S-Paxos: Offloading the Leader for High Throughput State Machine Replication

Martin Biely, Zarko Milosevic, Nuno Santos, André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Email: firstname.lastname@epfl.ch

***Abstract*—Implementations of state machine replication are prevalently using variants of Paxos or other leader-based protocols. Typically these protocols are also leader-centric, in the sense that the leader performs more work than the non-leader replicas. Such protocols scale poorly, because as the number of replicas or the load on the system increases, the leader replica quickly reaches the limits of one of its resources. In this paper we show that much of the work performed by the leader in a leader-centric protocol can in fact be evenly distributed among all the replicas, thereby leaving the leader only with minimal additional workload. This is done (i) by distributing the work of handling client communication among all replicas, (ii) by disseminating client requests among replicas in a distributed fashion, and (iii) by executing the ordering protocol on ids. We derive a variant of Paxos incorporating these ideas. Compared to leader-centric protocols, our protocol not only achieves significantly higher throughput for any given number of replicas, but also increases its throughput with the number of replicas.**

Keywords: Paxos, High Throughput, Scalability, State Machine Replication, Performance

I. Introduction

As online services increase in importance, their availability becomes a critical feature. One general approach for providing a highly available service is state machine replication (SMR) [1], [2]. By replicating a service on multiple servers, clients are guaranteed that even if some replica fails, the service is still available. The state machine replication approach has been widely considered by both the theoretical [3], [4], [5] and systems research community [6], [7], and is also used in several real-world systems [8], [9], [10].

If executing requests is more costly than ordering them, then the system performance is limited by the performance of the service being replicated. However, online services are frequently lightweight, like the Chubby lock service [8] or the Zookeeper coordination service [9] and can sustain very high throughput provided that the ordering protocol can keep up. The authors of Zookeeper report in [11] that *write throughput* of Zookeeper is limited by the throughput of the underlying ordering protocol.

We observe that implementations of state machine replication are prevalently using variants of Paxos [3] or other leader-based protocols. Typically these protocols are also leader-centric, in the sense that most of the work is done by the leader. Therefore the bottleneck is found at the leader and the maximum throughput is limited by the leader's resources (such as CPU and network bandwidth), although there are still available resources at other replicas.

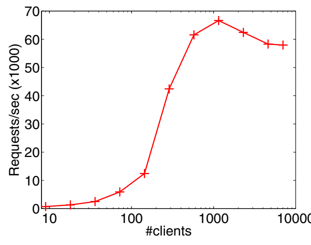
We illustrate our observation with JPaxos [12], an efficient multi-threaded implementation of Paxos in Java. Figure 1a shows the throughput of JPaxos (cf. Section III for details of the experimental setup) for different number of clients. The maximum throughput is achieved for 1000 clients. At this point, the leader's CPU becomes the bottleneck (cf. Figure 1b) and further increasing the number of clients results in a decrease of the throughput. The other replicas, however, are only lightly loaded. Since the bottleneck is at the leader, introducing additional replicas will not improve performance; in fact it will lead to a slight decrease in throughput since the leader will need to process additional messages.

In order to overcome this shortcoming of leader-centric protocols, we propose S-Paxos (S stands for scalable), a novel SMR protocol for clustered networks derived from Paxos. S-Paxos achieves high throughput by balancing the load among replicas to use otherwise idle resources, thereby avoiding the bottleneck at the leader. Furthermore, its throughput keeps increasing with the number of replicas (up to a reasonable number). This way, S-Paxos overcomes the traditional trade-off between fault tolerance and throughput present in most state machine replication protocols: in S-Paxos higher fault tolerance actually leads to higher throughput.

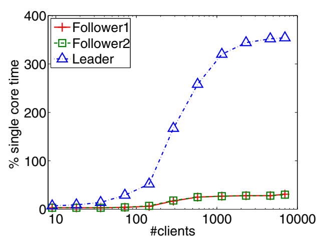
In order to derive S-Paxos, we start by observing that a replicated state machine performs the following tasks:

- *Request dissemination:* receiving requests from clients and distributing them to all replicas
- *Establishing order:* reaching agreement on the order of requests
- *Request execution:* executing requests in the determined order and sending replies to clients.

In leader-centric protocols such as Paxos, most responsibility for these three tasks rests with the leader, while the followers are left only with acknowledging the order proposed by the leader and executing requests. The S-Paxos key design guideline is to *distribute the three tasks evenly across replicas*. In order to do so, we turn the first two tasks into separate layers: dissemination and ordering. The *dissemination layer* is then balanced among all replicas, with all of them accepting and disseminating client requests. Since request dissemination is handled by the dissemination layer, the role of the *ordering layer* is only to determine the order in which requests will be executed. We use normal Paxos for this layer with a



(a) Throughput



(b) CPU usage

Fig. 1. Performance of a typical leader-centric Paxos. Request size 20 bytes, four core machines. See Section V for experimental settings.

difference that it orders request ids instead of full requests. Therefore, the additional overhead of being the leader in S-Paxos consists only in ordering ids. After *executing* the request, the replica that received the request from the client sends the corresponding reply.

We have implemented a prototype of S-Paxos[1] and performed a detailed experimental evaluation, which shows how our protocol circumvents the different bottlenecks at the leader that appear in leader-centric protocols like Paxos. As we will demonstrate, our protocol is able to reach unprecedented performance for an application-agnostic ordering service. For example, for typical replicated state machine deployment sizes (3-7 replicas according to [13]), S-Paxos achieves a throughput of 300'000 requests per second for 3 replicas, and keeps increasing up to 500'000 requests per second for 7 replicas.

Note that such high throughput comes at a price: While under high load higher throughput also means lower response time this is not the case under low load. Indeed, we observe a slight increase in latency for low to medium load when comparing S-Paxos to a basic Paxos implementation. However, we believe that given the increase in maximum throughput, the increase in latency is a price worth paying.

The remainder of the paper is organized as follows. Section II provides background on state machine replication and the Paxos algorithm. In Section III we identify bottlenecks in Paxos and how they limit its performance. We then introduce S-Paxos in Section IV. In Section V we evaluate the performance gains of S-Paxos over Paxos, and explain how S-Paxos circumvents the limitations of Paxos. Section VI summarizes related work and Section VII concludes the paper.

II. Background

Most services can be modeled as state-machines with state transitions being triggered by client requests. Such services can be made fault-tolerant through state machine replication (SMR) [1], [2]. In particular, the server is replaced by an ensemble of replicas starting in the same state, and executing client requests in the same order. For this to work it is necessary that processing of requests is deterministic such that the result of executing a request is the same at all replicas.

