Hydra: Serialization-Free Network Ordering for Strongly Consistent Distributed Applications

Inho Choi[1], Ellis Michael[2], Yunfan Li[1], Dan R. K. Ports[3], and Jialin Li[1]

[1]*National University of Singapore,* [2]*University of Washington,* [3]*Microsoft Research*

Abstract

Many distributed systems, e.g., state machine replication and distributed databases, rely on establishing a consistent order of operations on groups of nodes in the system. Traditionally, this ordering has been established by application-level protocols like Paxos or two-phase locking. Recent work has shown significant performance improvements are attainable by making ordering a network service, but current network sequencing implementations require routing all requests through a single sequencer – leading to scalability, fault tolerance, and load balancing limitations.

Our work, Hydra, overcomes these limitations by using a *distributed* set of network sequencers to provide network ordering. Hydra leverages loosely synchronized clocks on network sequencers to establish message ordering across them, per-sequencer sequence numbers to detect message drops, and periodic timestamp messages to enforce progress when some sequencers are idle. To demonstrate the benefit of Hydra, we co-designed a state machine replication protocol and a distributed transactional system using the Hydra network primitive. Compared to serialization-based network ordering systems, Hydra shows equivalent performance improvement over traditional approaches in both applications, but with significantly higher scalability, shorter sequencer failover time, and better network-level load balancing.

1 Introduction

Replication is ubiquitous in data center applications. Consensus protocols like Paxos, Viewstamped Replication, and Raft are used to maintain multiple copies of data, providing the illusion of a single correct service that remains available even as individual replicas fail and recover. However, these protocols impose substantial latency and throughput overhead.

A recent line of work demonstrated that in-network processing can alleviate this cost [42, 43, 56]. This *network sequencing* approach routes requests through a sequencer – implemented in a programmable switch or middlebox – which assigns a monotonically increasing sequence number to each request. By pre-establishing a total order of all requests, they enable lighter weight consensus protocols, ultimately yielding impressive performance gains: Network-Ordered Paxos achieves throughput within 2% and latency within 10% of an unreplicated, non-fault-tolerant system [43].

However, employing this approach in practice is not easy. Fundamentally, the difficulty stems from the fact that *network sequencing requires serialization*: all traffic for a replicated service must pass through a single sequencer. This poses three major challenges in production networks. First, the single sequencer must process all request traffic, posing a scalability bottleneck. Second, it imposes a new routing requirement for specific application traffic, which network operators are loath to accept. Restricting path diversity interferes with existing policies, carefully engineered for load balancing and fault tolerance. Finally, it introduces an undesirable coupling between network and application-level recovery. Replacing a failed or unreachable sequencer requires coordinating a simultaneous update to the network routes and recovery of the sequencer state (via a consensus protocol). This adds deployment complexity and increases system downtime during the recovery process. All three are serious barriers to adoption, based on our experiences with large-scale production data centers.

This paper asks whether network sequencing can be achieved *without* serialization. We answer that question in the affirmative by presenting the design of Hydra,[1] a new protocol for network sequencing that allows packets to be sequenced by multiple active sequencers. Hydra's sequencers themselves run a lightweight coordination protocol, in which each sequencer independently assigns sequence numbers to requests that can be merged to establish a total order of operations. Specifically, Hydra leverages a combination of per-sequencer sequence numbers and loosely synchronized physical clocks across sequencers to assign a global ordering while still efficiently detecting dropped messages.

Hydra is a practical protocol; we have built both a software implementation that runs on end hosts and one in P4 [11] that runs on an Intel Tofino programmable switch; the latter uses only a small fraction of switch resources, demonstrating its practicality for modern network devices. Hydra's sequencing functionality allows it to run the existing NOPaxos [43] and Eris [42] replication and transaction processing protocols with minimal modification, while making them more resilient to sequencer faults with marginal performance cost.

Our evaluation results demonstrate that Hydra achieves a 378% increase in throughput and 42% reduction in latency compared to an atomic multicast baseline, while scaling to high numbers of receivers, multicast groups, and sequencers. Comparing to systems that use a network serialization approach, Hydra significantly improves network-level load bal-

[1]Hydra is named after the Lernean Hydra of Greek mythology, a multi-headed serpent that could regrow a new head if one was chopped off [31].

ancing and reduces system downtime by 5×. Moreover, Hydra achieves these benefits without sacrificing performance: our Hydra-based state machine replication system gets latency within 5 μs and throughput within 17% of NOPaxos, and our transactional system attains 47% higher throughput than Eris.

2 Background

Establishing a consistent order of operations is fundamental to many distributed systems: state machine replication [43,52,53, 61] requires all correct replicas to execute a totally ordered set of client operations; distributed transactional systems [5, 14, 17, 19, 26, 39] mandate that all shards of the data store process transactions in a serializable order; distributed caches [50, 55] require consistent updates to ensure coherence.

Traditionally, guaranteeing strong consistency necessitates running complex application-level distributed protocols which involve coordination among servers. For instance, many state machine replication protocols [52, 53, 61] designate a single leader to assign an order to operations, and require it to communicate with replicas before returning a result to the client, and existing distributed databases execute concurrency control, atomic commitment, and consensus protocols for each client transaction. This expensive coordination is at odds with the demanding throughput, latency, and scalability requirements of modern data center applications.

2.1 Request Ordering in the Network

The need for these protocols stems from the fundamental assumption of a fully *asynchronous* network which can arbitrarily drop, reorder, or delay messages. A classic line of work in distributed computing proposes stronger communication primitives to simplify distributed applications, including virtual synchrony [8, 9], atomic broadcast [10, 34], and atomic multicast [28]. These provide broadcast or multicast operations that ensure all correct receivers will deliver the *same set* of messages in the *same order*. Such guarantees can obviate the need for consensus protocols – but implementing them is a problem equivalent to consensus [13], so applications do not enjoy a performance benefit.

Network ordering without reliability guarantees. A recent line of work [42, 43, 56] proposes a new network model that balances guarantees and implementation efficiency. This new model moves the responsibility of consistent message ordering into the network, but leaves reliable delivery of messages to application-level protocols. By providing ordering guarantees in the network, this network/protocol co-design approach allows faster replication protocols than traditional designs; by *not* enforcing reliability, the network model is simple enough to implement efficiently.

A key mechanism employed by these systems to implement network ordering is *in-network serialization*. For instance, Speculative Paxos [56] routes all client requests first to a designated switch in the network before multicasting to the replica servers. The single switch serves as a serialization point and ensures that, with high probability, all replicas receive client requests in the same order. NOPaxos [43] extends this serialization approach by using programmable switch ASICs to provide *guarantees* of request ordering. The designated switch stamps a sequence number into each client request. Receivers then ensure consistent ordering by processing requests only in sequence number order. Additionally, sequence numbers allow replicas to identify dropped messages (by detecting gaps in the sequence).

Eris [42] further generalizes the sequencing approach to support requests that are sent to multiple replication groups (e.g., to implement fault-tolerant distributed transactions). The sequencer switch maintains a counter for each group, and on each client request, *atomically* increments the counter value for all destination groups. These counter vectors ensure a consistent ordering of all multi-group operations, while still allowing receivers to independently detect dropped messages.

2.2 Limitations of In-Network Serialization

Prior work [42, 43, 56] has demonstrated the performance benefits of the network ordering approach and its applicability to several classes of distributed systems. However, the in-network serialization approach employed by existing network ordering solutions has important limitations.

Scalability bottleneck. A key requirement in the serialization approach is that all client requests need to go through a *single sequencer device*. This device can become a performance bottleneck. While in-switch sequencers can sustain a far higher sequencing rate than the server-based replicas that actually execute operations, sequencer capacity can still be a scalability limit for sharded database systems like Eris [42] where one sequencer serves many replica groups. Moreover, if the sequencer is implemented on an end host, as can be more practical for many deployments, poor CPU-based packet processing performance is at odds with the horizontal scaling capability of the system.

Prolonged system downtime. Being a serialization point of all client requests, a failure in the sequencer will result in unavailability of the entire distributed system. Sequencer failover is more complicated than traditional recovery (e.g., changing leaders in a Paxos deployment) because it couples network rerouting with application-level recovery. To resume operation, the network control plane must first detect the failure and carry out network-wide routing changes to redirect client traffic to a new sequencer, and afterwards begin a view change procedure to ensure system state is consistent; only then can replicas process requests from the new sequencer. Rerouting in a large data center network is expensive: previous studies [38, 42, 43] show that updating forwarding tables in a single switch alone can take more than 200 ms. Before this lengthy rerouting procedure is complete, the system will remain unavailable.

Worsened data center network properties. Data center networks are carefully engineered to provide high reliability, performance, and cost efficiency [2, 27, 49]. By adding redundant paths to the network and using protocols like ECMP, these networks can effectively load balance network traffic, tolerate link and switch failures, and sustain high bisection bandwidth. Serializing traffic through a single switch, however, reduces the number of available paths and can easily nullify these desirable network properties. For example, it can lead to link congestion at the sequencer switch.

Incompatible with multi-pipeline switches. Many switch ASICs scale out processing capacity by using multiple (e.g., 2–8 [18]) separate pipelines, with few or no shared resources. Existing sequencing approaches, however, update and atomically read a single copy of the sequence number. This requirement restricts deploying network sequencing logic to a single switch pipeline [36]. This not only limits the maximum throughput of the network sequencer to a fraction of the switch capacity, it complicates cabling and routing because specific physical ports are bound to each pipeline.

3 Sequencing with Multiple Sequencers

Hydra allows multiple active sequencers to work concurrently, preventing a single sequencer from becoming a scalability bottleneck or a single point of failure. This allows Hydra to support new deployment models for network sequencers.

3.1 Deployment Options

Hydra supports a spectrum of deployment models.

Root switches. Prior work envisioned using programmable switches at the root of a tree topology as sequencers, leveraging their centrality in the data center network. Such switches can handle high request load, making scalability beyond a single switch's capacity a less urgent concern, but they do so through the use of multiple ASICs and forwarding pipelines which prior sequencer designs do not support. Hydra can also improve availability by decoupling sequencer failover from reroute latency of the underlying network (§7.5). In addition, using multiple Hydra sequencers rather than routing all sequenced traffic through one switch provides path diversity, which allows better link-level load balancing and resilience to link failures (§7.4).

ToR switches. Many existing data center architectures cannot deploy programmable switches at the network core: they use large, multi-ASIC chassis switches at the root layer [15, 27], and programmable switches are not available in this configuration. For example, Tofino-based switches are only available in smaller 32/64-port configurations. For many scenarios, using top-of-rack switches as sequencers is thus a more practical alternative. In such deployments, scalability and fault tolerance are acute concerns: ToR switches fail more commonly [25] and frequently experience congestion on their uplinks. Hydra can avoid both problems by employing multiple sequencers (§7.4).

Sequencer appliances. In our experience, incrementally deploying new functionality in existing switches, ToR or otherwise, can be a challenge: coordinating updates with existing switch functionality and validating the correctness of a custom data plane are both obstacles. An appealing alternate approach is to employ a cluster of switches as dedicated "sequencer appliances" attached to the network as edge devices rather than being part of the fabric [57], as proposed for other network function accelerators [37, 64]. Again, fault tolerance of individual sequencers and congestion on their network links (which may not exploit the full bandwidth of the switch) are major concerns, which Hydra can alleviate.

End hosts. A final approach eschews specialized hardware in favor of using end hosts as sequencers [43]. This offers obvious deployment benefits and may be the only practical approach for many environments. However, both scalability and fault tolerance are critical here: Eris's end-host sequencer barely handles the load of a 15-shard database [42], making it an option only for smaller deployments. Hydra's multiple-sequencer approach allows it to go beyond this limit, providing a practical, scalable approach for environments where specialized hardware is unavailable (§7.1.3).

3.2 Addressing and Routing

Regardless of deployment options, Hydra integrates easily with existing data center routing structures. Each Hydra deployment has a unique IP address. Each sequencer in the deployment advertises its IP address via BGP anycast, allowing routes to be dynamically updated as sequencers join or leave the deployment. Messages are routed to individual sequencers using traditional shortest-path routing and load balancing techniques, e.g., ECMP. Alternatively, in an SDN-oriented design with a centralized controller, the network controller can install appropriate anycast routes for the group of sequencers.

