Chasing the tail of atomic broadcast protocols

Daniel Cason *Parisa J. Marandi*
Luiz E. Buzato *Fernando Pedone*

Technical Report - IC-15-03 - Relatório Técnico

May - 2015 - Maio

Chasing the tail of atomic broadcast protocols

Daniel Cason*[‡] Parisa J. Marandi[†][‡] Luiz E. Buzato* Fernando Pedone[‡]

Abstract

Many applications today rely on multiple services, whose results are combined to form the application's response. In such contexts, the most unreliable service and the slowest service determine the application's reliability and response time, respectively. State-machine replication and atomic broadcast are fundamental abstractions to build highly available services. In this paper, we consider the latency variability of atomic broadcast protocols. This is important because atomic broadcast has a direct impact on the response time of services. We study four high performance atomic broadcast protocols representative of different classes of protocol design and characterize their latency tail distribution under different workloads. Next, we assess how key design features of each protocol can possibly be related to the observed latency tail distributions. Our observations hint at request batching as a simple yet effective way to shorten the latency tails of some of the studied protocols; an improvement within the reach of application implementers. Indeed, our observation is not only verified experimentally, it allows us to assess which of the protocol's key design principles favor the construction of latency predictable protocols.

1 Introduction

Atomic broadcast, also known as total order broadcast, is a fundamental group communication abstraction that lies at the core of different approaches to replication, such as state-machine replication [11, 15]. In state-machine replication, servers replicate the service's state and deterministically execute client requests such that the failure of one or more replicas does not prevent client requests from being executed against non-faulty replicas. State-machine replication requires that (i) every non-faulty replica receive all requests and (ii) no two replicas disagree on the order in which requests are received. These requirements are captured by atomic broadcast.

Atomic broadcast has been extensively studied by the scientific community and many protocols under various assumptions have been proposed [8]. From a performance perspective, improving throughput and reducing latency have been the two main driving forces for new protocols. Years of research combined with developments in communication technology have led to atomic broadcast protocols that can meet most application requirements

*Institute of Computing, University of Campinas, 13081-970 Campinas, SP, Brasil. Research partially founded by FAPESP under grants 2011/23705-4 and 2013/21651-0, and CNPq grant 473340/2009-7.

[†]Microsoft Research, Cambridge, UK

[‡]Faculty of Informatics, Università della Svizzera italiana, CH-6904, Lugano, Switzerland

with respect to throughput and latency. In this paper, we consider an angle that has been unexplored in the design space of atomic broadcast protocols, namely *latency variability.* Latency variability is important because in many applications today, the execution of a request triggers the execution of multiple services, whose results are aggregated to form the request's response. In such environments, the responsiveness of the application will be determined by the latency of the slowest service; thus, each service must be as predictably responsive as possible. Although state-machine replication improves the availability of a service, it does not eliminate variations in the service's latency.

Two components account for the latency variability of a replicated service: the *ordering of requests*, that is, the atomic broadcast protocol, and the *execution of requests.* Taming the latency variability of the execution of requests has received much attention in the literature recently [6, 13]. It turns out that state-machine replication naturally reduces this source of variability in latency because commands are executed by all replicas and clients consider a command executed upon receiving the first (and thus the fastest) response from the replicas [6]. Reducing latency variations in the ordering of requests, however, requires different strategies since replicas interact in complex ways, depending on the total order protocol used, and slowdowns in one replica during the ordering of messages can easily slow down other replicas.

In this context, this paper analyses four different total order protocols with two goals in mind: (i) characterizing their design and implementation decisions in the light of their latency variability (i.e., latency tail distribution), and (ii) finding and assessing mechanisms that can be used to reduce the latency variability of these protocols.

The remainder of the paper is structured as follows. Section 2 presents the atomic broadcast protocols considered in our study. Section 3 explains our experimental methodology. Section 4 describes the results of our experimental evaluation. Section 5 discusses the lessons learned. Sections 6 reviews related work and Section 7 concludes the paper.

2 Atomic broadcast protocols

In this section, we overview the four atomic broadcast protocols considered in this study. These protocols are representatives of different engineering approaches to the problem of designing and implementing atomic broadcast.

2.1 System model

We consider a distributed system composed of processes that communicate with each other exclusively through message passing and do not have access to a common source of time. We assume processes may fail by crashing, but do not behave maliciously. The system is asynchronous and extended with additional assumptions to circumvent the FLP impossibility result [9]. For brevity, we do not describe these extensions here; detailed descriptions can be found in the papers that introduce the protocols assessed.

Atomic broadcast is formally defined by the primitives *broadcast*(m) and *deliver*(m), with the following guarantees: (i) if a non-faulty process broadcasts message m, it eventually delivers m; (ii) if a process delivers m, then all non-faulty processes deliver m; and (iii) no two processes deliver any two messages in different orders.

2.2 Ring Paxos

Ring Paxos is a high-throughput atomic broadcast protocol derived from Paxos [12]. Ring Paxos implements Paxos by distributing processes in a logical uni-directional ring [14]. Establishing a ring among processes maximizes throughput as it enables a balanced usage of the available bandwidth. In brief, Ring Paxos works as follows. Clients submit their requests to the processes in the ring. The dissemination of client requests is decoupled from the ordering of requests, which is accomplished using consensus on message identifiers. Requests are forwarded along the ring until they reach the leader, which creates the message identifiers used to totally order the requests. The leader tags each request with a unique identifier and initiates a consensus instance using a subset of the processes in the ring (i.e., Paxos's acceptors). The result of each consensus instance is circulated in the ring until all processes receive it. The execution of successive consensus instances induces a total order on requests.

2.3 S-Paxos

S-Paxos is an atomic broadcast protocol based on Paxos whose main design concern is to relieve the leader's load [4]. S-Paxos offloads the Paxos leader by balancing the load among the processes that execute the protocol. Similarly to Ring Paxos, S-Paxos separates request dissemination from ordering. Clients send their requests to a replica. Replicas create unique identifiers for the requests and send both (uid, request) to all other replicas. Requests become stable when acknowledged by a majority of replicas. When a request becomes stable, the leader passes its uid to the ordering layer, which uses instances of consensus. Batching is used by both the dissemination and the ordering layers to optimize throughput; acknowledgements are piggybacked on the messages used to transport the batches. S-Paxos operates on batches of requests and batches of uids. All processes execute all requests but only the replica that originally received a request forwards the result to the client. S-Paxos strives to balance CPU and network resources, the cost for which is paid by the number of messages that must be exchanged before a request can be ordered.