[1]The source code of S-Paxos is available at https://github.com/nfsantos/S-Paxos.

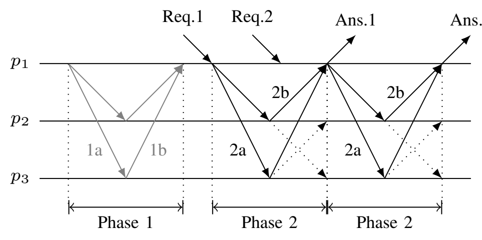


Fig. 2. Paxos message pattern

Paxos [3] is probably the most widely used state machine replication protocol [7], [10], [14]. Specifically, Paxos is a consensus protocol and MultiPaxos its extension to multiple consensus instances. As commonly done in the literature, we will refer to both protocols, as well as the resulting state machine replication protocol, as Paxos.

Paxos is designed for the partially synchronous system model and requires $n \geq 2f + 1$ replicas to tolerate f crash failures. Although Paxos is usually described in terms of the roles of proposer, acceptor and learner, this distinction is not relevant for the work in this paper; we assume that every process is at the same time proposer, acceptor and learner. Paxos can be seen as a *sequencer-based* protocol [15], where the sequencer orders requests received from the clients. In the context of Paxos, the sequencer is called *leader*. Moreover, we will refer to a replica which is not the leader as *follower*.

We now briefly describe the Paxos SMR protocol, focusing on the aspects relevant for the purpose of the paper. In our exposition we will focus on how a request is processed at different replicas in a typical Paxos based solution. Often, all communication with the clients is done by the leader, *i.e.*, the leader receives all requests and sends all replies[2]. The leader proposes a client request for a certain sequence number using the Paxos consensus protocol. The decision is a client request together with a sequence number, and requests are executed in the order of their sequence number.

The Paxos consensus protocol is structured in two phases, as shown in Figure 2. Phase 1 is executed by a newly elected leader as a preparation to order requests. Afterwards, the leader orders a series of client requests by executing several instances of Phase 2, with the leader proposing a request using a Phase 2a message and order being established once a majority of Phase 2b messages is received. We use the term *instance* as an abbreviation for *one instance of Phase 2*. The followers learn the decided sequence either (a) from the leader through a "decision" message (along with the next Phase 2 message), or (b) when receiving a majority of Phase 2b messages. In case (a) the followers send their Phase 2b message only to the leader, while in case (b) they must broadcast this message.

III. Limitations of Paxos

In order to design a balanced variant of Paxos, we start by looking in more detail at the traditional leader-centric variants

[2]Some implementations allow the clients to contact any replica which will then forward the request to the (current) leader [13], [16].

of Paxos, with the goal of identifying the limits this imposes. From the previous section is obvious that the Paxos protocol is heavily leader-centric, with the leader doing a disproportionate amount of work as compared to the followers. Therefore, the leader will be the first replica running out resources[3].

In the rest of the section we explain what system resources are critical and when they become a bottleneck. There are three types of resources that can cause a bottleneck. The first resource that we consider is network bandwidth, which is required to send and receive messages. The CPU is also a critical resource, as it is used to serialize and deserialize messages, to execute the replication protocol, and to run the service itself. A less obvious resource that can also become the bottleneck is the network subsystem (network interface card, network driver and operating system networking stack), which can only handle a certain number of packets per second.

Bandwidth: When the average request size is large enough, the outgoing channel of the leader saturates before the CPU or network subsystem limits are reached. The reason for this is that proposing an order for a request requires the leader to inform the other replicas of the request. Recall that this is done through the Phase 2a message. Therefore, the maximum throughput of Paxos is limited by $\frac{B}{n-1}$, where B is the available bandwidth on the leader's outgoing link (ignoring communication with clients). In this case, we say that the system is *bandwidth bound.*

Note that when a multicast primitive is available (for instance, IP Multicast), then the maximum throughput is roughly the leader's total outgoing bandwidth (e.g., [17]). However, support for multicast is not always available, and if available it may be prohibited from use. This is the case in many data-centers, as Vigfusson et al. point out [18]. As we are focusing on clusters in data-centers we do not consider multicast in the remainder of this paper.

CPU: When the average request size is small (contrasting the above case), the leader's CPU can become the bottleneck long before its outgoing link gets saturated [13], [17]. In this case, throughput is determined by the processing power of the leader and we say the system is *CPU bound*.

Network subsystem: Since the leader is responsible for communication with the clients (receiving requests and sending replies), the network subsystem can become the bottleneck before the leader saturates its bandwidth and CPU. More precisely, the network subsystem at the leader can reach its maximum capacity in terms of packets per second [19], [20]. In this case, we say the system is *network subsystem bound*.

IV. S-Paxos

In the previous section we have seen that in Paxos the bottleneck at the leader is mainly caused by the leader having to *receive client requests* and then *disseminate them* to all other replicas. The latter is done through the Phase 2a messages, which also serve another semantically very different purpose: they are used to *propose an order* in which requests should be executed. Proposing an order in a leader-centric fashion is a central idea of Paxos and S-Paxos retains this idea. Conversely, request dissemination (together with receiving requests from clients) is a task that can be performed by all replicas. For this reason S-Paxos separates these two roles into two different layers: the *dissemination layer* and the *ordering layer*.

By moving request dissemination into a separate layer, we are able to turn request dissemination into a distributed protocol that is more balanced: all replicas accept requests from clients and disseminate them directly to all other replicas. Moreover, to complete the separation of the two layers, the ordering layer only determines (using Paxos) the order of ids instead of requests.

Since S-Paxos is a variant of Paxos, we make the same assumptions about the system as Paxos (see Section II). We defer explaining how S-Paxos deals with failures to Section IV-B, and first focus our attention on the normal case.

A. Normal Operation

We now describe in detail how our dissemination layer works and how it interacts with the ordering layer (Paxos). In the pseudo-code of Algorithm 1 we use $propose(v)$ (line 17) to pass a value to the ordering layer which (later) signals decision in the i^{th} instance of Paxos by the $decide\langle i, v\rangle$ event (line 19). Moreover, since in our ordering layer only the leader can propose an order on ids, the dissemination layer also needs to be aware of the identity of the leader once it successfully finishes Phase 1.

As for Paxos we focus our exposition of S-Paxos on how a request is processed. Clients simply sends their requests to some replica.[4] When a replica receives a request from a client, it forwards the request and its unique id, to all other replicas (lines 9-10). When a replica receives a forwarded request (either from itself or from another replica), it records the id and the request in the $requests$ set. It then sends an acknowledgment containing only the id to all (lines 11-13). Since messages can be lost during periods of asynchrony (partial synchronous system), all acknowledgments are periodically retransmitted (lines 23-25).

