Achieving high-throughput State Machine Replication in multi-core systems

Nuno Santos, André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
Email: firstname.lastname@epfl.ch

***Abstract*—State machine replication is becoming an increasingly popular technique among online services to ensure fault-tolerance using commodity hardware. This has led to a renewed interest in its throughput, as these services have typically a large number of users. Recent work has shown how to improve throughput of the ordering phase by improving the replication protocol, using techniques like Ring-topologies, IP multicast, and rotating leaders. The resulting systems, when deployed in modern fast networks, achieve unprecedented levels of throughput. But these systems are increasingly becoming limited by the CPU of the replicas, especially with small client requests. However, the problem is not lack of performance of the CPUs, but instead the inability of typical implementations to effectively use the multiple cores of modern multi-core CPUs. In this work, we show how to architect a replicated state machine whose performance scales with the number of cores. We do so by applying several good practices of concurrent programming to the specific case of state machine replication, including staged execution, workload partitioning, actors, and non-blocking data structures. We describe and test a Java prototype of our architecture, based on the Paxos protocol. With a workload consisting of small requests, we achieve a 6 times improvement in throughput using 8 cores. More generally, in all our experiments we have consistently reached the limits of network subsystem by using up to 12 cores, and do not observe any degradation when using up to 24 cores. Furthermore, the profiling results of our implementation show that even at peak throughput contention between threads is minimal, suggesting that the throughput would continue scaling given a faster network.**

I. Introduction

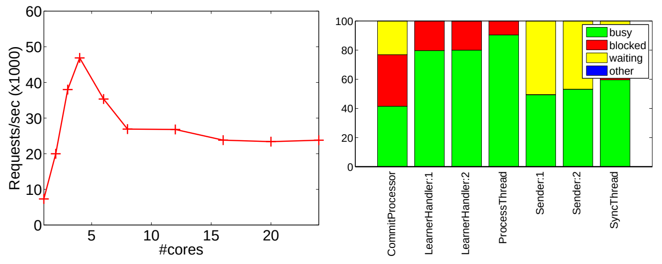
State machine replication is an effective mechanism for achieving fault-tolerance by replication on commodity hardware. For this reason, it is frequently used by online services as a low-cost solution to achieve reliability and availability. However, deploying state machine replication in such a context creates new challenges to the design and implementation of such services. In particular, online services are exposed to a very large number of potential users, and therefore must be engineered for high-throughput.

This raises the question of whether state machine replication can support the required levels of throughput. If the service being replicated is itself expensive, then this is a moot point, as the system will be limited by the performance of the service. But often the services are lightweight, like lock servers [1], and coordination services [2], and can sustain a very high-throughput, provided that the underlying state machine replication layer can also sustain it. Additionally, when multiple services within the same data center must be fault-tolerant, it is often easier to delegate ordering to a single, high-performance ordering service service instead of having every service implement its own ordering layer [3]. As this shared ordering service is potentially used by a large number of other services, its throughput is critical.

Recently there has been a renewed interest in improving the throughput of the ordering phase. For instance, [4], [5] show how to achieve high-throughput with algorithmic improvements to the replication protocol. These protocols use the network very highly efficiently, relying on techniques like ring topologies and IP multicast to achieve an efficiency of over 90% on Gigabit Ethernet, as measured by the amount of data ordered over the bandwidth of the network. However, this high efficiency is achieved only if requests are large enough, usually $\geq$8KB. For smaller request sizes (<8KB) they become CPU-bounded and their efficiency drops significantly [4]. A similar CPU bottleneck with small request sizes has been identified in many other systems, including in production-quality implementations. For instance, in [2] the authors report that the atomic broadcast protocol at the core of ZooKeeper becomes CPU-bounded at the leader process when running with a workload 1KB write requests. As small request sizes are common in practice (*e.g.*, coordination services, lock servers), the CPU bottleneck is a significant limitation to achieving higher throughput.

It is worth looking more carefully at the reasons behind this CPU bottleneck. In recent years the single-thread performance of CPUs improved only marginally, while the number of cores increased greatly. A modern CPU for servers contains anywhere between 4 and 16 cores, with even higher numbers in the horizon. Although each cores may have a relatively modest single-thread performance, their aggregate performance is considerable. In the case of state machine replication, current implementations are mostly unable to take full advantage of multi-core CPUs, thereby being limited by the single-thread performance.

As an example, consider Figure 1a, which shows the throughput of ZooKeeper with increasing number of cores. Although ZooKeeper scales well up to 4 cores, reaching a peak of 50K requests per second, its performance degrades as the number of cores increases, achieving less than 30K requests per second when all 24 cores are used. The per-thread profiling results with 24 cores (Figure 1b) shed some light on the cause of this poor behavior. For each thread, we show the time spent



(a) Throughput

(b) Per-thread CPU utilization at the leader, when using 24 cores.

Fig. 1. Performance of ZooKeeper with increasing number of cores. Ensemble of 3 replicas, 128 bytes write-request (`setData()` on a ephemeral node). ZooKeeper 3.3.3 with the default configuration except that the leader does not serve clients and logs and snaphsot directories were mapped to a RAM drive (`/dev/shm`). More details and experiments in [6].

executing (*busy*), blocked trying to acquire a lock (*blocked*), waiting to receive work (*waiting*), and in other states (*other*, see Section VI-B for a detailed explanation). One problem is a high level of contention, as shown by the large fraction of time the threads spend blocked. Another problem is poor load balancing, as several threads spend 100% of their time either busy or blocked, creating a single-thread bottleneck.

We believe this is typical of many replicated state machine (RSM) implementations. The traditional threading architecture used by RSMs is based on an event-driven model, with a event loop (thread) doing most of the work (with maybe the exception of some IO operations). There are good reasons why RSMs implementations use this design. First, it matches closely the way replication protocols are typically expressed, which is as a set of event handlers. It also simplifies thread coordination, trivially preventing race conditions by not sharing data among threads. This is especially important, because RSMs have a complex internal state which is shared by many different internal tasks (*e.g.*, ordering, retransmission, failure detection, snapshotting, and state transfer). Finally, before the multi-core era, a single-thread event-driven design was a good choice, as it avoided the cost of context switches and concurrency control. But this traditional architecture is reaching its limits and must be revised, in order to reach the full potential of modern Gigabit networks, multi-core CPUs and of the new generation of highly efficiency replication protocols.

