Kernel Paxos

Emanuele Giuseppe Esposito*, Paulo Coelho*† and Fernando Pedone*
*Università della Svizzera italiana, Switzerland
†Federal University of Uberlândia, Brazil

***Abstract*—State machine replication is a well-known technique to build fault-tolerant replicated systems. The technique guarantees that replicas of a service execute the same sequence of deterministic commands in the same total order. At the core of state machine replication is consensus, a distributed problem in which replicas agree on the next command to be executed. Among the various consensus algorithms proposed, Paxos stands out for its optimized resilience and communication. Much effort has been placed on implementing Paxos efficiently. Existing solutions make use of special network topologies, rely on specialized hardware, or exploit application semantics. Instead of proposing yet another variation of the original Paxos algorithm, this paper proposes a new strategy to increase performance of Paxos-based state machine replication. We introduce Kernel Paxos, an implementation of Paxos that significantly reduces communication overhead by avoiding system calls and TCP/IP stack. To reduce the number of context switches related to system calls, we provide Paxos as a kernel module. We present a detailed performance analysis of Kernel Paxos and compare it to a user-space equivalent implementation.**

***Index Terms*—fault-tolerance, replication, state machine replication, consensus, linux kernel.**

I. Introduction

State machine replication is a widely used technique to render applications fault tolerant [1], [2]. State machine replication underlying principle is surprisingly simple: server replicas must deterministically execute the same sequence of client commands. Determinism implies that the state changes and results created by the execution of a command only depend on the replica state when the command is executed and on the command itself. As a result, every replica transitions through the same sequence of states and produces the same results for each command. In a system prone to crash failures, clients only need to receive the reply from one replica, and thus, the application can tolerate replica failures.

Propagating and ordering commands in a distributed system subject to failures is no easy matter [3], [4]. At the core of state machine replication, replicas must solve consensus, a distributed problem in which replicas must agree on a value (i.e., the i-th command to be executed). Many consensus algorithms have been developed in the literature and Paxos [5] is probably the most prominent consensus algorithm proposed to date. In part this is attributed to the fact that Paxos is resilience-optimum (i.e., it ensures progress with a majority-quorum of non-faulty replicas) and delay-optimum (i.e., it requires a minimum number of network delays to reach a decision) [6].

While relying on an algorithm that optimizes resilience and communication is important, quickly ordering commands in state machine replication also requires an efficient implementation of Paxos. Therefore, much effort has been placed on implementing Paxos efficiently. Existing proposals fall in three categories: protocols that take advantage of special network topologies (i.e., overlay networks) [7], [8]; protocols that resort to specialized hardware (e.g., part of Paxos is deployed in the network) [9], [10]; and protocols that exploit the semantics of applications (e.g., if the order of two messages does not matter to the application, they do not have to be ordered) [11], [12], [13]. This paper presents Kernel Paxos, an alternative approach, which does not depend on special network topologies, specialized hardware, or application semantics. The main idea behind Kernel Paxos is simply to reduce the main sources of overhead in traditional Paxos implementations. Since Paxos does not involve complex computation, most of the overhead stems from communication. Kernel Paxos reduces communication overhead in two ways: (a) by eliminating context switches needed to send and receive messages and (b) by avoiding the TCP/IP stack. More concretely, Kernel Paxos eliminates context switches by placing the protocol in the kernel; instead of TCP/IP, Kernel Paxos uses raw Ethernet frames.

We have fully implemented Kernel Paxos (publicly available as open source) and carefully assessed its performance. Kernel Paxos is derived from LibPaxos, a popular Paxos library. We compared the performance of Kernel Paxos to LibPaxos and an improved version of LibPaxos. We have performed experiments using different message sizes, system load, and communication hardware. Our experiments show that Kernel Paxos largely outperforms these libraries. In some cases, messages can be ordered by Kernel Paxos in approximately one round-trip time.

This practical experience report makes the following contributions.

- It introduces a number of optimizations to LibPaxos, a popular Paxos library. These optimizations have consistently improved the performance of LibPaxos.
- It describes Kernel Paxos, a kernel-based library that outperforms both LibPaxos and its optimized version.
- It assesses the performance of Kernel Paxos under different conditions and experimentally compares it to LibPaxos and its improved version.

The rest of the paper is organized as follows. Section II introduces the system model and definitions. Section III introduces Kernel Paxos. Section V details our prototype. Section VI describes our experimental evaluation. Section VII surveys related work and Section VIII concludes the paper.

II. Model and Definitions

In this section, we detail our system model (§II-A), and recall the definitions of consensus (§II-B) and state machine replication (§II-C).

A. Processes, communication and failures

We consider a distributed system with an unbounded set of client processes $\mathcal{C} = \{c_1, c_2, ...\}$ and a bounded set of server processes $\mathcal{S} = \{p_1, ..., p_n\}$, where clients and servers are disjoint. Client and server processes are subject to the following assumptions.

- Processes communicate by exchanging messages and do not have access to a shared memory or a global clock.
- The system is asynchronous: messages may experience arbitrarily large (but finite) delays and there is no bound on relative process speeds.
- Processes may fail by crashing (i.e., processes do not behave maliciously). A process that never crashes is *correct*; otherwise it is *faulty*.
- Communication links are fair-lossy, i.e., links do not create, corrupt, or duplicate messages, and guarantee that for any two *correct* processes p and q, and any message m, if p sends m to q infinitely many times, then q receives m an infinite number of times.

B. Consensus

Consensus is a fundamental problem in fault-tolerant distributed systems. Informally, consensus allows processes to propose values and ensures that eventually one of the proposed values is decided.

Typically, applications of consensus use a sequence of consensus instances. Therefore, a process p proposes a value x (e.g., a message, a set of messages) in instance i by invoking propose$_i(x)$, and decides on value y in instance i with decide$_i(y)$. More formally, consensus satisfies the following properties:

- *uniform integrity:* if a process decides x in instance i, then x was previously proposed by some process in i.
- *termination:* if a correct process p proposes a value in instance i, then every correct process eventually decides exactly one value in i.
- *uniform agreement:* if a process p decides x in instance i, then no process decides $y \neq x$ in i.

Consensus has been shown to be impossible to solve in asynchronous distributed systems where processes are subject to failures [3]. To make consensus solvable in such settings, we further assume that processes have access to a weak leader election oracle [4]. The oracle outputs a single process denoted $leader_p$ such that there is (a) a correct process l and (b) a time after which $leader_p = l$ for every process p.