2.4 Spread

Spread is a widely used toolkit that provides several process group communication services, including atomic broadcast. Spread's initial atomic broadcast protocol was based on Totem [1] and was recently modified to include the Accelerated Ring protocol [2]. Both protocols distribute the processes in a logical ring and use a token to ensure total order. The token circulates along the ring and only the process that possesses the token can broadcast messages. In Totem, the process that holds the token increments its sequence number, broadcasts a message, and passes the token to the next participant. In the Accelerated Ring protocol, the token holder can pass the token to its successor before completing the broadcast of messages. In both the protocols, if the token holder needs to broadcast multiple messages, the sequence number in the token must be updated once per each message. Spread implements a sophisticated flow control mechanism combining a global message window and two per-process message windows ("personal" and "accelerated") [2]. These windows are managed by the protocol in order to implement a ring-based reliable communication channel. The practical consequence of the focus on efficient communication is a token that circulates with predictable latency along the ring.

2.5 THyTOB

THyTOB uses rounds to total order client requests [5]. The idea is to emulate synchronous rounds of communication atop of an asynchronous distributed system. At the beginning of each round, all processes broadcast messages (batches of requests). Each broadcast message includes its sender's identifier and a sequence number. Rounds are communication-closed, meaning that all messages broadcast in a round are received by all processes within that same round. However, since the underlying network is unreliable and asynchronous, messages may be lost or arrive after the end of the round in which they were broadcast. THyTOB guarantees progress only in synchronous rounds. A process is allowed to broadcast a message with sequence number $i+1$ only if it has received messages with sequence number i from all the other processes. If this is not the case, processes retransmit messages in order to resynchronize themselves. The total order is established by the sequence numbers and senders' identifiers. Messages are only delivered after becoming stable, which requires another synchronous round. The second round ensures that the system state can be restored in case of process failures by relying on consensus instances, which are also employed to reconfigure the system. THyTOB exploits the fact that processes and the underlying network behave synchronously for reasonably long periods of time to build a trivial total order among the messages, while ensuring safety under asynchrony and in the presence of process failures.

2.6 Discussion

The four protocols compared represent interesting classes of design choices. The first class includes two implementations of (multi-)Paxos optimized for local-area networks. Ring Paxos was designed to maximize throughput. It uses a ring topology to organize the dissemination and ordering of requests. We wanted to assess whether its ring design has implications on its latency distribution. S-Paxos is also a protocol designed for maximizing throughput. Like Ring Paxos, S-Paxos separates request dissemination from ordering but uses a classic multicast communication strategy. S-Paxos implements a fairly sophisticated request batching mechanism that can in theory contribute to keep latency tails short. Both protocols are implemented based on reliable unicast communication (TCP/IP).

Spread, the most mature of the systems tested, represents a class of protocols based on a ring where a control token circulates. The possession of the token dictates the broadcast, retransmission, and delivery of messages. It contains a careful congestion control mechanism, which may contribute to curb latency variations. The third class of protocols is based on the notion of synchronous rounds and represented by THyTOB. Its key design assumption regards network latency predictability (i.e., synchrony) that allows round-based broadcasts to be built and their use allow a very fine control of the load applied to the network. We hypothesize that well-behaved latency distributions should result from THyTOB's capacity to condition network load. Both Spread and THyTOB implementations are based on unreliable broadcast communication (UDP/IP).

3 Experimental methodology

This section details the key experimental aspects of our study: the metrics used to report results, the workloads and workload generator, and the computational environment.

3.1 Metrics and reported results

The metric used throughout the experiments is the latency (response time) experienced by requests submitted to the atomic broadcast protocol under test. Latency variability, the result of interest, is reported using the cumulative distribution function (CDF) of the latencies measured. The CDF describes the probability (shown in the y-axis in the graphs) that a random variable, in our case the latency, has a value less than or equal to a certain value (shown in the x-axis in the graphs, in milliseconds). For example, coordinate $(x = 10, y = 0.99)$ means that with probability 0.99 (99%) latency is less than or equal to 10 milliseconds. We report probabilities and latencies in linear-linear and in log-log scales. A probability of 0.99, for example, corresponds to the distribution's 99th percentile.

3.2 Workload generator

To conduct our experiments, we have built a simple workload generator inspired by the software architecture of S-Paxos [4], depicted in Figure 1. The workload generator has two components: a client node and a replica node.

The client node can generate different workloads by varying the number of client threads. Each thread starts its execution by randomly selecting and connecting to a replica node. The replica node contacted by the client assigns a unique id to the client thread. The client thread id, together with a request sequence number, uniquely identify all requests submitted. Client threads execute in a closed loop: the thread submits a request and waits for its response before it submits another request. If a request times out, the client tries to connect to other replicas, in a round-robin fashion; as soon as it is able to connect to a different replica, the client resends its last request.

The replica node has three components: a client proxy, a connector, and a service load emulator. The client proxy is responsible for managing connections with client threads, receiving requests from them, and sending responses back to them. The *client proxy* keeps the last response computed per client thread so that responses can be retransmitted when clients (re-)connect to a replica. The *connector* allows the connection of any of the atomic broadcast protocols to the service load emulator. Client requests are delivered to and executed by the *service load emulator*. A service load emulator can implement any function considered a legitimate load for the replicated service node. In our setup, it implements a null service, that is, a function whose only purpose is to return a fixed-size response for each request it processes. The service load emulator processes requests sequentially, in the order they are delivered to it, with corresponding responses being relayed back to the client proxy. Prior to the processing of a request, a service load emulator checks the message's sequence number to ensure a request is executed at most once.

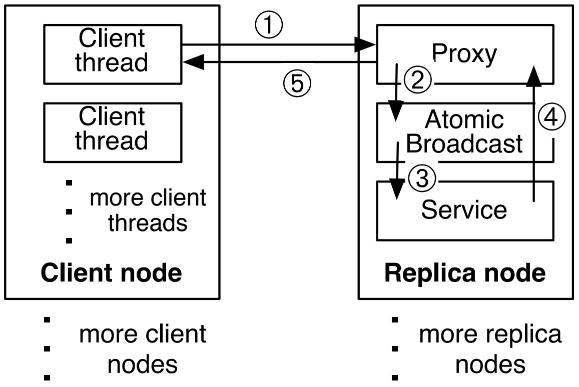

Figure 1: Architecture of a replicated service: (1) client submits a request to the client proxy of a replica, which (2) broadcast the request to all replicas; (3) upon delivering a client request, the service executes it, and (4) responds to the proxy, that acknowledges (5) the execution of the request to the client.

