5% packet dropping, demonstrating NOPaxos's largest advantage over Speculative Paxos. When 1% of packets are dropped, Speculative Paxos's maximum throughput falls to that of Paxos. As discussed in §5.3, Speculative Paxos performs an expensive reconciliation protocol when messages are dropped and replica states diverge. NOPaxos is much more resilient to packet drops and reorderings. It achieves higher throughput than Paxos with batching and much higher throughput than Speculative Paxos at high drop rates. Even with a 1% message drop rate, NOPaxos's throughput does not drop significantly. Indeed, NOPaxos maintains throughput roughly equivalent to an unreplicated system, demonstrating its strong resilience to network anomalies.

6.3 Scalability

To test NOPaxos's scalability, we measured the maximum throughput of the five protocols running on increasing number of replicas. Figure 10 shows that both Paxos and Fast Paxos suffer throughput degradation proportional to the number of replicas because the leaders in those protocols have to process more messages from the additional replicas. Replicas in NOPaxos and Speculative Paxos, however, process a constant number of messages, so those protocols maintain their throughput when more replicas are added.

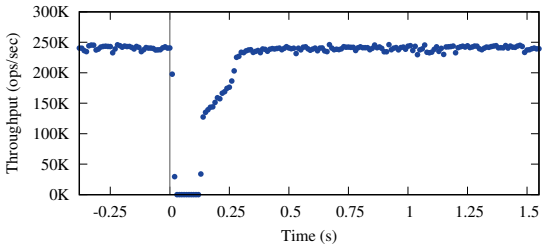

Figure 11: NOPaxos throughput during a sequencer failover.

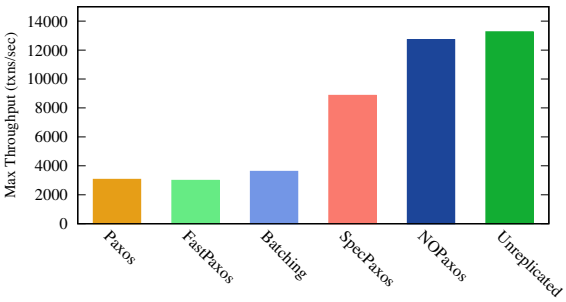

Figure 12: Maximum throughput achieved by a replicated transactional key-value store within 10 ms SLO.

6.4 Sequencer Failover

NOPaxos relies on the sequencer to order client requests. We measured the throughput of NOPaxos during a sequencer failover (Figure 11). We ran NOPaxos at peak throughput for approximately 7 seconds. We then simulated a sequencer failure by sending the controller a notification message. The controller modified the routing rules in the network and installed a new session number in the sequencer (as described in §3). The throughput of the system drops to zero during the failover and takes approximately 110 ms to resume normal operations and approximately 270 ms to resume processing operations at peak throughput. Most of this delay is caused by the route update rather than the NOPaxos view change.

6.5 Application Performance

To further demonstrate the benefits of the NOPaxos protocol, we evaluated the performance of a distributed, in-memory key-value store. The key-value store uses two-phase commit and optimistic concurrency control to support serializable transactions, and each shard runs atop our replication framework. Clients issue `GET` and `PUT` requests within transactions. We benchmarked the key-value store using a workload based on the Retwis Twitter clone [39].

Figure 12 shows the maximum throughput of the key-value store with a 10ms SLO. NOPaxos outperforms all other variants on this metric: it attains more than 4 times the throughput of Paxos, and outperforms the best prior protocol, Speculative Paxos, by 45%. Its throughput is also within 4% that of an unreplicated system.

7 Related Work

Our work draws on techniques from consensus 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 [37], Speculative Paxos [54], and Optimistic Atomic Broadcast [30, 51, 52]; Zyzzyva [33] applies a similar idea in the context of Byzantine fault tolerant replication. These protocols can reduce consensus 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 [38].

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 [12, 36, 46, 61]; 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 [6, 7]. Years ago, a great debate raged in the SOSP community about the effectiveness and limits of this causal and totally ordered communication support (CATOCS) [5, 15]. 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 [15, 56].