Apart from these routing changes, Hydra does not require any changes to any other elements in the network besides the sequencers themselves. This is a key design constraint, and one that differentiates Hydra from other ordering approaches like 1Pipe [41], which exchanges timestamps between every switch, as well as complementary techniques like RDMA, which requires complex in-network flow control [29].

4 Hydra: Serialization-Free Network Ordering

4.1 High-Level Abstraction

The core abstraction provided by Hydra is a group communication protocol. A Hydra deployment consists of receiver *groups*, and each group contains one or more *receivers*. Hydra offers a *groupcast* primitive, where a sender specifies one or multiple groups as the destination, and the message is multicast to the receivers in the destination groups. The Hydra groupcast primitive provides the following properties to the

participants:

- **Partial Ordering.** Hydra groupcast messages are partially ordered (the partial order relation is denoted as $\prec$), with all groupcast messages with overlapping destination groups are comparable. If groupcast message m_1 is ordered before m_2 ($m_1 \prec m_2$) and a receiver receives both m_1 and m_2, then every receiver delivers m_1 before m_2.
- **Unreliable Delivery.** Hydra only offers best effort message delivery. A groupcast message is not guaranteed to be delivered to any of its recipients.
- **Drop Detection.** If a groupcast message is not delivered to all its recipients, the primitive will notify the remaining receivers by delivering a DROP-NOTIFICATION. More formally, let R be the set of receiver groups for message m, then either one of the following two conditions holds: **all** receiver groups in R deliver m or a DROP-NOTIFICATION for m, or **none** of the receiver groups in R delivers m or a DROP-NOTIFICATION for m.

These are the same guarantees provided by the network abstractions in NOPaxos and Eris [42, 43]. However, critically, Hydra allows scalability, fast failure recovery, and load balancing across sequencers, where previous designs fell short.

4.2 Prior Approach: Centralized Sequencer

A recent line of work [6, 7, 42, 43, 63] proposed to use dedicated devices in the network – a programmable switch, a network processor, or an end-host server – as a centralized sequencer to establish message ordering. In particular, Eris [42] builds a *multi-sequenced groupcast* primitive that provides the same set of guarantees as we specified in §4.1.

To implement multi-sequenced groupcast, a centralized sequencer maintains a *sequence number* for each group in the system. Senders of a groupcast message encode all recipient groups in a special packet header, and the packet is first routed to the sequencer. Upon receiving a groupcast packet, the sequencer atomically increments the sequence number for each recipient group, and writes a *multi-stamp* into the packet. The multi-stamp contains a set of $\langle$*group-id*, *sequence-num*$\rangle$, one for each recipient group. The groupcast packet is then forwarded to each receiver in each receiver group.

Groupcast receivers track the next sequence number they expect from the sequencer. When a receiver receives a groupcast packet, it checks the sequence number that corresponds to its group ID in the multi-stamp. The receiver rejects the packet if the sequence number is lower than the expected value (indicating out-of-order or duplicated messages), and delivers a DROP-NOTIFICATION to the application if the sequence number is higher than expected.

Multi-sequencing provides the three properties of §4.1. By incrementing sequence numbers *atomically*, the sequencer ensures that if two groupcast messages have overlapping groups, all receivers in those groups will deliver the two messages in a consistent order. By maintaining per-group sequence numbers, any packet loss from the sequencer to a receiver will result in a gap in the received sequence numbers, and hence a DROP-NOTIFICATION. However, using a centralized sequencer introduces the limitations previously described.

Algorithm 1 SequencerHandlePacket(pkt)

```
id: sequencer ID
N: total number of Hydra groups
clk: switch physical clock
seq[N]: sequence number for each group
1: pkt.id ← id
2: pkt.c ← clk
3: for grp in pkt.grps do
4:     pkt.seq[grp] ← seq[grp]++
5: end for
6: Forward pkt
```

4.3 Consistent Ordering with Multiple Sequencers

Naïvely applying multi-sequenced groupcast to a multi-sequencer deployment violates the guarantees listed in §4.1. Suppose each sequencer independently maintains sequence numbers for each receiver group, and groupcast messages can be forwarded to any of the sequencers. Consider two groupcast messages m_1 and m_2, both destined to group G_1, but routed through two sequencers. The two sequencers may write the same sequence number (since they maintain sequence numbers independently) for G_1 into m_1 and m_2. When receivers in G_1 receive m_1 and m_2, they cannot consistently order the messages while providing drop detection. Breaking ordering ties with sequencer ID, for example, would be consistent across receivers, but a receiver that only received the "larger" of m_1 and m_2 would have no way of inferring the existence of the "smaller."

To enforce all the guarantees in §4.1 while scaling to multiple sequencers, we propose a new approach by combining *loosely synchronized clocks* across sequencers and *per-sequencer sequence numbers* to establish consistent ordering and detect drops, and using *periodic flush messages* to ensure receiver progress.

4.3.1 Physical Clocks for Message Ordering

Hydra uses a combination of sequence numbers and physical clocks to order messages. Concretely, each Hydra sequencer possesses a local *physical clock* that is strictly monotonically increasing; each sequencer also maintains a sequence number for each receiver group. Physical clocks are loosely synchronized across sequencers, but this is *not* required for safety; clock skew can only slow progress. Safety of Hydra only depends on physical clocks not drifting *backwards*. This requirement is already common practice: existing clock synchronization protocols such as NTP ensure that clocks can only move forward [51].

Each Hydra groupcast message is routed to one sequencer before being forwarded to all receivers in each destination group. When a sequencer receives a groupcast message, in addition to incrementing the sequence number for each recip-

Algorithm 2 ReceiverHandlePacket(pkt)

M: total number of sequencers
gid: receiver group ID
buf: ordered queue of undelivered messages
$s[M]$: latest sequence number delivered (per sequencer)
$c[M]$: largest clock value received (per sequencer)

```
1:  if pkt.seq[gid] ≤ s[pkt.id] then
2:      return
3:  end if
4:  c[pkt.id] ← max{c[pkt.id], pkt.c}
5:  Deliver DROP-NOTIFICATION for
        (s[pkt.id] + 1 ... pkt.seq[gid] − 1) (inclusive)
6:  s[pkt.id] ← pkt.seq[gid] − 1
7:  if pkt is not a flush message ∧ pkt ∉ buf then
8:      Add pkt to buf
9:  end if
10: for p in buf do
11:     if p.c ≤ min{c[m] : m ∈ (1...M)} then
12:         Dequeue p from buf and deliver p
13:         s[p.id] ← p.seq[gid]
14:     else
15:         break
16:     end if
17: end for
```

ient group and inserting a multi-stamp, it writes its current clock value into the packet (Algorithm 1 line 2-5). Note that reading the clock value and incrementing sequence numbers must be done in an atomic block. Strict monotonicity of physical clocks and the above atomicity requirement ensure the following: for any two groupcast messages m_1 and m_2 with an overlapping recipient group g sequenced by the same sequencer, $s_1 \neq s_2 \wedge (s_1 < s_2 \iff c_1 < c_2)$, where s_1 and s_2 are the assigned sequence numbers for g, and c_1 and c_2 are the assigned clock values.

With a clock value inserted into each groupcast message, Hydra defines the partial ordering ($\prec$) of groupcast messages in the following way: for groupcast messages m_1 and m_2 with overlapping recipient groups and clock values c_1 and c_2 sequenced by sequencers with IDs i and j, $m_1 \prec m_2$ if $c_1 < c_2 \vee (c_1 = c_2 \wedge i < j)$. Breaking ties between equal clock values using sequencer IDs is necessary in guaranteeing the partial order property in §4.1.

Hydra groupcast receivers deliver groupcast messages to their users *according to our partial order*. If a receiver receives groupcast messages m_2 before m_1 with $m_1 \prec m_2$, it must either deliver a DROP-NOTIFICATION for m_1 before delivering m_2 or add m_2 to a buffer until it receives m_1. However, delivery based on physical clocks alone is not strong enough to detect message drops.

4.3.2 Combining Physical Clocks and Multi-Stamps for Drop Detection

Attaching sequence numbers to messages offers the useful property that any dropped message can be detected by observing gaps in the number sequence. Unfortunately, this property is lost when using physical clocks to order messages – a receiver seeing a message with a clock value c cannot determine if it missed any message with $c' < c$. To detect message drops, Hydra combines physical clock values and sequence numbers from multiple sequencers. Hydra receivers buffer incoming messages and deliver them in clock value order, but *only* once they have determined – based on sequence numbers – that no message with a lower clock value from another sequencer will be delivered.

Specifically, each Hydra receiver maintains two values for each sequencer i (Algorithm 2): the latest group sequence number $s[i]$ it has delivered from i, and the largest clock value $c[i]$ among messages it has received from i. Let c_{min} be the minimum value among all $(c[i],\ i)$ tuples, ordered by $\prec$. A Hydra receiver delivers messages using the following rules:

(i) it delivers pending groupcast messages in clock value and sequencer ID order (line 10),
(ii) it will only deliver a single message or DROP-NOTIFICATION for each sequence number from each sequencer (lines 1, 6, and 13),
(iii) it only delivers groupcast messages m if $m \preceq c_{min}$ (line 11), and
(iv) when receiving groupcast message m with sequence number s from sequencer i, if $s > s[i] + 1$, it delivers a DROP-NOTIFICATION for each message between $s[i] + 1$ and s (line 5).

From our discussion in §4.3.1, rules (i) and (ii) ensure the partial ordering property of Hydra groupcast. To show how rules (iii) and (iv) enforce drop detection, we leverage a key invariant: for a receiver r in group g and for any groupcast message m that has g as a recipient group, if $m \preceq c_{min}$, then r has either received m, or has received another groupcast message m' stamped with a higher sequence number from the same sequencer. With this invariant, the drop detection property of Hydra groupcast is guaranteed, since r either delivers m (m is received and rule (i)(iii)), or delivers a DROP-NOTIFICATION for m (m' is received and rule (iv)). As an optimization, Hydra receivers can delay the delivery of DROP-NOTIFICATIONs until a message is needed to advance c_{min}. Because Hydra is robust to message reordering, this does not affect the correctness of the receiver protocol.

4.3.3 Ensuring Progress with Flush Messages

Our groupcast design so far ensures all the properties listed in §4.1, but has one remaining issue. In order for a receiver to make progress in delivering messages, it needs to receive groupcast messages from *all* sequencers to advance c_{min}. For instance, if a receiver has received message $m \succ c_{min}$, to deliver m, the receiver needs to receive messages from other sequencers to advance c_{min}. Consequently, any single sequencer that stays idle for an extended period of time would impede the progress of all groupcast receivers in the system.

To ensure progress in message delivery, each sequencer periodically sends a *flush* message to all groupcast receivers

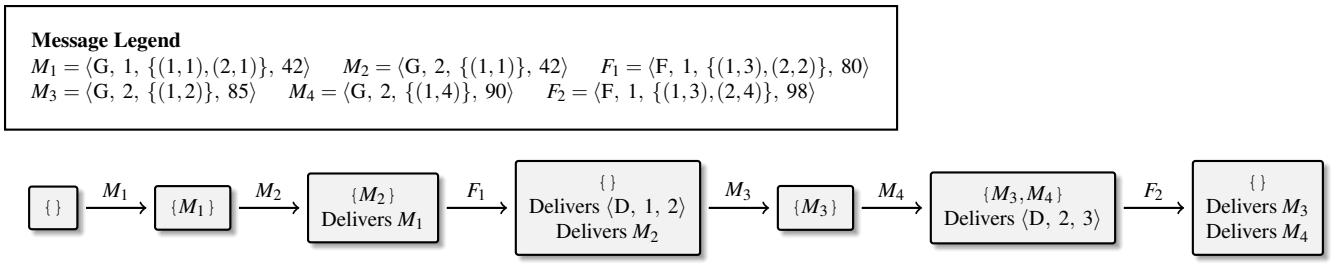


Figure 1: An example execution of the Hydra message delivery protocol. At every step, the state of the Hydra buffer is displayed along with any messages delivered to the application. Hydra groupcasts are written ⟨G, `sequencer`, `multi-stamp`, `timestamp`⟩*, while flush messages are written* ⟨F, `sequencer`, `multi-stamp`, `timestamp`⟩*, and* DROP-NOTIFICATIONS *are written* ⟨D, `sequencer`, `sequence_num`⟩*. A multi-stamp is a set of* (`group`, `sequence_num`) *tuples. This execution follows a receiver in group 1 receiving messages from two sequencers, 1 and 2. Transitions between states of the receiver show the message being received.*

containing its current clock value and the next sequence number that sequencer will send to each receiver group (without incrementing). When a receiver receives a flush message from sequencer i, it follows the same procedure (§4.3.2) to advance $c[i]$, c_{min}, and $s[i]$. Applications are unaware of flush messages. Receivers, however, still use the sequence number in flush messages to deliver DROP-NOTIFICATIONS, following rule (iv) (algorithm 2 line 5). Again, Hydra receivers can delay the delivery of DROP-NOTIFICATIONS until they are needed as an optimization.