After receiving $f + 1$ acknowledgments for a particular request id a replica records this fact by adding the id to its $stableIds$ set (we will discuss the importance of this set in the next subsection). Moreover, the leader replica also passes the request id to the ordering layer, which will then use the Paxos protocol to order it (lines 14-18).

As in Paxos the order in which requests are executed is defined by the sequence of decisions of the consensus instances. The only difference is that our ordering layer only orders ids. As it is possible (especially in periods of asynchrony) that an id is ordered before the corresponding request is received, S-Paxos cannot immediately execute requests as they are decided. Instead the decision is recorded (line 20) and executed

[3]We assume homogeneous settings, where there are no significant differences between hardware properties of replicas.

[4]We do not address the issue of load balancing in this paper. However, when all clients generate similar loads, a simple strategy is to let clients randomly pick a replica.

Algorithm 1 Dissemination layer (code of process p).

```
 1: Parameters:
 2:   $\eta$                         /* delay for retransmission of ACK */
 3:   $\Delta$                       /* delay before polling for request */
 4: Initialization:
 5:   $requests_p \leftarrow \emptyset$      /* requests/ids known to dissemination layer */
 6:   $proposed_p \leftarrow \emptyset$                      /* Tracks requests proposed */
 7:   $decided_p \leftarrow \emptyset$                      /* Maps instances to request ids */
 8:   $stableIds_p \leftarrow \emptyset$   /* ids acknowledged by at least $f+1$ replicas */

 9: upon receive request $\langle uid, r\rangle$ from a client do
10:   send $\langle$FORWARD$, uid, r\rangle$ to all

11: upon receive $\langle$FORWARD$, uid, r\rangle$ from $q$ do
12:   $requests_p \leftarrow requests_p \cup \{\langle uid, r\rangle\}$
13:   send $\langle$ACK$, \{uid\}\rangle$ to all

14: upon receive $f+1$ distinct acks for $uid$ do
15:   $stableIds_p \leftarrow stableIds_p \cup \{uid\}$
16:   if $p$ is leader and $uid \notin proposed_p$ then
17:     propose($uid$)
18:     $proposed_p \leftarrow proposed_p \cup \{uid\}$

19: upon decide $\langle i, uid\rangle$ do
20:   $decided_p \leftarrow decided_p \cup \{\langle i, uid\rangle\}$

21: upon exists $uid, r, i$ such that $\langle i, uid\rangle \in decided_p$
      and $\langle uid, r\rangle \in requests_p$ and $r$ not executed do
22:   execute $r$ in order $i$

23: every $\eta$ time do
24:   $S \leftarrow \{uid : \langle uid, r\rangle \in requests_p\}$
25:   send $\langle$ACK$, S\rangle$ to all

26: upon receive $\langle$ACK$, S\rangle$ from $q$ do
27:   for all $uid \in S : \langle uid, r\rangle \notin requests_p$ do
28:     if within $\Delta$ time request $\langle uid, r\rangle$ was not yet received then
29:       send $\langle$RESENDFORWARD$, uid\rangle$ to $q$

30: upon receive $\langle$RESENDFORWARD$, uid\rangle$ from $q$ do
31:   send $\langle$FORWARD$, uid, r\rangle$ to $q$

32: upon view change finished $\langle reproposed\rangle$ do
33:   /* executed by new leader only */
34:   $proposed_p \leftarrow proposed_p \cup reproposed$
                                   $\cup \{uid : \langle uid, r\rangle \in decided_p\}$
35:   for all $uid \in stableIds_p \setminus proposed_p$ do propose($uid$)
36:   $proposed_p \leftarrow proposed_p \cup stableIds_p$
```

only once the request is available at the replica (lines 21–22). After executing the request the replica that received the request from the client sends the corresponding reply.

B. Handling Failures

As S-Paxos uses Paxos as its ordering layer, and no internals of Paxos were changed, failure detection, leader election, view change, and retransmission of messages in the ordering layer are done as in Paxos. However, due to the separation of dissemination from ordering, failures can affect S-Paxos in ways not handled by the ordering layer alone. Therefore we now discuss the fault tolerance of the dissemination layer.

a) Ensuring stability of client requests: One aspect that makes performing ordering on ids instead of requests non trivial is the need to ensure that once an id is decided, the corresponding request is available in the system [21]. Requests that will remain available in the system even in the presence of crashes are called *stable*. In Paxos, stability is ensured by waiting for at least $f+1$ Phase 2b messages before deciding, which indicates that at least one correct process received the request as part of the Phase 2a message. As S-Paxos disseminates requests in a separate layer (i.e., outside Paxos), it cannot rely on the ordering layer to ensure stability of ordered requests. Rather, S-Paxos ensures that a request is stable before proposing the corresponding id. More specifically, the leader proposes an id only once it has received $f+1$ acknowledgments for the corresponding request (line 14).

This ensures that decided ids always correspond to client requests that reached at least one correct replica, say p. As p will include the corresponding request id in the ACK message sent periodically to all, other replicas can (upon receiving such ACK message) retrieve the request from p (lines 14–18, 23–25, 26–29). The same mechanism ensures (as $n \geq 2f+1$) that any request received by at least one correct replica will eventually be ordered.

Conversely, if the request does not reach any correct replica, then it will never be proposed by the leader (as there will never be $f+1$ acks). In this case, the clients that sent the requests will have to time out and retransmit it to another replica. Here unique request ids do not only prevent duplicate execution, but also allow replicas to send the correct response to previously executed requests.

b) View change: As long as there are no leader changes the mechanisms discussed above are sufficient to ensure that a request that becomes stable will be proposed. However, in the presence of leader changes this is no longer the case, because a request can become stable at a replica before that replica is promoted to leader. In order to ensure these requests are proposed we have introduced a call-back (lines 32–36) from the ordering layer which informs the dissemination layer that the new leader has finished Phase 1 of Paxos (Figure 2). At this point the dissemination layer (of the leader) will propose all stable ids (line 35), unless it is known that these ids have been decided (the ids are in the *decided* set) or they are already (re)proposed from within Paxos (the *reproposed* set).[5] Recall that the latter occurs as part of Paxos' view-change protocol and is based on information collected in Phase 1b. After this call-back is executed the new leader can again use the set *proposed* to ensure that no id is proposed more than once.

C. Optimizations

Since the ordering layer uses the standard Paxos protocol, it can use the traditional optimizations of batching and pipelining, as well as any other optimization that applies to Paxos.

