Designing Distributed Systems Using Approximate Synchrony in Data Center Networks

Dan R. K. Ports   Jialin Li   Vincent Liu   Naveen Kr. Sharma   Arvind Krishnamurthy
University of Washington
{*drkp,lijl,vincent,naveenks,arvind*}*@cs.washington.edu*

Abstract

Distributed systems are traditionally designed independently from the underlying network, making worst-case assumptions (*e.g.,* complete asynchrony) about its behavior. However, many of today's distributed applications are deployed in data centers, where the network is more reliable, predictable, and extensible. In these environments, it is possible to co-design distributed systems with their network layer, and doing so can offer substantial benefits.

This paper explores network-level mechanisms for providing *Mostly-Ordered Multicast (MOM)*: a best-effort ordering property for concurrent multicast operations. Using this primitive, we design *Speculative Paxos*, a state machine replication protocol that relies on the network to order requests in the normal case. This approach leads to substantial performance benefits: under realistic data center conditions, Speculative Paxos can provide 40% lower latency and 2.6× higher throughput than the standard Paxos protocol. It offers lower latency than a latency-optimized protocol (Fast Paxos) with the same throughput as a throughput-optimized protocol (batching).

1 Introduction

Most distributed systems are designed independently from the underlying network. For example, distributed algorithms are typically designed assuming an asynchronous network, where messages may be arbitrarily delayed, dropped, or reordered in transit. In order to avoid making assumptions about the network, designers are in effect making worst-case assumptions about it.

Such an approach is well-suited for the Internet, where little is known about the network: one cannot predict what paths messages might take or what might happen to them along the way. However, many of today's applications are distributed systems that are deployed in data centers. Data center networks have a number of desirable properties that distinguish them from the Internet:

- Data center networks are more *predictable*. They are designed using structured topologies [8,15,33], which makes it easier to understand packet routes and expected latencies.
- Data center networks are more *reliable*. Congestion losses can be made unlikely using features such as Quality of Service and Data Center Bridging [18].
- Data center networks are more *extensible*. They are part of a single administrative domain. Combined with new flexibility provided by modern technologies like software-defined networking, this makes it possible to deploy new types of in-network processing or routing.

These differences have the potential to change the way distributed systems are designed. It is now possible to co-design distributed systems and the network they use, building systems that rely on stronger guarantees available in the network and deploying new network-level primitives that benefit higher layers.

In this paper, we explore the benefits of co-designing in the context of state machine replication—a performance-critical component at the heart of many critical data center services. Our approach is to treat the data center as an approximation of a synchronous network, in contrast to the asynchronous model of the Internet. We introduce two new mechanisms, a new network-level primitive called *Mostly-Ordered Multicast* and the *Speculative Paxos* replication protocol, which leverages approximate synchrony to provide higher performance in data centers.

The first half of our approach is to engineer the network to provide stronger ordering guarantees. We introduce a *Mostly-Ordered Multicast* primitive (MOM), which provides a best-effort guarantee that all receivers will receive messages from different senders in a consistent order. We develop simple but effective techniques for providing Mostly-Ordered Multicast that leverage the structured topology of a data center network and the forwarding flexibility provided by software-defined networking.

Building on this MOM primitive is *Speculative Paxos*, a new protocol for state machine replication designed for an environment where reordering is rare. In the normal case, Speculative Paxos relies on MOM's ordering guarantees to efficiently sequence requests, allowing it to execute and commit client operations with the minimum possible latency (two message delays) and with significantly higher throughput than Paxos. However, Speculative Paxos remains correct even in the uncommon case where messages are delivered out of order: it falls back on a reconciliation protocol that ensures it remains safe and live with the same guarantees as Paxos.

Our experiments demonstrate the effectiveness of this approach. We find:

- Our customized network-level multicast mechanism ensures that multicast messages can be delivered in a consistent order with greater than 99.9% probability in a data center environment.
- In these environments, Speculative Paxos provides 40% lower latency and 2.6× higher throughput than leader-based Paxos.
- Speculative Paxos can provide the best of both worlds: it offers 20% lower latency than a latency-optimized protocol (Fast Paxos) while providing the same throughput as a throughput-optimized protocol (batching).

We have used Speculative Paxos to implement various components of a transactional replicated key-value store. Compared to other Paxos variants, Speculative Paxos allows this application to commit significantly more transactions within a fixed latency budget.

2 Background

Our goal is to implement more efficient replicated services by taking advantage of features of the data center environment. To motivate our approach, we briefly discuss existing consensus algorithms as well as the design of typical data center networks upon which they are built.

2.1 Replication and Consensus Algorithms

Replication is widely used to provide highly available and consistent services in data centers. For example, services like Chubby [4] and ZooKeeper [17] provide applications with support for distributed coordination and synchronization. Similarly, persistent storage systems like H-Store [39], Granola [10], and Spanner [9] require multiple replicas to commit updates. This provides better availability than using a single replica and also provides better performance by eschewing costly synchronous disk writes in favor of maintaining multiple copies in RAM.

Replication systems rely on a consensus protocol (*e.g.,* Paxos [24, 25], Viewstamped Replication [29, 35], or atomic broadcast [3, 19]) to ensure that operations execute in a consistent order across replicas. In this paper, we consider systems that provide a *state machine replication* interface [23, 38]. Here, a set of nodes are either *clients* or *replicas*, which both run application code and interact with each other using the replication protocol. Note that, here, clients are application servers in the data center, not end-users. Clients submit *requests* containing an operation to be executed. This begins a multi-round protocol to ensure that replicas agree on a consistent ordering of operations before executing the request.

As an example, consider the canonical state machine replication protocol, leader-based Paxos. In the normal case, when there are no failures, requests are processed as shown in Figure 1. One of the replicas is designated as the *leader*, and is responsible for ordering requests; the leader can be replaced if it fails. Clients submit requests to the leader. The leader assigns each incoming request a *sequence number*, and sends a PREPARE message to the other replicas containing the request and sequence number. The other replicas record the request in the log and acknowledge with a PREPARE-OK message. Once the leader has received responses from enough replicas, it executes the operation and replies to the client.

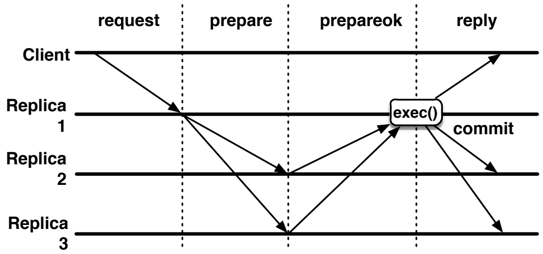


Figure 1: Normal-case execution of Multi-Paxos/Viewstamped Replication

Data center applications demand high performance from replicated systems. These systems must be able to execute operations with both high throughput and low latency. The latter is an increasingly important factor for modern web applications that routinely access data from thousands of storage servers, while needing to keep the total latency within strict bounds for interactive applications [37]. For replication protocols, throughput is typically a function of the load on a bottleneck entity, *e.g.,* the leader in Figure 1, which processes a disproportionate number of messages. Latency is primarily a function of the number of message delays in the protocol—for Paxos, four message delays from when a client submits its request until it receives a reply.