Network-level processing NOPaxos takes advantage of flexible network processing to implement the OUM model. Many designs have been proposed for flexible processing, including fully flexible, software-based designs like Click [31] and others based on network processors [58] or FPGA platforms [48]. At the other extreme, existing software defined networking mechanisms like OpenFlow [45] can easily achieve line-rate performance in commodity hardware implementations but lack the flexibility to implement our multicast primitive. We use the P4 language [9], which supports several high-performance hardware designs like Reconfigurable Match Tables [10].

These processing elements have generally been used for classic networking tasks like congestion control or queue management. A notable exception is SwitchKV [42], which uses OpenFlow switches for content-based routing and load balancing in key-value stores.

...and their intersection Recent work on Speculative Paxos and Mostly-Ordered Multicast proposes co-designing network primitives and consensus protocols to achieve faster performance. Our work takes the next step in this direction. While Speculative Paxos assumes only a best-effort ordering property, NOPaxos requires an ordering guarantee. Achieving this guarantee requires more sophisticated network support made possible with a programmable data plane (Speculative Paxos's Mostly-Ordered Multicast requires only OpenFlow support). However, as discussed in §5.3, NOPaxos achieves a simpler and more robust protocol as a result, avoiding the need for superquorums and speculation.

A concurrent effort, NetPaxos [18], also explores ways to use the network layer to improve the performance of a replication protocol. That work proposes moving the Paxos logic into switches, with one switch serving as a coordinator and others as Paxos acceptors. This logic can also be implemented using P4 [17]. However, as the authors note, this approach requires the switches to implement substantial parts of the logic, including storing potentially large amounts of state (the results of each consensus instance). Our work takes a more practical approach by splitting the responsibility between the OUM network model, which can be readily implemented, and the NOPaxos consensus protocol.

Other related work uses hardware acceleration to speed communication between nodes in a distributed system. FaRM [19] uses RDMA to bypass the kernel and minimize CPU involvement in remote memory accesses. Consensus in a Box [26] implements a standard atomic broadcast protocol entirely on FPGAs. NOPaxos provides more flexible deployment options. However, its protocol could be integrated with RDMA or other kernel-bypass networking for faster replica performance.

8 Conclusions

We presented a new approach to high-performance, fault-tolerant replication, one based on dividing the responsibility for consistency between the network layer and the replication protocol. In our approach, the network is responsible for ordering, while the replication protocol ensures reliable delivery. "Splitting the atom" in this way yields dramatic performance gains: network-level ordering, while readily achievable, supports NOPaxos, a simpler replication protocol that avoids coordination in most cases. The resulting system outperforms state-of-the-art replication protocols on latency, throughput, and application-level metrics, demonstrating the power of this approach. More significantly, it achieves both throughput and latency equivalent to an unreplicated system, proving that replication does not have to come with a performance

cost.

Acknowledgments

We thank Irene Zhang, the anonymous reviewers, and our shepherd Dawn Song for their helpful feedback. This research was supported by the National Science Foundation under awards CNS-1518702 and CNS-1615102 and by gifts from Google and VMware.

References

[1] 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.

[2] Arista Networks. 7150 series ultra low latency switch. `https://www.arista.com/assets/data/pdf/Datasheets/7150S_Datasheet.pdf`.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, Oct. 2014. USENIX.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, Melbourne, Australia, 2010. ACM.

[5] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *ACM SIGOPS Operating Systems Review*, 28(1), Jan. 1994.

[6] 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.

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

[8] 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.

[9] 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), July 2014.

[10] 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*. ACM, 2013.

[11] 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.

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

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, USA, Feb. 1999.

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

[15] 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.

[16] 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.

[17] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. Network hardware-accelerated consensus. Technical Report USI-INF-TR-2016-03, Università della Svizzera italiana, May 2016.

[18] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, New York, NY, USA, 2015. ACM.

[19] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, Apr. 2014. 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), Apr. 1985.

[21] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Annual Tech-*

nical Conference, San Jose, CA, USA, June 2013. USENIX.

[22] 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.

[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), 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] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, Mar. 2016. USENIX Association.

[27] 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.

[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, 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)*, Denver, CO, June 2016. USENIX Association.

[30] 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.

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

[32] 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.

[33] 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.

[34] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[35] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), Dec. 2001.