The above protocol guarantees that, in the absence of failures (we discuss failure handling below), all received groupcast messages on every receiver will eventually be delivered. Clock divergence on different sequencers can delay message delivery (up to the clock skew), since c_{min} on each receiver depends on the sequencer with the slowest clock value, but cannot violate any of the safety properties.

Figure 1 shows an example execution of Hydra from the point of view of a single receiver receiving groupcast and flush messages from two sequencers. At every step of the execution, the receiver accepts an incoming message and delivers groupcasts and DROP-NOTIFICATIONS and retains pending groupcasts in its local buffer according to the rules defined in §4.3.2.

4.4 Handling Sequencer Failures

If a sequencer fails or link failures occur between a sequencer and some of the groups, some (or all) groupcast receivers will stop delivering messages, since they no longer receive messages from the failed sequencer and are unable to advance c_{min}. We use a reconfiguration protocol to address this issue.

Concretely, each Hydra deployment uses a centralized, fault-tolerant configuration service to manage a sequence of configurations. Each configuration specifies the set of sequencers and groupcast receivers (here we only discuss changes to sequencers across configurations). Groupcast receivers also store the current configuration locally. When a receiver suspects that sequencer j in the current configuration n has failed, e.g., when it has not received messages from sequencer j in a timeout period, it notifies the configuration service. The configuration service creates a new configuration $n+1$ with sequencer j removed, and sends the configuration to all groupcast receivers.

When groupcast receivers receive the new configuration, they run an agreement protocol to agree on the last sequence number each receiver group should deliver from the failed sequencer. To do so, each Hydra receiver additionally stores, for each sequencer, the largest sequence number it has seen in a multi-stamp for each receiver group (not just its own). To continue the sequencer removal process, each receiver sends a message with the largest sequence numbers (for all groups) it has received from the failed sequencer to the configuration service and stops processing messages with higher sequence numbers from that sequencer. Once the configuration service receives a quorum of these messages from each receiver group, it aggregates them to derive the highest sequence number each receiver group has or should have received from the failed sequencer. The configuration service sends a removal message to each group. A groupcast receiver delivers all necessary DROP-NOTIFICATIONS based on this removal message and continues to deliver messages following the rules in §4.3.2; the removal message serves as a final flush from the failed sequencer (with an infinitely large timestamp). Once all pending messages from the failed sequencer have been delivered, the receiver can safely transition to the new configuration. To avoid inconsistencies caused by different configurations, a receiver always attaches its current configuration number when delivering messages to the application.

Discussion. How does Hydra's recovery protocol compare, in terms of availability, to single-sequencer systems like those originally used by NOPaxos and Eris? Like these systems, Hydra experiences an interruption in message delivery caused by the failure of a sequencer. However, Hydra receivers can resume delivering messages from the other sequencers once they run the above protocol, which requires coordination only between the receivers, not the network layer. Thus, its unavailability period depends only on failure detection time and the agreement protocol latency, which can be orders-of-magnitude shorter than the duration of network rerouting. It

can also support more aggressive sequencer removal with shorter timeouts. Even though deploying more sequencers increases sequencer failure probability, by avoiding network rerouting on the critical path, Hydra still achieves overall improvement in system availability.

Adding new sequencers. To add a sequencer k to the system, the configuration service similarly creates a new configuration $n+1$ with sequencer k added, and sends the configuration to all receivers. Once a receiver receives the new configuration, it stops delivering groupcast messages to the application, and waits until it receives a flush message from the new sequencer k that has a higher timestamp than its latest delivered message. It then sends that flush message to the configuration service. When the configuration service receives a quorum of flush messages from each receiver group, it picks the flush message with the highest timestamp, denoted as t_k, and broadcasts that flush to all receivers; t_k effectively serves as the starting time of the new configuration. A receiver then resumes delivering messages for the previous configuration, until the next-to-be-delivered message has timestamp bigger than t_k. At that point, the receiver transitions to the next configuration, sets $s[k]$ to the sequence number in the flush message it receives from the configuration service, and starts delivering messages from the new sequencer. Note that if an old sequencer (removed previously) rejoins in a new configuration, the sequencer's ID is reassigned to a value unique from all other IDs, and its sequence numbers are all reset.

4.5 Correctness

We provide a detailed discussion of the safety of the Hydra protocol in Appendix B. In addition, a TLA$^+$ specification [40] of the Hydra groupcast and sequencer addition/removal protocols (Appendix C) has been model checked against Hydra's safety guarantees.

The liveness of the Hydra protocol is straightforward: as long as (1) receivers continue to receive groupcasts or flush messages from non-failed sequencers, and (2) the configuration service remains available and can communicate with a quorum of each receiver group to remove failed sequencers and complete the addition of new sequencers, Hydra groupcasts will be delivered.

4.6 Optimizations

Flush messages facilitate progress of Hydra receivers. However, generating flush messages at an overly aggressive rate will adversely affect a receiver's performance, as these flush messages consume network, CPU, and I/O resources. To strike a balance between message delivery latency and throughput, we propose two optimizations: receiver-side flush message solicitation and in-network flush message aggregation.

4.6.1 Receiver-Side Flush Message Solicitation

In our basic protocol described in §4.3.3, sequencers periodically send flush messages to all receivers. We can manually tune the flush message generating interval `T` on sequencers to adjust the latency/throughput trade-off: a smaller `T` improves message delivery latency but increases the load on the receivers, while a larger `T` has the opposite effect. However, since flush messages are broadcast to all receivers, this one-value-for-all policy cannot account for the different processing capacities and load levels on different receivers. Moreover, blindly sending flush messages every `T` time unit, particularly when `T` is small, can result in significant amount of unnecessary traffic. To see why this is the case, consider a receiver that currently has no message to deliver. Any flush message sent to this receiver, before the next Hydra message (with a higher clock value) arrives, will have no effect on the receiver's delivery progress and thus are strictly unnecessary.

Our key observation is that *receivers*, not sequencers, have perfect knowledge of when flush messages are required: receivers only need flush messages to make progress *when they possess undelivered messages*. We therefore propose a receiver-centric optimization, in which sequencers do not actively generate flush messages; instead, receivers explicitly request flush messages when needed. This optimization also enables various solicitation policies on the receivers. To optimize for latency, a receiver can immediately request flush messages when it receives groupcast messages that cannot be delivered. To optimize for throughput, it can delay requesting flush messages, equivalent to a batching approach. It can also apply a more sophisticated approach where it determines the solicitation delay based on the current load of the receiver, adaptively optimizing for both latency and throughput.

4.6.2 In-Network Flush Message Aggregation

Our message delivery rules (§4.3.2) require that a receiver delivers a groupcast message if and only if it has received messages with higher clock values from **all** other sequencers. The implication of this rule is that, the number of flush messages required to deliver a groupcast message increases *linearly* with the number of sequencers. To further reduce the processing overhead caused by excessive flush messages, we propose an advanced optimization technique inspired by recent in-network aggregation work. Concretely, we leverage ToR programmable switches connected to Hydra receivers to track each sequencer's clock value and sequence numbers. These numbers are updated when a ToR switch receives a flush message, but it does not immediately forward the flush message to the receiver. Only when the minimum stored clock value becomes large enough, the switch sends a single aggregated flush message containing all the clock values and sequence numbers to the receiver. To accurately determine this threshold, receivers attach the largest clock value among all undelivered messages in its flush message solicitation request. The ToR switch uses this value as the clock threshold, which guarantees that the aggregated flush message would allow the receiver to deliver all undelivered messages in the buffer (those when the solicitation request was made). By

applying our in-network aggregation optimization, the number of flush messages a receiver processes remains constant regardless of the number of sequencers.

5 Hydra Implementation

A Hydra deployment contains a dynamic set of groupcast senders, receivers, and sequencers, managed by a configuration service. We use a centrally-controlled SDN approach for managing groupcast routing: a POX [58]-based SDN controller installs rules that route groupcast messages to a randomly-selected reachable sequencer. When using end-host sequencers, we use a source routing approach: the configuration service tracks addresses of sequencers, which are cached on Hydra senders. When sending groupcast messages, senders randomly pick one of the sequencers and send to it via unicast. No special network routing is required.

Hydra sequencers each maintain minimal state: a unique sequencer ID, a sequence number for each receiver group, and a physical clock that is monotonically increasing. One of our key design principles is *simplicity*. It enables us to implement a Hydra sequencer *efficiently* on different hardware platforms.

In-network sequencing using programmable switches. Implementing Hydra sequencers in the data plane of network switches offer the highest sequencing performance, as current programmable switches can process billions of packets per second, with switching latency consistently under a few hundred nanoseconds. Hydra groupcast is implemented as an application-level protocol atop UDP. We reserve a special UDP port for Hydra groupcast, and append a customized Hydra header after the UDP header. The Hydra header includes a bitmap to specify the destination groups, a vector of sequence numbers (one for each destination group), and a single clock value. The switch implementation uses one switch register array element for each receiver group to store its current sequence number. The switch checks each bit of the bitmap, and for each enabled bit, increments the corresponding sequence number register and writes the sequence number into the Hydra header. Since there is no dependency across groups when processing a groupcast message, bit checking and sequence number updating for all destination groups can be done in parallel. This enables us to significantly reduce the required pipeline stages, allowing us to scale to higher number of groups. Subsequently, the switch stamps the hardware clock time into the header, and uses the replication engine to multicast the packet to receivers.

End-host Sequencers. Implementing Hydra sequencers on end-host servers offers better flexibility and portability, particularly attractive for deployments that cannot deploy specialized hardware. The downside is comparatively lower packet processing performance. Our Hydra protocol, however, enables scaling sequencing performance by adding additional sequencers. As we will show in our evaluation (§7.1.3), throughput of Hydra scales linearly with the number of sequencers.

Sender and receiver libraries. Hydra provides user-space libraries for sending and receiving groupcast messages. In addition to coordinating with the configuration service to track active sequencers and groups, this library also implements receiver-side buffering to deliver messages in the right order, and the flush message solicitation policies of §4.6.1. We have implemented two I/O stacks for the libraries. First, a polling-based DPDK [23] stack for efficient, kernel-bypassed packet processing. Second, a Linux-based transport using sockets and libevent [45] for better compatibility. Our evaluation in §7 uses the DPDK stack.

6 Building Distributed Systems using Hydra

Our Hydra groupcast primitive has a unique set of trade-offs between its guarantees and efficiency of the implementation. Compared to best effort primitives such as unicast and IP multicast, Hydra offers strong message ordering guarantees; compared to atomic broadcast and atomic multicast primitives, Hydra does not guarantee reliable message delivery, but can be implemented efficiently using a single phase protocol. In order to show the benefits its design, we applied Hydra to two recent distributed systems – NOPaxos and Eris [42, 43] – and built a state machine replication called HydraPaxos and a distributed transaction processing system called HydraTxn.

Hydra's groupcast provides the same guarantees as the network protocols used in NOPaxos and Eris (Ordered Unreliable Multicast and Multi-sequenced Groupcast). Therefore, Hydra is readily composed with these existing protocols. HydraPaxos and HydraTxn use the NOPaxos and Eris protocols to tolerate server faults and handle DROP-NOTIFICATION, while use Hydra to provides message ordering guarantees and allows the adding and removing of sequencers. The only necessary modifications to NOPaxos and Eris are the disabling of their sequencer failure handling protocols, as this is handled by Hydra itself. Both HydraPaxos and HydraTxn can commit operations in a *single round trip* in the normal case.