Batching can also be used as an optimization at the dissemination layer: Instead of forwarding requests directly, a replica p can first group them into batches, assign them a unique batch id, e.g., of the form $\langle p : sn\rangle$, where sn is a local sequence number, and then broadcast them. The remainder of

[5]If this information is not available an alternative approach is to filter out duplicate requests in the ordering layer or during execution. While the former requires changing the ordering layer which we try to avoid, the latter solution incurs some wasted resources. Therefore using the *reproposed* set can also be seen as an optimization.

the protocol will then operate on batch ids and batches of requests rather than request ids and individual requests.

As presented in the algorithm above, the size of the periodic ACK messages will grow forever. This can be easily avoided, by using the id schema for requests proposed above for the batching optimization. This allows a replica to group all consecutive batch ids generated by one replica into intervals, and only transmit the bounds of these intervals. When one uses TCP for communication between the replicas the dissemination and reception of batches will occur in a FIFO manner (as long as connections are not interrupted). Therefore—in practice—the number of transmitted intervals is very low, and so this approach is sufficient to keep the size of acknowledgment messages within reasonable bounds.

Another possible optimization is to piggy-back the acknowledgments on the messages used to forward batches, with explicit acknowledgments sent only as a fall-back mechanism if no other messages are being exchanged. This optimization is especially effective when the system is under high load.

D. Discussion

The protocol above distributes *request reception*, *request dissemination* and *sending replies* across all replicas. Moreover, the only remaining leader-centric task, ordering, is now very light-weight since it is done only on ids. Thus, S-Paxos uses the resources of all replicas more evenly, which allows it to perform better than traditional leader-centric protocols. Indeed, the results in Section V confirm that S-Paxos balances the load across all replicas such that there is no perceptible difference between the leader and the followers (i.e., the other replicas).

The two parts of the protocol that were identified as main sources of bottlenecks in Section III, i.e., client request reception and dissemination, are now performed by all replicas. Therefore adding more replicas for fault tolerance can have the positive side-effect of improving the performance of the system (for reasonable numbers of replicas), since S-Paxos can use the additional resources of these replicas (cf. Figures 3b and 5). This contrasts with conventional leader-centric protocols, where higher fault tolerance typically results in lower performance.

The fact that only ids are proposed, also allows the dissemination of requests to continue throughout leader change. It is only the ordering layer that does not make progress while a new leader is determined. Moreover ordering only ids also leads to another advantage during view change: the state that has to be transferred from the replicas to the new leader during Phase 1b can be significantly smaller for S-Paxos when compared with Paxos. All this makes view changes very lightweight in S-Paxos (cf. Section V-C).

S-Paxos is designed for high throughput, which does not come for free. Compared to Paxos, it requires a higher number of communication steps to order a client request. Additionally, the two levels of batching (at the dissemination and at the ordering layer) can also harm response time. As our evaluation in Section V-B shows, we can indeed observe a moderate increase (less than 10ms) in average response time with low and medium client load. However, for high load the situation is reversed: the performance of Paxos eventually reaches its peak, leading to dramatically higher client response time, while for S-Paxos the throughput continues to increase and the response time remains low.

Cluster	CPUs	Network	Used in Section
Helios	2×2cores@2.2GHz	1Gbps	V-A1, V-B
Parapluie	2×12cores@1.7GHz	1Gbps	V-A2
Paradent	2×4cores@2.5GHz	1Gbps	V-A3, V-C

TABLE I
GRID5000 CLUSTERS FOR EXPERIMENTS

		S-Paxos		Paxos
Cluster	Req Size	*cbsz*	*bsz*	*bsz*
Helios	20	1450	50	1450
Parapluie	20	1450	50	1450
Paradent	1024	8700	50	8700

TABLE II
EXPERIMENTAL SETTINGS. SIZES IN BYTES. CBSZ - CLIENT BATCH SIZE (DISSEMINATION LAYER), BSZ - ORDERING LAYER BATCH SIZE.

V. PERFORMANCE EVALUATION

We have implemented S-Paxos by modifying JPaxos [12], an efficient multi-threaded implementation of Paxos in Java. The ordering layer of S-Paxos uses batching and pipelining, just like a typical Paxos implementation. Additionally, the dissemination layer implements the optimizations described in Section IV-C, *i.e.*, batching of client requests, compact representation of acknowledgments, and piggybacking of acknowledgments.

The experiments were run in several clusters of the Grid5000 testbed (see Table I). The network was always a Gigabit Ethernet with an effective inter-node bandwidth of 930Mbps (measured using `iperf`). Nodes were running Linux, kernel version 2.6.32-5, and the Java Virtual Machine used was Oracle's JRE version 1.6.0_25.

The workload was generated by nodes located in the same cluster as the replicas, each running several client threads in a single Java process. The clients use persistent TCP connections to communicate with replicas. In Paxos they communicate with the leader only, while in S-Paxos clients connect to a random replica. Clients send requests in a closed loop, waiting for the answer to the previous request before sending the next one[6]. Each experiment was run for 3 minutes, with the first 10% ignored in the calculation of the results. To focus our evaluation on the ordering protocol, we used a null service that discards the payload of the request, sends back a fixed 8 bytes response, and does not use stable storage. The overhead of using a more complex service or stable storage would easily dominate the ordering part.

[6]Since clients can have only one outstanding request, in order to saturate the system we had to use a high number of clients. In case clients can issue several parallel requests, a smaller number of clients would be sufficient.

In all experiments the bound on the number of parallel Paxos instances (pipelining) was set to 30. Table II summarizes other experimental settings (for S-Paxos, $cbsz$ refers to the batching of request ids of Section IV-C, and bsz to the batching of ids in the Paxos ordering layer; for Paxos, bsz refers to the batching of requests in Paxos). The values were chosen to match the natural limits of the underlying Ethernet network (1500 bytes maximum payload of a frame, and optimal performance usually with messages of around 8KB).

We use as metrics the throughput in requests per second and in data ordered per second (Mbps), the client response time, and the CPU utilization. The *client response time* is the time from when the client sends a request until it receives the corresponding reply, which includes dissemination, ordering, and execution. The *CPU utilization* of a replica is measured using the GNU `time` command and is shown as a percentage of one core, *i.e.*, 100% is equivalent to one core being fully utilized.