In order to submit all protocols tested to the same experimental conditions, all experiments use the same Java-based implementation of the workload generator. In the case of Spread, it is worth noting that the literature reports better performance results with a workload generator implemented in C [2]. The client proxy instantiates a pool of four threads. Each thread executes in an interruption-driven closed-loop that blocks in the absence of communication requests. The client proxies used by S-Paxos have been originally programmed so that sending events do not interrupt the thread if it has already entered its 10 millisecond sleep. As a consequence, communication between replicas and clients in S-Paxos occurs at regular intervals.

3.3 Workloads

We use two workloads to chase the tail distributions of atomic broadcast protocols, as described next. The workloads were generated using requests with three sizes: small (256 B), medium (2 KB) and large (32 KB).

The *peak workload* is the workload that takes the replicated service to 98–99% of its maximum throughput, for each of the protocols tested. We have determined the peak workload by observing when workload increments resulted in minor throughput variation but significant increase in response time, indicating that the atomic protocol under test had reached its saturation point, that is, any increment in the number of threads used to generate the workload did not result in a significant increase in the throughput.

The *operational workload* is the workload under which the replicated service provides around 70% of the peak throughput, for each protocol tested. Since it was not always possible to find a number of clients that matches this percentage and ensures a fair distribution of clients per replica node, we report in the graphs the exact percentages of peak throughput obtained. The operational workload is motivated by the fact that production systems provision resources to operate within certain safety margins in order not to risk major response time fluctuations in the presence of workload peaks.

3.4 Computational environment

All experiments were executed on a cluster with HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of main memory. Nodes were connected to a 1 Gbps HP ProCurve 2910al-48G switch, with round-trip time of approximately 0.1 ms.

The replicated service was implemented by five processes, each one hosted in a different node. Experiments lasted 200 seconds. Requests submitted in the first 20 seconds were disregarded when computing the latency distributions, which then refer to 3 minutes of execution. The setups for the protocols, insofar as possible kept unchanged, were the following.

We used the Java version of Ring Paxos, based entirely on unicast communication (TCP/IP) [3]. The five nodes instantiated implemented all the three Paxos roles: proposer, learner, and acceptor. The quorum size was set to 3 acceptors. Proposers implemented a batching procedure in which a batch of client requests was broadcast when either: (i) its size reached 32 KB, or (ii) no new request was received within 0.5 ms.

For S-Paxos, we adopted the settings presented in [4] for experiments with the best performance. The batch sizes for dissemination and ordering layers were set to 8700 and 50 bytes; the maximum batch delays to 5 and 10 ms. Batches were broadcast when either their size exceeded the maximum size, or the corresponding maximum delays expired. Parallelism was limited to 30 pending consensus instances at a time.

Five Spread daemons were instantiated, belonging to the same Spread segment. Each replica connected to the daemon hosted in the same node, and all replicas joined the same group. The service type for the messages broadcast was set to safe, as it ensures that messages delivered by a daemon have been received by all daemons. Based on [2], sizes of accelerated, personal and global windows were set to 15, 20, and 160.

For THyTOB, the maximum batch size was set to 32 KB and the round duration to 2.6 ms. Processes broadcast a batch with up to the maximum size at the beginning of each round. With five processes, this lead to an "optimal" (target) throughput of about 500 Mbps, which on average is the best reported in [5], for various maximum batch sizes.

4 Experimental evaluation

In this section, we present and analyze the results of our experiments with the peak and the operational workloads, and discuss our findings from the perspective of the different protocol designs.

4.1 Peak workload

Table 1 summarizes the performance of the four protocols when subject to the peak workload. The latency distributions at peak workload are depicted in Figure 2. We present results for this workload using CDFs with linear-linear (left) and log-log (right) scales. CDFs with log-log scales emphasize the latency tail, which in many cases cannot be easily distinguished in the linear-linear CDFs.

For medium and large request sizes, all protocols reached high throughput, with Ring Paxos and S-Paxos sporting performance closer to the network capacity (1 Gbps). Spread and THyTOB presented lower and also less dispersed latencies, with mean latencies always below 10 ms. For small requests, we observed more variation among the protocols.

Workload	*Request size*	*Latency (mean ± stdev)*	*Throughput*
		Ring Paxos	
40 clients	32 KB	11.8 ± 4.0 ms	887 Mbps
160 clients	2 KB	11.6 ± 18.8 ms	568 Mbps
1280 clients	256 B	13.1 ± 33.3 ms	198 Mbps
		S-Paxos	
75 clients	32 KB	23.8 ± 5.4 ms	825 Mbps
600 clients	2 KB	13.3 ± 2.5 ms	736 Mbps
7680 clients	256 B	27.0 ± 6.0 ms	578 Mbps
		Spread	
20 clients	32 KB	8.2 ± 1.7 ms	643 Mbps
100 clients	2 KB	4.8 ± 0.7 ms	345 Mbps
400 clients	256 B	9.2 ± 1.4 ms	89 Mbps
		THyTOB	
15 clients	32 KB	7.8 ± 0.6 ms	502 Mbps
240 clients	2 KB	7.8 ± 0.5 ms	503 Mbps
1800 clients	256 B	8.0 ± 1.2 ms	461 Mbps

Table 1: Performance with peak workload.