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

[37] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2), Oct. 2006.

[38] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2), Oct. 2006.

[39] C. Leau. Spring Data Redis – Retwis-J, 2013. `http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/`.

[40] 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. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, Nov. 2016. USENIX Association.

[41] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the 5th Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, USA, Nov. 2014. ACM.

[42] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, Mar. 2016. USENIX Association.

[43] 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.

[44] 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.

[45] 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), Apr. 2008.

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

[47] 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.

[48] 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, New York, NY, USA, 2008. ACM.

[49] 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.

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

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

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

[53] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, Oct. 2014. USENIX.

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

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

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

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

[58] 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.

[59] 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.

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

[61] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. of SOSP*, 2015.

A Correctness of NOPaxos

A.1 Comparable Views

First, we need to show that the leader's selection of candidate log in §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 (§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

successful. 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 TLA+ Specification

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 $\mapsto n \in (1 \,..\,)$]

$RequestReply$
$[\ mtype \mapsto MRequestReply,$
$\ sender \mapsto r \in Replicas,$
$\ viewID \mapsto v \in ViewIDs,$
$\ request \mapsto v \in Values \cup \{NoOp\},$
$\ logSlotNum \mapsto n \in\ (1\ ..\ )\ ]$

$SlotLookup$
$[\ mtype \mapsto MSlotLookup,$
$\ dest \mapsto r \in Replicas,$
$\ sender \mapsto r \in Replicas,$
$\ viewID \mapsto v \in ViewIDs,$
$\ sessMsgNum \mapsto n \in\ (1\ ..\ )\ ]$

$GapCommit$
$[\ mtype \mapsto MGapCommit,$
$\ dest \mapsto r \in Replicas,$
$\ viewID \mapsto v \in ViewIDs,$
$\ slotNumber \mapsto n \in\ (1\ ..\ )\ ]$

$GapCommitRep$
$[\ mtype \mapsto MGapCommitRep,$
$\ dest \mapsto r \in Replicas,$
$\ sender \mapsto r \in Replicas,$
$\ viewID \mapsto v \in ViewIDs,$
$\ slotNumber \mapsto n \in\ (1\ ..\ )\ ]$

$ViewChangeReq$
$[\ mtype \mapsto MViewChangeReq,$
$\ dest \mapsto r \in Replicas,$
$\ viewID \mapsto v \in ViewIDs\ ]$

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

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

$SyncPrepare$
$[\ mtype \mapsto MSyncPrepare,$
$\ dest \mapsto r \in Replicas,$
$\ sender \mapsto r \in Replicas,$
$\ viewID \mapsto v \in ViewIDs,$

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

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

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

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

Set of all vars

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

Initial state

$Init \triangleq \land InitNetworkState$
$\land InitSequencerState$
$\land InitReplicaState$

Helpers

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

View ID Helpers

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

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.