In this paper we show how to implement RSMs such that their performance scales with the number of cores. For this, we assume that the workload and the system are such that the bottleneck is the single-thread performance of the CPU, which is often the case with small request sizes and fast networks (Gigabit or more). In order to scale the performance with the number of cores, we must ensure that 1) the tasks performed by a replica are evenly balanced among threads, so that no replica becomes limited by the single-thread performance of its CPU, and 2) threads make progress mostly independent of others, with minimal time wasted in coordination (*e.g.*, contention).

Some of the tasks performed by a replica, *e.g.*, I/O, are fairly easy to parallelize, But other tasks, like the execution of the ordering protocol and the various housekeeping operations, pose a much greater challenge, either because they are inherently serial or because their state is complex and shared between several tasks. A naive separation into threads is likely to make extensive use of locks, increasing contention, limiting concurrency, and being prone to race conditions and deadlocks. We propose an architecture that avoids these problems, by grouping tasks into threading modules according to carefully chosen boundaries that minimize shared state and contention.

We use a hybrid design, with a mixture of event-driven and thread-based modules. This design draws inspiration from SEDA [7] and from the concept of Actors in languages like Erlang and Scala [8]. Each module consists of private state, one or more threads, and a well-defined interface for communicating with threads from other modules (usually through message queues). With few exceptions, the private state is accessible only by the threads managed by the module. This organization keeps complex state isolated inside a module, with well-defined access points, simplifying reasoning about thread-safety and parallelism.

We have implemented this architecture in JPaxos [9], a fully-functional implementation in Java of state machine replication based on the Paxos protocol. We performed an experimental study using a CPU-intensive workload on a cluster of 24-core machines connected by a Gigabit Ethernet.

The results show that the throughput increases with the number of cores, either linearly or very close to linearly, until reaching the limit of the network subsystem, which happens when using between 8 to 12 cores. Furthermore, the profiling results of our implementation show that, even at peak throughput, contention between threads is minimal, suggesting that performance would continue scaling in the absence of non-threading related bottlenecks. This shows that even with a demanding workload, the throughput of state machine replication can be further improved by leveraging multi-core CPUs.

To summarize, our main contributions are:

- Show that further improvements in throughput of state machine replication require implementations capable of exploiting the potential of multi-core CPUs.
- A multi-core scalable design for RSM.
- An experimental study showing the scalability of this architecture in multi-core CPUs.

The remainder of the paper is organized as follows. Section II discusses the related work, Section III provides the background by describing how state machine replication works and how it is typically implemented, Section IV discusses the challenges in parallelizing RSM implementations and describes our general approach, Section V describes in detail the scalable threading architecture we propose, Section VI presents the experimental results, and Section VII concludes the paper.

II. Related Work

The interest in state machine replication has been increasing in the last years, not only in academia [4], [5], [10] but also in industry. In the latter case, some of the noteworthy examples are the Chubby lock-server [1] and the ZooKeeper coordination service [2].

Recently, several works have focused on the performance of state machine replication. Mencius [10] proposes a rotating leader protocol that performs well in WANs. In [3], the authors propose a protocol based on similar ideas as Mencius but adapted to the LAN scenario, which improves throughput by distributing over multiple replicas the load that usually is concentrated on the leader. LCR [5] and Ring-Paxos [4] focus instead on achieving high network efficiency on fast Gigabit LANs. Zab [2], the atomic broadcast protocol at the core of ZooKeeper, is a modified version of Paxos with focus on high-performance. However, all these works are concerned only with improving the replication protocol, and do not address the potential CPU bottleneck. And in fact, some of results presented in [4] and [2] show the implementations of these protocols reach the single-thread performance bottleneck of the CPU with workloads consisting of small requests. Our work complements the algorithmic improvements proposed in these papers, by showing how to leverage multi-core CPUs to address the CPU bottleneck.

Outside the field of state machine replication, there is a rich body of work on tools and techniques to exploit the parallelism available on multi-CPU/multi-core systems. Our work combines ideas from SEDA [7] and from the notion of Actors from languages like Erlang and Scala [8]. SEDA is an architecture for highly-scalable servers consisting of interconnected event-driven stages, each implemented by one or more threads, and using message queues to communicate asynchronously. Actors can be regarded as an extension of the concept of data encapsulation to threading: an Actor encapsulates both data and threading, forbidding explicit sharing of state and communicating with other Actors only by means of message passing. Like in SEDA, our architecture is partly organized as a set of stages, but we also use thread-based modules. Like Actors, we tried to encapsulate threading within a module using message passing, but deviated from this rule where needed to achieve performance.

III. Background

This section gives an overview of how state machine replication works. Then it briefly describes the main tasks performed internally by a replica and presents the generic design of an implementation of a Replicated State Machine. This generic design is the basis of the threading architecture proposed in the following section.

A. State Machine Replication

A RSM makes a (deterministic) service fault-tolerant by replicating it on several replicas, and ensuring that every replica executes the same sequence of requests. This ensures that the state of the replicas remains consistent and available, in spite of a (limited) number of faults.

The key component at the heart of a RSM is the ordering protocol used to ensure agreement on the sequence of requests, which in most cases is one of the many variants of Paxos [11]. Paxos is a leader-based consensus protocol that tolerates the crash of up to f replicas by using $n \geq 2f + 1$ replicas. Here, we give only a rough overview of the protocol, as the details are not important for our work. Using a leader election protocol, one of the replicas assumes the role of *leader*, thereby becoming responsible for coordinating the other replicas. The leader orders requests received from the clients by executing a series of ballots. In each ballot, the leader assigns to one or more client requests a tentative sequence number and proposes it to the other replicas. Once a majority of replicas receive and acknowledge this proposal, the request is permanently assigned to the given sequence number.