Ring Paxos. Although not very noticeable on the overall latency distribution (CDF at the top left of Figure 2), Ring Paxos has the longest tails among the tested protocols, with tails becoming longer as the request sizes get smaller. For example, from the 90th-percentile to the 99th-percentile (CDF at the top right of Figure 2, y-axis at 0.9 and 0.99) latency increases by more than 8 times with medium (2 KB) requests and by more than 13 times with small (256 B) requests. Large (32 KB) requests experience an increase of 67% from the 90th-percentile to the 99th-percentile. Longer tails imply, on average, latencies larger than the medians (around 8.0 ms for medium and small requests), and larger standard

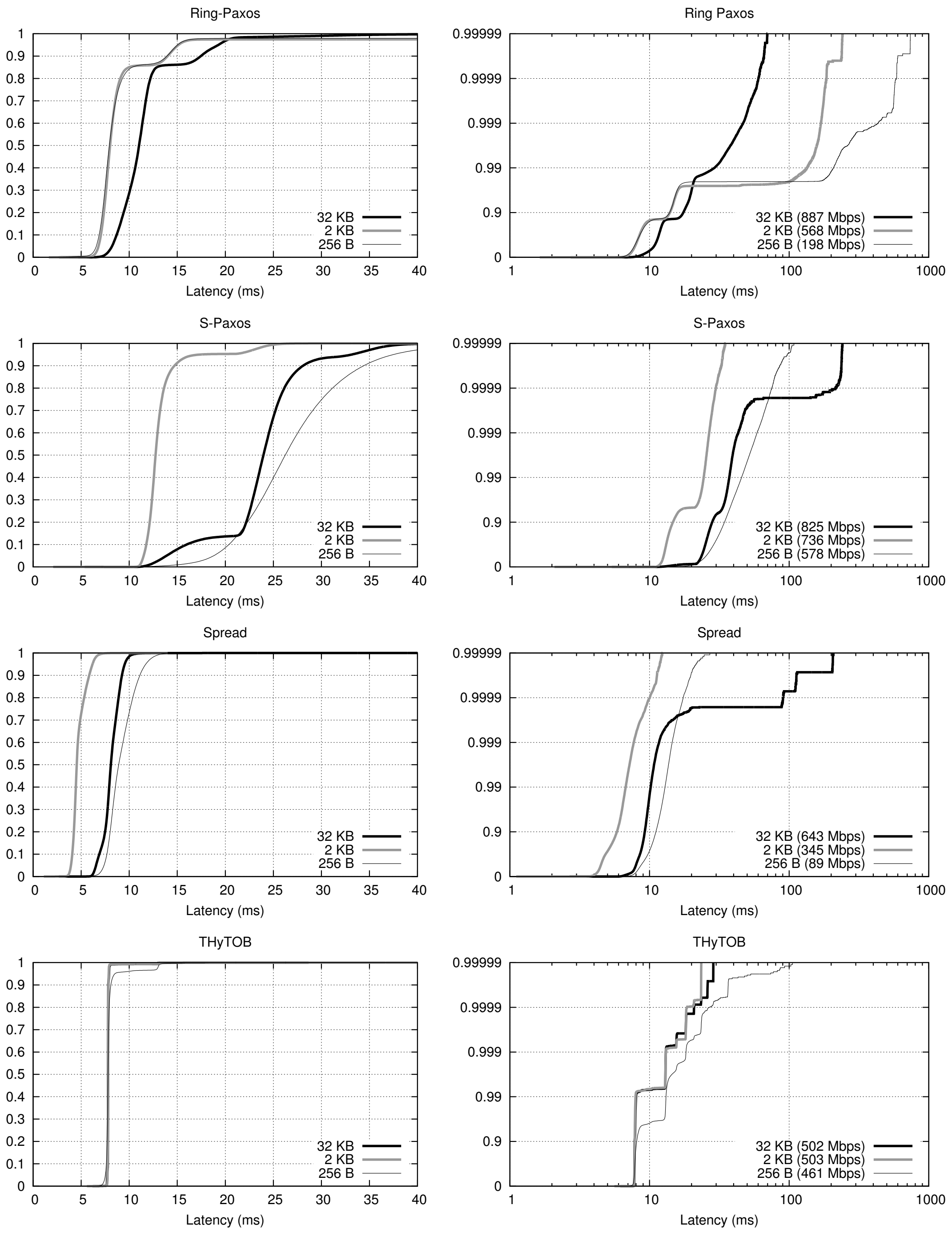

Figure 2: Overall latency distributions (linear-linear CDFs, on the left) and the corresponding latency tail distributions (log-log CDFs, on the right) for Ring Paxos, S-Paxos, Spread and THyTOB under peak workload.

deviations.

S-Paxos. S-Paxos presents the higher latencies among the four protocols, and distinct tail distributions for each request size. For large requests, 99% of the latencies are from 12 and 37 ms, while more concentrated around the average, which coincided with the median. The 99.9th-percentile is about 80% bigger than the average latency, and a long tail is observed around the 99.99th-percentile, where latencies exceeded 215 ms. For medium requests, a better behavior is observed, with latencies much more concentrated: 99.99th-percentile around 30 ms, while the average was 13.3 ms. Finally, latencies for small requests are the highest and most scattered: average at 27.0 ms and 99.99th-percentile about three times higher, at 75 ms.

Spread. The shape of the tail distribution curves for Spread at peak workload resemble those for S-Paxos, although Spread presents significantly lower and less dispersed latency values. The largest tail is observed for large requests, although latency scatters only at high percentiles. For example, while from the 99.9th to the 99.99th-percentile latency increases by almost 7 times, from the 99th to the 99.9th-percentile it increases by only 15%. Both small and medium requests have smaller latency tails than large requests, with medium requests presenting the lowest average latency and standard deviation. It is worth noting, however, that in our experiments Spread saturated at considerably low loads for smaller requests. As a consequence, the throughput Spread reached for medium requests was 46% lower than for large requests; and for small requests it was 7 times lower, reaching only 89 Mbps.

THyTOB. THyTOB presented the most predictable and stable behavior among the protocols assessed, with little performance variation with different request sizes. To understand the latency distributions of THyTOB, recall that it operates in synchronous rounds, whose duration was set to 2.6 ms. Since in the best case requests needed three rounds to be delivered (two rounds for ordering and one for client-replica interaction), we can expect a lower bound of about 7.8 ms. When a round fails, due to message loss or timing faults, latencies of all ongoing requests are increased by an additional round. For medium and large requests, about 99.3% of requests fitted THyTOB's best case, and were delivered in three rounds. For small requests, 96% of requests fell in this category. We can also observe that tails become significant only at high percentiles.

4.2 Operational workload

We now observe the behavior of Ring-Paxos, S-Paxos, Spread, and THyTOB at the operational workload, i.e., at 70% of peak workload. We have selected this workload to assess whether the latency tails observed at peak workload are the result of the saturation of the servers, or are intrinsic to the protocols. The tail distributions (CDFs) are presented in Figure 3, while the overall performance and the workload for these experiments are summarized in Table 4.2.