C. State machine replication

State machine replication is a well-established approach to fault tolerance [1], [2]. The idea is that by executing service requests deterministically in the same order, correct replicas will transition through the same sequence of state changes and produce the same output for every request.

With state machine replication, every server has a full copy of the service state, executes every operation, and replies to clients. Since the order and the execution is deterministic, any server may reply to clients. A sequence of consensus instances can be used to ensure that servers execute operations in the same order: consensus instance i decides on the i-th operation executed by each server.

State machine replication provides linearizability, a consistency criteria. A system is linearizable if it satisfies the following requirements [14]: (i) It respects the real-time ordering of requests across all clients. There exists a real-time order among any two requests if one request finishes at a client before the other request starts at a client. (ii) It respects the semantics of the requests as defined in their sequential specification.

III. Background on Paxos

This section briefly overviews the Paxos algorithm (§III-A), explains the use of Paxos in state machine replication (§III-B), and discusses some optimizations (§III-C).

A. Basic Paxos

Paxos is a fault-tolerant consensus protocol with important characteristics: it has been proven safe under asynchronous assumptions (i.e., when there are no timing bounds on message propagation and process execution), live under weak synchronous assumptions, and resilience-optimum [5].

Paxos distinguishes the following roles that a process can play: *proposers*, *acceptors* and *learners*. An execution of Paxos proceeds in two phases. During the first phase, a proposer that wants to submit a value selects a unique round number and sends a prepare request to a group of acceptors (at least a quorum). Upon receiving a prepare request with a round number bigger than any previously received round number, the acceptor responds to the proposer promising that it will reject any future prepare requests with smaller round numbers. If the acceptor already accepted a request for the current instance (explained next), it will return the accepted value to the proposer, together with the round number received when the request was accepted. When the proposer receives answers from a quorum of acceptors, it proceeds to the second phase of the protocol.

The proposer selects a value according to the following rule. If no acceptor in the quorum of responses accepted a value, the proposer can select a new value for the instance; however, if any of the acceptors returned a value in the first phase, the proposer chooses the value with the highest round number. The proposer then sends an accept request with the round number used in the first phase and the value chosen to at least a quorum of acceptors. When receiving such a request, the acceptors acknowledge it by sending a message to the learners, unless the acceptors have already acknowledged another request with a higher round number. Some implementations extend the protocol such that the acceptor also sends an acknowledge to the proposer. Consensus is reached when a quorum of acceptors accepts a value.

Paxos ensures consistency despite concurrent leaders and progress in the presence of a single leader. Paxos is resilience-optimum in that it tolerates the failure of up to f acceptors (or replicas) from a total of $2f + 1$ acceptors to ensure progress (i.e., a quorum of $f + 1$ acceptors must be non-faulty) [6].

B. Paxos and state-machine replication

In practice, replicated services run multiple executions of the Paxos protocol to achieve consensus on a sequence of values. We refer to multiple executions, or instances, of Paxos chained together as Multi-Paxos [15].

Clients of a replicated service are typically proposers, and propose operations that need to be ordered by Paxos before they are learned and executed by the replicated state machines. These replicas typically play the roles of acceptors and learners. If multiple proposers simultaneously execute the described procedure for the same instance, then no proposer may be able to execute the two phases of the protocol and reach consensus. To avoid scenarios in which proposers compete indefinitely in the same instance, a *leader* process can be chosen. In this case, proposers submit values to the leader, which executes the first and second phases of the protocol. If the leader fails, another process takes over its role. In typical state machine replication implementations one of the replicas assumes the role of leader and proposes operations forwarded by clients.

C. Optimizations

If the leader identity does not change between instances, then the protocol can be optimized by pre-initializing acceptor state with previously agreed upon instance and round numbers, avoiding the need to send first phase messages [5]. This is possible because only the leader sends values in the second phase of the protocol. With this optimization, consensus can be reached in three communication steps: the message from the proposer to the leader, the accept request from the leader to the acceptors, and the response to this request from the acceptors to the leader and learners. Moreover, the leader pre-initialization (first phase of the protocol) for a further instance could be piggybacked in the second phase message for the current instance. Fig. 1 exhibits such optimizations.

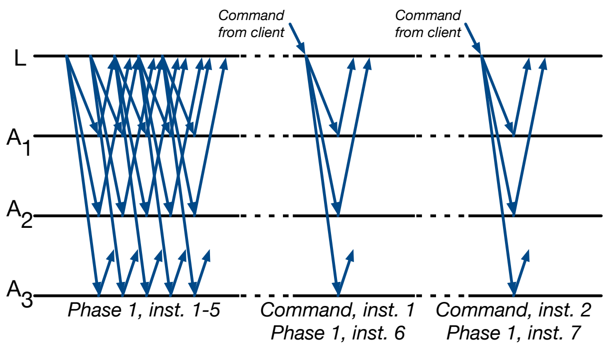


Fig. 1: Multi-Paxos optimization (example with five pre-initialized instances followed by two client commands).

IV. Paxos in the kernel

In this section, we motivate the use of in-kernel Paxos (§IV-A) and describe Kernel Paxos's architecture (§IV-B) and message flow (§IV-C).

A. Linux kernel and TCP/IP stack

The Linux kernel is an open-source software that implements the basic functionalities provided by an operating system (e.g., interface to the hardware devices, memory and processes management). Due to its openness, the Linux kernel is easily extensible by means of loadable kernel modules (LKM). A module can add support to a new device or introduce a new operating system feature. Besides, it allows separating the minimum core functionalities from on-demand loadable complementary utilities.

Linux implements the traditional user and kernel separation of concerns: user applications run in user-space while kernel-related tasks run in the kernel-space. This separation protects the operating system and the hardware against intentional and unintentional misbehave of applications. Every time a user application wants to send a message through the network, for instance, it uses a system call to context switch from user-space to kernel-space. The system call is necessary to access the network interface card (NIC) and execute the actual sending of data.

A clear trade-off comes from such an approach: if on the one hand the system calls protect the operating system and control the execution of programs, on the other hand every system call involves two context switches, one from user-space to kernel-space so that the call can be executed and another context switch to return from kernel-space to user-space.