In Section V-A we describe the throughput of Paxos and S-Paxos in different scenarios when Paxos' bottleneck is the CPU (Section V-A1), the networking subsystem (Section V-A2), and the bandwidth (Section V-A3). For each of these scenarios, we choose the workload (number of clients and request/response size), the algorithm parameters (*e.g.*, batch size), and clusters (Table I) such that Paxos hits the intended bottleneck. In each case, S-Paxos is able to avoid the bottleneck that limits Paxos, achieving higher performance. In Section V-B we measure the client response time of Paxos and S-Paxos under different client loads, and in Section V-C we study the effect of view changes and crashes on performance.

In the tests that show the throughput with increasing number of replicas (n), we were interested in the maximum throughput of each protocol. However, Paxos and S-Paxos achieve their maximum throughput with very different number of clients (see Figure 3a): S-Paxos reaches its maximum with a number of clients that is significantly higher than what Paxos can support without its performance starting to degrade. Therefore, we repeated each experiment using different number of clients, and report the highest throughput for each value of n.

A. Throughput

1) When the CPU is the Bottleneck: For testing with the CPU as the bottleneck, we have chosen a request size of 20 bytes. As discussed in Section III, small request sizes put a much greater stress on the CPU than on the bandwidth. Moreover, we used a cluster with a small number of cores, *e.g.*, 2×2cores@2.2GHz. As our results show, in these conditions the CPU is indeed the bottleneck both for Paxos and S-Paxos (although at very different throughput levels).

When $n = 3$, the throughput of Paxos increases with the number of clients until it reaches a maximum at around 75K requests per second with just over 1'000 clients, (Figure 3a). At this point, we can see in Figure 4a that one replica, the leader, is using over 300% of CPU, which is close to the maximum of 400% for this setup (recall that the nodes in the Helios cluster are four-core machines). The other replicas,

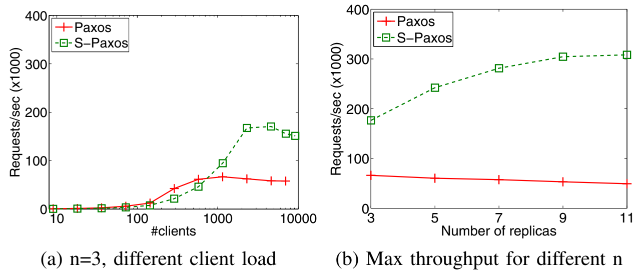


Fig. 3. Throughput of Paxos and S-Paxos when CPU is the bottleneck in Paxos. Helios cluster.

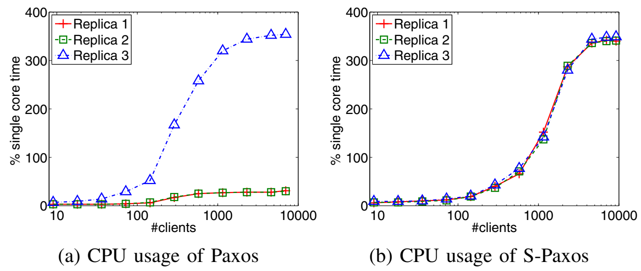


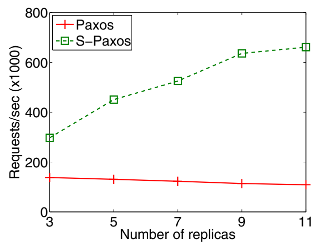
Fig. 4. CPU usage at different replicas with a) Paxos and b) S-Paxos with increasing number of clients. Helios cluster.

however, are only lightly loaded. In contrast, with S-Paxos (Figure 4b) the CPU usage is equally distributed among all replicas and grows slower than on the leader when using Paxos. Like with Paxos, the throughput also levels off when the CPU usage exceeds 300%. However, S-Paxos reaches a throughput of almost three times the one of Paxos, with 200K requests per second while serving 8'000 clients.

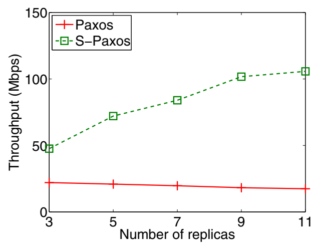
When the number of replicas is increased (Figure 3b), S-Paxos throughput increases, reaching a maximum of 300K requests per second with 11 replicas. This contrasts sharply with Paxos, whose throughput decreases with the number of replicas, going down from around 75K with $n = 3$ to 50K with $n = 11$. With $n = 11$, S-Paxos achieves six times the throughput of Paxos. The reason is that S-Paxos is able to spread the workload of handling client connections and dissemination among all replicas, while in Paxos the leader must perform all these tasks, thus quickly maxing out its CPU bottleneck.

2) When the Network Subsystem is the Bottleneck: For these tests, we use the Parapluie cluster. In contrast to the Helios cluster used in the previous experiments, the nodes in the Parapluie cluster have a high number of cores (24 versus 4). And since both Paxos and S-Paxos are designed to make efficient use of multiple cores, this means that in the Parapluie cluster neither of them is CPU bound anymore. Instead, the bottleneck shifts to the network subsystem which can only use a single core, due to limitations on the network stack of the version of Linux used for the tests.

In Figures 5a and 5b we show the throughput of Paxos and S-Paxos in requests per second, and data ordered per second



(a) Throughput(req/sec)



(b) Throughput(Mbps)

Fig. 5. Throughput of Paxos and S-Paxos when network subsystem is the bottleneck in Paxos. Parapluie cluster.

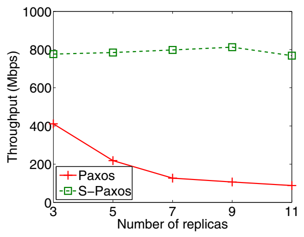


Fig. 6. Throughput of Paxos and S-Paxos when bandwidth is the bottleneck in Paxos. Paradent cluster.

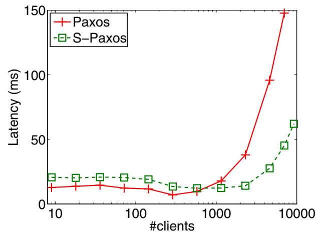


Fig. 7. Response time with Paxos and S-Paxos with different number of clients (log scale). Helios cluster.

(Mbps). The results confirm that the bottleneck in this case is indeed the network subsystem: The CPU usage during the tests is always below 600% out of a maximum of 2400% (results not shown here), and the peak data rate of 100Mbps is far from the maximum bandwidth of a Gigabit network (Figure 5b). Through separate experiments, we have confirmed that both Paxos and S-Paxos have reached the maximum number of packets per second that the network subsystem is able to handle, which is around 150K.