2.2 Data Centers

Today's data centers incorporate highly engineered networks to provide high availability, high throughput, low latency, and low cost. Operators take advantage of the following properties to tune their networks:

Centralized control. All of the infrastructure is in a single administrative domain, making it possible for operators to deploy large-scale changes. Software-defined networking tools (*e.g.,* OpenFlow) make it possible to implement customized forwarding rules coordinated by a central controller.

A structured network. Data center networks are multi-rooted trees of switches typically organized into three levels. The leaves of the tree are Top-of-Rack (ToR) switches that connect down to many machines in a rack, with a rack containing few tens of servers. These ToR switches are interconnected using additional switches or routers, which are organized into an *aggregation tier* in the middle and a *core tier* at the top. Each ToR switch is connected to multiple switches at the next level, thus providing desired resilience in the face of link or switch failures. Racks themselves are typically grouped into a *cluster* (about ten to twenty racks) such that all connectivity within a cluster is provided by just the bottom two levels of the network.

Within the data center, there may be many replicated services: for example, Google's Spanner and similar storage systems use one replica group per shard, with hundreds or thousands of shards in the data center. The replicas in each group will be located in different racks (for failure-independence) but may be located in the same cluster to simplify cluster management and scheduling. The service will receive requests from clients throughout the data center.

Switch support for QoS. The controlled setting also makes it possible to deploy services that can transmit certain types of messages (*e.g.,* control messages) with higher priority than the rest of the data center traffic. These priorities are implemented by providing multiple hardware or software output queues—one for each priority level. When using strict priorities, the switch will always pull from higher priority queues before lower priority queues. The length and drop policy of each queue can be tuned to drop lower priority traffic first and can also be tuned to minimize latency jitter.

3 Mostly-Ordered Multicast

The consensus algorithms described in the previous section rely heavily on the concept of ordering. Most Paxos deployments dedicate a leader node to this purpose; approaches such as Fast Paxos [27] rely on requests to arrive in order. We argue instead that the structured, highly-engineered networks used in data centers can themselves be used to order operations in the normal case. To that end, this section explores different network-layer options for providing a *mostly-ordered multicast (MOM)* mechanism. We show that simple techniques can effectively provide best-effort ordering in a data center.

3.1 Model

We consider multicast primitives that allow clients to communicate simultaneously with a group of receivers N.

In this category, the traditional *totally-ordered multicast* provides the following property: if $n_i \in N$ processes a multicast message m followed by another multicast message m', then any other node $n_j \in N$ that receives m' must process m before m'. Primitives like this are common in group communication systems [3]. Ensuring that this property holds even in the presence of failures is a problem equivalent to consensus, and would obviate the need for application code to run protocols like Paxos at all.

Instead, we consider a relaxed version of this property, which does not require it to hold in every case. A multicast implementation is said to possess the *mostly-ordered multicast* property if the above ordering constraint is satisfied with high frequency. This permits occasional *ordering violations*: these occur if n_i processes m followed by m' and either (1) n_j processes m after m', or (2) n_j does not process m at all (because the message is lost).

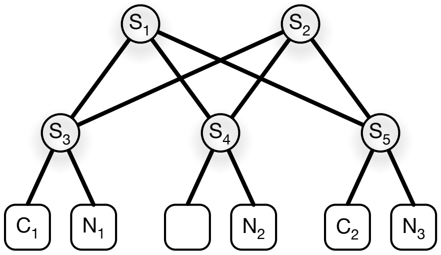


Figure 2: Clients C_1 and C_2 communicating to a multicast group comprising of N_1, N_2, and N_3.

This is an empirical property about the common-case behavior, not a strict guarantee. As a result, MOMs can be implemented as a best-effort network primitive. We seek to take advantage of the properties of the data center network previously described in order to implement MOMs efficiently in the normal case. The property may be violated in the event of transient network failures or congestion, so application-level code must be able to handle occasional ordering violations.

In this section, we first examine why existing multicast mechanisms do not provide this property, and then describe three techniques for implementing MOMs. Each stems from the idea of equalizing path length between multicast messages with a topology-aware multicast. The second adds QoS techniques to equalize latency while the third leverages in-network serialization to guarantee correct ordering. In Section 3.4, we evaluate these protocols using both an implementation on an OpenFlow switch testbed and a simulation of a datacenter-scale network. We show that the first two techniques are effective at providing MOMs with a reordering rate of ∼0.01–0.1% and the third eliminates reorderings entirely except during network failures.

3.2 Existing Multicast is Not Ordered

We first consider existing network-layer multicast mechanisms to understand why ordering violations occur.

Using IP multicast, a client can send a single multicast message to the target multicast address and have it delivered to all of the nodes. Multicast-enabled switches will, by default, flood multicast traffic to all the ports in a broadcast domain. Unnecessary flooding costs can be eliminated by using IGMP, which manages the membership of a multicast group.

Using a network-level multicast mechanism, packets from different senders may be received in conflicting orders because they traverse paths of varying lengths and experience varying levels of congestion along different links. For example, suppose the clients C_1 and C_2 in Figure 2 each send a multicast message to the multicast group $\{N_1, N_2, N_3\}$ at times $t = 0$ and $t = \varepsilon$ respectively. Let $p_{i \to j}$ represent the network path traversed by the multicast operation initiated by C_i to N_j, and $l(p_{i \to j})$ represent its latency. In this setting, the ordering property is violated if N_1 receives m_1 followed by m_2 while N_3 receives m_2 followed by m_1, which could occur if $l(p_{1 \to 1}) < l(p_{2 \to 1}) + \varepsilon$ and $l(p_{1 \to 3}) > l(p_{2 \to 3}) + \varepsilon$. This is distinctly possible since the paths $p_{1 \to 1}$ and $p_{2 \to 3}$ traverse just two links each while $p_{2 \to 1}$ and $p_{1 \to 3}$ traverse four links each.

In practice, many applications do not even use network-level multicast; they use application-level multicast mechanisms such as having the client send individual unicast messages to each of the nodes in the target multicast group. This approach, which requires no support from the network architecture, has even worse ordering properties. In addition to the path length variation seen in network-level multicast, there is additional latency skew caused by the messages not being sent at the same time.

3.3 Our Designs

We can improve the ordering provided by network-level multicast by building our own multicast mechanisms. Specifically, we present a sequence of design options that provide successively stronger ordering guarantees.

1. Topology-aware multicast: Ensure that all multicast messages traverse the same number of links. This eliminates reordering due to path dilation.
2. High-priority multicast: Use topology-aware multicast, but also assign high QoS priorities to multicasts. This essentially eliminates drops due to congestion, and also reduces reordering due to queuing delays.
3. In-network serialization: Use high-priority multicast, but route all packets through a single root switch. This eliminates all remaining non-failure related reordering.

The common intuition behind all of our designs is that messages can be sent along predictable paths through the data center network topology with low latency and high reliability in the common case.

We have implemented these three designs using OpenFlow [31] software-defined networking, which allows it to be deployed on a variety of switches. OpenFlow provides access to switch support for custom forwarding rules, multicast replication, and even header rewriting. Our designs assume the existence of a SDN controller for ease of configuration and failure recovery. The controller installs appropriate rules for multicast forwarding, and updates them when switch failures are detected.