Applications that heavily rely on system calls (e.g., I/O-bound applications) suffer the most from the overhead of context switches. In an attempt to measure the impact of context switches in network applications, we have developed a simple echo application that runs either in user-space or entirely in kernel-space, as a loadable kernel module. Clients send outstanding UDP messages to a server, which simply replies with the same message it has received from the client. We ran experiments with increasing number of clients until the point when the throughput stopped increasing. The first line of Table I shows the peak performance for both the user application and the loadable kernel module.

	User-space application	*Loadable kernel module*
UDP echo	120000 msgs/sec	135000 msgs/sec
Ethernet echo	318000 msgs/sec	**520000 msgs/sec**

TABLE I: Performance of user-space and kernel-space echo with UDP and Ethernet.

In controlled environments such as local-area networks (LAN), network application performance may also benefit from bypassing the TCP/IP stack, i.e., instead of having IP and UDP headers inside a Ethernet frame to differentiate between

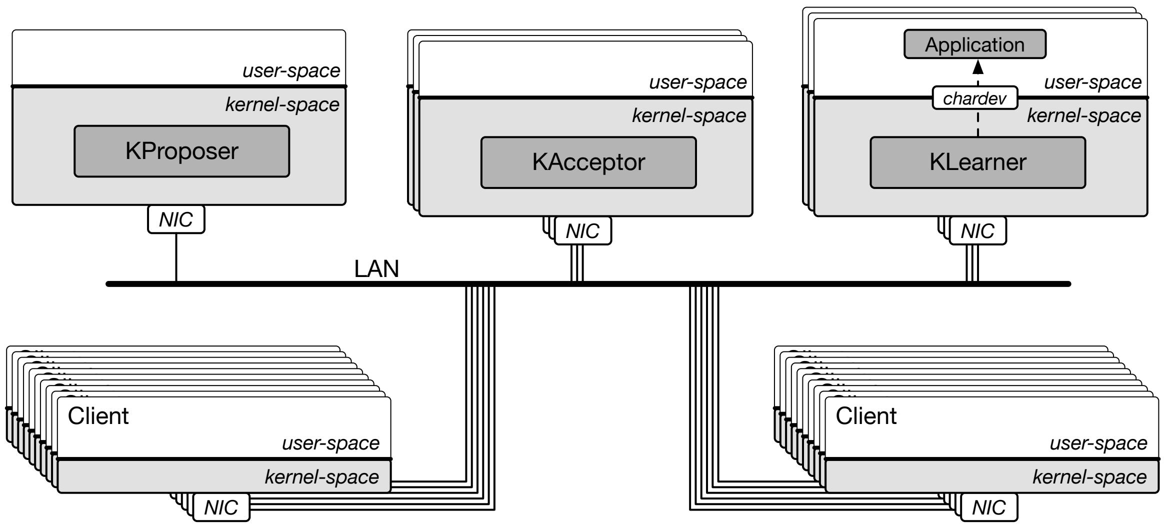


Fig. 2: Kernel Paxos architecture.

application addresses and ports, we use custom Ethernet types and send application data directly as the Ethernet frame payload. To assess the impact of this modification we extended the initial UDP echo application to send application data directly on top of Ethernet frames. The second line of Table I shows the improvement with respect to the UDP version for both user and kernel module versions.[1]

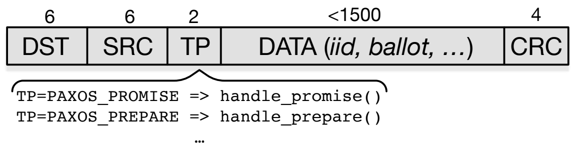
While a typical user-space UDP-based echo application can reach peak throughput at 120K messages per second, a kernel-based Ethernet-based system attains 520K messages per second, a 4x performance improvement. These results motivate Kernel Paxos. Moreover, as we show later in the paper, latency also largely benefits from such an approach.

B. *Kernel Paxos architecture*

Paxos relies on a leader for liveness. Therefore, typical Paxos implementations have their performance limited by what the leader can achieve. While some proposals extend Paxos with multi-leader or generalized solutions [11], [12], [13], we intend to reduce the overhead imposed by the communication layer and the operating system to improve the performance of classic Paxos. Fig. 2 shows the proposed architecture.

In Kernel Paxos, each one of the Paxos roles, proposer, acceptor and learner, can run either on the same machine (defined as a *replica* in such case) or deployed in independent nodes, as depicted in Fig. 2. Each role runs entirely in kernel space and communicates with the other Paxos players by means of message passing. For the reasons discussed earlier, we bypass the TCP/IP stack and exchange messages right on top of Ethernet frames.

The kernel adopts an event-driven model regarding the reception of Ethernet frames, i.e., the LKM must assign a callback function to each Ethernet type, as exhibited in Fig. 3. As soon as a new frame arrives, the kernel invokes the corresponding handler automatically. The great advantage of such an approach is that we do not require kernel threads that block while waiting for messages. In fact, each module is a single event-driven kernel thread and no synchronization is required with respect to the data accessed by registered handlers. In Kernel Paxos specifically, we statically associate each Paxos-related message, like a *promise* or a *prepare* message, to an Ethernet type and the corresponding handler function. This way, each Paxos player registers itself to receive the types its roles requires. For example, a KProposer registers to receive both *promise* and *accepted* messages representing the replies from acceptors for phases one and two, besides the *client_value* type for clients requests.



```
TP=PAXOS_PROMISE  => handle_promise()
TP=PAXOS_PREPARE  => handle_prepare()
                  ...
```

Fig. 3: Kernel Paxos event-driven approach for Ethernet types.

C. *Message flow in Kernel Paxos*

When a KProposer is loaded into the kernel, it contacts a quorum of KAcceptors and pre-executes the first phase for a configurable number of instances. Upon receipt of commands from clients, the KProposer uses such instances to send *accept* messages to the KAcceptors, which, under normal execution, reply with *accepted* messages to both KProposer and KLearners. A KLearner waits until it receives a quorum of such messages before it can deliver the command to the application in user-space. The application in the user-space receives the command from a KLearner reading from a character device, as depicted in Fig. 2. The application blocks until the command is ready and the KLearner has written

[1]We describe the hardware setup used in this evaluation in §VI. The source code used in these experiments is publicly available at https://github.com/paulo-coelho/kether.

it to the character device. Commands are delivered to the application in increasing instance order.