Like in the case where the bottleneck is the CPU, as the number of replicas increases the throughput of Paxos gets worse while the one of S-Paxos improves (Figure 5a). Additionally, the throughput of S-Paxos is always substantially higher than the one of Paxos: For $n = 3$ S-Paxos has double the throughput (300K versus 150K), while for $n = 11$ the advantage increases to six times (600K versus 100K).

The reason is once again that in Paxos the leader is the sole replica interacting with the clients, while in S-Paxos this work is shared among all replicas. Therefore, for the same number n of replicas, S-Paxos is capable of handling approximately n times more client connections. So, as n increases, S-Paxos can handle more client connections, where Paxos is still limited to what the leader can handle.

3) When the Network Bandwidth is the Bottleneck: For the tests focusing on the bandwidth, we use a request size of 1KB. With this request size, in all the clusters we used for the experiments, both Paxos and S-Paxos are easily able to saturate the bandwidth available before hitting the limits of the CPU or of the network subsystem of the replicas. Even so, we chose the cluster with the fastest CPUs (Paradent, see Table I); this further ensures that neither the network subsystem nor the CPU are the bottleneck. Recall that the effective inter-node bandwidth in the clusters used for all the experiments, including Paradent, is approximately $B = 930$Mbps.

Figure 6 shows the throughput in Mbps of Paxos and S-Paxos with an increasing number of replicas. On the one hand, the maximum throughput of Paxos is $B/(n-1)$, since the leader has to send requests to $n-1$ followers. This explains the throughput being approximately 400Mbps for $n = 3$, and dropping as additional replicas are added to the system. On the other hand S-Paxos orders around 800Mbps of application data (without TCP/IP headers and protocol headers) irrespective of the number of replicas. As expected, for S-Paxos the limiting factor is the amount of data any replica can receive. This difference between Paxos and S-Paxos comes from the fact that in both protocols every request has to be received by every replica, but only in Paxos the leader has to send every request.

B. Response Time

In this section we study the response time of Paxos and S-Paxos under different client loads. We run the experiments on the Helios cluster, using a request size of 20 bytes.

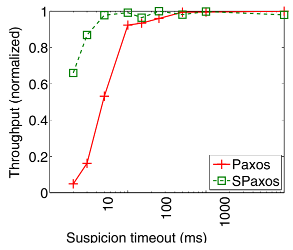
Figure 7 shows the results. Under small and medium load (up to 1'000 clients), S-Paxos has a slightly higher response time than Paxos (approximately 20ms versus 13ms). However, as the client load increases, the situation is reversed, with the response time of Paxos deteriorating faster than the one of S-Paxos (note the log-scale). These results are a good example of the trade-off that typically exists between latency and throughput. To optimize for throughput, S-Paxos uses a longer request processing pipeline, with more levels of batching and more communication steps than Paxos. Therefore, under low load it does not perform as well as Paxos. But as the client load increases, Paxos is not able to keep up with the demand, forcing the client requests to queue for a long time before being ordered and executed. S-Paxos, however, can cope with higher client loads, therefore avoiding the queuing delays and providing a much better response time.

C. Failures

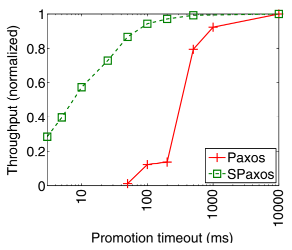
For the evaluation of the failure case we return to the Paradent cluster, and a request size of 1KB (See Table II).

1) Cost of View Change: We start by studying the cost of view change in two scenarios: view changes due to false suspicions and leader induced view changes. For the first scenario, we perform several runs where we lower the suspicion timeout of the failure detector, therefore increasing the frequency of false suspicions. For the second scenario, we rotate the role of leader periodically between replicas, by having the replica with id immediately greater than the one of the leader (modulo n) promote itself some time after the current leader was elected. The results are shown in Figure 8.

With timeouts as low as 100ms, false suspicions are rare so neither Paxos nor S-Paxos show any drop in performance (Figure 8a). For smaller timeouts, false suspicions become

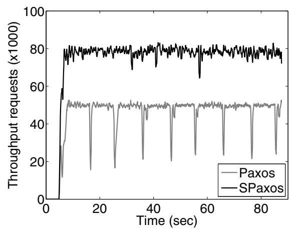


(a) Suspicions

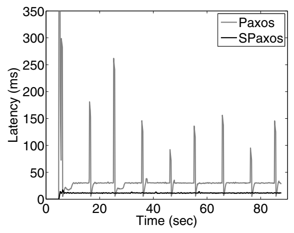


(b) Promotions

Fig. 8. Performance with varying failure suspicion timeout and leader promotion timeout. $n = 3$

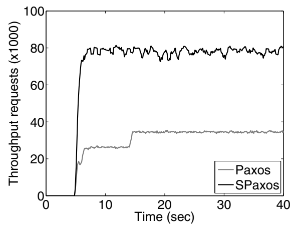


(a) Throughput

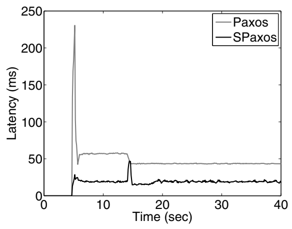


(b) Latency

Fig. 9. Performance over time with leader promotion every 10s. $n = 3$

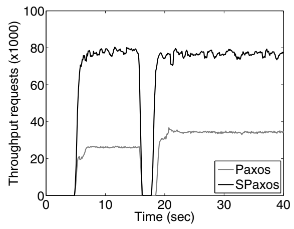


(a) Throughput

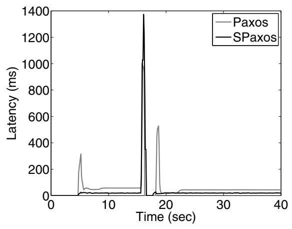


(b) Response time

Fig. 10. Crash of a follower. $n = 5$



(a) Throughput



(b) Response time

Fig. 11. Crash of the leader. $n = 5$

common enough to affect performance. While S-Paxos has a graceful behavior, achieving still a respectable 60% of its peak performance even with suspicions timeouts of 3ms, the performance of Paxos quickly collapses with timeouts smaller than 10ms. Since the failure detector used is the same in both cases, the number of false suspicions is roughly equivalent, which leaves the cost of view change itself as the cause of the difference. The results of the test with leader induced view changes (Figure 8b) confirm that this is indeed the case. With Paxos, the throughput degrades quickly with view changes every 1 second, and collapses with view change intervals of 200ms. S-Paxos, however, tolerates view changes much better, achieving still 70% of its peak throughput with view change intervals of 50ms.