Two optimizations that greatly improve the performance of Paxos are batching and pipelining [12]. *Batching* consists of grouping several client requests in the same ballot, while *pipelining* consists of executing several ballots concurrently. As these optimizations are common practice, in the following we will assume they are used.

B. Processing a request

We now look at the tasks performed internally by a RSM. When a replica receives a request from a client, it first queries a cache of previously executed requests to check if the request was already executed. If so, the replica sends the previously computed reply back to the client. Otherwise, it initiates ordering of the request by adding it to a batch. Once the batch is ready, either because it reached the maximum size or its timeout expired, the replica initiates ordering of the batch. This involves exchanging several messages with the other replicas according to the replication protocol, until enough messages are received to decide the order of the batch. At this point the requests inside the batch are assigned a final order, executed sequentially, and the replies sent to the appropriate client.

C. Generic design of an RSM

Although implementations of replicated state machines differ in many aspects, they are usually organized around the same set of modules, whose functionality and state are roughly equivalent across implementations. We will therefore present our work in the context of such typical design (see [13] for an example). However, the threading architecture we propose and the underlying guiding principles can be easily adapted to other designs.

An implementation of a RSM consists roughly of four modules: ClientIO, ReplicaIO, ReplicationCore, and Service-Manager (Figure 2).

The *ClientIO* module manages the communication with the clients, which is usually done using TCP connections. Its main tasks are accepting new connections, receiving requests and sending replies, and its state consists of the connection information (sockets) and I/O buffers with partly read/written

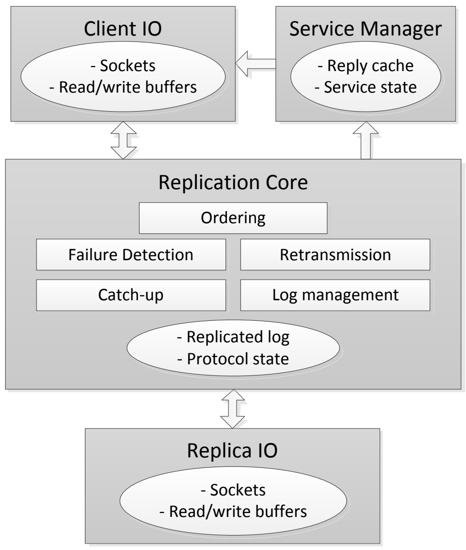


Fig. 2. Main modules of a Replicated State Machine

packets. To achieve high-throughput even with small requests, this module must be designed to handle thousands of connections and up to hundreds of thousands of small messages per second.

The *ReplicaIO* is similar to the ClientIO module, managing the communication with the other replicas. However, it must be designed for a very different workload, *i.e.*, for a small number of connections (one for every other replica), each transmitting a high amount of data. And contrary to the ClientIO module, the size of the messages exchanged with other replicas is partly under the control of the batching policy of the RSM, which can choose a size that provides good bulk throughput.

The *ReplicationCore* executes the ordering protocol and all the auxiliary services, like failure detection, log management, message retransmission and catch-up/state transfer. Its state consists of the replicated log containing the information on every known instance of the ordering protocol, and a few other control variables.

Finally, the *ServiceManager* module receives the ordered sequence of requests, executes them on the service, and sends the reply to the clients. Apart from the state of the service, this module may manage a reply cache (used to ensure at-most-once execution of requests) and some additional information to manage snapshots.

IV. Challenges and Design principles

Our goal is to design a threading architecture whose performance scales with the number of cores. This goal requires first *finding and exploiting parallelism* among the tasks performed by a RSM. The difficulty of this depends greatly on the nature of the particular module of the RSM, which can range from embarrassingly parallel to inherently serial. Furthermore, scaling the performance requires good *load balancing among threads*, to avoid single-thread performance bottlenecks and ensure that all threads are able to make progress concurrently. Once again, this is easily done inside the modules with homogeneous workload, like I/O, but difficult to do across heterogeneous modules. Finally, to *ensure correctness* threads must coordinate when accessing shared state. This is a hard problem, susceptible to several classes of errors that can lead to safety violations (race conditions), to liveness violations (deadlock and livelock), or to poor performance (contention).

Our design draws inspiration from the architecture proposed by SEDA [7], and from the concept of Actors in languages like Scala and Erlang. It consists of a set of modules, with some of them forming a pipeline used to process and order requests, and the others providing auxiliary services.

Like Actors, each module encapsulates both state and threading, and the primary means of communication between threads in different modules are message queues. However, a strict enforcement of this rule would at some places either harm performance or scalability, or result in an unnecessarily complex design. Therefore, we have allowed some carefully designed exceptions where threads access state directly in other modules.

Modules have one or more threads, and may be either event-driven or thread-based. Note that single-threaded modules with only private state are naturally thread-safe. Multi-threaded modules, however, must use either locks or state partitioning to protect the internal state. But this is an easier problem than in a monolithic design because the module provides boundaries to the shared state and threading.

This organization has several advantages for RSMs implementations. In addition to the well-known advantages of state and thread encapsulation, it also allows each module to have its own design. This is key in scaling the performance of RSMs with the number of cores while keeping complexity under control, since the many tasks performed by the implementation of a RSM differ substantially in their structure, so that a single homogeneous design would not provide the best results. The critical factors that guided our design of the modules are the potential for parallelism, the complexity of the state and of the operations performed, the frequency and complexity of interactions with other tasks, and the nature of the task (sequential or event-handlers). The choice of the appropriate design involves a series of tradeoffs between these factors.

As the CPU-intensive tasks have the greatest potential for parallelism, they should be made into multi-threaded modules whenever possible. This is easy in some cases; the I/O tasks which are embarrassingly parallel in nature. However, the ReplicationCore and ServiceManager modules pose a greater challenge; although they are potential single-thread bottlenecks, they are hard to divide into independent tasks executing concurrently because of their complex state and inter-dependencies. In these cases, we have used the natural single-threaded, event-driven implementation for the core tasks of the module, while offloading as much work as possible from the main event-dispatch thread to auxiliary threads. For the modules that have little or no potential for parallelism (*e.g.*, retransmission and failure detection), we chose the design with the simplest implementation, both in terms of code complexity and thread safety.