Both KProposer and KLearner take care of possible message losses. The KProposer resends first- and second-phase messages for timed-out instances. A KLearner restarts an instance i when it receives a quorum of second-phase messages for an instance $j > i$, but not for i. Restarting an instance i means the KLearner tries to propose a *NULL* value for instance i in order to learn the previous decided value (if any) or skip the instance otherwise. Clients and application are not aware of Kernel Paxos internals. Clients simply send requests in an Ethernet frame. From the application point of view, receiving a command is as simple as reading from a file.

V. Implementation

Kernel Paxos was implemented in C and the source code is publicly available.[2] Kernel Paxos borrows Paxos algorithm logic implementation from LibPaxos,[3] an open-source user-space Multi-Paxos library written in C as well. Each Paxos role comprises five components responsible for specific tasks. Fig. 4 shows the relation among such components.

The *network* component handles the creation of Ethernet frames with application specific payload and type to be sent as well as the unpacking of received frames and invocation of registered handlers.

The *storage* component is a general-purpose in-memory key-value store, used mainly to keep the acceptor state and "live" proposer and acceptor instances.

The *kpaxos* component consists of two sub-components: (a) a network-independent set of objects for each Paxos role and functions to update the state from received messages returning the replies to be sent over the network; (b) the set of handlers for each Ethernet type, which are responsible for (de)serializing messages and invoking the corresponding function in the other sub-component.

To keep statistical information about Kernel Paxos performance, a *statistics* component is also notified by *kpaxos* handlers every time a new message arrives. When the kernel module is unloaded the collected statistical data is persisted to disk.

The last component, *chardev*, manages the creation, reading, writing and destruction of character devices and represents the interface between Kernel Paxos and the actual replicated application. It is only present in a KLearner, which reads from this component to deliver commands to the application in increasing instance order.

VI. Performance

In this section, we describe the main motivations that guided the design of our experiments (§VI-A), detail the environment in which we conducted the experiments (§VI-B), and then present and discuss the results (§VI-C – §VI-G). We conclude with a summary of the main findings (§VI-H).

[2]https://github.com/esposem/Kernel_Paxos
[3]https://bitbucket.org/sciasciad/libpaxos

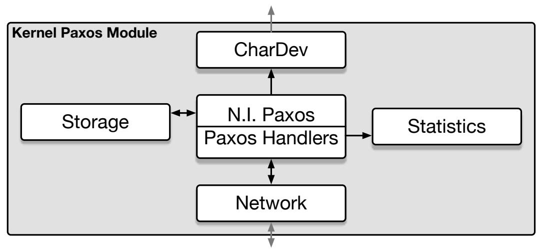


Fig. 4: Main components of Kernel Paxos module.

A. Evaluation rationale

All experiments are conducted in a local-area network (LAN). The LAN provides a controlled environment, where experiments can run in isolation. In all experiments, we have a single proposer representing the leader and three acceptors, deployed in different machines. To measure the performance of the ordering protocol, each client is also a learner. Thus, the latency of a command represents the time between the command has been sent by the client and delivered to the learner in the same process or kernel module.

We use a micro-benchmark with 64-byte and 1000-byte messages to evaluate particular scenarios in isolation. In this benchmark, we consider executions with a single client to understand the performance of Kernel Paxos without queueing effects and executions with multiple clients to evaluate the protocol under stress. We perform executions with multiple clients by controlling the number of outstanding messages of up to three learner processes (or kernel modules) in different machines.

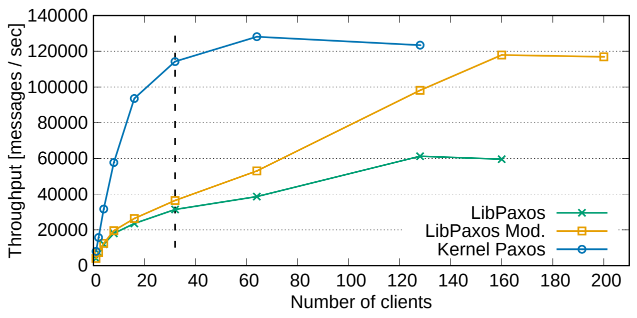
B. Environment

We compare Kernel Paxos to LibPaxos, a user-space library that provides the basic Paxos logic present in the "network-independent" part of our KPaxos component (see Fig. 4). Besides comparing with the original LibPaxos version, we have also implemented and assessed a modified version of LibPaxos that includes all the optimizations applied to Kernel Paxos and described in §III-A.

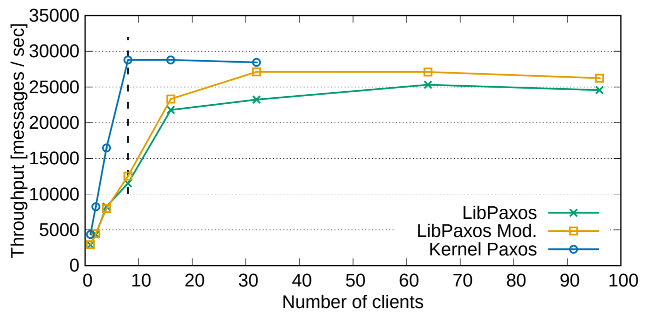
The LAN environment consists of a set of nodes, each one with an eight-core Intel Xeon L5420 processor working at 2.5GHz, 8GB of memory, SATA SSD disks, and 1Gbps ethernet card. Each node runs CentOS 7.1 64 bits. The round-trip time (RTT) between nodes in the cluster is around 0.1ms. We have also assessed Kernel Paxos on nodes with 10Gbps ethernet card, where the RTT is around 0.04ms. The experiments in the 1Gbps nodes compare Kernel Paxos to LibPaxos, while the experiments in the 10Gbps nodes assess the performance of Kernel Paxos in two different configurations.