$CombineLogs(ls) \triangleq$
LET
$combineSlot(xs) \triangleq \text{IF } NoOp \in xs \text{ THEN } NoOp$

```
                                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]
```

```
  ∧ vGapCommitReps' =
      [vGapCommitReps EXCEPT ![r] = vGapCommitReps[r] ∪ {m}]
  When there's enough, resume StNormal and process more messages
  ∧ LET isViewPromise(M) ≜ ∧ {n.sender : n ∈ M} ∈ Quorums
                           ∧ ∃ n ∈ M : n.sender = r
        gCRs             ≜ {n ∈ vGapCommitReps'[r] :
                              ∧ n.mtype      = MGapCommitRep
                              ∧ n.viewID     = vViewID[r]
                              ∧ n.slotNumber = vCurrentGapSlot[r]}
    IN
      IF isViewPromise(gCRs) THEN
        vReplicaStatus' = [vReplicaStatus EXCEPT ![r] = StNormal]
      ELSE
        UNCHANGED vReplicaStatus
  ∧ UNCHANGED ⟨sequencerVars, networkVars, vLog, vViewID, vCurrentGapSlot,
               vSessMsgNum, vLastNormView, vViewChanges, vSyncPoint,
               vTentativeSync, vSyncReps⟩

  Failure Cases
  Replica r starts a Leader change
StartLeaderChange(r) ≜
  LET
    newViewID ≜ [sessNum    ↦ vViewID[r].sessNum,
                 leaderNum ↦ vViewID[r].leaderNum + 1]
  IN
  ∧ Send({[mtype  ↦ MViewChangeReq,
           dest   ↦ d,
           viewID ↦ newViewID] : d ∈ Replicas})
  ∧ UNCHANGED ⟨replicaVars, sequencerVars⟩

  View Change Handlers
  Replica r gets MViewChangeReq, m
HandleViewChangeReq(r, m) ≜
  LET
    currentViewID ≜ vViewID[r]
    newSessNum    ≜ Max({currentViewID.sessNum, m.viewID.sessNum})
    newLeaderNum  ≜ Max({currentViewID.leaderNum, m.viewID.leaderNum})
    newViewID     ≜ [sessNum ↦ newSessNum, leaderNum ↦ newLeaderNum]
  IN
  ∧ currentViewID  ≠ newViewID
  ∧ vReplicaStatus' = [vReplicaStatus EXCEPT ![r] = StViewChange]
  ∧ vViewID'        = [vViewID EXCEPT ![r] = newViewID]
  ∧ vViewChanges'   = [vViewChanges EXCEPT ![r] = {}]
  ∧ Send({[mtype    ↦ MViewChange,
           dest     ↦ 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

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

Replica r receives *MSyncRep*, m

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

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

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

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

C P4 Sequencer Implementation

Headers.p4

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

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

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

        /* These are nopaxos specific fields
         * appended to a standard UDP packet. */
        replicaGroup : 16;
        sessionNumber : 16;
        sequenceNumber : 32;
    }
}
```

Parser.p4

```
#define ETHERTYPE_IPV4 0x0800
#define IP_PROTOCOL_NOPAXOS 253

parser start {
    return parse_ethernet;
}

header ethernet_t ethernet;

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

header ipv4_t ipv4;

parser parse_ipv4 {
    extract(ipv4);
    return select(latest.protocol) {
```

```
        IP_PROTOCOL_NOPAXOS : parse_nopaxos;
        default: ingress;
    }
}

header udp_nopaxos_t nopaxos;

parser parse_nopaxos {
    extract(nopaxos);
    return ingress;
}
```

NOPaxos.p4

```
#include "Headers.p4"
#include "Parser.p4"

#define MAX_REPLICA_GROUPS 65536

/* This structure contains the data that moves through the packet
 * processing pipeline. */
header_type ingress_metadata_t {
    fields {
        sessionNumber : 16;
        sequenceNumber : 32;
    }
}

metadata ingress_metadata_t ingress_data;

/* This stateful array of registers keep track of the current OUM session
 * each replica group is in.
 * session_number [i] = OUM session of replica group i */
register session_number {
    width : 16;
    instance_count : MAX_REPLICA_GROUPS;
}

/* This array of registers keep track of the current sequence number
 * of each replica group.
 * sequence_number [i] = sequence number of replica group i */
register sequence_number {
    width : 32;
    instance_count : MAX_REPLICA_GROUPS;
}

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

    // Reads the values from stateful memory into ingress metadata.
    register_read(ingress_data.sessionNumber, session_number, nopaxos.replicaGroup);
    register_read(ingress_data.sequenceNumber, sequence_number, nopaxos.replicaGroup);

    // Updates the packet header fields to above read session and sequence number.
    modify_field(udp_nopaxos.sessionNumber, ingress_data.sessionNumber);
    modify_field(udp_nopaxos.sequenceNumber, ingress_data.sequenceNumber);

    // Increment the sequence number by 1.
    add_to_field(ingress_data.sequenceNumber, 1);

    // Write back the updated value into stateful memory.
    register_write(sequence_number, nopaxos.replicaGroup, ingress_data.sequenceNumber);
}

/* M+A table that applies nopaxos actions. */
table nopaxos {
    actions { fill_nopaxos_fields; }
```

```
55  }
56
57  /* Packet processing starts here. */
58  control ingress {
59      // Apply the nopaxos update actions.
60      apply(nopaxos);
61
62      // The usual forwarding.
63      apply(forward);
64  }
```