Ring Paxos. We note that the shape of the tail distributions at peak and operational workload are similar. The main difference, in particular for medium and small requests,

Workload	*Request size*	*Latency (mean ± stdev)*	*Throughput*
		Ring Paxos	
10 clients	32 KB	4.2 ± 1.7 ms	619 Mbps
120 clients	2 KB	4.9 ± 10.5 ms	400 Mbps
420 clients	256 B	6.1 ± 19.4 ms	140 Mbps
		S-Paxos	
25 clients	32 KB	11.4 ± 1.5 ms	576 Mbps
360 clients	2 KB	11.2 ± 0.5 ms	526 Mbps
3200 clients	256 B	15.3 ± 2.1 ms	428 Mbps
		Spread	
10 clients	32 KB	5.4 ± 0.5 ms	483 Mbps
30 clients	2 KB	2.0 ± 0.3 ms	241 Mbps
80 clients	256 B	2.6 ± 0.4 ms	64 Mbps
		THyTOB	
10 clients	32 KB	7.8 ± 0.3 ms	336 Mbps
170 clients	2 KB	7.8 ± 0.3 ms	357 Mbps
1280 clients	256 B	7.8 ± 0.8 ms	335 Mbps

Table 2: Performance with operational workload.

is that the tails were displaced to higher percentiles: while under peak workload they affected about 3% of requests, under operational workload this percentage dropped to 0.75%. Further reducing the load leads to similar tails, although affecting even smaller portions of requests. This suggests that latency tails are intrinsic to the protocol.

S-Paxos. The most remarkable change when the load is reduced to 70% is the better behavior for large requests. The tail observed in peak performance is no longer noticeable, indicating that it is not inherent to the protocol, but due to queuing effects at high load. In addition, there is a “step” around the 98th-percentile, with a leap from about 13 ms to beyond 20 ms. This behavior results from a peculiarity of the protocol. As pointed out in Section 3, the client proxies of S-Paxos were originally programmed to communicate with clients periodically, every 10 ms. S-Paxos presumably assumes that most requests will be ordered within this period, which results in latencies starting at 10 ms, independently of the load. Moreover, requests for which this assumption fails, about 2% in this case, will only be replied in the next iteration of the respective client proxy threads; their latencies will then start at 20 ms. This same effect can also be observed for medium requests under peak workload (Figure 2, left side).

Spread. The most well-behaved latency distributions among those presented in Figure 3 belong to Spread. Latencies for the three request sizes did not exceed 14 ms, while the

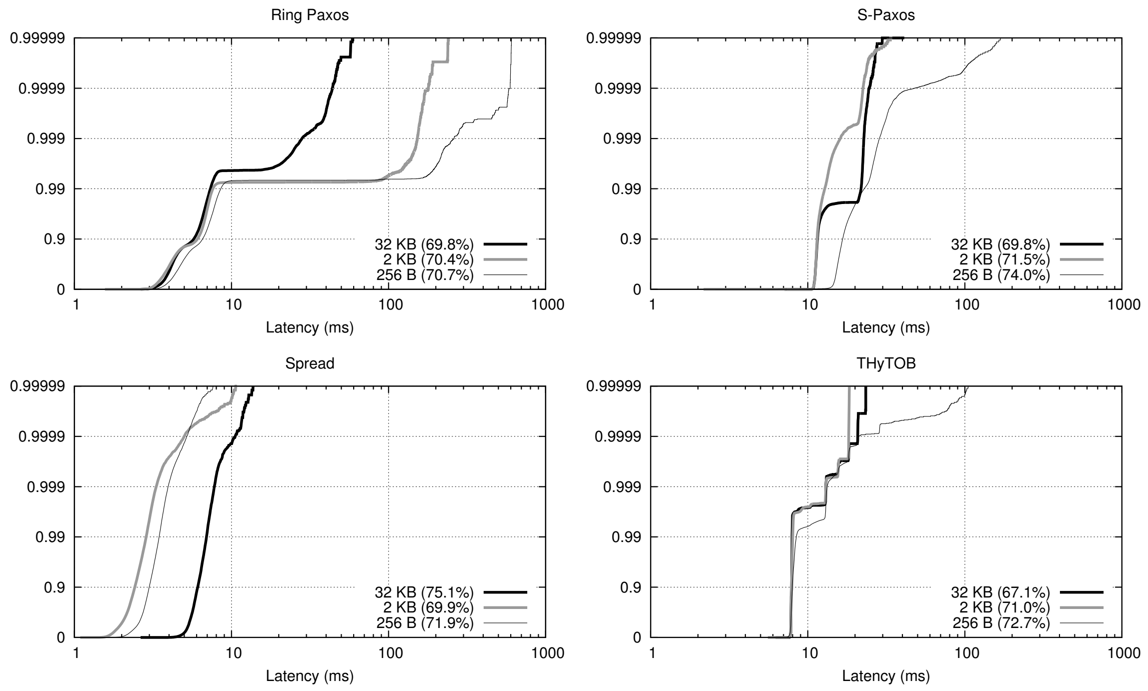

Figure 3: Tail distributions for Ring Paxos, S-Paxos, Spread and THyTOB at operational workload; actual percentages of peak throughput shown in parentheses.

99.99th-percentiles were 10.6 ms, 5.0 ms, and 5.2 ms for large, medium and small requests, respectively. We repute this behavior to the sophisticated flow control mechanisms adopted by Spread. They ensure a strict control of the load applied to the network, enabling the broadcast of requests to be completed, with high probability, within stipulated deadlines. However, it is worth noting that the workload applied, specially for small requests, is much lower than for the remaining protocols.

THyTOB. As for Ring Paxos, there is not significative change in the shape of the latency distribution when the load is reduced to 70%. There is an increase in the portion of requests that fit in the best case of the protocol, with latencies of three rounds: from 96% to about 99.4% for small requests, and from 99.3% to 99.8% for medium and large requests. For the latter two, in particular, latencies did not exceed nine rounds (23.4 ms). However, there is still a tail at very high percentiles for small requests. Apparently, the processing overhead induced by such small requests causes, in very few occasions, a long sequence of consecutive rounds to fail, affecting all ongoing requests.