C. Throughput in a LAN

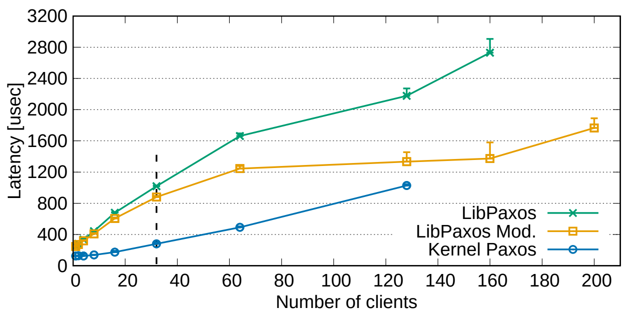
These experiments measure the throughput of standard LibPaxos, improved LibPaxos, and Kernel Paxos as we increase the number of clients in scenarios with 64-byte and 1000-byte messages, as depicted in Fig. 5(a) and Fig. 5(b), respectively. In summary, Kernel Paxos outperforms both the original



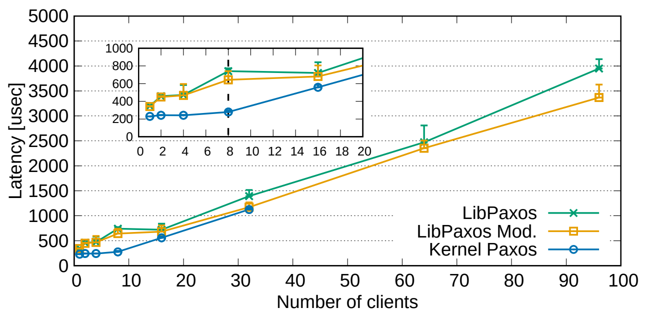
(a)



(b)



(c)



(d)

Fig. 5: Throughput and median latency for increasing number of clients with 64-byte (left) and 1000-byte messages (right). Whiskers represent the 95-th percentile.

LibPaxos and the modified version for any number of clients and message size.

1) Performance with small (64-byte) messages: With a single client, Kernel Paxos is twice as fast as LibPaxos with throughput around 8000 msgs/sec. The difference increases even more as the number of clients is augmented, with Kernel Paxos throughput close to 100000 msgs/sec with 16 clients, 4x better than both versions of LibPaxos. The modification to LibPaxos starts paying off with more clients. The performance of the modified version with 128 clients is close to 100000 msgs/sec while the original library already saturates the proposer and cannot go beyond 60000 msgs/sec. The maximum performance of Kernel Paxos, LibPaxos and LibPaxos Modified is 128000, 61000 and 117000 msgs/sec with 64, 128 and 160 clients, respectively. The maximum represents the point where the *proposer* is overloaded, i.e., its CPU reaches 100%.

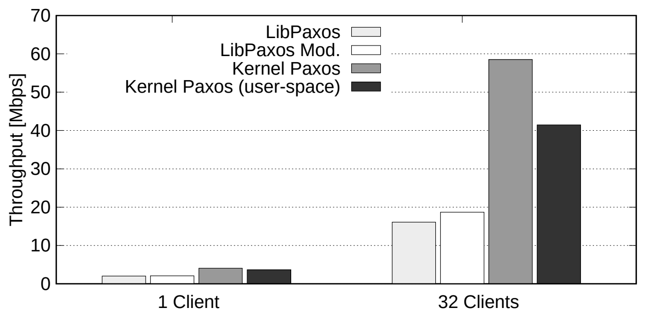
2) Perfomance with large (1000-byte) messages: With a single client Kernel Paxos is still faster than LibPaxos with throughput around 4500 msgs/sec against less than 3000 msgs/sec for LibPaxos. In this setup, larger messages saturate the network before *proposers* are overloaded. For such reason, both versions of LibPaxos have similar performance and reach a maximum of 25000 and 27000 msgs/sec for the original and the modified version. Kernel Paxos can still push close to 29000 msgs/sec. For 1000-byte messages, this represents a throughput of around 235 Mbps in the clients, amounting to an aggregated throughput in the proposer of 940 Mbps: 235 Mbps from the clients and 3×235 Mbps from the acceptors. This value was confirmed with *iptraf* on the proposer machine.

D. Latency in a LAN

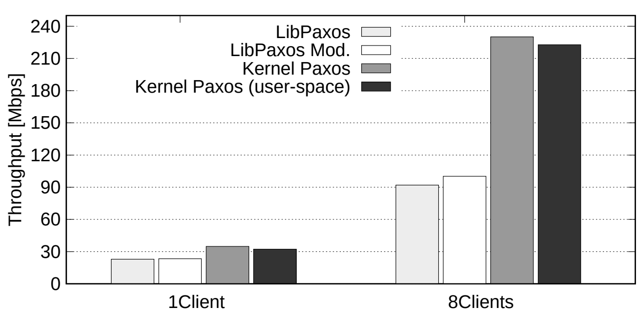
The overall better performance of Kernel Paxos in terms of throughput repeats for latency. Fig. 5(c) and Fig. 5(d) exhibits the results for small and large client messages.

1) Latency with small (64-byte) messages: With a single client, the latency of Kernel Paxos is very close to 1 round-trip time (RTT) and as low as 124 microseconds. LibPaxos, on the contrary, has a median latency of 248 and 252 microseconds for the modified and original versions, what represents more than 2 RTTs. The modifications in LibPaxos have direct positive impact on the latency. With 160 clients, the modified version has a median latency of around 1400 microseconds against almost twice the value for the original one with 2700 microseconds. In all implementations the measured latencies are quite stable with very low variance around the median values. With the maximum throughput, Kernel Paxos' latency is still below 500 microseconds with 64 clients, 3x to 5x less than the values for LibPaxos.

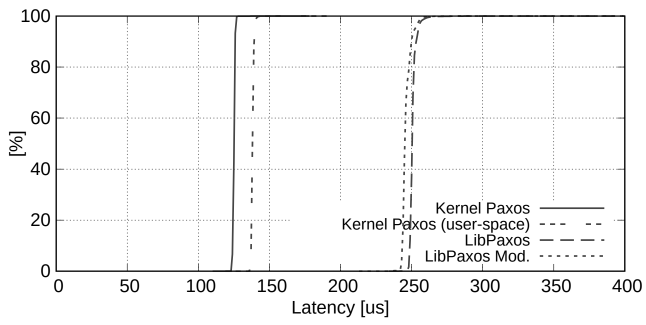
2) Latency with large (1000-byte) messages: With bigger messages, the latency for all protocols increase. With a single client, messages need close to 350 microseconds to be ordered with LibPaxos and 230 microseconds with Kernel Paxos. At maximum throughput, Kernel Paxos can keep a latency as low as 279 microseconds while LibPaxos needs 8x and 9x more time to deliver a message for the modified and original