3.3.1 Topology-Aware Multicast

In our first design, we attempt to minimize the disparity in message latencies by assuring that the messages corresponding to a single multicast operation traverse the same number of links. To achieve this, we route multicast messages through switches that are equidistant from all of the nodes in the target multicast group. The routes are dependent on the scope of the multicast group.

For instance, if all multicast group members are located within the same cluster in a three-level tree topology, the aggregation switches of that cluster represent the nearest set of switches that are equidistant from all members. For datacenter-wide multicast groups, multicast messages are routed through the root switches.

Equalizing path lengths can cause increased latency for some recipients of each multicast message, because messages no longer take the shortest path to each recipient. However, the maximum latency—and, in many cases, the average latency—is not significantly impacted: in datacenter-wide multicast groups, some messages would have to be routed through the root switches anyway. Moreover, the cost of traversing an extra link is small in data center networks, particularly in comparison to the Internet. As a result, this tradeoff is a good one: Speculative Paxos is able to take advantage of the more predictable ordering to provide better end-to-end latency.

Addressing. Each multicast group has a single address. In the intra-cluster case, the address shares the same prefix as the rest of the cluster, but has a distinct prefix from any of the ToR switches. In the datacenter-wide case, the address should come from a subnet not used by any other cluster.

Routing. The above addressing scheme ensures that, with longest-prefix matching, multicast messages are routed to the correct set of switches without any changes to the existing routing protocols. The target switches will each have specific rules that convert the message to a true multicast packet and will send it downward along any replica-facing ports. Lower switches will replicate the multicast packet on multiple ports as necessary.

Note that in this scheme, core switches have a multicast rule for every datacenter-wide multicast group while aggregation switches have a rule for every multicast group within its cluster. Typical switches have support for thousands to tens of thousands of these multicast groups.

As an example of end-to-end routing, suppose the client C_1 in Figure 2 wishes to send a message to a three-member multicast group that is spread across the data center. The group's multicast address will be of a subnet not shared by any cluster, and thus will be routed to either S_1 or S_2. Those core switches will then replicate it into three messages that are sent on each of the downward links. This simple mechanism guarantees that all messages traverse the same number of links.

Failure Recovery. When a link or switch fails, the SDN controller will eventually detect the failure and route around it as part of the normal failover process. In many cases, when there is sufficient path redundancy inside the network (as with a data center network organized into a Clos topology [15]), this is sufficient to repair MOM routing as well. However, in some cases—most notably a traditional fat tree, where there is only a single path from each root switch down to a given host—some root switches may become unusable for certain replicas. In these cases, the controller installs rules in the network that "blacklist" these root switches for applicable multicast groups. Note, however, that failures along upward links in a fat tree network can be handled locally by simply redirecting MOM traffic to any working root without involving the controller [30].

3.3.2 High-Priority Multicast

The above protocol equalizes path length and, in an unloaded network, significantly reduces the reordering rate of multicast messages. However, in the presence of background traffic, different paths may have different queuing delays. For example, suppose clients C_1 and C_2 in Figure 2 send multicasts m_1 and m_2 through S_1 and S_2 respectively. The latency over these two switches might vary significantly. If there is significant cross-traffic over the links $S_1 - S_5$ and $S_2 - S_4$ but not over the links $S_1 - S_4$ and $S_2 - S_5$, then N_2 is likely to receive m_1 followed by m_2 while N_3 would receive them in opposite order.

We can easily mitigate the impact of cross-traffic on latency by assigning a higher priority to MOM traffic using Quality of Service (QoS) mechanisms. Prioritizing this traffic is possible because this traffic is typically a small fraction of overall data center traffic volume. By assigning MOM messages to a strict-priority hardware queue, we can ensure that MOM traffic is always sent before other types of traffic. This limits the queuing delays introduced by cross-traffic to the duration of a single packet. With 10 Gbps links and 1500 byte packets, this corresponds to a worst-case queuing delay of about 1.2 μs per link or a total of about 2.4 μs from a core switch to a ToR switch. Our evaluation results show that this leads to a negligible amount of reordering under typical operating conditions.

3.3.3 In-Network Serialization

Even with QoS, currently-transmitting packets cannot be preempted. This fact, combined with minor variations in switch latency imply that there is still a small chance of message reordering. For these cases, we present an approach that uses the network itself to *guarantee* correct ordering of messages in spite of cross-traffic.

Our approach is to route all multicast operations to a given group through the same switch. This top-level switch not only serves as a serialization point, but also ensures that messages to a given group node traverse the same path. As a result, it provides perfect ordering as long as the switch delivers packets to output ports in order (we have not observed this to be a problem, as discussed in Section 3.4.1) and there are no failures.

Addressing/Routing. As before, we assign a single address to each multicast group. In this design, however, the SDN controller will unilaterally designate a root or aggregation switch as a serialization point and install the appropriate routes in the network. By default, we hash the multicast addresses across the relevant target switches for load balancing and ease of routing.

In certain network architectures, similar routing functionality can also be achieved using PIM Sparse Mode multicast [12], which routes multicast packets from all sources through a single "rendezvous point" router.

Failure Recovery. Like the failure recovery mechanism of Section 3.3.1, we can also rely on an SDN controller to route around failures. If a switch no longer has a valid path to all replicas of a multicast group or is unreachable by a set of clients, the controller will remap the multicast group's address to a different switch that *does* have a valid path. This may require the addition of routing table entries across a handful of switches in the network. These rules will be more specific than the default, hashed mappings and will therefore take precedence. Downtime due to failures is minimal: devices typically have four or five 9's of reliability [14] and path recovery takes a few milliseconds [30].

We can further reduce the failure recovery time by letting the end hosts handle failover. We can do this by setting up n different multicast serialization points each with their own multicast address and pre-installed routing table entries. By specifying the multicast address, the clients thus choose which serialization point to use. A client can failover from one designated root switch to another immediately after it receives switch failure notifications from the fabric controller or upon encountering a persistent communication failure to a target MOM group. This approach requires some additional application-level complexity and increases routing table/address usage, but provides much faster failover than the baseline.

Load Balancing. In-network serialization is not inherently load-balancing in the same way as our previous designs: all multicast traffic for a particular group traverses the same root switch. However, as described previously, there are many replica groups, each with message load far below the capacity of a data center core switch. Different groups will be hashed to different serialization switches, providing load balancing in aggregate. If necessary, the SDN controller can explicitly specify the root switch for particular groups to achieve better load balancing.

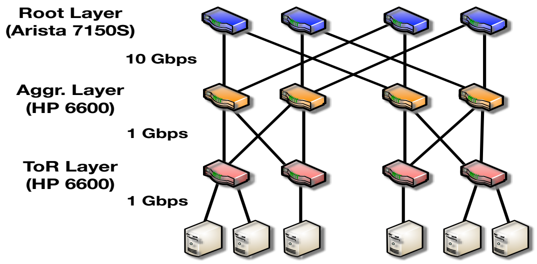


Figure 3: Testbed configuration

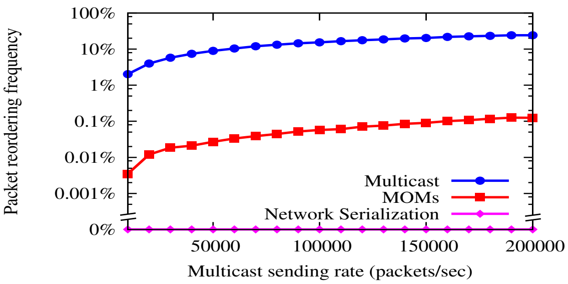


Figure 4: Measured packet reorder rates on 12-switch testbed

3.4 Evaluation of MOMs

Can data center networks effectively provide mostly-ordered multicast? To answer this question, we conducted a series of experiments to determine the reorder rate of concurrent multicast transmissions. We perform experiments using a multi-switch testbed, and conduct simulations to explore larger data center deployments.