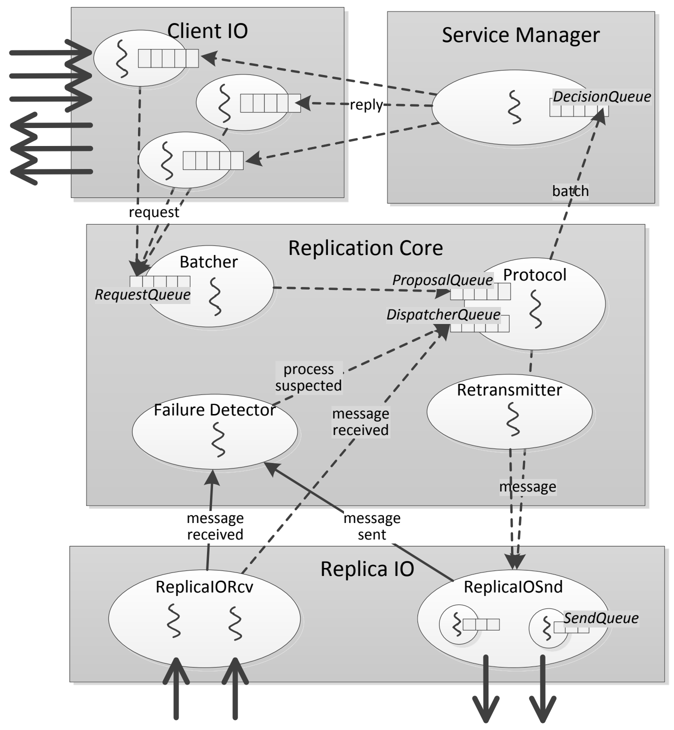


Fig. 3. Scalable threading architecture. Dashed lines represent asynchronous calls (putting a message on the message queue of the module), thin solid lines represent synchronous calls, and thick solid lines represent network communication.

V. Threading Architecture

To simplify our presentation, we will not distinguish between the case of a replica acting as leader from the case where it is a non-leader replicas, even though the tasks performed are not the same. Instead, we discuss a single case, where the replica does both the leader and non-leader related tasks. Note that this is often the case in reality, where leadership is just an additional responsibility for the replica.

Figure 3 shows the threading architecture. Our design uses several types of message queues. The *RequestQueue* connects the ClientIO threads to the Batcher thread, while the *ProposalQueue* does the same between the Batcher and Protocol threads. The *DispatcherQueue* is the queue from where the Protocol thread takes events to process. Each ReplicaIOSnd thread has a queue with the messages waiting to be sent. The *DecisionQueue* is used by the Protocol thread to pass the ordered batches to the ServiceManager. Finally, each ClientIO thread has a queue for the replies to be sent to the respective client.

A. ClientIO module

Since clients connect to a replica using TCP and remain connected for potentially a long time, the ClientIO module has to handle thousands of concurrent connections. In this scenario blocking I/O with a thread-per-connection model is inefficient, so the ClientIO module uses instead non-blocking I/O (Java NIO) and an event-driven architecture. For parallelism and load balancing, it keeps a static pool of I/O threads and assigns new connections to a thread in this pool using a round-robin strategy[1]. For each connection, a ClientIO thread is responsible for reading and deserializing requests, checking the reply cache, and then either sending the cached reply back to the client or putting the request in the Batcher queue. After the request is executed, the ServiceManager thread places the answer in the message queue of the ClientIO thread that is handling the connection to the corresponding client. The ClientIO thread will later serialize and send the reply.

Our profiling tests (Section VI-B) show that reading and writing requests represent a significant fraction of the CPU utilization in state machine replication. By using a configurable number of ClientIO threads, this module can easily take advantage of the cores available in the system.

B. ReplicaIO module

This module uses blocking I/O and a thread-based design, with two threads per socket, one for reading and another for writing. The reader thread for replica p reads and deserializes the messages received from p, then passes the messages to the Protocol thread using the *DispatcherQueue*. Any thread wanting to send a message to replica p places the message on the queue of the respective sender thread. This thread will later take the message, serialize and send it to p.

Although the reader thread is necessary in a blocking I/O design, the sender thread is not strictly required because other threads can write directly to the socket. However, having a dedicated thread to send messages has several benefits. First, it improves parallelism by offloading to a dedicated thread the work of serialization and of writing to a socket. Second, it prevents the thread running the main event-loop (*e.g.*, the Protocol thread) from blocking on a socket write, which can happen if other replicas are slow in reading from the network or stop reading altogether because of crashing. Blocking in this situation would at best slow down the main event-loop and, at worst, lead to a distributed deadlock if the Protocol threads in multiple replicas block trying to send messages to each other. By having a dedicated send thread, this situation is detected by other threads without blocking when the *SendQueue* becomes full.

We chose blocking I/O for this module, because the number of connections between replicas is relatively small, usually comparable to the number of replicas, so it did not justify the additional complexity of non-blocking I/O. As our experiments show, a single thread can easily handle the load of reading or writing to one other replica. Additionally, this design scales well with the number of replicas, since the number of ReplicaIO threads is proportional to the number of replicas. Given enough available cores, we can expect that the performance of reading/writing to other replicas will not degrade as the number of replicas increases.

[1]Our design can easily accommodate more sophisticated load-balancing strategies.

C. ReplicationCore Module

This module contains four threads: Batcher, Protocol, FailureDetector and Retransmitter. The Protocol thread has the central role, because it executes the core operations of the replication protocol. As such, it is the critical path for the performance of both the local replica and of the system as a whole. Therefore, we have reduced to a minimum the work done by this thread, delegating as much as possible to other threads. This is challenging, because the tasks done by this module are closely related, sharing and manipulating the same underlying state. Using locks to protect the shared state would lead to a complex design, being prone to contention and race conditions. Therefore, we have enforced in this module a 'no-lock rule': coordination between threads is done either by message passing using queues, or by shared state if concurrent access is not harmful. In spite of this strict rule, we have identified several tasks that are mostly self-contained, having only a few isolated interactions with other threads and sharing only a few variables. We only allowed shared variables if accessing them can be done without locks, *i.e.*, by relying only on atomic operations or on the semantics of volatile variables. Several conditions must be met for this to be true, mainly the variable cannot be used in a condition variable and it must be possible to access the variable independently without exposing inconsistent state.