HydraPaxos. HydraPaxos is a state machine replication system based on NOPaxos that tolerates crash failures of less than half of the replicas (or equivalently, with $2f+1$ replicas, HydraPaxos tolerates f crash failures). It guarantees linearizability [30] as long as the application state machine is deterministic. Each HydraPaxos deployment registers a unique Hydra groupcast address. HydraPaxos clients send state machine operations as a groupcast message with a single destination group. Each replica in a HydraPaxos deployment acts as a single Hydra receiver of the group. HydraPaxos operations are handled in a single round trip in the normal case. Once Hydra delivers an operation to the replicas, the replicas use the NOPaxos protocol to ensure operations are committed durably. Briefly, each replica adds the operation to its log, and the *leader* replica executes the operation against the current state. Clients wait for consistent replies from a majority of replicas (including the leader) before considering a reply committed. When a DROP-NOTIFICATION is delivered

to a replica, replicas need to reach consensus on the fate of the message – either to process or to permanently ignore – to ensure linearizability. The replica first attempts to recover the missing message by contacting other replicas in the group. If replicas fail to recover the dropped message, they coordinate (driven by the leader) to commit the message as a NO-OP.

HydraTxn. HydraTxn, is a fault-tolerant, distributed transaction processing system. HydraTxn partitions the entire data store into multiple shards with each shard replicated on multiple servers. Clients wrap data reads and writes into transactions. HydraTxn guarantees atomic, strict serializable execution of the transactions, and tolerates failures of less than half of the replicas in each shard. Similar to HydraPaxos, each HydraTxn deployment uses a unique Hydra groupcast address. Each shard of the deployment is assigned a unique group, and replicas in the shard each register a Hydra group receiver, delivering Hydra messages and DROP-NOTIFICATIONs. For transactions that qualify as independent transactions – stored procedures that has no dependency across shards and requires no client interactions – clients send the transaction in a single Hydra groupcast message destined to all the involved shards. HydraTxn also supports more general transactions by dividing them into multiple independent transactions and using two-phase locking on the servers to ensure isolation. As in HydraPaxos, independent transactions are handled in a single round trip in the normal case. Replicas in each shard involved in the transaction log the transaction and reply to the client, with the leader of each shard additionally executing the transaction. Clients wait for majority quorums from each shard to reply before considering a transaction committed. Since a transaction groupcast may involve multiple shards, DROP-NOTIFICATION requires all involved shards, not just the local group, to reach consensus on the reception/dropping decision. Similar to Eris [42], we use a logically separate, fault-tolerant failure coordinator service to manage this agreement protocol.

7 Evaluation

Our Hydra implementation includes Hydra host libraries, switch data and control planes, end-host sequencers, and Hydra co-designed replication (HydraPaxos) and transactional (HydraTxn) protocol implementations. The switch data plane is implemented in 1040 lines of P4 [11] code, and the switch control plane is written in 493 lines of Python code. We implemented the end-host sequencer, Hydra host libraries, the HydraPaxos protocol, and the HydraTxn protocol in approximately 8000 lines of C++ code.

Our evaluation testbed consists of 10 nodes connected to an APS BF6064X-T (Barefoot Tofino-based) programmable switch. We ran servers/replicas on nodes with dual 2.90GHz Intel Xeon Gold 6226R processors (32 total cores), 256 GB RAM. We used the remaining nodes to run clients and end-host sequencers. Clients use 2.10GHz Intel Xeon Gold 6230 processors (20 total cores) and 96 GB RAM. All nodes ran Ubuntu Linux 20.04 and use Mellanox ConnectX-5 100 GbE NICs. We statically partitioned resources on the programmable switch to implement multiple switch sequencers.

7.1 Hydra Groupcast Microbenchmarks

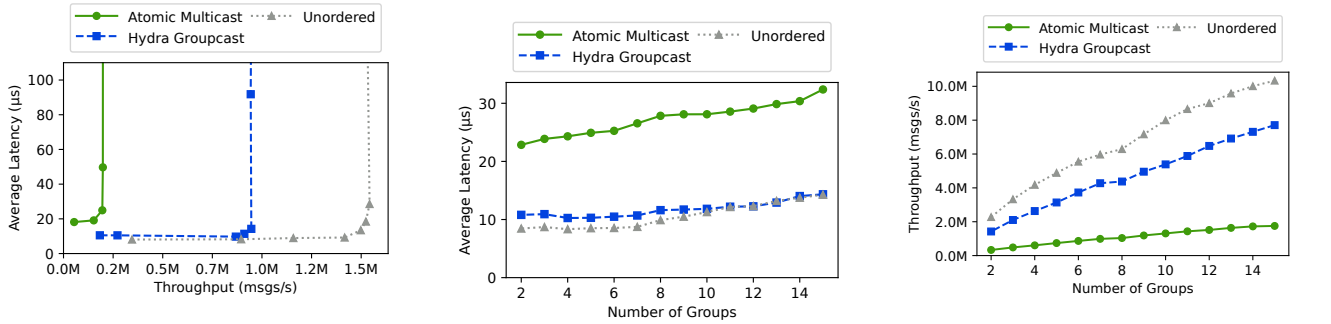
We first used microbenchmarks to evaluate the performance of our Hydra groupcast primitive. We ran closed-loop clients, each sending groupcast messages to a set of receiver groups. When a Hydra receiver *delivers* a groupcast message, it immediately replies to the client. Clients send the next groupcast message when they receives replies from each receiver in all destination groups (we assume no receivers fail).

We compared Hydra to two other groupcast implementations. First, we implemented a version of genuine atomic multicast [28]. To atomic multicast a message, a client first sends the message to all the receivers in each destination group. Receivers in each group run a consensus round to agree on a message timestamp and send the group timestamp to the client. The client picks the highest timestamp as the final message number, and forwards the message number to all involved group receivers. Receivers deliver messages in message number order. Second, we implement an unordered multicast – receivers immediately deliver client messages without any ordering guarantee – as a baseline.

7.1.1 Latency and Throughput

In the first experiment, we used a single group with three receivers to evaluate the base case performance. Two switch sequencers were deployed when evaluating Hydra, and we applied the receiver-side solicitation optimization (§4.6.1). We gradually increased the offered client load, and measured both the latency and the throughput of each system. As shown in Figure 2a, Hydra achieves a 378% increase in throughput and 42% reduction in latency compared to atomic multicast. Running consensus among group receivers for each message adds substantial throughput and latency overheads to atomic multicast. On the contrary, Hydra receivers require no coordination among each other to deliver messages in consistent order. In the worst case, they wait for a half RTT (receiver $\rightarrow$ switch $\rightarrow$ receiver) to receive flush messages in order to deliver a groupcast. This overhead is reflected in Hydra's small latency penalty (3 μs) compared to the baseline. As Hydra receivers can deliver messages without explicit flush messages when sequencers receive enough traffic, Hydra is able to attain throughput within 39% of the baseline.

Increasing group size. Next, we added more receivers to the group. When we increased the group size threefold (from three to nine), throughput of Hydra dropped only by 8%, and its latency remained the same. Hydra scales well to larger group sizes because receivers can independently determine the correct order of messages with no coordination. Performance of atomic multicast, however, degrades proportionally to the number of receivers as the cost of consensus increases. At group size of nine, Hydra outperforms atomic multicast by 567% in throughput and 56% in latency.



(a) Latency and throughput for a single group of three receivers

(b) Latency with increasing number of groups

(c) Maximum throughput with increasing number of groups

Figure 2: Latency and throughput of running a micro multicast benchmark. We use two switch sequencers for Hydra, and compare its performance to an atomic multicast and an unordered multicast protocol. For (b) and (c), we use a group size of three.

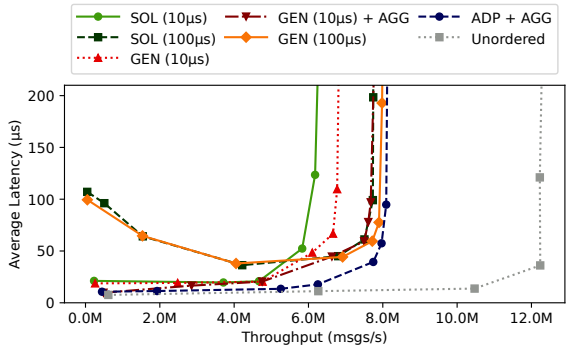


Figure 3: Impact of different flush message policies and parameters on performance of Hydra. The policies are: the receiver solicitation strategy (SOL), sequencer generation strategy (GEN), adaptive solicitation strategy (ADP), ToR switch aggregation (AGG).

Scaling to more groups. To test how well Hydra scales to larger number of groups, we fixed the group size to three receivers, and increased the total number of groups. We used a workload where 80% of the groupcast messages were destined to a single group, and the remaining 20% had two destination groups. Clients chose destination groups following a uniform distribution. As shown in Figure 2b and Figure 2c, Hydra's throughput and latency continue to closely match the baseline. At 15 groups, throughput of Hydra is within 25% of the baseline, and 340% higher than atomic multicast.

7.1.2 Impact of Flush Messages

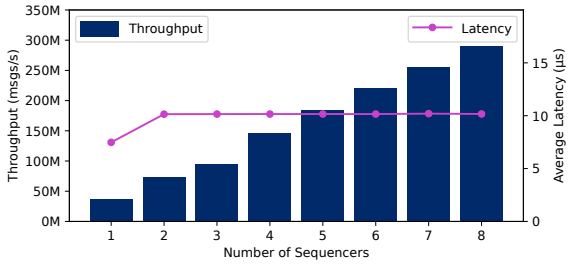
As we discussed in §4.6, policies for generating and handling flush messages can affect Hydra performance. To evaluate their effectiveness, we ran 15 groups each with three receivers, deployed four switch sequencers, and measured the latency and throughput of Hydra with increasing client load. We apply three flush message policies and show their impact in Figure 3: (1) sequencers periodically generate flush messages (GEN), (2) receivers solicit flush messages from sequencers after a delay (SOL), (3) receivers adaptively solicit flush messages based on current load (ADP). We also examine the impact of having ToR switches aggregate flush messages from sequencers (AGG).

When we use a higher delay for generating or soliciting flush messages, receivers on average need to wait longer to deliver messages. This effect is validated by the higher average latency experienced by GEN and SOL when their delay is at 100 μs. By decreasing the delay, both policies enjoy better message delivery latency. Unfortunately, it also degrades maximum throughput, as receivers use more CPU cycles to process flush messages – up to 14% lower throughput for GEN. ToR switch aggregation reduces the impact of frequent flush messages: AGG improves the throughput of GEN by 14%. Finally, by using an adaptive solicitation strategy, ADP achieves both low latency – it immediately requests flush messages when it has spare CPU cycles – and high throughput – it does not receive excessive flush messages at high utilization. As shown in Figure 3, it attains latency within 3μs and throughput within 33% that of the baseline result.

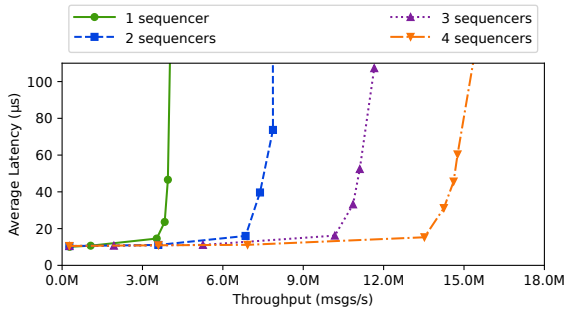
7.1.3 Sequencer Scalability

To evaluate the sequencer scalability of Hydra, we emulated an increasing number of switch sequencers (up to eight) on the same physical switch. For each emulated sequencer, we allocated a dedicated queue in the switch traffic manager that rate limits to 10 Gbps. Due to the limited number of physical servers, we only deployed 15 real Hydra groups, each with three receivers. To saturate the sequencers' capacity, we deployed additional virtual groups, whose request traffic were simply dropped at the switch egress ports. Figure 4a shows that throughput of the system increases linearly with more deployed switch sequencers. With eight sequencers, Hydra can process more than 250 million groupcast per second. The additional switch sequencers also have minimum impact on groupcast latency.