3.4.1 Testbed Evaluation

We evaluate the ordering properties of our multicast mechanism using a testbed that emulates twelve switches in a fat-tree configuration, as depicted in Figure 3. The testbed emulates four Top-of-Rack switches in two clusters, with two aggregation switches per cluster and four root switches connecting the clusters. Host-ToR and ToR-aggregation links are 1 Gbps, while aggregation-root links are 10 Gbps. This testbed captures many essential properties of a data center network, including path length variance and the possibility for multicast packets to arrive out of order due to multi-path effects. The testbed can deliver multicast messages either using native IP multicast, topology-aware MOM, or network serialization.

The testbed is realized using VLANs on five switches. Four HP ProCurve 6600 switches implement the ToR and aggregation switches, and an Arista 7150S-24 10 Gbps switch implements the root switches. All hosts are Dell PowerEdge R610 servers with 4 6-core Intel Xeon L5640 CPUs running Ubuntu 12.04, using Broadcom BCM5709 1000BaseT adapters to connect to the ToRs.

A preliminary question is whether individual switches will cause concurrent multicast traffic to be delivered in conflicting orders, which could occur because of parallelism in switch processing. We tested this by connecting multiple senders and receivers to the same switch, and verified that all receivers received multicast traffic in the same order. We did not observe any reorderings on any of the switches we tested, including the two models in our testbed, even at link-saturating rates of multicast traffic.

With the testbed configured as in Figure 3, we connected three senders and three receivers to the ToR switches. The receivers record the order of arriving multicasts and compare them to compute the frequency of ordering violations.

In this configuration, ordering violations can occur because multicast packets traverse different paths between switches. Figure 4 shows that this occurs frequently for conventional IP multicast, with as many as 25% of packets reordered. By equalizing path lengths, our topology-aware MOM mechanism reduces the reordering frequency by 2–3 orders of magnitude. Network serialization eliminates reorderings entirely.

3.4.2 Simulation Results

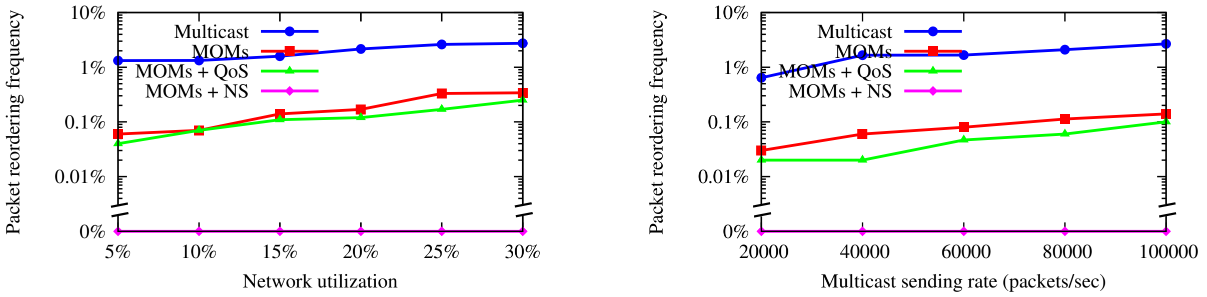
To evaluate the effectiveness of our proposed multicast designs on a larger network, we use a parallel, packet-level, event-driven simulator. In addition to the experiments described below, we have validated the simulator by simulating the same topology as our testbed, using measured latency distributions for the two switch types. The simulated and measured results match to within 8%.

The simulated data center network topology consists of 2560 servers and a total of 119 switches. The switches are configured in a three-level FatTree topology [1] with a total oversubscription ratio of about 1:4. The core and aggregation switches each have 16 10 Gbps ports while the ToRs each have 8 10 Gbps ports and 40 1 Gbps ports. Each switch has a strict-priority queue in addition to standard queues. Both queues use drop-tail behavior and a switching latency distribution taken from our measurements of the Arista 7150 switch.

In this setup, we configure end hosts to periodically transmit MOM traffic to a multicast group of 3 nodes where we observe the frequency of ordering violations. In addition to MOM traffic, hosts also send background traffic to one another. We derive the distribution of interarrival times and ON/OFF-period length for the background traffic from measurements of Microsoft data centers [2].

In the experiments in Figure 5, we measure reordering rates for the four options previously discussed: standard multicast, topology-aware multicast (MOMs), MOMs with QoS, and MOMs with in-network serialization.

Reordering. In Figure 5(a), we fix the MOM sending rate to 50,000 messages per second and vary the amount of simulated background traffic. The range equates to about 5–30% average utilization. Note that data center traffic is extremely bursty, so this range is typical [2]. In



(a) Reorder rates with varying network utilization; MOM sending rate fixed at 50,000 per second.

(b) Reorder rates with varying MOM throughput; background traffic fixed at 10% average utilization.

Figure 5: Simulated packet reorder rates, in a 160-switch, three-level fat-tree network

Figure 5(b), we vary the MOM sending rate and fix the average utilization to 10%. Results are similar for other utilization rates.

As expected, the standard multicast approach has a relatively high rate of packet reorderings because packets traverse paths of varying lengths. Simply being aware of the topology reduces the rate of reorderings by an order of magnitude, and employing QoS prioritization mitigates the impact of congestion caused by other traffic. The in-network serialization approach achieves perfect ordering: as packets are routed through a single switch, only congestion losses could cause ordering violations.

Latency. As we previously observed, MOM can increase the path length of a multicast message to the longest path from a sender to one of the receivers. As a result, the time until a message arrives at the first receiver increases. However, for more than 70% of messages, the *average* latency over all receivers remains unchanged. The latency skew, i.e., the difference between maximum and minimum delivery time to all recipients for any given message, is in all cases under 2.5 μs for the in-network serialization approach.

4 Speculative Paxos

Our evaluation in the previous section shows that we can engineer a data center network to provide MOMs. How should this capability influence our design of distributed systems? We argue that a data center network with MOMs can be viewed as *approximately synchronous*: it provides strong ordering properties in the common case, but they may occasionally be violated during failures.

To take advantage of this model, we introduce *Speculative Paxos*, a new state machine replication protocol. Speculative Paxos relies on MOMs to be ordered in the common case. Each replica speculatively executes requests based on this order, before agreement is reached. This speculative approach allows the protocol to run with the minimum possible latency (two message delays) and provides high throughput by avoiding communication between replicas on each request. When occasional ordering violations occur, it invokes a reconciliation protocol to rollback inconsistent operations and agree on a new state. Thus, Speculative Paxos does not rely on MOM for correctness, only for efficiency.