1) Batcher thread: This thread takes requests from the Request queue, forms batches according to the batching policy, and puts them in the *ProposalQueue*. This thread accesses directly the state owned by the Protocol thread to read the number of ballots that are currently in execution (volatile variable).

The Batcher thread removes from the critical path the task of building a batch, doing it concurrently with the execution of the ordering protocol by the Protocol thread. This both reduces latency of request ordering and improves parallelism. The latency is reduced because when the Protocol thread needs a batch to start a new ballot, it can simply take one from the *ProposalQueue*, which is faster than generating a new batch from a list of requests. The parallelism is improved because, as shown by the experiments in Section VI-B, Figure 8, the total execution time of the Batcher thread can exceed 50% of a CPU, which justifies having a separate thread to offload this work from the Protocol thread.

2) Protocol thread: This thread implements the core replication protocol, consisting of an event-loop taking events from the *DispatcherQueue*. The events that it processes consist of messages from other replicas, suspicions raised by the failure detector, batches ready to be proposed, and other housekeeping events related to log management. This thread has exclusive write access to the bulk of the state of the ReplicationCore module, including the replicated log and the variables describing the current state of the protocol. In addition to the *DispatcherQueue*, the Protocol thread uses a second queue (*ProposalQueue*) to receive batches from the Batcher thread. This second queue is needed to enforce flow control (explained below) and to allow the Batcher thread to produce a (limited) number of batches in advance.

This design matches closely the logical structure of a replication protocol, usually expressed as a collection of handlers, and ensures that the implementation of its core is thread-safe.

3) FailureDetector thread: Depending on the role of the replica, this thread either sends hearbeats to the other replicas (leader role) or waits for heartbeats from the leader. When the leader is suspected, it enqueues a suspect event on the *DispatcherQueue*. It also receives notifications from the Protocol thread whenever the view changes. For every other replica, the FailureDetector thread keeps timestamps with the time when the last message was received or sent. These timestamps are updated directly by the ReplicaIO threads. In order to avoid context switches, the ReplicaIO threads do not notify the FailureDetector thread when timestamps are updated. This is safe, because as timestamps never decrease, updating a timestamp always results in delaying the corresponding event (send heartbeat or suspect process), so the failure detector thread can safely wait for the original delay and then decide what to do based on the current values of the timestamps.

Using a dedicated thread for failure detection provides significantly better timing guarantees than using an event-loop, and as such significantly improves the chances that the failure detector will work correctly, even under high-load.

4) Retransmitter thread: This thread ensures that messages essential to the progress of the protocol are eventually delivered. This service is also needed when using TCP, because messages may be lost when a connection is broken and reestablished. Internally, the Retransmitter thread uses a priority blocking queue containing the messages to be retransmitted sorted by time of retransmission. When the Protocol thread sends a message for the first time, it also enqueues it in the Retransmitter queue. As instances are decided, the Protocol thread cancels the retransmission of the messages. This operation must be very efficient, because under normal conditions it will be done for all messages sent. We do it without acquiring locks and without waking up the Retransmitter thread. The Protocol thread simply sets a (volatile) flag on the control structure associated with the message. Later, when the retransmission timeout of the message expires, the Retransmitter thread wakes up, sees that it was canceled and drops the message.

D. ServiceManager module

This module contains a single thread, which receives the ordered batches that the Protocol thread puts in the *DecisionQueue*. For each batch, it extracts the requests, passes them to the service in the final order, updates the reply cache with the results of the execution of the request, and finally hands over the reply to the ClientIO thread responsible for the connection to the respective client.

The reply cache is a potential source of contention: it is queried by each ClientIO thread when a client request is received, and updated by the ServiceManager thread when a request is executed. Under high load, it can be accessed several thousands of times per second from multiple threads.[2] A conventional hash table based on coarse-grained locking performs poorly in this situation, as confirmed by our initial tests. Instead, this table should be implemented using fine-grained locking. In our implementation we have used the class `ConcurrentHashTable` from `java.util.concurrent`, which eliminated any signs of contention in the reply cache.

E. Queues and flow control

Flow control based on backpressure can easily be implemented in the architecture described above. This is achieved by setting appropriate limits to each queue, so that when a stage is not able to keep up with the incoming workload, the queue fills up, which allows the stages before it to detect the overload and take corrective action.

For instance, under high load the Protocol thread will usually not be able to order batches as quickly as the Batcher thread generates them. The *ProposalQueue* will therefore become full, which in turn stops the Batcher thread from taking requests from the *RequestQueue*. This is detected by the ClientIO threads, which in turn temporarily stop reading new requests from the clients. This activates the flow control mechanisms of TCP, resulting in the send buffers at the client side filling up, and the client being blocked from sending more data.

This mechanism proved to be effective in our tests. Even when high load, the resource usage at the replicas remains bounded, without any noticeable degradation in performance.

VI. Performance Evaluation

We now evaluate the multi-core scalability of the architecture described above (see [6] for the full results). We have implemented it in JPaxos, which is a library for state machine replication based on the Paxos protocol. JPaxos is based on the generic design described in Section III, and includes the optimizations of batching and pipelining. The details can be found in [9].

The experiments were run on two different clusters of the Grid5000[3] testbed, one with 8 core machines and the other with 24 core machines. For the 8 core setup, we used the *edel* cluster at the Grenoble site. This cluster consists nodes with two quad-core CPUs (Intel Xeon E5520 CPU) running at 2.27GHz. The experiments with 24 core machines were run on the *parapluie* cluster of the Rennes Grid5000 site. Each node is equipped with two 12-core CPUs (AMD Opteron 6164 HE) running at 1.7Ghz. Both clusters used a 1 Gigabit Ethernet network with an effective inter-node bandwidth of 114MB/s. Nodes were running Linux, kernel version 2.6.26-2, and the JVM used was Oracle's JRE version 1.6.0_25.