To better understand the effect of view change, we show in Figure 9 the time series of the response over a single run. For each x coordinate, the plot shows the average response time of the requests sent at time x. Note that this time includes re-transmissions in case of connection or replica failures. Figure 9 shows clearly that view changes do not affect the performance of S-Paxos in any noticeable way. However, in Paxos, each view change causes a temporary drop in performance.

S-Paxos performs better in these cases because of two main reasons. First, the amount of data exchanged by S-Paxos during view change is, on average, much smaller than in Paxos; this is because Phase 1b messages of S-Paxos contain only ids, while in Paxos they contain full batches. Second, in S-Paxos clients do not need to reconnect to a different replica, while in Paxos all clients must disconnect from the previous leader and reconnect to the new one. This second problem can be minimized by allowing every replica to receive client connections and having replicas forward requests to the leader, like Zab [13] does. But even with this improvement, view changes will still be more expensive than in S-Paxos, because after a view change replicas may have to forward to the new leader some requests that were already forwarded to the previous leader, thus wasting resources. This is not necessary in S-Paxos.

2) Performance with Crashes: We now look at the effect of a crash on the performance of the system. Each experiment lasts for 40 seconds, with a crash being triggered 15 seconds after startup. As we can see from Figures 10 and 11, requests start to be ordered about 5 seconds after startup.

The crash of a follower has little impact on the performance of the system (Figure 10). The throughput of S-Paxos is not at all affected, while the one of Paxos increases because now the leader is sending messages to one less follower, since the TCP connection to the crashed follower is closed.[7] For the same reason, the response time of Paxos improves slightly after the crash. On the other hand, the response time of S-Paxos has a spike during the crash, then quickly returning to the same levels as before the crash. This spike is caused by the clients that were connected to the replica that crashed (approximately, 1/5 of all clients), which have to reconnect to another replica after a small random backoff delay (between 0.1 and 0.5 seconds). This is not the case for Paxos because clients do not connect to followers.

When the leader crashes (Figure 11) both Paxos and S-Paxos stop ordering requests until a new leader is elected.[8] The gap in the plots is dominated by the suspicion timeout

[7]Recall from Section V-A3 that with these experimental settings the bandwidth is the bottleneck.

[8]The response time for periods where no request is ordered is shown as 0.

of the failure detector (2 seconds). S-Paxos recovers faster than Paxos because clients remain connected to the other 4 replicas and, therefore, as soon as a leader is elected these clients and the replicas resume normal operation, with only a small portion of the clients having to reconnect to some other replica. In Paxos all clients have to reconnect to the new leader. Depending on the timeouts and reconnection strategy, this may happen only some time after a new leader is elected. The response time of both Paxos and S-Paxos shows a spike just before the crash, corresponding to the requests that were sent before the crash of the leader, but ordered only after the recovery of the system. The second spike of Paxos is, once again, a consequence of the clients having to connect directly to the leader, which introduces additional delays.

VI. Related Work

According to the classification of [15], Paxos belongs to the group of fixed sequencer protocols because a distinguished process (the leader) is responsible for establishing the order of messages. Another example of a high-throughput fixed sequencer protocol is Ring-Paxos [17]. Ring-Paxos relies on efficient use of IP Multicast to disseminate messages to servers. This, together with the fact that it executes consensus on ids (like S-Paxos), makes it a very efficient protocol in the case where the outgoing channel of the leader is the bottleneck (large requests). Another fixed sequencer protocol of interest is Zab [13], a variant of the Paxos protocol used in primary-backup systems such as Zookeeper. While Zab distributes client communication to all replicas, it follows the conventional leader-centric design by having replicas forward requests to the leader.

All the above fixed sequencer protocols share the same short-coming: They burden the sequencer to the point it becomes the system's bottleneck. In this paper, we have shown that by offloading the work from the leader, we can derive a fixed sequencer protocol that has good performance and that uniformly utilizes system resources at all replicas.

Mencius [22] and the position paper [23] take an alternative approach to prevent the leader from becoming the bottleneck. These protocols are based on the observation that rotating the role of the sequencer avoids contention on a single replica.[9]

In Mencius the sequence of consensus protocol instances is partitioned among all replicas with each taking the job of being (initial) leader of an instance in a round-robin fashion. In order to exclude failed replicas from this schedule some reconfiguration is necessary in case of failures. Contrary to S-Paxos, which is designed for clustered environment, Mencius is designed with the goal of being an efficient SMR protocol for WAN environments. Since Mencius and S-Paxos are based on different high level concepts (moving sequencer vs. fixed sequencer), a detailed analysis would be necessary to understand trade-offs between the two approaches. However, we expect the performance in the failure-free case to be comparable (both approaches balance the work among replicas). In the failure case we expect S-Paxos to be more efficient: the crash of any replica in Mencius stops progress, while in S-Paxos this is the case only if the leader replica fails (the crash of a follower does not affect system progress).

[9]According to the classification [15], the protocols that rotate the sequencer role among replicas are called *moving sequencer protocols*.

Kapritsos and Junqueira [23] assign each consensus instance to different "virtual clusters", instead of to a different leader as in Mencius. Each virtual cluster consists of $2f + 1$ virtual replicas which are mapped to overlapping sets of physical replicas. To reach maximal throughput each virtual cluster's leader should be on a different physical replica. This clearly necessitates some inter-cluster coordination in case of leader changes. Otherwise, one physical replica could become the bottleneck as it has the job of being leader in multiple virtual clusters. Although this approach can potentially achieve the same goal as S-Paxos, we cannot at the time of writing compare them in more detail because [23] is the only publicly available description of the protocol and it does not provide many important details (in particular, how the system maintains a balance distribution of work in the presence of failures).

In both solutions just discussed, multiple streams of decisions have to be merged into a single total order. This is done by taking one request from each process/virtual cluster in a round robin fashion. If one process/cluster does not have any request to order, it has to propose a special *skip* request, to allow other replicas to continue ordering requests. There is no need for such mechanism in S-Paxos.

While all aforementioned work is based on the ideas behind Paxos and thus are leader-based, this is not the case for all protocols. One protocol that belongs to this group is LCR [24], which arranges replicas along a logical ring and uses vector clocks for message ordering. LCR is a high-throughput protocol where work is equally divided among servers, thereby utilizing all available system resources. The drawback of the approach taken by LCR is that latency increases linearly with the number of processes in the ring; moreover, maintaining the ring structure adds overhead to the protocol. Although LCR has a slightly better bandwidth efficiency than S-Paxos for large requests (according to [17] it achieves 95% efficiency in a cluster setting), it requires *perfect failure detection*. Perfect failure detection implies stronger synchrony assumption than required by S-Paxos. While not limited by a leader being the bottleneck, it is unclear whether adding replicas increases the throughput of LCR.