Client interface

- `invoke`(*operation*) → *result*

Replica interface

- `speculativelyExecute`(*seqno*, *operation*) → *result*
- `rollback`(*from-seqno*, *to-seqno*, *list<operations>*)
- `commit`(*seqno*)

Figure 6: Speculative Paxos library API

4.1 Model

Speculative Paxos provides state machine replication, following the model in Section 2.1. In particular, it guarantees linearizability [16] provided that there are no more than f failures: operations appear as though they were executed in a consistent sequential order, and each operation sees the effect of operations that completed before it. The API of the Speculative Paxos library is shown in Figure 6.

Speculative Paxos differs from traditional state machine replication protocols in that it executes operations speculatively at the replicas, *before* agreement is reached about the ordering of requests. When the replica receives a request, the Speculative Paxos library makes a `speculativelyExecute` upcall to the application code, providing it with the requested operation and an associated sequence number. In the event of a failed speculation, the Speculative Paxos library may make a `rollback` upcall, requesting that the application undo the most recent operations and return to a previous state. To do so, it provides the sequence number and operation of all the commands to be rolled back. The Speculative Paxos library also periodically makes `commit` upcalls to indicate that previously-speculative operations will never be rolled back, allowing the application to discard information (*e.g.,* undo logs) needed to roll back operations.

Importantly, although Speculative Paxos executes operations speculatively on replicas, *speculative state is not exposed to clients*. The Speculative Paxos library only returns results to the client application after they are known to have successfully committed in the same order at a quorum of replicas. In this respect, Speculative Paxos is similar to Zyzzyva [22], and differs from systems that employ speculation on the client side [41].

Failure Model Although the premise for our work is that data center networks can provide stronger guarantees of ordering than distributed algorithms typically assume, Speculative Paxos does not rely on this assumption for correctness. It remains correct under the same assumptions as Paxos and Viewstamped Replication: it requires $2f+1$ replicas and provides safety as long as no more than f replicas fail simultaneously, even if the network drops, delays, reorders, or duplicates messages. It provides liveness as long as messages that are repeatedly resent are eventually delivered before the recipients time out. (This requirement is the same as in Paxos, and is required because of the impossibility of consensus in an asynchronous system [13].)

4.2 Protocol

Speculative Paxos consists of three sub-protocols:

- *Speculative processing* commits requests efficiently in the normal case where messages are ordered and $< f/2$ replicas have failed (Section 4.2.2)
- *Synchronization* periodically verifies that replicas have speculatively executed the same requests in the same order (Section 4.2.3)
- *Reconciliation* ensures progress when requests are delivered out of order or when between $f/2$ and f nodes have failed (Section 4.2.4)

4.2.1 Replica State

Each replica maintains a *status*, a *log*, and a *view number*.

The replica's status indicates whether it can process new operations. Most of the time, the replica is in the NORMAL state, which allows speculative processing of new operations. While the reconciliation protocol is in progress, the replica is in the RECONCILIATION state. There are also RECOVERY and RECONFIGURATION states used when a failed replica is reconstructing its state and when the membership of the replica set is changing.

The log is a sequence of operations executed by the replica. Each entry in the log is tagged with a *sequence number* and a state, which is either COMMITTED or SPECULATIVE. All committed operations precede all speculative operations. Each log entry also has an associated *summary hash*. The summary hash of entry n is given by

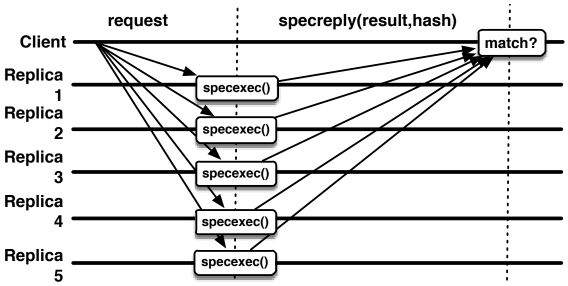


Figure 7: Speculative processing protocol.

$$summary_n = H(summary_{n-1} \,||\, operation_n)$$

Thus, it summarizes the replica's state up to that point: two replicas that have the same summary hash for log entry n must agree on the order of operations up to entry n. To simplify exposition, we assume replicas retain their entire logs indefinitely; a standard checkpointing procedure can be used to truncate logs.

The system moves through a series of views, each with a designated view number and leader. Each replica maintains its own idea of the current view number. The leader is selected in a round-robin ordering based on the view number, *i.e.,* for view v, replica v mod n is the leader. This is similar to leader election in [6] and [29], but the leader does not play a special role in the normal-case speculative processing protocol. It is used only to coordinate synchronization and reconciliation.

4.2.2 Speculative Processing

Speculative Paxos processes requests speculatively in the common case. When a client application initiates an operation, the Speculative Paxos library sends a REQUEST message to all replicas. This message includes the operation requested, the identity of a client, and a unique per-client request identifier. The REQUEST message is sent using our MOM primitive, ensuring that replicas are likely to receive concurrent requests in the same order.

Replicas participate in speculative processing when they are in the NORMAL state. Upon receiving a REQUEST message, they immediately speculatively execute the request: they assign the request the next higher sequence number, append the request to the log in the SPECULATIVE state, and make an upcall to application code to execute the request. It then sends a SPECULATIVE-REPLY message to the client, which includes the result of executing the operation as well as the sequence number assigned to the request and a summary hash of the replica's log.

Clients wait for SPECULATIVE-REPLY messages from a *superquorum* of $f + \lceil f/2 \rceil + 1$ replicas, and compare the responses. If all responses match exactly, *i.e.,* they have the same sequence number and summary hash, the client treats the operation as committed. The matching responses indicate that the superquorum of replicas have executed the request (and all previous operations) in the same order. The replicas themselves do not yet know that the operation has committed, but the reconciliation protocol ensures that any such operation will persist even if there are failures, and may not be rolled back. If the client fails to receive SPECULATIVE-REPLY messages from a superquorum of replicas before a timeout, or if the responses do not match (indicating that the replicas are not in the same state), it initiates a reconciliation, as described in Section 4.2.4.

Why is a superquorum of responses needed rather than a simple majority as in Paxos? The reasoning is the same as in Fast Paxos[1]: even correct replicas may receive operations from different clients in inconsistent orders. Consider what would happen if we had used a quorum of size $f+1$, and one request was executed by $f+1$ replicas and a different request by the other f. If any of the replicas in the majority subsequently fail, the recovery protocol will not be able to distinguish them. As a result, the size a superquorum must be chosen such that, if an operation is successful, a majority of active replicas will have that operation in their log [28].

4.2.3 Synchronization

In the speculative processing protocol, clients learn that their requests have succeeded when they receive matching speculative replies from a superquorum of replicas. The replicas, however, do not communicate with each other as part of speculative processing, so they do not learn the outcome of the operations they have executed. The synchronization protocol is a periodic process, driven by the leader, that verifies that replicas are in the same state.

Periodically (every t milliseconds, or every k requests), the leader initiates synchronization by sending a $\langle$SYNC, v, $s\rangle$ message to the other replicas, where v is its current view number and s is the highest sequence number in its log. Replicas respond to SYNC messages by sending the leader a message $\langle$SYNC-REPLY, v, s, $h(s)\rangle$, where $h(s)$ is the summary hash associated with entry s in its log. If v is not the current view number, or the replica is not in the NORMAL state, the message is ignored.