For clusters without programmable switches, sequencers can be deployed on end-host servers, offering an immediately deployable solution. End-host sequencers, however, have limited processing capacity compared to an in-switch implementation. Figure 4b shows that Hydra can avoid this dilemma by adding more end-host sequencers, with near-linear scaling. With enough Hydra traffic, request load were evenly



(a) Latency and sustainable throughput of Hydra with increasing number of switch sequencers



(b) Latency and throughput of Hydra with increasing number of end-host Hydra sequencers

Figure 4: Scalability of Hydra with increasing number of sequencers. We use 15 groups, three receivers per group, and deploy sequencers on switches and end-host servers. We also generate additional group-cast traffic to virtual Hydra groups to saturate the sequencers.

distributed among all sequencers. Since receivers need to wait for at least one message from each sequencer for message delivery, latency of Hydra increases slightly with more sequencers.

7.2 HydraPaxos Evaluations

Next, we evaluate the performance benefits of co-designing state machine replication (SMR) systems with Hydra. We compared our HydraPaxos to three other SMR protocols: Paxos (with the Multi-Paxos optimization), Fast Paxos, and NOPaxos. We also ran an unreplicated system with no fault tolerance as a baseline. All protocols were implemented in the same codebase for a fair comparison. We deployed each SMR system on three replica servers, ran an echo-RPC application, and measured the end-to-end latency and throughput of each system with increasing client load. We used two switch sequencers for HydraPaxos, and one switch sequencer for NOPaxos. Figure 5 shows HydraPaxos achieves significantly higher throughput than Paxos (204%) and Fast Paxos (180%), by avoiding replica coordination in the common case. Figure 5 also shows that our design can attain performance comparable to a network serialization approach: HydraPaxos achieves latency within 5 μs and throughput within 17% of NOPaxos. Like NOPaxos, HydraPaxos can sustain its throughput even with a moderate rate of packet drops ($\leq 0.1\%$), because drop recovery uses a lightweight protocol; a full evaluation appears in Appendix A.

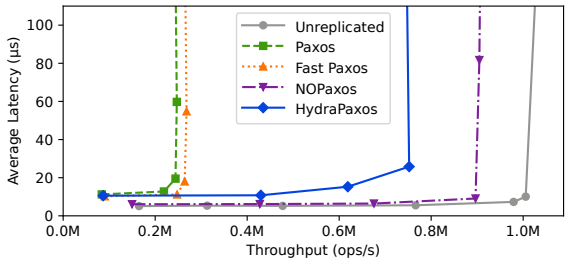


Figure 5: State machine replication system comparison. We measure the latency and throughput of HydraPaxos and other SMR protocols with three replicas. HydraPaxos uses two switch sequencers.

7.3 HydraTxn Evaluations

The second distributed application we evaluated was a fault-tolerant, distributed transactional system. We compared HydraTxn with three other systems: Granola [19], Eris [42], and a standard distributed transactional system [17] called Lock-Store that uses two-phase commit, two-phase locking, and Paxos. All systems are implemented in the same code base. We deployed each system on 15 database shards, each replicated on three servers. HydraTxn uses two switch sequencers, while Eris uses only one. Similar to our experiment in §7.1.3, we rate limit each sequencer's bandwidth in the switch traffic manager and generate traffic to virtual Hydra groups.

We use the YCSB+T [16] benchmark that wraps `read` and `write` operations into stored-procedure style transactions. The workload we used consists of single-shard transactions with a read/write ratio of 1:1. Keys are selected using a uniform distribution. As shown in Figure 6, HydraTxn avoids server coordination overhead when processing transactions, leading to a 3.1× and 1.9× throughput, and a 49% and 13% latency reduction compared to Lock-Store and Granola. Performance of Eris is bottlenecked by the single switch sequencer. Excessive client load can even cause sequenced packets to be dropped in the network, leading to throughput collapse due to more frequent drop agreement protocol [42]. Hydra enables HydraTxn to scale beyond the central sequencer bottleneck, achieving a 47% throughput improvement over Eris.

We also tested HydraTxn's resilience to network anomalies by injecting simulated packet drops. As in the SMR experiment, small to moderate levels of packet drops have minimal impact on HydraTxn's performance (Appendix A).

7.4 Network-Level Load Balancing

To evaluate the impact of our approach on network properties, we simulated a data center network with a three-layer FatTree topology in NS3. The network consisted of 2560 servers and 112 switches. All servers generate background traffic following a Poisson distribution. We set up 16 multicast receiver groups in the network, each with three receivers. We selected a few servers across the data center to generate periodic multicast messages, each message destined to a randomly selected group. We compared two approaches: a network serialization approach where all multicast messages are routed through

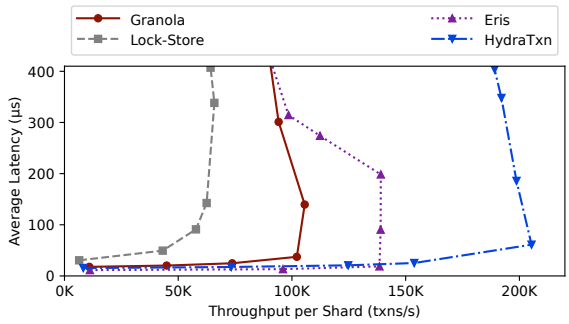
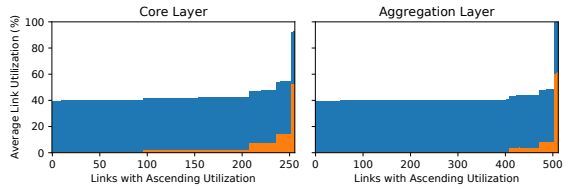
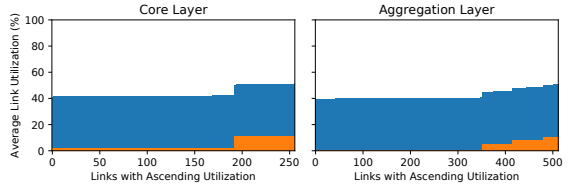


Figure 6: Distributed transactional system comparison. We measure the latency and per-shard throughput of HydraTxn and other transactional systems when running on 15 shards each replicated on three servers. HydraTxn uses two sequencers, while Eris uses one sequencer.



(a) Network link utilization when multicast messages traverse a single ToR switch sequencer



(b) Network link utilization when multicast messages are randomly routed to one of the eight ToR switch sequencers

Figure 7: Average link utilization of a simulated data center network. We simulate a three-layer FatTree topology with 2560 servers and 112 switches. Links between servers and ToR switches are 1 Gbps, and all other links are 10 Gbps.

a single ToR switch, and the Hydra approach where eight ToR switches are deployed as sequencers. Figure 7 shows the link utilization of each aggregation and core layer link for each approach. In the network serialization deployment, several aggregation layer links were fully saturated due to concentrated multicast traffic. By distributing multicast traffic across multiple sequencer switches, utilization of all core and aggregation links stayed below 50% in the Hydra deployment, demonstrating the load balancing benefit of our approach.

We then studied the impact of network congestion by generating bursty background traffic to one of the sequencer switches. Figure 8 shows that in a network serialization deployment, congestion at the sequencer switch caused median multicast latency to degrade by more than 13×. By distributing multicast traffic to multiple sequencer switches, Hydra reduces the impact of local congestion and improves the median latency by 11×. We also simulated *congestion-aware routing* [3] for Hydra. By preferentially routing to non-congested sequencers, Hydra further improves multicast tail latency.

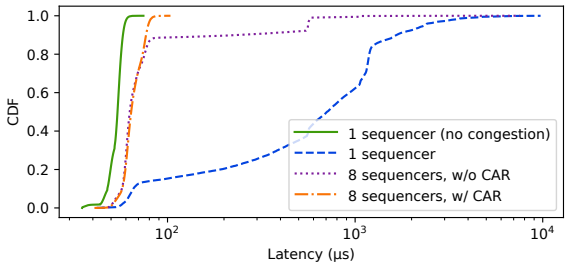
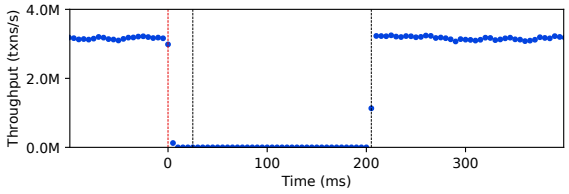
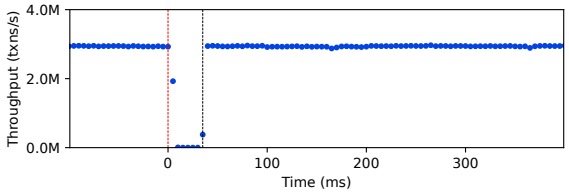


Figure 8: Latency distribution of multicast in the same simulated data center network as Figure 7. We generate bursty traffic to a single sequencer switch that causes congestion.



(a) Throughput of Eris during a sequencer failover



(b) Throughput of HydraTxn during a sequencer failover

Figure 9: Throughput of Eris and HydraTxn during a sequencer failover. For both, we injected a sequencer failure at time 0.

7.5 Sequencer Failover

Lastly, we evaluated the effectiveness of Hydra in handling sequencer failures and compared it to the network serialization approach. To do so, we used the same transactional system setup in §7.3, triggered a sequencer failure during the run, and measured the sustained throughput over time. As shown in Figure 9b, after the Hydra receivers detected the failure of one of the sequencers (we used a 20 ms timeout value), they immediately ran a reconfiguration protocol to remove the failed sequencer. The protocol only took a few hundred microseconds. HydraTxn was able to resume normal operation and returned to its maximum throughput afterwards, using the remaining sequencers. By contrast, Eris (Figure 9a) relies on the network control plane needed to perform rerouting once a failure is detected, to forward client requests to a new sequencer switch. We simulated a 100 ms rerouting delay which matches results in the literature [38]. Unlike HydraTxn, Eris remained unavailable during network rerouting, demon-

strating the benefit of our redundant sequencer approach.

8 Related Work

Ordered group communication primitives such as atomic multicast [28] have a long history, dating back to virtual synchrony [8], and have been implemented and used widely [4, 9, 32, 34, 62]. The classic atomic broadcast model is equivalent to consensus [13]. Our work explicitly adopts the *ordered but unreliable* communication model introduced by NOPaxos [43] and Eris [42], which enables network-accelerated sequencing.

Other distributed systems also use sequencers. CORFU [63] combines an unreliable sequencer with replicated storage on flash to build a shared log that can be used to build distributed data structures [7]. vCorfu [63] extends it to a multi-log abstraction analogous to multi-sequencing. Scalog [22] addresses the blocking reconfiguration and sequencer scalability issues of previous shared log designs by distributing log data to replicated data shards and periodically order log entries using an Paxos-based ordering layer. Hydra does not guarantee message persistence, so it avoids the overhead of intra-shard replication. Hydra also eliminates coordination among the sequencers on the critical path, which reduces message delivery latency and avoids potential bottlenecks of a centralized ordering service. Percolator [54] uses a sequencer for transaction processing, and deterministic databases like Calvin [60], SLOG [60], and Aria [47] combine sequencers with transaction schedulers for concurrency control.

Hydra builds on work on improving the scalability of consensus protocols. Its use of multiple active sequencers and flush messages is analogous to Mencius's rotating leader [48]. Hydra uses loosely synchronized clocks [46] to establish a total order, an idea used in concurrency control protocols like CLOCC [1], Spanner [17], and TAPIR [66]. Protocols like PTP [59] make clock synchronization widely available in data centers, and recent work like Sundial [44] and DPTP [35] demonstrates the precision available. Hydra's approach of using timestamps to order operations is similar to that of TEMPO [24]. However, TEMPO requires at least one and a half RTTs to commit a timestamp. Hydra, using network sequencers, can commit timestamps in half of an RTT even in the presence of concurrent requests. Similar to TEMPO, Hydra also waits for higher timestamps from other sequencers to ensure a timestamp is stable.