To restrict the number of cores used by the JVM, we used the GNU command `taskset` to control the process affinity. In choosing the cores from the two CPUs, we tried to co-locate cores in a single CPU as much as possible. This strategy performs in general better than if CPUs are mixed, as the cores in the same CPU share L3 cache and thereby can communicate very quickly.

The workload was generated by nodes located in the same cluster as the replicas, each running several client threads in the same Java process. The clients send the requests directly to the leader, using persistent TCP connections. After establishing the connection, they send requests in a loop, waiting for the answer to the previous request before sending the next one. Each experiment was run for 3 minutes, with the first 10% ignored in the calculation of the results. The request size was 128 bytes and the answer size was 8 bytes. To focus our evaluation on the ordering protocol, we used a null service, which discards the payload of the request and sends back a byte array of the size required by the test. Additionally, we have not used stable storage, as that would introduce an additional bottleneck making it harder to test the multi-core scalability.

We used a total of 1800 clients distributed over six machines. For the pipelining optimization we set the maximum number of parallel ballots to 10, and for batching we used a maximum batch size to 1300 bytes. With these settings the system is CPU-bound, thereby allowing us to better observe the gains from parallel execution.

For each experiment we show the throughput, the speedup, CPU utilization and the total thread blocking time. The *throughput* is measured in requests per second. The *speedup* is defined as the ratio between the throughput with k cores and with 1 core The *CPU utilization* of a replica is measured using the GNU `time` command and is shown as a percentage of 1 core, *i.e.*, 100% is equivalent to one core being fully utilized. These values are the average over the full run, including the warm up period. The plots labeled *Total blocked time* show the sum across all threads of the time spent blocked trying to acquire a lock. The values displayed are normalized to the run time, *i.e.*, 100% corresponds to 3 minutes. This metric gives an indication of the level of contention inside the JVM, and therefore, the efficiency of the threading architecture. These values are obtained using the Java Management interface (`ThreadMXBean`) by a dedicated background thread. This thread takes samples every second, starting 10 seconds after JVM startup and continuing until shutdown. The values reported below are the cumulative times between the first and last samples.

Recall from Section V that the number of ClientIO threads is configurable. This parameter has a significant impact on performance in multi-core machines: too low and the cores are underutilized, too high and performance drops due to contention. Therefore, for each data point (number of CPUs) in the plots in Section VI-A we have repeated the experiments with various number of ClientIO and show the best results.

[2]The optimal number of ClientIO threads depends on the number of cores and of clients, but in our tests it was usually between 3 and 6.

[3]https://www.grid5000.fr

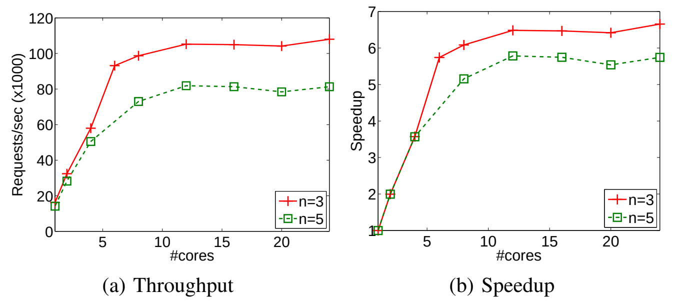


Fig. 4. JPaxos performance with increasing number of cores, parapluie cluster.

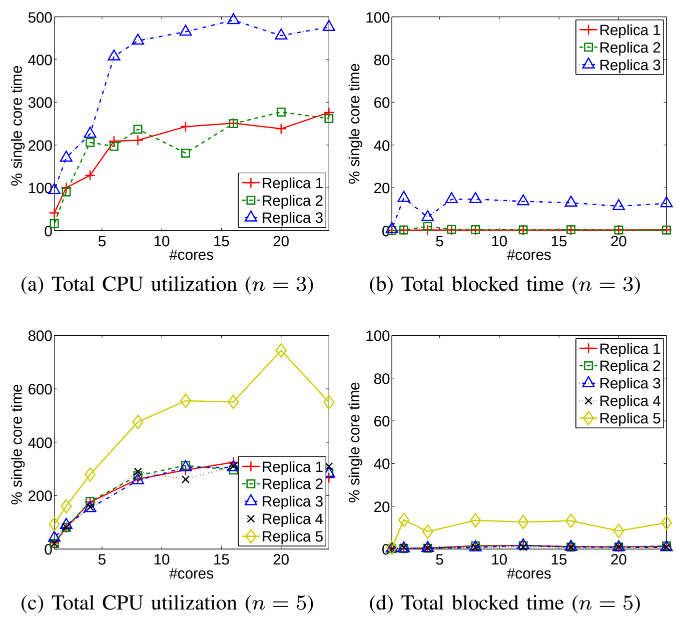


Fig. 5. JPaxos CPU usage and contention, parapluie cluster.

A. Multi-core Scalability

We performed the experiments using configurations with three and five replicas. In each set of experiments, we varied the number of cores from 1 to the maximum number of cores in the nodes. Figures 4 and 5 show the results for the parapluie cluster.

For $n = 3$, the speedup (Figure 4b) is linear up to six cores, then sublinear up to twelve where it reaches the maximum speedup of over 6.5, with a throughput of around 100K requests/sec (Figure 4a). The throughput then remains stable up to the maximum of 24 cores.