When the leader has received $f+\lceil f/2\rceil+1$ SYNC-REPLY messages for a sequence number s, including its own, it checks whether the hashes in the messages match. If so, the replicas agree on the ordering of requests up to s. The leader promotes all requests with sequence number less than or equal to s from SPECULATIVE to COMMITTED state, and makes a `commit`(s) upcall to the application. It then sends a message $\langle$COMMIT, v, s, $h(s)\rangle$ to the other replicas. Replicas receiving this message also commit all operations up to s if their summary hash matches.

If the leader receives SYNC-REPLY messages that do not have the same hash, or if a replica receives a COMMIT message with a different hash than its current log entry, it initiates a reconciliation.

4.2.4 Reconciliation

From time to time, replica state may diverge. This can occur if messages are dropped or reordered by the network, or if replicas fail. The reconciliation protocol repairs divergent state and ensures that the system makes progress.

The reconciliation protocol follows the same general structure as view changes in Viewstamped Replication [29]: all replicas stop processing new requests and send their log to the new leader, which selects a definitive log and distributes it to the other replicas. The main difference is that the leader must perform a more complex *log merging* procedure that retains any operation that successfully completed even though operations may have been executed in different orders at different replicas.

When a replica begins a reconciliation, it increments its view number and sets its status to RECONCILIATION, which stops normal processing of client requests. It then sends a $\langle$START-RECONCILIATION, $v\rangle$ message to the other replicas. The other replicas, upon receiving a START-RECONCILIATION message for a higher view than the one they are currently in, perform the same procedure. Once a replica has received START-RECONCILIATION messages for view v from f other replicas, it sends a $\langle$RECONCILE, v, v_ℓ, $log\rangle$ message to the leader of the new view. Here, v_ℓ is the last view in which the replica's status was NORMAL.

Once the new leader receives RECONCILE messages from f other replicas, it merges their logs. The log merging procedure considers all logs with the highest v_ℓ (including the leader's own log, if applicable) and produces a combined log with two properties:

- If the same prefix appears in a majority of the logs, then those entries appear in the combined log in the same position.
- Every operation in any of the input logs appears in the output log.

The first property is critical for correctness: it ensures that any operation that might have successfully completed at a client survives into the new view. Because clients treat requests as successful once they have received matching summary hashes from $f+\lceil f/2\rceil+1$ replicas, and f of those replicas might have subsequently failed, any successful operation will appear in at least $\lceil f/2\rceil+1$ logs.

The second property is not required for safety, but ensures that the system makes progress. Even if all multicast requests are reordered by the network, the reconciliation

[1] Fast Paxos is typically presented as requiring quorums of size $2f+1$ out of $3f+1$, but like Speculative Paxos can be configured such that $2f+1$ replicas can make progress with f failures but need superquorums to execute fast rounds [27].

procedure selects a definitive ordering of all requests.

The procedure for merging logs is as follows:

- The leader considers only logs with the highest v_ℓ; any other logs are discarded. This ensures that the results of a previous reconciliation are respected.
- It then selects the log with the most COMMITTED entries. These operations are known to have succeeded, so they are added to the combined log.
- Starting with the next sequence number, it checks whether a majority of the logs have an entry with the same summary hash for that sequence number. If so, that operation is added to the log in SPECULATIVE state. This process is repeated with each subsequent sequence number until no match is found.
- It then gathers all other operations found in any log that have not yet been added to the combined log, selects an arbitrary ordering for them, and appends them to the log in the SPECULATIVE state.

The leader then sends a $\langle$START-VIEW, v, *log*$\rangle$ message to all replicas. Upon receiving this message, the replica *installs* the new log: it rolls back any speculative operations in its log that do not match the new log, and executes any new operations in ascending order. It then sets its current view to v, and resets its status to NORMAL, resuming speculative processing of new requests.

Ensuring progress with f failures. The reconciliation protocol uses a quorum size of $f+1$ replicas, unlike the speculative processing protocol, which requires a superquorum of $f+\lceil f/2 \rceil+1$ replicas. This means that reconciliation can succeed even when more than $f/2$ but no more than f replicas are faulty, while speculative processing cannot. Because reconciliation ensures that all operations submitted before the reconciliation began are assigned a consistent order, it can commit operations even if up to f replicas are faulty.

Upon receiving a START-VIEW message, each replica also sends a $\langle$IN-VIEW, $v\rangle$ message to the leader to acknowledge that it has received the log for the new view. Once the leader has received IN-VIEW messages from f other replicas, it commits all of the speculative operations that were included in the START-VIEW message, notifies the clients, and notifies the other replicas with a COMMIT message. This allows operations to be committed even if there are more than $f/2$ failed replicas. This process is analogous to combining regular and fast rounds in Fast Paxos: only $f+1$ replicas are required in this case because only the leader is is allowed to propose the ordering of requests that starts the new view.

4.2.5 Recovery and Reconfiguration

Replicas that have failed and rejoined the system follow a *recovery* protocol to ensure that they have the current state. A *reconfiguration* protocol can also be used to change the membership of the replica group, *e.g.,* to replace failed replicas with new ones. For this purpose, Speculative Paxos uses standard recovery and reconfiguration protocols from Viewstamped Replication [29]. The reconfiguration protocol also includes the need to add or remove newly-joined or departing replicas to the multicast group. For our OpenFlow multicast forwarding prototype, it requires contacting the OpenFlow controller.

Reconfiguration can also be used to change the system from Speculative Paxos to a traditional implementation of Paxos or VR. Because reconfiguration can succeed with up to f failures, this can be a useful strategy when more than $f/2$ failures occur, or during transient network failures that can cause packets to be reordered.

4.3 Correctness

Speculative Paxos treats an operation as successful (and notifies the client application) if the operation is COMMITTED in at least one replica's log, or if it is SPECULATIVE in a common prefix of $f+\lceil f/2 \rceil+1$ replica's logs.

Any successful operation always survives in the same serial order. We only need to consider reconciliations here, as speculative processing will only add new operations to the end of the log, and synchronization will cause successful operations to be COMMITTED. Consider first operations that succeeded because they were speculatively executed on $f+\lceil f/2 \rceil+1$ replicas. These operations will survive reconciliations. The reconciliation process requires $f+1$ out of $2f+1$ replicas to respond, so $\lceil f/2 \rceil+1$ logs containing these operations will be considered, and the log merging algorithm ensures they will survive in the same position.

Operations can also be committed through reconciliation. This can happen only once $f+1$ replicas have processed the START-VIEW message for that view. All of these replicas agree on the ordering of these operations, and at least one will participate in the next reconciliation, because $f+1$ replicas are required for reconciliation. The reconciliation procedure only merges logs with the highest v_ℓ, and the log merging procedure will ensure that the common prefix of these logs survives.

Only one operation can succeed for a given sequence number. By itself, the speculative processing protocol allows only one operation to succeed for a given sequence number, because an operation only succeeds if speculatively committed by a superquorum of replicas. The reconciliation protocol will not assign a different operation to any sequence number that could potentially have speculatively committed at enough replicas, nor one that committed as the result of a previous reconciliation.