Hydra is designed to support programmable devices as sequencers, including PISA/RMT switch ASICs [12]. Recent work shows that these switches can implement complex protocols including consensus [20, 21] and chain replication [33]. Like NOPaxos and Eris, Hydra intentionally implements a limited set of sequencing functionality on the switch, leaving most of the protocol complexity at the end hosts. RedPlane [37] and SwiSh [64, 65] provide abstractions for replicating switch data plane state for reliability and scalability, respectively; sequencing, which requires strong consistency and frequent updates, represents a worst-case performance scenario for both, necessitating a different approach.

A concurrent effort, 1Pipe [41] uses programmable switches in a data center to implement causally and totally ordered communication. Senders in 1Pipe attach local timestamps to messages, and receivers deliver messages strictly in timestamp order. To determine when a timestamp is safe to deliver, switches in 1Pipe track barrier information from all ingress links and write the aggregated barrier timestamp into each packet. Hosts and switches periodically send beacon messages on idle links to ensure progress.

Hydra similarly uses timestamps to order messages. A key difference is that 1Pipe uses sender-generated timestamps, while in Hydra timestamps are generated by the sequencers. Consequently, 1Pipe requires synchronized clocks on *all* nodes in the network and in-network computation at *each* switch, a deployment challenge in heterogeneous networks where not all switches are programmable [57]; Hydra accommodates more practical deployments by only running logic on the sequencers and replicas, and only synchronizing clocks across sequencers. Moreover, in a 1Pipe deployment, any node, link, or switch failure in the network would stall the progress of all receivers; in Hydra, only failures local to the sequencers may impact progress.

9 Conclusion

The deployment of network sequencing approaches has been hindered because they require serializing messages through a single sequencer. Hydra addresses this with a new protocol that allows the concurrent use of multiple sequencers. A Hydra deployment serves as a drop-in replacement for sequencers in systems like NOPaxos and Eris, making their benefits more widely accessible. In particular, it scales beyond the performance of a single sequencer, which allows commodity servers rather than programmable switches; reduces system downtime during sequencer failures; and improves network load balancing by avoiding serialization.

Acknowledgments

We thank our shepherd Shuai Mu and the anonymous reviewers for their valuable feedback. We also thank Xin Zhe Khooi and Raj Joshi for their helpful comments on the P4 implementaion. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship. Jialin Li was supported by an MOE AcRF Tier 1 grant, an ODPRT SUG grant, and a Huawei research grant.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, June 1995. ACM.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of ACM SIGCOMM 2014*, 2014.

[4] Y. Amir and J. Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, The Johns Hopkins University, Baltimore, MD, USA, 1998.

[5] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research*, CIDR '11, Asilomar, California, 2011.

[6] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, San Jose, CA, USA, 2012. USENIX Association.

[7] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, Farminton, Pennsylvania, 2013. Association for Computing Machinery.

[8] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 123–138, New York, NY, USA, 1987. Association for Computing Machinery.

[9] K. P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, page 79–86, New York, NY, USA, 1985. Association for Computing Machinery.

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

[11] 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.

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of ACM SIGCOMM 2013*, Hong Kong, China, Aug. 2013. ACM.

[13] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '92, page 147–158, New York, NY, USA, 1992. Association for Computing Machinery.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, Seattle, Washington, 2006. USENIX Association.

[15] Cisco data center infrastructure design guide 2.5. https://www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. 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 Conference on Operating Systems Design and Implementation*, OSDI '12, Hollywood, CA, USA, 2012. USENIX Association.

[18] I. Corporation. Intel Tofino 3 Intelligent Fabric Processor Brief. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html.

[19] J. Cowling and B. Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the*

2012 USENIX Conference on Annual Technical Conference, USENIX ATC '12, Boston, MA, 2012. USENIX Association.

[20] 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.

[21] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, N. Zilberman, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. Technical Report USI-INF-TR-2018-01, Università della Svizzera italiana, May 2018.

[22] C. Ding, D. Chu, E. Zhao, X. Li, L. Alvisi, and R. Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, USA, Feb. 2020. USENIX.

[23] Data Plane Development Kit. https://www.dpdk.org/.

[24] V. Enes, C. Baquero, A. Gotsman, and P. Sutra. Efficient replication via timestamp stability. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 178–193, New York, NY, USA, 2021. Association for Computing Machinery.

[25] 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.

[26] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable Consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, Cascais, Portugal, 2011. Association for Computing Machinery.

[27] 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 the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery.

[28] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1–2):297–316, mar 2001.

[29] C. Guo, H. Wu, Z. Deng, J. Y. Gaurav Soni, J. Padhye, and M. Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of ACM SIGCOMM 2016*, Florianopolis, Brazil, Aug. 2016. ACM.

[30] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[31] Hesiod. Theogony. c. 730 BCE.

[32] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems*, 36(2):1–49, Apr. 2019.

[33] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT coordination. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, USA, Apr. 2018. USENIX.

[34] 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 and Networks*, DSN '11, page 245–256, USA, 2011. IEEE Computer Society.

[35] P. G. Kannan, R. Joshi, and M. C. Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 Symposium on SDN Research (SOSR '19)*, Santa Jose, CA, USA, Mar. 2019. ACM.

[36] X. Z. Khooi, L. Csikor, J. Li, and D. M. Divakaran. In-network applications: Beyond single switch pipelines. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 1–8, 2021.

[37] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan. RedPlane: Enabling fault tolerant stateful in-switch applications. In *Proceedings of ACM SIGCOMM 2021*, Virtual Conference, Aug. 2021. ACM.

[38] 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 Conference on Operating Systems Design and Implementation*, OSDI'10, page 351–364, USA, 2010. USENIX Association.

[39] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, Prague, Czech Republic, 2013. Association for Computing Machinery.

[40] L. Lamport. The TLA$^+$ home page. https://lamport.azurewebsites.net/tla/tla.html.

[41] B. Li, G. Zuo, W. Bai, and L. Zhang. 1pipe: Scalable total order communication in data center networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 78–92. Association for Computing Machinery, 2021.

[42] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, Shanghai, China, 2017. Association for Computing Machinery.

[43] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, Savannah, GA, USA, 2016. USENIX Association.

[44] Y. Li, G. Kumar, H. Hariharan, H. Wassel, P. Hochschild, D. Platt, S. Sabato, M. Yu, N. Dukkipati, P. Chandra, and A. Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, Banff, AL, Canada, Nov. 2020. USENIX.

[45] libevent – an event notification library. https://libevent.org/.

[46] B. Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC '91)*, Montreal, QC, Canada, Aug. 1991. ACM.

[47] Y. Lu, X. Yu, L. Cao, and S. Madden. Aria: A fast and practical deterministic oltp database. *Proceedings of the VLDB Endowment*, 13(12):2047–2060, July 2020.

[48] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, San Diego, California, 2008. USENIX Association.

[49] R. Niranjan 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 the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 39–50, New York, NY, USA, 2009. Association for Computing Machinery.

[50] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, Lombard, IL, 2013. USENIX Association.

[51] NTP clock discipline algorithm. https://www.eecis.udel.edu/~mills/ntp/html/discipline.html.

[52] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, Toronto, Ontario, Canada, 1988. Association for Computing Machinery.

[53] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '14, Philadelphia, PA, 2014. USENIX Association.

[54] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, Oct. 2010. USENIX.

[55] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, Vancouver, BC, Canada, 2010. USENIX Association.

[56] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI '15, Oakland, CA, 2015. USENIX Association.

[57] D. R. K. Ports and J. Nelson. When should the network be the computer? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019. ACM.

[58] POX SDN controller. https://github.com/noxrepo/pox.

[59] IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. https://www.nist.gov/el/intelligent-systems-division-73500/ieee-1588.

[60] K. Ren, D. Li, and D. J. Abadi. SLOG: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11):1747–1761, July 2019.

[61] R. Van Renesse and D. Altinbuken. Paxos Made Moderately Complex. *ACM Computing Survey*, 47(3), Feb. 2015.

[62] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr. 1996.

[63] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, Boston, MA, USA, 2017. USENIX Association.

[64] L. Zeno, D. R. K. Ports, J. Nelson, D. Kim, S. L. Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*, Renton, WA, USA, Apr. 2022. USENIX.

[65] L. Zeno, D. R. K. Ports, J. Nelson, and M. Silberstein. SwiShmem: Distributed shared state abstractions for programmable switches. In *Proceedings of the 16th Workshop on Hot Topics in Networks (HotNets '20)*, Chicago, IL, USA, Nov. 2020. ACM.

[66] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems*, 35(4):12, Dec. 2018.

A Additional Evaluation

A.1 Clock Skew

Clock skew among sequencers does not affect Hydra correctness, but can delay message delivery progress. To evaluate the impact of clock skews, we deployed eight switch sequencers, 15 groups (no virtual groups), and three receivers in each group. We injected artificial clock skews to different sequencers, and measured both the latency and throughput of Hydra. As shown in Figure 10, clock skew does not impact Hydra throughput. Messages stamped by sequencers with faster clocks are buffered temporarily on the receivers, but the rate of delivering messages remains the same. At small to medium clock skews (1 to 10 μs), Hydra experiences marginal latency penalties (0 to 5 μs). Such clock skews are realistic: modern clock synchronization protocols [59] can maintain clock skews in the sub-microsecond range, and recent work has demonstrated synchronization error under 50 ns between programmable switches [35]. Even a 200 μs clock skew only resulted in less than 100 μs of added latency.

A.2 Message Loss for HydraPaxos and HydraTxn

Handling message drops. When Hydra messages are dropped in the network, HydraPaxos replicas need to coordinate to handle DROP-NOTIFICATIONs. To evaluate HydraPaxos's resilience to network anomalies, we measured its maximum throughput when an increasing percentage of packets were artificially dropped in the network. Figure 11 shows that HydraPaxos is able to sustain its high throughput even with a moderate rate of packet drops ($\leq 0.1\%$). HydraPaxos uses a lightweight protocol to recover from DROP-NOTIFICATIONs, as long as the message is not dropped on *all* replicas. At higher drop rates, throughput of HydraPaxos starts to decline due to more frequent coordination. We observe a similar level of throughput reduction for NOPaxos at these high drop rates.

We conduct the same experiment for HydraTxn. As in the SMR experiment, small to moderate levels of packet drops have minimal impact on HydraTxn's performance (Figure 12): its peak throughput decreased only by 11% even when the network dropped 1% of packets, and remained higher than that of Eris.

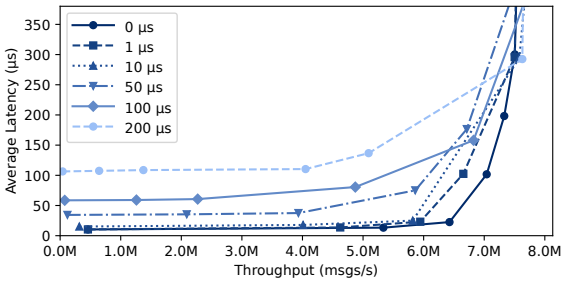


Figure 10: Latency and throughput of Hydra with increasing clock skew among sequencers. We use 15 groups, three receivers per group, and eight switch sequencers. Clock skew shows the maximum skew between any two sequencers.

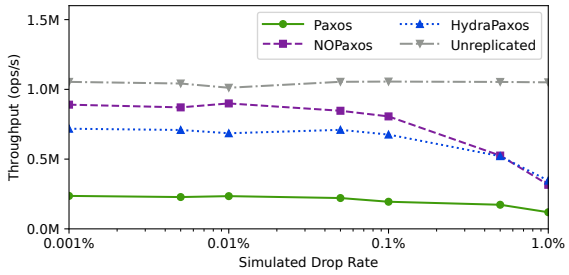


Figure 11: Maximum throughput of SMR systems with increasing packet drop rate. All systems run on three replicas. HydraPaxos uses two switch sequencers.

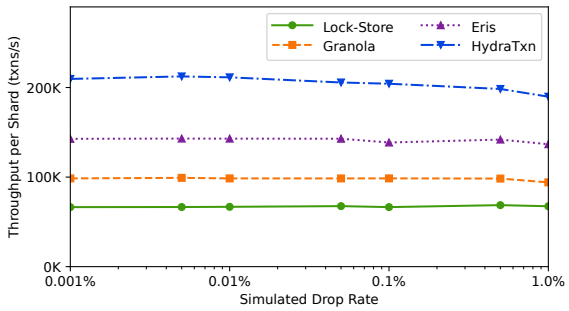


Figure 12: Maximum throughput of transactional systems with increasing packet drop rate. All systems run on 15 shards each replicated three-way. HydraTxn uses two switch sequencers.