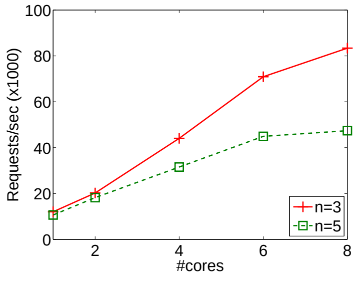
A common pattern in all our results is that the replica in the role of leader (replica 3 and 5 in the tests with three and five replicas, respectively) has significantly higher CPU utilization and contention than the other replicas (Figure 5), which is to be expected from leader-based protocols. Therefore, in the following we will focus our analysis on the leader. Interestingly, the CPU utilization (Figure 5a) increases slower than the throughput (Figure 4a): at the leader, from one to six cores it goes from 100% to 400%, while the throughput increases six times. A possible explanation is that with more cores available, threads run for longer without doing context switches, resulting in less overhead and better caching behavior than with fewer cores. Note that the CPU utilization (Figure 5a) and total blocked time (Figure 5b) remain stable up to the maximum number of cores. In particular, the total blocked time remains under 20%, showing that increasing the number of cores up to 24 does not cause additional contention Recall from the results in Figure 1 that this is not always the case, even in production quality implementations like ZooKeeper.

For $n = 5$ the results are similar, with the exception of a smaller speedup, reaching a maximum of 5.5 instead of 6.5. This is likely an indirect consequence of the higher number of messages (approximately the double) that the leader has to handle. Receiving and sending are done by two dedicated threads per replica, so these tasks scale linearly with the number of cores. However, the main event-loop has to handle all those messages, and being single threaded (Protocol thread) it is limited by the single-thread performance of the CPU, which accounts for the lower speedup. This effect is likely more pronounced with higher number of replicas.

At 100K requests/second the leader hits the limit of the network subsystem. At this point, the leader is exchanging 100K network packets per second with the clients, in addition to the packets exchanged with other replicas, which amounts to 150K packets/second in each direction. In Section VI-D) we analyze this bottleneck, performing experiments with different parameters including smaller requests, larger batch sizes, and larger maximum number of parallel ballots, obtaining slightly better results in throughput in some cases. But in all cases the throughput reached a peak when the leader was handling around 150K network packets/second. As we discuss in section VI-D, this bottleneck is likely in the Linux kernel, since the version we used in the experiments (2.6.26) is known to have several scalability bottlenecks when running in multi-core machines, some of which affect the networking subsystem [14].

Note that although by using a larger batch size it is possible to pack more requests into a network packet, thereby making more efficient use of the network, this works only among the replicas. The communication pattern between clients and replicas depends on the particular workload of the application, which is generally not under the control of the RSM implementation. This highlights the importance of considering external clients when designing and benchmarking replicated systems.

The experiments performed in the edel cluster show similar trends, except that with only eight cores in each node the system does not reach the limit of the network subsystem. With $n = 3$, the speedup increases almost linearly, reaching a maximum of 7 with eight cores (Figure 6b), for a throughput of just above 80K requests per second (Figure 6a). The shape of the curve suggests that with more cores available, the performance will increase further. Another indication that JPaxos would scale further, comes from the results in the parapluie cluster, which reached a maximum throughput of 100K requests per


(a) Throughput

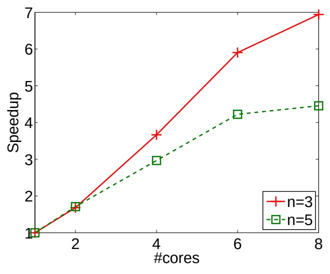
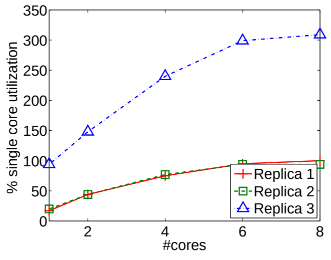
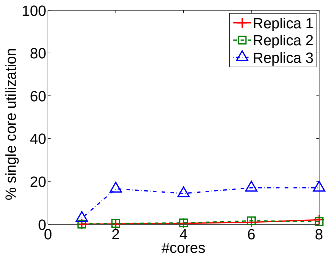
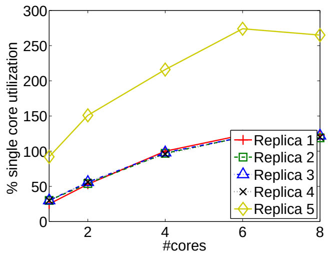

(b) Speedup

Fig. 6. JPaxos performance with increasing number of cores, edel cluster.


(a) Total CPU utilization ($n = 3$)


(b) Contention ($n = 3$)


(c) Total CPU utilization ($n = 5$)

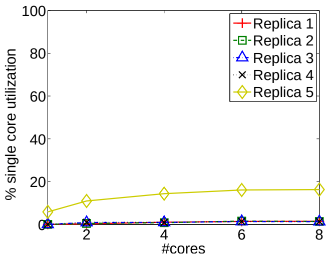
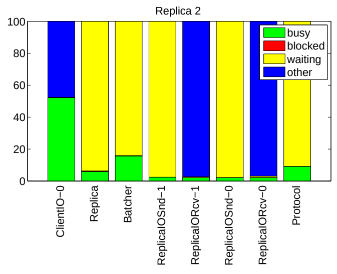
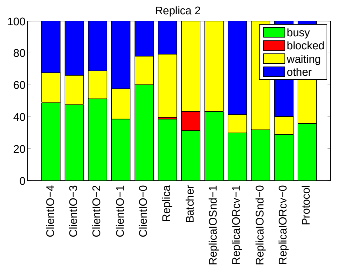
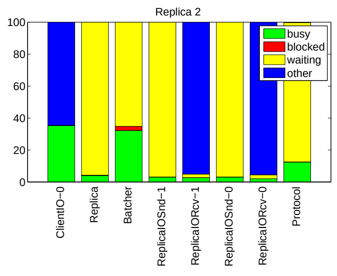

(d) Contention ($n = 5$)

Fig. 7. JPaxos CPU usage and total blocked time, edel cluster.