4.3 Discussion

We could identify two aspects in the design of Ring Paxos that may explain the long tails observed for the protocol. First, there is no flow control in the proposers. A proposer can circulate requests in the ring at the same rate they are received from the clients. Without any flow control, proposers may overcharge their outgoing links, as they are responsible for circulating requests and protocol control messages along the ring, leading to an increase in latency variability. Second, the ring topology adopted by the protocol tightly couples the communication behavior of the processes. In effect, there is a single stream of messages circulating the ring, implying that every message has to be received, processed and forwarded by every process in the ring. When a process blocks because of background activities (i.e., garbage collection), it neither broadcasts new requests nor forwards messages received from its predecessor in the ring. Consequently, irregular delays experienced by a process are potentially propagated throughout the ring, affecting (i.e., delaying the deliver of) ongoing requests.

To verify these two hypotheses, we profiled the proposers of Ring Paxos and measured their request sending rates and delays. With increasing sending rates, average delays increase, reaching about 0.12 ms for large, 0.65 ms for medium, and 1.7 ms for small requests. The higher delays for smaller messages are due to the batching procedure, which requires more smaller messages to fill a batch. When analyzing their distributions, we could observe some abnormal sending delays, exceeding 40 ms for large, 150 ms for medium, and 500 ms for small requests. We were able to associate most of these outliers to garbage collection events, which were more frequent and had longer durations than those observed for smaller requests.

The abnormal delays measured in the proposers necessarily affect the overall request latencies, and in part explain the long tails observed for Ring Paxos. However, only these outliers alone cannot account for the magnitude and the frequency of the latency peaks measured by the clients. This lead us to suspect that other effects contribute to Ring Paxos's latency variability. In fact, when analyzing the peaks of latency as measured by the clients of different replicas, we observed that they were often correlated to each other in terms of the moment in which their executions occurred in the replicas (ring). This indicates that the occurrence of an abnormal delay at one process of the ring is actually propagated throughout the ring, a shortcoming that seems intrinsic to the topology.

The same behavior was not observed for S-Paxos, despite some similar design principles it shares with Ring Paxos: both protocols separate request dissemination from ordering to mitigate the bottleneck in the coordinator, and use TCP unicast to achieve throughput close to the network capacity. S-Paxos's behavior can be in part explained by the adoption of several flow control mechanisms, including a conservative batching policy, and limitation on the number of concurrent broadcast instances at both dissemination and ordering layers. But the main reason is probably the adoption of a fully connected communication graph topology plus majority-based mechanisms for the dissemination and ordering of requests. Since with this approach processes tend to be loosely coupled, the blocking of a single process is less likely to affect requests that were broadcast by other processes.

Another interesting aspect of S-Paxos's design is the effect of its intricate batching mechanisms. On the one hand, for medium requests they proved quite efficient in conditioning latency. Under both peak and operational performance, the 99.99th-percentiles for medium requests were approximately twice the average latencies. In particular, the mechanisms were effective in amortizing the load applied by replicas to the network. Thus, despite a throughput only 10.8% lower than that for large requests (which are not subject to batching at the dissemination layer), there was no significant effect of network contention under peak workload. On the other hand, latencies for small requests were considerably dominated by queuing delays: they were significantly higher and quite dispersed. The probable reason is the larger number of requests required to fill a batch. As a result, batches were more likely to be sent by timeout than by size. The latency variability was then caused by contention and can be seen as an overhead of batching.

Spread needs to establish a reliable communication channel among a group of processes atop an unreliable datagram service (UDP). For this end, the protocol relies on a complex flow control mechanism, orchestrated by a token that circulates a logical ring of processes. To improve performance, requests are disseminated using ip-multicast, an efficient broadcast primitive in local-area networks, but prone to message loss [14]. By restricting the number of concurrent broadcasts, Spread manages to minimize the shortcomings of this primitive. In fact, when monitoring the protocol, we noted that the message loss and retransmissions rates were negligible. This explains the good behavior of its latency distributions.

Another interesting design decision of Spread, not present in the other protocols, is to use a separate process, the Spread daemon, to handle the inter-process communication. Once the replica executes in a different process, the possible overheads of handling clients and executing requests are less likely to be propagated to the protocol, which contributes to its stability. However, the interaction between replicas and local daemons, accomplished through a TCP connection, may have curbed the performance of Spread in our experiments, specially with small requests. Thus, if on the one hand separating the protocol implementation from the replica execution can be useful in controlling latencies, on the other hand the communication between replica and daemon may limit performance.

THyTOB's key design feature is to emulate a synchronous communication service atop an asynchronous (and unreliable) broadcast network. It organizes the execution in synchronous rounds, by means of which it controls the load applied to the network. In fact, the duration of the rounds, as a function of the number of processes and maximum batch size, determines the protocol's maximum throughput and minimum latency. The longer the rounds, the lower the throughput, the higher the latency, and the lower the probability of a round to fail because of message loss or timing faults. This trade-off between achievable performance and latency predictability is at the core of the synchronous design of THyTOB. When considering that more than 99% of latencies, in particular for large and medium requests, fitted the protocol's best case, it is possible to affirm that this strategy can lead to quite controlled latencies, while providing good throughput.

5 Lessons learned

A remarkable aspect of the results of our experiments is the significative performance degradation—in terms of latency distribution and throughput—of the four protocols with small requests. Although affecting all protocols, two aspects allow us to distinguish two groups of protocols, in terms of performance with small requests. First, Ring Paxos and Spread presented a noticeable throughput degradation as requests got smaller, an effect that was not observed in the same proportion for S-Paxos and THyTOB. For instance, from large to medium requests, the peak throughput dropped by 46% for Spread, and by 36% for Ring Paxos; for S-Paxos it dropped by only 11%, and it remained fairly stable, almost constant, for THyTOB. Second, while the latency distribution of Ring Paxos became more scattered for medium requests, for this request size (2 KB) both S-Paxos and THyTOB presented the least latency variations.