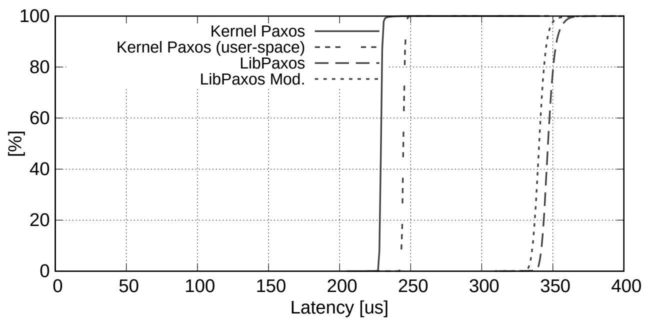
(a)



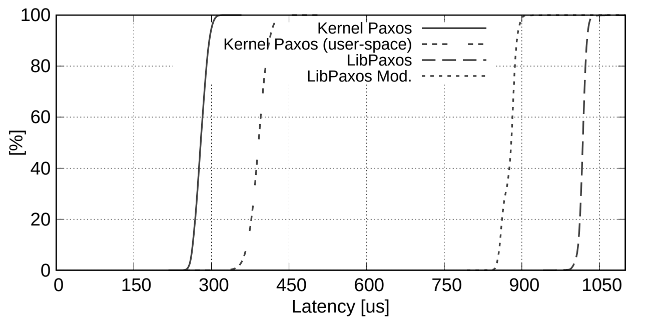
(b)



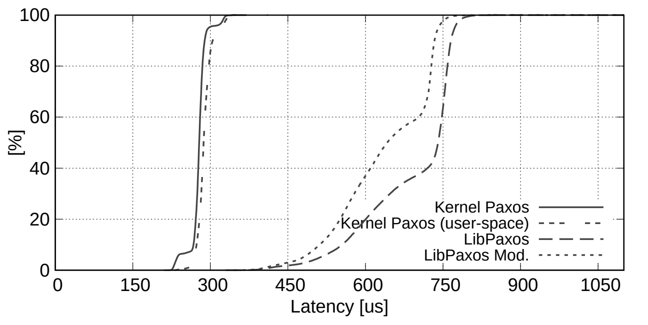
(c) 1 client.



(d) 1 client.



(e) 32 clients.



(f) 8 clients.

Fig. 6: Throughput and latency CDF with the selected number of clients for 64-byte (left) and 1000-byte messages (right).

library, respectively. Kernel Paxos has smaller latency, higher throughput and needs only 8 clients to saturate the data link, while LibPaxos needs at least 32 clients.

E. Performance with similar number of clients

As observed in Fig. 5, depending on the message size we can saturate either the proposer's CPU or the communication links. In such cases, the maximum throughput of Kernel Paxos is only 10% superior to the modified version of LibPaxos in the same setup.

We now compare the various protocols in scenarios with the same number of clients. We have chosen two specific scenarios from the previous experiments and zoomed in on the behavior of the protocols. These scenarios represent (a) the experiment with a single client and thus no contention or queue effects, to isolate the benefits of eliminating context switches and TCP/IP stack; and (b) the performance under stress, specifically with 32 clients for 64-byte massages and 8 clients for 1000-byte messages, since none of the implementations are saturated at these points. The dashed lines in Fig. 5 highlight the selected points.

Besides, we also evaluate the behavior of Kernel Paxos with client/learner in user-space sending messages to the proposer using raw sockets and learning decided values from the character device created by the kernel-space learner.

1) Throughput and latency with small (64-byte) messages: Fig. 6(a) exhibits he throughput advantage introduced by Kernel Paxos. With a single client, Kernel Paxos reduction in communication overhead and context switches is enough to double the throughput. With 32 clients, Kernel Paxos can reach 60 Mbps, almost 4x the performance of the original library and 3x the performance of the modified version.

Besides, the Cumulative Distribution Function (CDF) with 1 client shown in Fig. 6(c) for Kernel Paxos is quite stable with the 99.9-th percentile equal to 128 microseconds while the median is 126 microseconds. These values represent at least half of the latencies for the user-space protocols.

We observe in Fig. 6(e) a similar behaviour with 32 clients, where the CDF is still stable in Kernel Paxos going from 280 to 305 microseconds in the 50-th and 95-th percentile, respectively. In the user-space library, the latency is at least 3x greater.

Another interesting result regards the difference between Kernel Paxos entirely in the kernel and the deployment with client and application in user-space. The throughput and latency distribution show that, although there is a small overhead in the latter deployment, Kernel Paxos is still 2x to 3x times better than LibPaxos.

2) Throughput and latency with large (1000-byte) messages: Fig. 6(b) shows the throughput for this scenario. With a single client, Kernel Paxos reaches a throughput 50% greater than LibPaxos, around 35 Mbps. With only 8 clients, Kernel Paxos almost saturated the proposer communication link: with 232 Mbps in the client, the proposer aggregated throughput is around 930 Mbps. LibPaxos, on the other hand, uses only 10% of the communication link capacity, since performance is capped by the processing power of the proposer.

Regarding the latency, both LibPaxos and Kernel Paxos are very stable for a single client, with latencies close to 340 and 230 microseconds, respectively, as shown in Fig. 6(d). With 8 clients, however, LibPaxos latency has a large variation, from 500 microseconds in the 5-th percentile to almost 800 microseconds in the 95-th percentile. Kernel Paxos remains quite stable with values between 220 and 295 microseconds for the same percentiles as exhibited in Fig. 6(f).

For large messages, the difference of having a deployment of Kernel Paxos with client and application in the user-space is minimal for both throughput and latency.

F. Context-switch overhead

Since LibPaxos is a user-space library, sending and receiving data imply invoking a system call to context-switch to kernel-space and access the network card, followed by another context-switch back to user-space when the system call returns.

Kernel Paxos runs entirely in kernel-space and the context-switch overhead at the proposer and acceptors is completely eliminated, contributing to improve the overall performance.

To quantify the difference, we used *dstat* to monitor the number of context-switches for the experiments with 1000-byte messages. Table II exhibits the approximate numbers for both libraries as the number of clients increase.

Clients	*LibPaxos*	*Kernel Paxos*
1	7000	150
2	9000	150
4	12000	150
8	12000	150

TABLE II: Approximate number of context-switches with increasing number of clients

G. Kernel Paxos in a 10Gbps network

The last experiments evaluates Kernel Paxos in a 10Gbps network and compares the results with the 1Gbps network. The 10Gbos setup represents environments typically used by hardware-based consensus solutions, which rely on high-throughput network cards. The experiments assess Kernel Paxos with small and large messages while increasing number of clients.