	Latency (Msg Delays)	**Message Complexity**	**Messages at Bottleneck Replica**
Paxos	4	$2n$	$2n$
Paxos + batching	4+	$2+\frac{2n}{b}$	$2+\frac{2n}{b}$
Fast Paxos	3	$2n$	$2n$
Speculative Paxos	2	$2n+\frac{2n}{s}$	$2+\frac{2n}{s}$

Table 1: Comparison of Paxos, Fast Paxos, and Speculative Paxos. n is the total number of replicas; b is the batch size for Paxos with batching, and s is the number of requests between synchronization for Speculative Paxos.

4.4 Discussion

Speculative Paxos offers high performance because it commits most operations via the fast-path speculative execution protocol. It improves on the latency of client operations, an increasingly critical factor in today's applications: a client can submit a request and learn its outcome in two message delays—the optimal latency, and a significant improvement over the four message delays of leader-based Paxos, as shown in Table 1. Speculative Paxos also provides better throughput, because it has no bottleneck replica that bears a disproportionate amount of the load. In Speculative Paxos, each replica processes only two messages (plus periodic synchronizations), whereas all $2n$ messages are processed by the leader in Paxos.

Speculative Paxos is closely related to Fast Paxos [27], which reduces latency by sending requests directly from clients to all replicas. Fast Paxos also incurs a penalty when different requests are received by the replicas in a conflicting order. In Fast Paxos, the message flow is

client → replicas → leader → client

and so the protocol requires three message delays. Speculative Paxos improves on this by executing operations speculatively so that clients can learn the result of their operations in two message delays. The tradeoff is that the reconciliation protocol is more expensive and speculative operations might need to be rolled back, making Speculative Paxos slower in conflict-heavy environments. This tradeoff is an example of our co-design philosophy: Fast Paxos would also benefit from MOM, but Speculative Paxos is optimized specially for an environment where multicasts are mostly ordered.

Speculative Paxos improves throughput by reducing the number of messages processed by each node. Despite the name, Fast Paxos does not improve throughput, although it reduces latency: the leader still processes $2n$ messages, so it remains a bottleneck. Other variants on Paxos aim to reduce this bottleneck. A common approach is to batch requests at the leader, only running the full protocol periodically. This also eliminates a bottleneck, increasing throughput dramatically, but *increases* rather than reducing latency.

4.5 Evaluation

We have implemented the Speculative Paxos protocol as a library for clients and replicas. Our library comprises about 10,000 lines of C++, and also supports leader-based Paxos (with or without batching) and Fast Paxos.

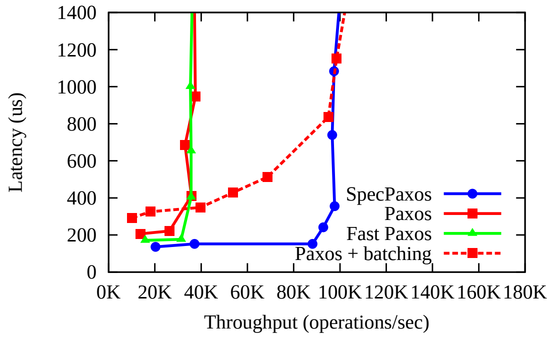


Figure 8: Latency vs throughput tradeoff for testbed deployment

We evaluate the performance of Speculative Paxos and compare it to Paxos and Fast Paxos using a deployment on the twelve-switch testbed shown in Figure 3, measuring the performance tradeoffs under varying client load. We then investigate the protocol's sensitivity to network conditions by emulating MOM ordering violations.

4.5.1 Latency/Throughput Comparison

In our testbed experiments, we use three replicas, so $f = 1$, and a superquorum of all three replicas is required for Speculative Paxos or Fast Paxos to commit operations on the fast path. The replicas and multiple client hosts are connected to the ToR switches with 1 Gbps links. Speculative Paxos and Fast Paxos clients use the network serialization variant of MOM for communicating with the replicas; Paxos uses standard IP multicast.

Figure 8 plots the median latency experienced by clients against the overall request throughput obtained by varying the number of closed-loop clients from 2 to 300. We compare Speculative Paxos, Fast Paxos, and Paxos with and without batching. In the batching variant, we use the latency-optimized sliding-window batching strategy from PBFT [6], with batch sizes up to 64. (This limit on batch sizes is not reached in this environment; higher maximum batch sizes have no effect.) This comparison shows:

- At low to medium request rates, Speculative Paxos provides lower latency (135 μs) than either Paxos (220 μs) or Fast Paxos (171 μs). This improvement can be attributed to the fewer message delays.

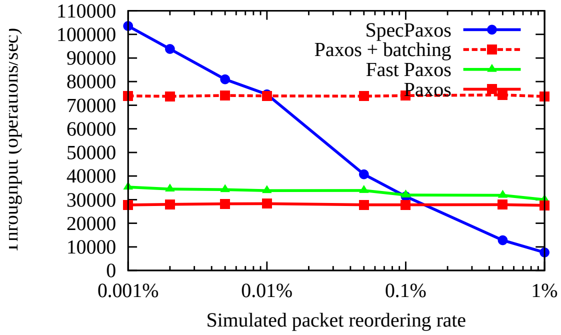


Figure 9: Throughput with simulated packet reordering

Application	Total LoC	Rollback LoC
Timestamp Server	154	10
Lock Manager	606	75
Key-value Store	2011	248

Table 2: Complexity of rollback in test applications

- Speculative Paxos is able to sustain a higher throughput level (∼100,000 req/s) than either Paxos or Fast Paxos (∼38,000 req/s), because fewer messages are handled by the leader, which otherwise becomes a bottleneck.
- Like Speculative Paxos, batching also increases the throughput of Paxos substantially by eliminating the leader bottleneck: the two achieve equivalent peak throughput levels. However, batching also increases latency: at a throughput level of 90,000 req/s, its latency is 3.5 times higher than that of Speculative Paxos.

4.5.2 Reordering Sensitivity

To gain further insight into Speculative Paxos performance under varying conditions, we modified our implementation to artificially reorder random incoming packets. These tests used three nodes with Xeon L5640 processors connected via a single 1 Gbps switch.

We measured throughput with 20 concurrent closed-loop clients. When packet reordering is rare, Speculative Paxos outperforms Paxos and Fast Paxos by a factor of 3×, as shown in Figure 9. As the reorder rate increases, Speculative Paxos must perform reconciliations and performance drops. However, it continues to outperform Paxos until the reordering rate exceeds 0.1%. Our experiments in Section 3.4 indicate that data center environments will have lower reorder rates using topology-aware multicast, and can eliminate orderings entirely except in rare failure cases using network serialization. Paxos performance is largely unaffected by reordering, and Fast Paxos throughput drops slightly because conflicts must be resolved by the leader, but this is cheaper than reconciliation.

5 Applications

We demonstrate the benefits and tradeoffs involved in using speculation by implementing and evaluating several applications ranging from the trivial to fairly complex.

Since Speculative Paxos exposes speculation at the application replicas, it requires rollback support from the application. The application must be able to rollback operations in the event of failed speculations.

We next describe three applications ranging from a simple timestamp server to a complex transactional, distributed key-value store inspired by Spanner [9]. We measure the performance achieved by using Speculative Paxos and comment on the complexity of rollback code.