B Proof of Safety

As specified in §3.1 Hydra provides the following guarantees to receivers of groupcast messages:

- **Partial Ordering.** All groupcast messages are partially ordered (the partial order relation is denoted as $\prec$), with all groupcast messages with overlapping destination groups are comparable. If groupcast message m_1 is ordered before m_2 ($m_1 \prec m_2$) and a receiver receives both m_1 and m_2, then every receiver delivers m_1 before m_2.
- **Unreliable Delivery.** Hydra only offers best effort message delivery. A groupcast message is not guaranteed to be delivered to any of its recipients.
- **Drop Detection.** If a groupcast message is not delivered to all its recipients, the primitive will notify the remaining receivers by delivering a DROP-NOTIFICATION. More formally, let R be the set of receiver groups for message m, then either one of the following two conditions holds: **all** receiver groups in R deliver m or a DROP-NOTIFICATION for m, or **none** of the receiver groups in R delivers m or a DROP-NOTIFICATION for m.

It is important to note that Hydra receiver groups, like NOPaxos and Eris receiver groups, use quorum-based protocols to decide which messages are delivered to the the group and which are permanently dropped. In order to tolerate the failure of some receivers in a receiver group, the drop detection requirement only considers each receiver group as a whole. Here, a receiver group delivers a message m or DROP-NOTIFICATION for m if every receiver in some quorum delivers m or a DROP-NOTIFICATION for m. Individual receivers in a group can diverge from the quorum when a sequencer is added or removed, and the receiver groups themselves must be able to handle this divergence. (NOPaxos and Eris do handle this case.) If an application using Hydra requires that the drop detection property apply uniformly to all receivers, then the quorum size for each receiver group is the size of the entire group.

Also important to note is that in the drop detection requirement, when we say a receiver delivers a DROP-NOTIFICATION for message m, what we mean is that the receiver delivers a DROP-NOTIFICATION for m *before delivering any message ordered after m in the partial order*. In this way, the drop detection requirement is indeed a safety requirement and not a liveness guarantee.

In the absence of sequencer failures, the correctness of Hydra's groupcast delivery is straightforward. Receivers deliver groupcast messages only when the message's clock value is less than or equal to $c_{\min}$, the minimum among the latest timestamps received from each of the sequencers. In fact, they only deliver messages whose timestamp is exactly $c_{\min}$, as ties in clock value are broken by sequencer ID. Because receivers only deliver messages in sequence number order, once message m is delivered by a receiver, no message with smaller sequence number or clock value from m's sequencer will be delivered by the receiver. Therefore, messages are always delivered in (timestamp, sequencer ID) order, which is a total order; the partial ordering guarantee is satisfied *a fortiori*. Furthermore, because receivers always deliver DROP-NOTIFICATION for smaller undelivered sequence numbers before delivering a message when there would be gaps in the sequence numbers delivered for that sequencer, the drop detection guarantee is satisfied.

In order to show that the sequencer removal process is correct, we first note that it is consistent with the Hydra safety guarantees for a receiver to at any time deliver a DROP-NOTIFICATION for the next sequence number yet to be delivered for some sequencer. The sequencer removal process is functionally equivalent to each receiver delivering infinitely many DROP-NOTIFICATIONs for all non-delivered sequence numbers for that sequencer. The agreement round is only necessary to determine exactly how many DROP-NOTIFICATIONs each receiver must explicitly deliver based on the results from each quorum. If a message m or a DROP-NOTIFICATION for m is delivered by a quorum from group g, and the sequencer that sequenced m is removed, then the configuration service is guaranteed to receive a multi-stamp with a sequence number for g at least as high as m's. Before transitioning to the new configuration (or delivering any message with a timestamp larger than m's), all other receiver groups that m was sent to must deliver a DROP-NOTIFICATION for m. Similarly, if no quorum from any receiver group received m or a message with sequence number larger than m's' before agreeing to

stop processing messages from the removed sequencer, then m will never be delivered by any receiver group (nor will an explicit DROP-NOTIFICATION for m be delivered by any receiver group). Therefore, for every groupcast sequenced by the removed sequencer, either all groups deliver the message or a DROP-NOTIFICATION for it or none do, satisfying the drop detection requirement.

When a sequencer is added, the flush message with timestamp t_k constructed by the configuration service when adding sequencer k is sent to all receiver groups. t_k is necessarily larger than the clock value of any message delivered by a quorum of receivers by construction. No message from sequencer k with clock value less than or equal to t_k will be delivered, nor will any DROP-NOTIFICATION for a message from sequencer k with clock value less than or equal to t_k. t_k was derived from a flush message that included sequence numbers for all groups, and upon entering the new configuration, a receiver immediately sets its sequence number for the added sequencer to the one included in this flush message. Conversely, once the new configuration starts, receivers in the new configuration will deliver messages or DROP-NOTIFICATIONs from the new sequencer with timestamp greater than t_k following the normal protocol for message delivery. Therefore, for any groupcast sequenced by the added sequencer, either all groups deliver the message or a DROP-NOTIFICATION for it or none do, satisfying the drop detection requirement.

C Hydra TLA+ Specification

MODULE *Hydra*

Specifies the *Hydra* protocol.

Receiver groups in this model are treated as single entities. This is done to increase model checking performance and avoid making assumptions about the protocol being run by the receiver groups. This specification focuses on the Hydra protocol and avoids the details of the quorum-based protocol being run by the receivers.

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

Constants and Variables

CONSTANTS *numSequencers*, *receivers*, *initialActiveSequencers*

ASSUME $numSequencers \in Nat$
ASSUME $numSequencers > 0$
ASSUME $IsFiniteSet(receivers)$
ASSUME $IsFiniteSet(initialActiveSequencers)$
ASSUME $initialActiveSequencers \in \text{SUBSET } Nat$

$sequencers \triangleq (1 \,..\, numSequencers)$

$mGroupcast \triangleq$ "mGroupcast"
$mFlush \triangleq$ "mFlush"
$mAddSequencer \triangleq$ "mAddSequencer"
$mFinishAdd \triangleq$ "mFinishAdd"
$mRemoveSequencer \triangleq$ "mRemoveSequencer"
$mFinishRemove \triangleq$ "mFinishRemove"

$vGroupcast \triangleq$ "vGroupcast"
$vDropNotification \triangleq$ "vDropNotification"

VARIABLES *messages*, *sequencerState*, *receiverState*, *configState*

$$
\begin{aligned}
Init \triangleq\ & \wedge messages = \{\} \\
& \wedge sequencerState = [s \in sequencers \mapsto \\
& \qquad [timestamp \mapsto 0, \\
& \qquad\ sequenceNums \mapsto [v \in receivers \mapsto 0] \\
& \qquad]] \\
& \wedge receiverState = [v \in receivers \mapsto [\\
& \qquad \text{Undelivered groupcasts} \\
& \qquad buffer \mapsto \{\}, \\
& \qquad \text{Delivered groupcasts and drop notifications} \\
& \qquad delivered \mapsto \langle\rangle, \\
& \qquad \text{Largest timestamps seen} \\
& \qquad timestamps \mapsto [s \in sequencers \mapsto 0], \\
& \qquad \text{Largest } sequenceNums \text{ seen} \\
& \qquad sequenceNums \mapsto [s \in sequencers \mapsto 0], \\
& \qquad \text{Currently active sequencers}
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad\quad activeSequencers \;\mapsto initialActiveSequencers, \\
\qquad\qquad\quad \text{Sequencers being added} \\
\qquad\qquad\quad addedSequencers \;\mapsto initialActiveSequencers, \\
\qquad\qquad\quad \text{Sequencers being removed} \\
\qquad\qquad\quad removedSequencers \mapsto \{\} \\
\qquad\qquad]] \\
\land\ configState = [addedSequencers \;\mapsto initialActiveSequencers, \\
\qquad\qquad\quad removedSequencers \mapsto \{\}]
\end{array}
$$

Helper and Utility Functions

$Min(S) \triangleq$ CHOOSE $s \in S : \forall\, sp \in S : sp \geq s$
$Max(S) \triangleq$ CHOOSE $s \in S : \forall\, sp \in S : sp \leq s$

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

RECURSIVE $SeqFromSet(\_)$
$SeqFromSet(S) \triangleq$
  IF $S = \{\}$ THEN $\langle\rangle$
  ELSE LET $x \triangleq$ CHOOSE $x \in S :$ TRUE
       IN $\langle x \rangle \circ SeqFromSet(S \setminus \{x\})$

Short-hand way of sending a message
$Send(m) \triangleq messages' = messages \cup \{m\}$

Main utility function for delivering groupcasts and drop notifications.

Receiver r adds G to its buffer, increments sequencer s's timestamp to t, increments s's *sequenceNum* to n, and delivers deliverable groupcasts.

It also permanently removes all sequencers in the removed set.

$DeliverAvailable(r,\ G,\ s,\ t,\ n,\ removed) \triangleq$
LET
  $rstate \triangleq receiverState[r]$

  $newRemovedSequencers \triangleq rstate.removedSequencers \cup removed$
  $newActiveSequencers \triangleq rstate.activeSequencers \setminus removed$

  $bg \triangleq rstate.buffer \cup G$
  $oldLog \triangleq rstate.delivered$

  $newTimestamps \triangleq [rstate.timestamps$ EXCEPT $![s] = Max(\{@,\ t\})]$
  $newSequenceNums \triangleq [rstate.sequenceNums$ EXCEPT $![s] = Max(\{@,\ n\})]$

  Groupcasts about to be delivered
  $deliverable \triangleq \{gp \in bg :$
      $Min(\{newTimestamps[sp] :$
        $sp \in newActiveSequencers\}) \geq gp.timestamp\}$

```
    Newly/previously delivered Groupcasts + previous drop notifications
    delivered  ≜  Range(oldLog) ∪ deliverable

    newBuffer  ≜  bg \ deliverable

    Necessary drop notifications
    dropNotifications(sequencer)  ≜  {
        [vtype          ↦ vDropNotification,
         sequencer      ↦ sequencer,
         sequenceNum ↦ k] : k ∈ {l ∈ (1 .. newSequenceNums[sequencer]) :
            ¬∃ gp ∈ Range(oldLog) ∪ bg :
                ∨ ∧ gp.vtype = vDropNotification
                  ∧ gp.sequencer = sequencer
                  ∧ gp.sequenceNum = l
                ∨ ∧ gp.vtype = vGroupcast
                  ∧ gp.sequencer = sequencer
                  ∧ gp.sequenceNums[r] = l}}
    allDropNotifications  ≜  UNION {dropNotifications(s) : sp ∈ sequencers}

    orderedDropNotifications  ≜  SortSeq(
        SeqFromSet(allDropNotifications),
        LAMBDA d1, d2 : d1.sequenceNum < d2.sequenceNum)

    orderedDeliverables  ≜  SortSeq(SeqFromSet(deliverable),
        LAMBDA g1, g2 : ∨ g1.timestamp < g2.timestamp
                        ∨ ∧ g1.timestamp = g2.timestamp
                          ∧ g1.sequencer  < g2.sequencer)

    newLog  ≜  oldLog ∘ orderedDropNotifications ∘ orderedDeliverables
IN
    ∧ receiverState' = [receiverState EXCEPT ![r] =
                        [@ EXCEPT !.buffer = newBuffer,
                                  !.timestamps = newTimestamps,
                                  !.delivered = newLog,
                                  !.sequenceNums = newSequenceNums,
                                  !.activeSequencers = newActiveSequencers,
                                  !.removedSequencers = newRemovedSequencers
                        ]]
    ∧ UNCHANGED ⟨messages, sequencerState, configState⟩
```