State partitioning [25] is another technique commonly used to achieve scalability. Recently, it has been considered in the context of state machine replication [26], [27]. In [26], Marandi et al. show that state partitioning leads to improvements in both throughput and response time for a replicated B-tree service. However, as they argue, perfect state partition is often not possible. One can gain performance even with imperfect state partitioning, as it lifts the requirement that every request needs to be received by every replica. In [26] the authors use Ring-Paxos as an ordering protocol. However, as they point out in [27], when the number of partitions increases (which is required for good throughput), Ring-Paxos becomes the bottleneck due to its leader-centric nature. In order to avoid Ring-Paxos being the bottleneck, [27] introduces groups, and let a single instance of Ring-Paxos be responsible for ordering messages within a single group. Then each learner subscribes only to the groups it wants to receive messages from. This approach, which is similar to [22], [23], requires a mechanism for merging requests from different groups and a skipping mechanism for inter-group coordination. Note that S-Paxos can be easily used in this context by configuring the dissemination layer to send requests only to interested learners, while ordering layer would deliver the order to all learners. The benefits would be using a single cluster instead of several, and the fact that there is no need for additional mechanisms (merging requests, skipping consensus instances, reconfiguration).

The benefit of running consensus on ids in the context of Atomic Broadcast is explored in [21]. Their approach requires modifying the consensus algorithm, so that it only decides an id when the corresponding message is stable. Our work differs in two main aspects. First, we consider a different problem, that is, balancing the load in State Machine Replication using Paxos. Second, we ensure stability of the request before initiating ordering of the id, and thereby avoid the need to modify the ordering protocol.

VII. Conclusion

In this paper we have shown that leader-based protocols do not have to be leader-centric. We presented S-Paxos, a variant of the Paxos protocol that offloads work from the leader by delegating it to the other replicas. Distributing the work done by the leader in leader-centric protocols over all replicas allows S-Paxos to benefit fully from the resources of all replicas and to also increase its performance with the number of replicas. We implemented a prototype of S-Paxos and evaluated its performance in different cluster settings. S-Paxos achieves between 2 and 3 times the request throughput of a leader-centric Paxos implementation for $n = 3$. Furthermore, the throughput of S-Paxos improves when additional replicas are added to the system, while the performance of most SMR implementations drops. Therefore, with S-Paxos there is no need to make a trade-off between fault tolerance and performance, contrary to what must be done in most SMR implementations. By increasing the number of replicas, the system not only becomes more fault tolerant but also achieves higher throughput than leader-centric Paxos implementations (up to 7 times, for $n = 11$).

As future work, we plan to add stable storage (using solid state drives) to S-Paxos in order to tolerate catastrophic failures, to apply our technique to Byzantine faults, and to perform experiments with S-Paxos in WAN settings.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). We would like to thank Darko Petrovic, Thomas Ropars and Omid Shahmirzadi for their comments on an earlier version of the paper.

References

[1] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, pp. 558–565, 1978.

[2] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, pp. 299–319, 1990.

[3] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, 1998.

[4] ——, “Fast Paxos,” *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[5] L. Lamport, D. Malkhi, and L. Zhou, “Reconfiguring a state machine,” *SIGACT News*, vol. 41, pp. 63–73, 2010.

[6] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, “Paxos replicated state machines as the basis of a high-performance data store,” in *NSDI'11*, 2011, pp. 11–11.

[7] J. Rao, E. J. Shekita, and S. Tata, “Using Paxos to build a scalable, consistent, and highly available datastore,” *Proc. VLDB Endow.*, vol. 4, pp. 243–254, 2011.

[8] M. Burrows, “The chubby lock service for loosely-coupled distributed systems,” in *OSDI '06*, 2006, pp. 335–350.

[9] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: wait-free coordination for internet-scale systems,” in *USENIXATC*, 2010, p. 11.

[10] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, “Boxwood: abstractions as the foundation for storage infrastructure,” in *OSDI'04*, 2004, pp. 8–8.

[11] B. Reed and F. P. Junqueira, “A simple totally ordered broadcast protocol,” in *LADIS '08*, 2008, pp. 2:1–2:6.

[12] N. Santos, J. Kończak, T. Żurkowski, P. Wojciechowski, and A. Schiper, “JPaxos - State machine replication in Java,” EPFL, Tech. Rep. 167765, Jul. 2011.

[13] F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *DSN'11*, 2011, pp. 245–256.

[14] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: an engineering perspective,” in *PODC '07*, 2007.

[15] X. Defago, A. Schiper, and P. Urban, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *ACM Computing Surveys*, vol. 36, p. 2004, 2004.

[16] J. Kirsch and Y. Amir, “Paxos for system builders,” Dept. of CS, Johns Hopkins University, Tech. Rep., 2008.

[17] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, “Ring Paxos: A high-throughput atomic broadcast protocol,” in *DSN'10*, 2010, pp. 527–536.

[18] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, G. Chockler, H. Li, and Y. Tock, “Dr. multicast: Rx for data center communication scalability,” in *EuroSys '10*, 2010.

[19] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, “An analysis of linux scalability to many cores,” in *OSDI'10*. Berkeley, USA: USENIX, 2010, pp. 1–8.

[20] N. Santos and A. Schiper, “Tuning Paxos for high-throughput with batching and pipelining,” in *13th International Conference on Distributed Computing and Networking (ICDCN 2012)*, Jan. 2012.

[21] R. Ekwall and A. Schiper, “Solving atomic broadcast with indirect consensus,” in *DSN'06*, 2006, pp. 156–165.

[22] Y. Mao, F. P. Junqueira, and K. Marzullo, “Mencius: building efficient replicated state machines for wans,” in *OSDI'08*, 2008, pp. 369–384.

[23] M. Kapritsos and F. P. Junqueira, “Scalable agreement: toward ordering as a service,” in *HotDep'10*, 2010, pp. 1–8.

[24] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, “Throughput optimal total order broadcast for cluster environments,” *ACM Trans. Comput. Syst.*, vol. 28, pp. 5:1–5:32, 2010.

[25] J. Gray, P. Helland, P. O'Neil, and D. Shasha, “The dangers of replication and a solution,” in *SIGMOD '96*, 1996.

[26] P. Marandi, M. Primi, and F. Pedone, “High performance state-machine replication,” in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, june 2011, pp. 454 –465.

[27] ——, “Multi-ring Paxos,” in *Dependable Systems Networks (DSN), 2012 IEEE/IFIP 42st International Conference on*, june 2012.