1) Throughput and latency with small (64-byte) messages: Fig. 7(a) and 7(c) show throughput and latency in both networks. The throughput for a single client in the faster network is more than 2x greater, close to 19000 msgs/sec or 10 Mbps. The median latencies are 52 and 126 microseconds for the 10Gbps and 1Gbps network, respectively, representing roughly the measure of one RTT.

The maximum throughput is 170000 msgs/sec (87 Mbps) for the 10Gbps network against 129000 msgs/sec (66 Mbps). In both cases the proposer is CPU bound and the difference in this case is mainly due to the better CPU in the nodes with 10Gbps NIC.

The CDF in Fig. 7(e) confirms that although the values are smaller for the 10Gbps network, the slower network has a much more stable curve.

2) Throughput and latency with large (1000-byte) messages: With 1000-byte messages, we expect to saturate the communication links before the proposer is CPU bound. As demonstrated in Fig. 5(b) and now in Fig. 7(b), the 1Gbps link is saturated in the proposer when the client reaches around 29000 msgs/sec or 240 Mbps. The 10Gbps link, on the other hand, has enough bandwidth to go beyond 1Gbps in the client, and close to 4.5Gbps in the proposer, which differently from the slower network, is saturated at around 130000 msgs/sec.

The median latencies are 83 and 230 microseconds with a single client and 224 and 279 microseconds with 8 and 32 clients for the 10Gbps and 1Gbps networks, as seen in Fig. 7(d).

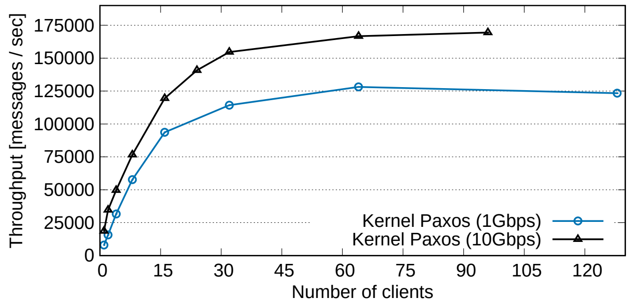
The CDF shown in Fig. 7(f) reinforces the overall stability in the 1Gbps network, which we believe is due to lower load in the proposer, since the 1Gbps link is not enough to overload it in this configuration.

H. Summary

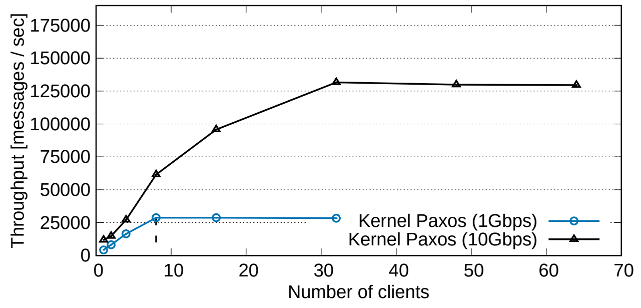
We now draw some general conclusions from our experimental evaluation:

- Kernel Paxos outperforms LibPaxos, both original and improved versions, in each and every experiment, with small and large messages, and presents superior throughput and smaller latency for the same number of clients.
- In setups with large messages (1000 bytes), the evaluated libraries are able to saturate the proposer's data link in a 1Gbps network.
- In a 10Gbps network, Kernel Paxos can deliver commands at a rate superior to 1Gbps, representing 130000 1000-byte messages per second.

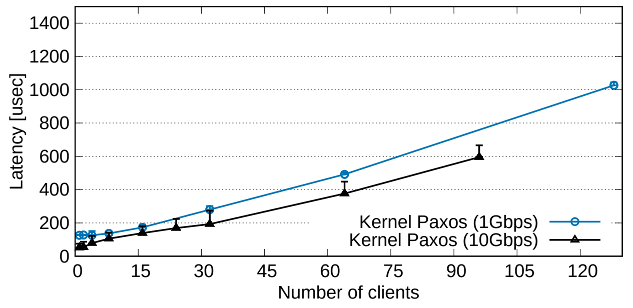
The last two observations suggest that batching could be quite effective in further boosting Kernel Paxos performance



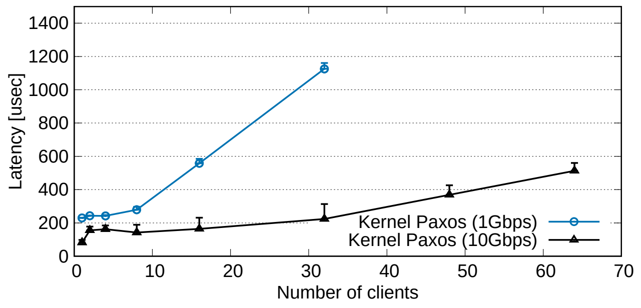
(a)



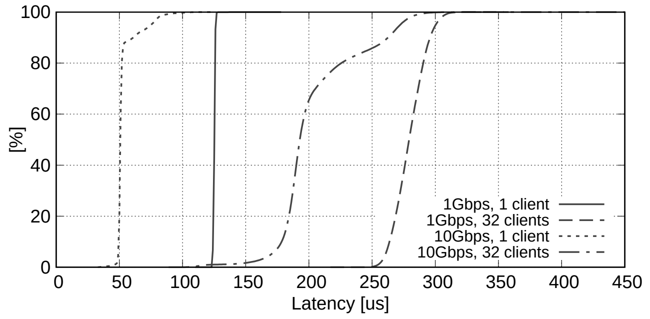
(b)



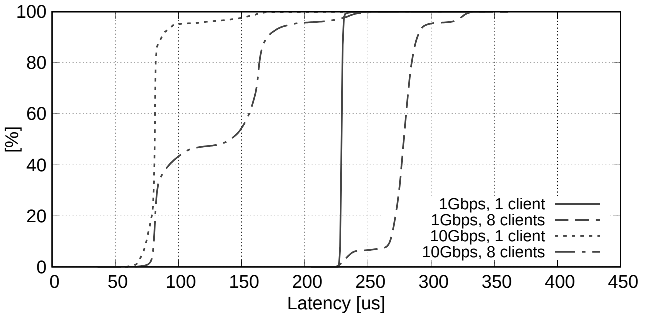
(c)



(d)



(e)



(f)

Fig. 7: Kernel Paxos performance in 1Gbps and 10Gbps setups for increasing number of clients with 64-byte (left) and 1000-byte messages (right). Whiskers represent the 95-th percentile.

at the cost of small increase in latency. For example, if the KProposer would batch 10 small client requests (i.e., 100 bytes) in 1000-byte messages, Kernel Paxos could potentially reach one million ordered messages per second.