Main Spec

```
If two receivers deliver groupcasts, they deliver them in the same order
GlobalOrder  ≜  ∀ r1, r2 ∈ receivers : LET
    d1  ≜  receiverState[r1].delivered
    d2  ≜  receiverState[r2].delivered
```

```
IN
∀ n1_1 ∈ (1 .. Len(d1)), n2_1 ∈ (1 .. Len(d2)) :
   ( ∧ d1[n1_1] = d2[n2_1]
     ∧ d1[n1_1].vtype = vGroupcast
     ∧ d2[n2_1].vtype = vGroupcast) ⇒
∀ n1_2 ∈ (1 .. n1_1), n2_2 ∈ (n2_1 + 1 .. Len(d2)) :
   ( ∧ d1[n1_2].vtype = vGroupcast
     ∧ d2[n2_2].vtype = vGroupcast) ⇒
   d1[n1_2] ≠ d2[n2_2]
```

If any receiver delivers a *Groupcast*, then all receivers deliver that Groupcast or a *DropNotification* before that timestamp

```
Delivery ≜ ∀ r1 ∈ receivers : LET
    d1 ≜ receiverState[r1].delivered
IN
∀ n1 ∈ (1 .. Len(d1)) :
   d1[n1].vtype = vGroupcast ⇒
LET
    g1 ≜ d1[n1]
    t ≜ g1.timestamp
IN
∀ r2 ∈ DOMAIN g1.sequenceNums :
LET
    d2 ≜ receiverState[r2].delivered
IN
∨ ∃ g2 ∈ Range(d2) : g1 = g2
∨ ¬∃ g2 ∈ Range(d2) : ∧ g2.vtype = vGroupcast
                      ∧ g2.timestamp ≥ t
∨ ∃ n2 ∈ (1 .. Min({x ∈ DOMAIN d2 :
                    d2[x].vtype = vGroupcast ∧ d2[x].timestamp ≥ t})) :
   ∧ d2[n2].vtype = vDropNotification
   ∧ d2[n2].sequencer = g1.sequencer
   ∧ d2[n2].sequenceNum = g1.sequenceNums[r2]
```

Groupcasts are always delivered in timestamp and sequence number order

```
LocalOrder ≜ ∀ r ∈ receivers :
LET
    deliveredGroupcasts ≜ SelectSeq(receiverState[r].delivered,
                                    LAMBDA g : g.vtype = vGroupcast)
    deliveredFromSequencer(s) ≜ SelectSeq(deliveredGroupcasts,
                                          LAMBDA g : g.sequencer = s)
    SeqNum(g) ≜ IF g.vtype = vGroupcast
                THEN g.sequenceNums[r]
                ELSE g.sequenceNum
IN
```

$\land \forall\, n \in (1\,..\,Len(deliveredGroupcasts) - 1) :$
$\quad deliveredGroupcasts[n].timestamp \leq deliveredGroupcasts[n+1].timestamp$
$\land \forall\, s \in sequencers : \forall\, n \in (1\,..\,Len(deliveredFromSequencer(s)) - 1) :$
$\quad SeqNum(deliveredFromSequencer(s)[n]) \leq$
$\quad SeqNum(deliveredFromSequencer(s)[n+1])$

$Safety \triangleq GlobalOrder \land Delivery \land LocalOrder$

Actions and Message Handlers

Advance time at sequencer s

$AdvanceTime(s) \triangleq$
$\land\ sequencerState' = [sequencerState \text{ EXCEPT } ![s] =$
$\qquad\qquad [@ \text{ EXCEPT } !.timestamp = @ + 1]]$
$\land$ UNCHANGED $\langle messages,\ receiverState,\ configState\rangle$

Sequencer s sends a groupcast to set of receivers R

$SendGroupcast(s) \triangleq \exists\, R \in \text{SUBSET } receivers :$
$\land\ Cardinality(R) > 0$
$\land\ Send([mtype \mapsto mGroupcast,$
$\qquad sequencer \mapsto s,$
$\qquad timestamp \mapsto sequencerState[s].timestamp + 1,$
$\qquad sequenceNums \mapsto [r \in R \mapsto$
$\qquad\qquad sequencerState[s].sequenceNums[r] + 1]$
$\qquad])$
$\land\ sequencerState' = [sequencerState \text{ EXCEPT } ![s] =$
$\quad [@ \text{ EXCEPT } !.sequenceNums = [r \in receivers \mapsto$
$\qquad\qquad \text{IF } r \in R \text{ THEN } @[r] + 1 \text{ ELSE } @[r]],$
$\qquad !.timestamp = @ + 1]]$
$\land$ UNCHANGED $\langle receiverState,\ configState\rangle$

Sequencer s sends a flush

$SendFlush(s) \triangleq$
$\land\ Send([mtype \mapsto mFlush,$
$\qquad sequencer \mapsto s,$
$\qquad timestamp \mapsto sequencerState[s].timestamp,$
$\qquad sequenceNums \mapsto sequencerState[s].sequenceNums$
$\qquad])$
$\land$ UNCHANGED $\langle sequencerState,\ receiverState,\ configState\rangle$

Receiver r receives an $mGroupcast$ message m from receiver i

$HandleGroupcast(r,\ m) \triangleq$
LET
$\quad rstate \triangleq receiverState[r]$
$\quad s \triangleq m.sequencer$
$\quad n \triangleq m.sequenceNums[r]$

```
    g ≜ [vtype          ↦ vGroupcast,
         timestamp      ↦ m.timestamp,
         sequencer      ↦ s,
         sequenceNums ↦ m.sequenceNums]
IN
    Don't accept Groupcasts if we're adding a sequencer
    ∧ rstate.addedSequencers ⊆
      (rstate.activeSequencers \ rstate.removedSequencers)
    Sequencer must be active and not being removed
    ∧ s ∈ rstate.activeSequencers
    ∧ s ∉ rstate.removedSequencers
    Don't receive if already handled
    ∧ g.sequenceNums[r] > rstate.sequenceNums[s]
    ∧ DeliverAvailable(r, {g}, s, m.timestamp, n, {})

 Receiver r receives an mFlush message m
HandleFlush(r, m) ≜
LET
    rstate ≜ receiverState[r]
    s ≜ m.sequencer
    t ≜ m.timestamp
    largestDeliveredTimestamp ≜ Max({0} ∪ {
        g.timestamp : g ∈ {gp ∈ Range(rstate.delivered) :
                           gp.vtype = vGroupcast}})
IN
    Don't accept flushes while adding sequencers
    ∨ ∧ rstate.addedSequencers ⊆
        (rstate.activeSequencers \ rstate.removedSequencers)
      ∧ s ∈ rstate.activeSequencers
      ∧ s ∉ rstate.removedSequencers
      ∧ DeliverAvailable(r, {}, s, t, m.sequenceNums[r], {})
    ∨ ∧ s ∈ rstate.addedSequencers \ rstate.activeSequencers
      ∧ s ∉ rstate.removedSequencers
      ∧ t > largestDeliveredTimestamp
      ∧ Send([mtype          ↦ mAddSequencer,
              receiver       ↦ r,
              sequencer      ↦ s,
              timestamp      ↦ t,
              sequenceNums ↦ m.sequenceNums])
      ∧ UNCHANGED ⟨sequencerState, receiverState, configState⟩

 Receiver r begins adding sequencer s
BeginAddSequencer(r, s) ≜
LET
    rstate ≜ receiverState[r]
```

```
IN
    ∧ s ∉ rstate.activeSequencers
    ∧ s ∉ rstate.addedSequencers
    ∧ s ∉ rstate.removedSequencers
    ∧ receiverState' = [receiverState EXCEPT ![r] =
                          [@ EXCEPT !.addedSequencers = @ ∪ {s}]]
    ∧ UNCHANGED ⟨messages, sequencerState, configState⟩

 The config service finalizes the addition of sequencer s
AddSequencer(s) ≜
LET
    adds ≜ {m ∈ messages : m.mtype = mAddSequencer ∧ m.sequencer = s}
    tStart ≜ Max({m.timestamp : m ∈ adds})
    seqsStart ≜ (CHOOSE m ∈ adds : m.timestamp = tStart).sequenceNums
IN
    ∧ s ∉ configState.addedSequencers
    ∧ ∀ r ∈ receivers : ∃ m ∈ adds : m.receiver = r
    ∧ Send([mtype        ↦ mFinishAdd,
            sequencer    ↦ s,
            timestamp    ↦ tStart,
            sequenceNums ↦ seqsStart])
    ∧ configState' = [configState EXCEPT !.addedSequencers = @ ∪ {s}]
    ∧ UNCHANGED ⟨sequencerState, receiverState⟩

 Receiver r receives an mFinishAdd message m
HandleFinishAdd(r, m) ≜
LET
    s ≜ m.sequencer
    rstate ≜ receiverState[r]
IN
    ∧ s ∉ rstate.activeSequencers
    ∧ s ∉ rstate.removedSequencers
    ∧ s ∈ rstate.addedSequencers
    ∧ receiverState = [receiverState EXCEPT ![r] =
      [@ EXCEPT !.activeSequencers = @ ∪ {s},
                !.timestamps = [@ EXCEPT ![s] = m.timestamp],
                !.sequenceNums = [@ EXCEPT ![s] = m.sequenceNums[r]]]]
    ∧ UNCHANGED ⟨messages, sequencerState, configState⟩


BeginRemoveSequencer(r, s) ≜
LET
    rstate ≜ receiverState[r]
    gs ≜ {g ∈ Range(rstate.delivered) ∪ rstate.buffer :
            ∧ g.vtype = vGroupcast
            ∧ g.sequencer = s}
```

```
    seqs ≜ [rp ∈ receivers ↦ Max({0} ∪
              {g.sequenceNums[rp] : g ∈
               {gp ∈ gs : rp ∈ DOMAIN gp.sequenceNums}})]
IN
    ∧ s ∉ rstate.removedSequencers
    ∧ Send([mtype          ↦ mRemoveSequencer,
            receiver       ↦ r,
            sequencer      ↦ s,
            sequenceNums ↦ seqs])
    ∧ receiverState' = [receiverState EXCEPT ![r] =
                          [@ EXCEPT !.removedSequencers = @ ∪ {s}]]
    ∧ UNCHANGED ⟨sequencerState, configState⟩

RemoveSequencer(s) ≜
LET
    removes ≜ {m ∈ messages :
                 m.mtype = mRemoveSequencer ∧ m.sequencer = s}
    lastSeqs ≜ [r ∈ receivers ↦ Max({0} ∪
                 {m.sequenceNums[r] : m ∈ removes})]
IN
    ∧ s ∉ configState.removedSequencers
    ∧ ∀ r ∈ receivers : ∃ m ∈ removes : m.receiver = r
    ∧ Send([mtype          ↦ mFinishRemove,
            sequencer      ↦ s,
            sequenceNums ↦ lastSeqs])
    ∧ configState' = [configState EXCEPT !.removedSequencers = @ ∪ {s}]
    ∧ UNCHANGED ⟨sequencerState, receiverState⟩

Receiver r receives an mFinishRemove message m
HandleFinishRemove(r, m) ≜
LET
    s ≜ m.sequencer
    rstate ≜ receiverState[r]
IN
    ∧ s ∉ rstate.removedSequencers
    ∧ DeliverAvailable(r, {}, s, 0, m.sequenceNums[r], {s})
```

Main Transition Function

```
Next ≜ ∨ ∃ s ∈ sequencers : ∨ AdvanceTime(s)
                            ∨ SendGroupcast(s)
                            ∨ SendFlush(s)
                            ∨ AddSequencer(s)
                            ∨ RemoveSequencer(s)
       ∨ ∃ m ∈ messages :
```

$$
\begin{array}{l}
\quad \lor \ \land \ m.mtype = mGroupcast \\
\qquad \land \exists \, r \in \text{DOMAIN } m.sequenceNums : HandleGroupcast(r,\ m) \\
\quad \lor \ \land \ m.mtype = mFlush \\
\qquad \land \exists \, r \in \text{DOMAIN } m.sequenceNums : HandleFlush(r,\ m) \\
\quad \lor \ \land \ m.mtype = mFinishAdd \\
\qquad \land \exists \, r \in receivers : HandleFinishAdd(r,\ m) \\
\quad \lor \ \land \ m.mtype = mFinishRemove \\
\qquad \land \exists \, r \in receivers : HandleFinishRemove(r,\ m) \\
\lor \exists \, r \in receivers : \exists \, s \in sequencers : \\
\quad \lor \ BeginAddSequencer(r,\ s) \\
\quad \lor \ BeginRemoveSequencer(r,\ s)
\end{array}
$$