The distinct behavior of these two groups of protocols can be explained by their *batching policies*. S-Paxos and THyTOB are protocols designed from scratch to operate on batches of requests, instead of single requests. Processes of both protocols wait for client requests and aggregate them into batches, which are then disseminated and ordered. Upon the delivery of a batch, the requests are retrieved, and then processed by the replicas. Ring Paxos and Spread do not expressly adopt batching of requests; instead, they consider alternative strategies to handle small requests. Ring Paxos is optimized for large requests; proposers try to aggregate small requests in order to send them together, but this is a "network batching", since requests are individually processed by all participants. Spread considers the size of pending requests to compute the message windows, which use "packets" as unit. Large requests may be split in several packets, while small requests can be aggregated into a single packet. Similarly, all requests are processed one by one, not in batches. In our experiments, these alternative strategies have proven ineffective to minimize the performance loss with small requests. This led us to consider the adoption of a batching policy at the replicas.

We conducted a second set of experiments to assess the performance of Ring Paxos and Spread with batched requests. To this end, we implemented a simple routine in the replicas, responsible for building batches of requests. Replicas broadcast a batch, composed by requests individually received from the clients, when either its size reaches 32 KB or the batching delay exceeds a maximum duration δ. The value of δ was computed so that if all replicas broadcast a full batch every δ units of time, the aggregate broadcast rate is equivalent to the maximum throughput of the protocol (a similar method is used in THyTOB to select the duration of rounds). For Ring Paxos, δ was set to 1.5 ms and for Spread it was set to 2.0 ms. The tail distributions for these protocols with batched requests, at peak and operational workload, are depicted in Figure 4, and the protocols performance is summarized in Table 3.

The best improvement in the overall performance when replicas batch client requests was observed in Ring Paxos. While in the original setup the latency distributions at peak throughput for the three request sizes were comparable up to around the 97th-percentile, with batched requests they are quite similar up to the 99.95th-percentile. In addition, for medium requests the peak throughput was 874 Mbps, 54% higher than in the original setup, and very close to the one achieved with large requests. Throughput also increased for small requests, by 2.6 times, while the average latency decreased by 2.5 ms. Since the performance for medium and small requests became more similar to the achieved for large requests, we can affirm that batching at replicas is more efficient to cope with smaller requests than the batching implemented at the proposers.

The reasons for this improvement were found by profiling the batching routine and, as done for the proposers, measuring the sending rates and delays. While the sending rates for small and medium requests increased, the average delays remained almost unchanged—1.6 ms and 0.6 ms, respectively—from the original experiments. However, abnormal sending

Workload	*Request size*	*Latency (mean ± stdev)*	*Throughput*
Ring Paxos with batched requests			
Peak workload			
40 clients	32 KB	11.9 ± 4.1 ms	883 Mbps
640 clients	2 KB	12.0 ± 4.0 ms	874 Mbps
2720 clients	256 B	10.6 ± 6.6 ms	521 Mbps
Operational workload			
15 clients	32 KB	5.6 ± 2.2 ms	707 Mbps
160 clients	2 KB	4.3 ± 1.8 ms	611 Mbps
960 clients	256 B	5.5 ± 2.0 ms	356 Mbps
Spread with batched requests			
Peak workload			
25 clients	32 KB	12.1 ± 0.9 ms	542 Mbps
320 clients	2 KB	10.3 ± 0.8 ms	509 Mbps
2560 clients	256 B	13.8 ± 1.9 ms	377 Mbps
Operational workload			
10 clients	32 KB	5.8 ± 0.6 ms	451 Mbps
100 clients	2 KB	4.7 ± 0.8 ms	349 Mbps
960 clients	256 B	7.4 ± 1.2 ms	264 Mbps

Table 3: Performance with batched requests.

delays, which in the original setup exceeded 150 ms for medium, and 500 ms for small requests, very few times exceeded 40 ms for all request sizes. The explanation for this difference is related to the number of requests a proposer has to handle, and the overhead of temporary information it stores for each of them. When batching is done at replicas, a single metadata instance is allocated for each batch of requests, instead of one instance for each client request in the original setup. Considering that metadata instances are sent together with the requests, and are potentially stored by all processes, this makes a big difference, specially for small requests. In fact, the time spent in garbage collection events at peak throughput for single small requests is up to 14 times higher than for batched small requests.

As a consequence, we also observed with batched requests a reduction on the range within which latencies are distributed, that is, on the length of the latency tail. While in the original setup of Ring Paxos at peak throughput the 99.9th-percentiles for medium and small requests were about 180 and 260 ms, with batched requests they were both around 30 ms. It is also true that tails at high percentiles are still observed, mainly with small requests. This possibly explains why the peak throughput for such requests is 40% lower than for the larger ones.

For Spread, the impact of batching at replicas was mostly on throughput for smaller

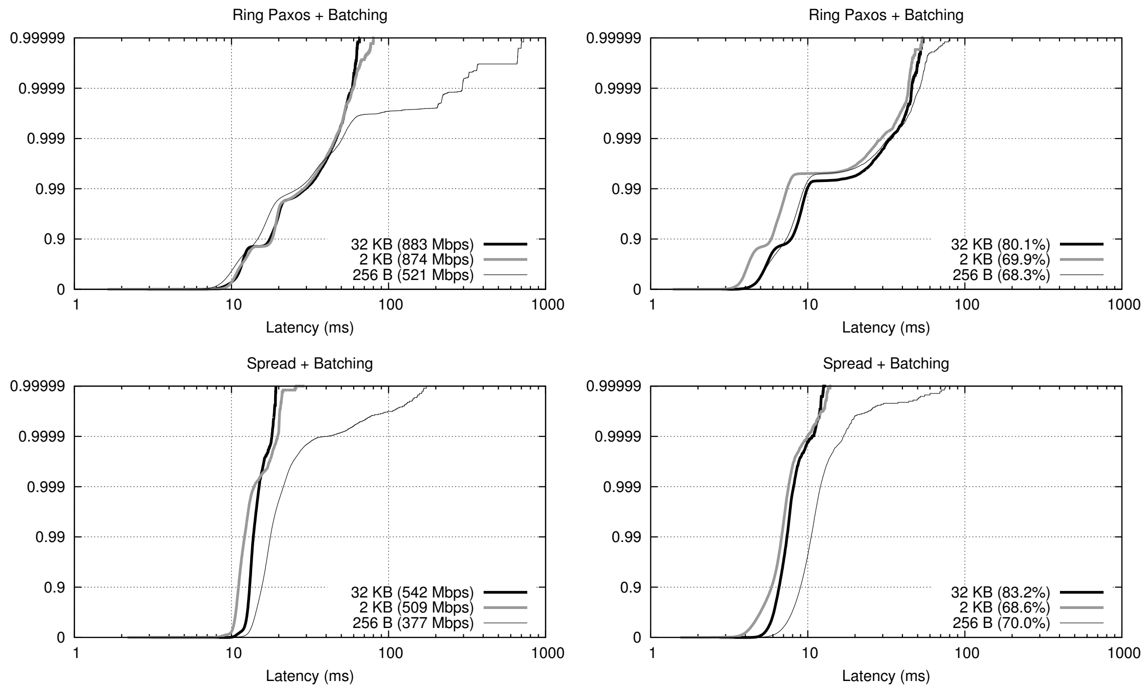

Figure 4: Tail distributions for Ring Paxos and Spread with batched requests under peak workload (on left) and at operational workload (on right).