VII. Related work

Improving the performance of Paxos has been a hot research topic. In the following, we consider related proposals from three perspectives: those that take advantage of special network topologies (i.e., overlays); those that resort to specialized hardware; and those that exploit the semantics of applications.

A. *Protocols that exploit special topologies*

Several approaches modify the Paxos [5] protocol to improve performance. In [7], the authors proved that ring topologies allow systems to achieve optimal throughput. Some protocols that benefit from such topologies are LCR [7], Spread [16] and Ring Paxos [8]. LCR arranges processes in a ring and uses vector clocks to ensure total order. Spread, which is based on Totem [17], relies on daemons interconnected as a ring to order messages, while message payloads are disseminated using IP-multicast. Finally, Ring-Paxos deploys Paxos processes in a ring to maximize throughput. A problem of all such ring-based protocols is that their latency is proportional to the size of the system times the network point-to-point latency.

B. *Protocols that exploit special hardware*

Some solutions provide ordering as a network service. Speculative Paxos [18] and NOPaxos [9] use a combination of techniques to eliminate packet reordering in a data center, including IP multicast, fixed-length network topologies, and a single top-of-rack switch acting as a serializer (although in case of NOPaxos a Linux server can act as a sequencer at the cost of higher latency). NetPaxos [10] deploys acceptors and coordinator in switches. Although this reduces the latency, it makes the recovery more complex: switches have limited memory, making it hard to keep the state of acceptors. Besides, replacing a failed coordinator is not trivial in such configuration. In [19], the authors bring the consensus logic to the hardware, implementing Zookeeper [20] in FPGA. Even though the throughput is quite high, such programmable hardwares are not suitable to store large amount of replicated state [9]. While Kernel Paxos cannot achieve the performance of hardware implementations for state machine replication protocols, we can still provide a considerably high throughput without depending on any network ordering guarantees or specialized hardware. Kernel Paxos can virtually run on any machine with commodity hardware and a Linux operating system.

C. *Protocols that exploit message semantics*

Several Paxos variants yield better performance with a multi-leader or generalized approach. EPaxos [11] improves on traditional Paxos [5] and reduces the leader overload by allowing any replica to order commands. As long as replicas observe common dependencies set, commands can be ordered in one round-trip. M2Paxos [12] is a implementation of Generalized Consensus [13] that allows different ordering of non-conflicting commands. It detects conflicts by looking into the objects the commands access and making sure a replica has exclusive access to such objects. M2Paxos guarantees that commands that access the same objects are ordered by the same replica. Any of such solutions could benefit from Kernel Paxos strategies to reduce latency and increase performance, i.e., the protocols could be moved to the Linux kernel as module and use raw Ethernet frames to exchange messages.

VIII. Conclusion

State machine replication is a fundamental technique in the design of highly available strongly consistent distributed systems. At the core of state machine replication, there is the consensus problem. While many protocols have been proposed to implement consensus, Paxos stands out. Therefore, much effort has been placed on implementing Paxos efficiently. This paper proposes Kernel Paxos, a different approach to improving the performance of Paxos. Differently from previous proposals, Kernel Paxos does not rely on special network topologies, specialized hardware or application semantics. Kernel Paxos addresses the performance challenge by reducing the communication overhead in two directions: (a) eliminating context switches by placing the protocol implementation in the Linux kernel, and (b) bypassing the TCP/IP stack and exchanging protocol messages inside raw Ethernet frames. We assessed the performance of Kernel Paxos and compared to user-space implementations. The results show that in most configurations of interest Kernel Paxos largely outperforms user-space protocols.

Acknowledgements

We thank the reviewers for the constructive suggestions. This work is supported in part by the Swiss Government Excellence Scholarships, and Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq (GDE Project 204558/2014-0).

References

[1] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *CACM*, vol. 21, pp. 558–565, July 1978.
[2] F. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.
[3] M. J. Fischer, N. A. Lynch, and M. S. Patterson, “Impossibility of distributed consensus with one faulty process,” *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.
[4] T. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.
[5] L. Lamport, “The part-time parliament,” *Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.
[6] L. Lamport, “Lower bounds for asynchronous consensus,” *Distributed Computing*, vol. 19, no. 2, pp. 104–125, 2006.
[7] R. Guerraoui and M. Vukolić, “Refined quorum systems,” *Distributed Computing*, vol. 23, no. 1, pp. 1–42, 2010.
[8] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, “Ring paxos: A high-throughput atomic broadcast protocol,” in *DSN*, 2010.
[9] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, “Just say no to paxos overhead: Replacing consensus with network ordering.,” in *OSDI*, 2016.
[10] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, “Netpaxos: Consensus at network speed,” in *SOSR*, ACM, 2015.
[11] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is more consensus in egalitarian parliaments,” in *SOSP*, 2013.
[12] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, “Making fast consensus generally faster,” in *DSN*, 2016.
[13] L. Lamport, “Generalized consensus and paxos,” tech. rep., Technical Report MSR-TR-2005-33, Microsoft Research, 2005.
[14] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *Trans. on Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990.
[15] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: an engineering perspective,” in *PODC*, 2007.
[16] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, “The spread toolkit: Architecture and performance,” *Johns Hopkins University, Tech. Rep. CNDS-2004-1*, 2004.
[17] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, “The totem single-ring ordering and membership protocol,” *ACM Transactions on Computer Systems (TOCS)*, vol. 13, no. 4, pp. 311–342, 1995.
[18] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, “Designing distributed systems using approximate synchrony in data center networks.,” in *NSDI*, 2015.
[19] Z. István, D. Sidler, G. Alonso, and M. Vukolic, “Consensus in a box: Inexpensive coordination in hardware.,” in *NSDI*, 2016.
[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for internet-scale systems.,” in *USENIX ATC*, 2010.