Timestamp Server. This network service generates monotonically increasing timestamps or globally unique identifier numbers. Such services are often used for distributed concurrency control. Each replica maintains its own counter which is incremented upon a new request. On rollback, the counter is simply decremented once for each request to be reverted.

Lock Manager. The Lock Manager is a fault-tolerant synchronization service which provides a fine-grained locking interface. Clients can acquire and release locks in read or write mode. Each replica maintains a mapping of object locks held by a client and the converse. On rollback, both these mappings are updated by the inverse operation, *e.g.,* RELEASE(X) for LOCK(X). Since these operations do not commute, they must be rolled back in the reverse order in which they were applied.

Transactional Key-Value Store. We built a distributed in-memory key-value store which supports serializable transactions using two-phase commit and strict two-phase locking or optimistic concurrency control (OCC) to provide concurrency control. Client can perform GET and PUT operations, and commit them atomically using BEGIN, COMMIT, and ABORT operations.

The key-value store keeps multiple versions of each row (like many databases) to support reading a consistent snapshot of past data, and to implement optimistic concurrency control. Rolling back PUT operations requires reverting a key to an earlier version, and rolling back PREPARE, COMMIT, and ABORT two-phase commit operations requires adjusting transaction metadata to an earlier state.

We test the key-value store on the previously-described testbed. In our microbenchmark, a single client executes transactions in a closed loop. Each transaction involves several replicated get and put operations. Speculative Paxos commits a transaction in an average of 1.01 ms, a 30% improvement over the 1.44 ms required for Paxos, and a 10% improvement over the 1.12 ms for Fast Paxos.

We also evaluate the key-value store on a more complex benchmark: a synthetic workload based on a profile of the Retwis open-source Twitter clone. The workload chooses

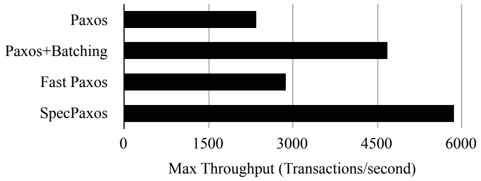


Figure 10: Maximum throughput attained by key-value store within 10 ms SLO

keys based on a Zipf distribution, and operations based on the transactions implemented in Retwis. Figure 10 plots the maximum throughput the system can achieve while remaining within a 10 ms SLO. By simultaneously providing lower latency and eliminating throughput bottlenecks, Speculative Paxos achieves significantly greater throughput within this latency budget.

6 Related Work

Weak Synchrony. The theoretical distributed systems literature has studied several models of weak synchrony assumptions. These include bounding the latency of message delivery and the relative speeds of processors [11], or introducing unreliable failure detectors [7]. In particular, a single Ethernet LAN segment has been shown to be nearly synchronous in practice, where even short timeouts are effective [40] and reorderings are rare [21]. The data center network is far more complex; we have shown that with existing multicast mechanisms, reorderings are frequent, but our network-level MOM mechanisms can be used to ensure ordering with high probability.

In the context of a single LAN, the Optimistic Atomic Broadcast [36] protocol provides total ordering of messages under the assumption of a spontaneous total ordering property equivalent to our MOM property, and was later used to implement a transaction processing system [21]. Besides extending this idea to the more complex data center context, the Speculative Paxos protocol is more heavily optimized for lower reorder rates. For example, the OAB protocol requires all-to-all communication; Speculative Paxos achieves higher throughput by avoiding this. Speculative Paxos also introduces additional mechanisms such as summary hashes to support a client/server state machine replication model instead of atomic broadcast.

Paxos Variants. Speculative Paxos is similar to Fast Paxos [27], which reduces latency when messages arrive at replicas in the same order. Speculative Paxos takes this approach further, eliminating another message round and communication between the replicas, in exchange for reduced performance when messages are reordered.

Total ordering of operations is not always needed. Generalized Paxos [26] and variants such as Multicoordinated Paxos [5] and Egalitarian Paxos [32] mitigate the cost of conflicts in a Fast Paxos-like protocol by requiring the programmer to identify requests that commute and permitting such requests to commit in different orders on different replicas. Such an approach could allow Speculative Paxos to tolerate higher reordering rates.

Speculation. Speculative Paxos is also closely related to recent work on speculative Byzantine fault tolerant replication. The most similar is Zyzzyva [22], which employs speculation on the server side to reduce the cost of operations when replicas are non-faulty. Like Speculative Paxos, Zyzzyva replicas execute requests speculatively and do not learn the outcome of their request, but clients can determine if replicas are in a consistent state. Zyzzyva's speculation assumes that requests are not assigned conflicting orders by a Byzantine primary replica. Speculative Paxos applies the same idea in a non-Byzantine setting, speculatively assuming that requests are not reordered by the network. Eve [20] uses speculation and rollback to allow non-conflicting operations to execute concurrently; some of its techniques could be applied here to reduce the cost of rollback.

An alternate approach is to apply speculation on the *client* side. SpecBFT [41] modifies the PBFT protocol so that the primary sends an immediate response to the client, which continues executing speculatively until it later receives confirmation from the other replicas. This approach also allows the client to resume executing after two message delays. However, the client cannot communicate over the network or issue further state machine operations until the speculative state is committed. This significantly limits its usability for data center applications that often perform a sequence of accesses to different storage systems [37]. Client-side speculation also requires kernel modifications to support unmodified applications [34].

7 Conclusion

We have presented two mechanisms: the Speculative Paxos protocol, which achieves higher performance when the network provides our Mostly-Ordered Multicast property, and new network-level multicast mechanisms designed to provide this ordering property. Applied together with MOM, Speculative Paxos achieves significantly higher performance than standard protocols in data center environments. This example demonstrates the benefits of co-designing a distributed system with its underlying network, an approach we encourage developers of future data center applications to consider.

Acknowledgements

We would like to thank Adriana Szekeres, Irene Zhang, our shepherd Robbert van Renesse, and the anonymous reviewers for their feedback. This work was supported in part by Google and the National Science Foundation (CNS-0963754, CNS-1217597, CNS-1318396, and CNS-1420703).

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM 2008*, Seattle, WA, USA, Aug. 2008.

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

[3] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, TX, USA, Oct. 1987.

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

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

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

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

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

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, USA, Oct. 2012.

[10] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012.

[11] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):228–323, Apr. 1988.

[12] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast-sparse mode (PIM-SM): Protocol specification. RFC 2117, June 1997. https://tools.ietf.org/html/rfc2117.

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

[14] 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.

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

[16] M. P. Herlihy and J. M. Wing. Linearizabiliy: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

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

[18] IEEE 802.1 Data Center Bridging. http://www.ieee802.org/1/pages/dcbridges.html.

[19] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[20] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, USA, Oct. 2012.

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

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

[23] L. Lamport. Time, clocks, and ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

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

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

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

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

[29] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[30] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, Apr. 2013.

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

[32] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 25rd ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, USA, Nov. 2013.

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

[34] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, Oct. 2005.

[35] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC '88)*, Toronto, Ontario, Canada, Aug. 1988.

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

[37] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's time for low latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS '11)*, Napa, CA, USA, May 2011.

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

[39] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, Vienna, Austria, Sept. 2007.

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

[41] B. Wester, J. Cowling, E. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, USA, Apr. 2009.