requests. Compared to the original setup, throughput was 4.2 times higher for small, and 32% higher for medium requests. The increase in throughput was accompanied by an increase in average latency, by 4.6 ms and 5.5 ms, respectively. Latencies also increased by 3.9 ms for large requests, which caused the corresponding throughput to drop by 16%. This was the consequence of requests being broadcast only by the batching routine (a single thread) rather than by the client proxies (four threads), as done in the original setup. The increase in latency reveals how costly, in terms of delay, calls to the broadcast method provided by Spread, responsible for transferring requests to the daemon, are.

Spread saturated at different loads with batched requests, and this had some impact on the latency distributions. When taking into account the increase in the average latencies, the distributions for small and medium requests were more dispersed for batched than they were for single requests. In particular, for small requests long tails were observed from the 99.99th-percentile. It is worth noting, however, that we are comparing here latencies from experiments that achieved different peak throughputs, and were subjected to different workloads: 400 clients with single requests versus 2560 clients with batched requests. The opposite behavior is observed for large requests, for which throughput was lower with batched requests. The latency distribution became more concentrated, and almost no tail was noticed, even at peak throughput. Thus, Spread is subject to a tradeoff involving the load it can sustain and the resulting latency distribution. The more the protocol amortizes the load applied, the better controlled latencies are, but also the lower the throughput is.

Thus, mechanisms that attenuate the protocol contention, as batching does, can indeed improve throughput, at the cost of less predictable latencies.

6 Related Work

The increased attention to the latency tail by companies such as Google and Amazon [6, 7] indicates the importance of end-to-end latency guarantees to distributed applications. This has raised interest in the scientific community, and several works have considered the problem (e.g., [20, 22, 18, 19]).

When a request enters a system it passes through many layers and stages, both locally on a single machine, and globally across machines. Moreover, current distributed applications are complex, composed of several services, each one operating independently. The latency a request experiences depends on each of these stages, layers, and services. Therefore, it is important that each component reduces the delays it imposes on the final response time of a request. A recent work [13] considers the effects of different layers on a request's lifetime in an individual machine, and makes suggestions to mitigate the latency tail, e.g., controlling background processes, scheduling tasks to specific threads, and using core-affinity.

In the context of atomic broadcast protocols, most early and recent work focused on maximizing throughput and minimizing latency, but not on latency variability. When the response time has been of interest, its average over periods of system operation has been considered, rather than its stability and equality across requests. An exception is Totem, Spread's atomic broadcast protocol, that had its latency distribution studied in some works [16, 17, 21]. The latency probability functions for Totem were determined analytically by [16], analysis that was experimentally verified in a subsequent work [17]. Finally, the end-to-end latency of remote methods invoked in a replicated CORBA server built atop Totem was analyzed in [21]. Our findings are consistent with the analysis presented in these studies: Spread provides controlled latency, as a result of carefully designed flow control mechanisms.

This paper expands on these previous studies as we consider the latency distribution of four different atomic broadcast protocols, including Spread's recent revamped implementation, based on the Accelerated Ring Protocol [2]. Moreover, we identify bottlenecks and propose ways to improve performance and reduce the latency tail of existing protocols.

7 Final remarks

Given the importance of atomic broadcast protocols to replicated services, this work offers a novel analysis of their behavior, focused on latency tail distribution instead of average latency or throughput. The results presented shed light on the design tradeoffs—topology, communication, role of synchrony, batching—made by four representative protocols to achieve a fair balance between throughput and the maintenance of short-tailed, predictable, response latency distributions. Among the design principles assessed, we have identified batching as having a key role to curb latency variability. While it is well-known that batching can boost throughput [10], it is remarkable that in some cases (i.e., Ring Paxos) it can also help shorten the latency tail of atomic broadcast.

References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM TOCS*, 13(4):311–342, 1995.

[2] Amy Babay. The accelerated ring protocol: Ordered multicast for modern data centers. Master's thesis, The Johns Hopkins University, 2014.

[3] Samuel Benz. Unicast Multi-Ring Paxos. Master's thesis, University of Lugano, 2013.

[4] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *SRDS*, 2012.

[5] Daniel Cason and Luiz E. Buzato. Time hybrid total order broadcast: Exploiting the inherent synchrony of broadcast networks. *JPDC*, 77:26–40, March 2015.

[6] Jeffrey Dean and Luiz André Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[8] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM CS*, 36(4):372–421, 2004.

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

[10] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *HPDC*, 1997.

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

[12] Leslie Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, 1998.

[13] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, OS, and Application-level sources of tail latency. In *SoCC*, 2014.

[14] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *DSN*, 2010.

[15] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM CS*, 22(4):299–319, 1990.

[16] E. Thomopoulos, L.E. Moser, and P.M. Melliar-Smith. Analyzing the latency of the Totem multicast protocols. In *ICCCN*, 1997.

[17] E. Thomopoulos, L.E. Moser, and P.M. Melliar-Smith. Latency analysis of the Totem single-ring protocol. *ACM NET*, 9(5):669–680, 2001.

[18] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level SLOs on shared storage systems. In *SoCC*, 2012.

[19] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien-An Lai, Chien-An Cho, Yuji Nomura, and Calton Pu. Lightning in the cloud: A study of very short bottlenecks on n-tier web application performance. In *TRIOS*, 2014.

[20] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *NSDI*, 2013.

[21] W. Zhao, L.E. Moser, and P.M. Melliar-Smith. End-to-end latency of a fault-tolerant CORBA infrastructure. In *ISORC*, 2002.

[22] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. PriorityMeister: Tail latency QoS for shared networked storage. In *SoCC*, 2014.