(a) Parapluie - 1 core


(b) Parapluie - 24 cores


(c) Edel - 1 cores

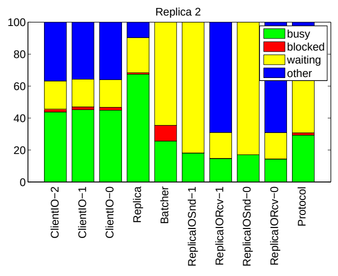

(d) Edel - 8 cores

Fig. 8. JPaxos per-thread CPU utilization of the leader process. $n = 3$.

second. As the both clusters use similar network infrastructure and operating system, we can expect that the bottleneck from the network subsystem will be reached at similar levels of throughput, suggesting the the network subsystem in the edel cluster still has room for higher performance.

Concerning CPU utilization and blocking time (Figure 7), the results are once again similar to the ones in the parapluie cluster, with the CPU utilization increasing slower than the speedup; more precisely, for a 7x speedup there is a 3x increase in CPU utilization. The total blocking time is once again relatively low, with a total aggregate time across threads of under 20% of a single core time. This shows that having more cores allows a well-designed multi-threaded application to run more efficiently, by avoiding the overhead associated with sharing a small number of cores among a larger number of threads.

B. *Scalability limits of threading architecture*

The results above confirm that the threading architecture scales efficiently with the number of cores up to the limits of the networking subsystem of the nodes. Although this shows that the initial goal of this work was reached, it leaves open the question of what are the scalability limits of the threading architecture itself. We can, however, use the previous results to try to infer what would happen if the networking subsystem were faster. For instance, the aggregate CPU utilization is far from the theoretical maximum (500% for a maximum of 2400%), and the contention does not increase with the number of cores, suggesting that the performance could continue scaling.

In this section, we take a more detailed look at what happens under the hood of JPaxos by analyzing how the threads spend their time. The goal is both to better understand how the architecture works internally, and to try to identify any potential architectural bottlenecks. We discuss only the results with three replicas, since the results with five replicas do not differ substantially.

Figure 8 shows the CPU usage of the main threads in JPaxos. For each thread, we show the time spent executing (*busy*), blocked trying to acquire a lock (*blocked*), waiting on a condition variable (*waiting*), and in other states (*other*). The *waiting* state is a sign that the thread is idle, either because its input queue is empty or because its output queue is full, and thus must wait for other threads to advance. The *blocked* state is a sign of contention for locks. Finally, the *other* state accounts for the remainder of the time including, among others, the time spent sleeping (*e.g.*, called `Thread.sleep()`), the time spent blocked on a system call (*e.g.*, waiting for I/O), and the time when the thread is ready to execute but is waiting to be scheduled to some core.

Figures 8a and 8c show the results for the tests with 1 core in the parpaluie and edel clusters, respectively. JPaxos is CPU-bound in this test, as the sum of the busy time of all threads is close to 100%, which is the maximum with one core. In both cases, the ClientIO and the Batcher threads account for most of the CPU utilization, with the sum of their busy time reaching 80%. With all cores enabled in the parapluie cluster (Figure 8b) all threads are busy between 30 and 60% of the time. In the edel cluster (Figure 8b), the Replica thread stands out somewhat, at more than 60% busy time, while the others are under 40%. These results show that the workload is mostly well-balanced between threads, which reduces the likelihood of any single thread becoming the bottleneck when the load increases. The only potential exception is the Replica thread, which stands out somewhat in some tests. We discuss this issue in more detail below.

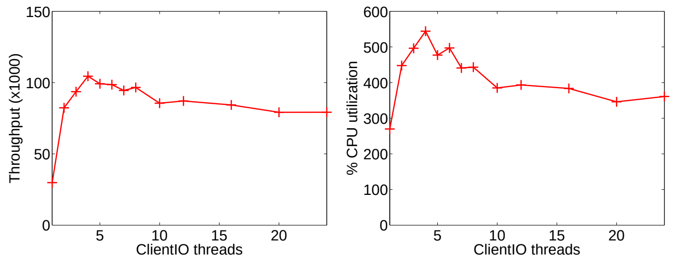
The results also confirm that there is little contention among threads, with most threads spending almost no time on the blocked state. The exception is the Batcher thread, which spends around 15% of its time blocked. Recall that this thread competes for locks both with the ClientIO threads (Request queue), and with the Protocol thread (Proposal queue), so it has a higher chance of being blocked. Although this is undesirable, it is not affecting the performance because the Batcher thread still spends over 50% of its time in the waiting state, *i.e.*, waiting for work.

From these results, we can extrapolate what is likely to happen in the absence of bottlenecks other than the ones inherent to the threading architecture. Interacting with clients is likely to continue scaling with the number of cores and with the workload. The absence of contention among these threads confirms that this is a highly parallelizable task, so adding additional ClientIO threads will improve performance if enough cores are available.

Communication with other replicas should also scale. By using a pair of dedicated send/receive threads per replica, if more replicas are deployed there will be also more ReplicaIO threads, which can run independently given enough cores. The only potential bottleneck is at the level of an individual connection, since all the reading from (resp. sending to) another replica is done by a single thread. However, the per-connection workload is mainly a function of the size of the batches (which is under control of the RSM), being mostly independent of the number of the clients and number of replicas. And in our tests, even at peak throughput, the ReplicaIO threads are busy less than 40% of the time, suggesting that there is room to more than double the throughput given a faster network.

The busy time of the Replica, Batcher and Protocol threads is between 40 and 50%, suggesting that if no other bottlenecks were present, the nodes used in this experiment could sustain up to double the current throughput before hitting the single-thread performance limits of the CPU. Further improvements would require changing the architecture.

Next are some ideas that could extend even further the scalability of our architecture. The creation of batches can be parallelized by using several Batcher threads, each with its own queue of incoming requests. However, this change is far from being trivial to implement, since it requires addressing load balancing among Batcher threads and can potentially increase latency as batches take longer to be formed. The work done by the Protocol thread cannot be easily parallelized because of the



(a) Throughput (b) Total CPU utilization at leader

Fig. 9. Varying the number of ClientIO threads. Parapluie cluster, $n = 3$

complexity of the state managed by this thread. But since the work of this thread is proportional to the number of batches ordered, it is possible to reduce its load by increasing the batch size. The Replica thread poses the biggest challenge, because its work is proportional to the number of requests and it cannot be easily parallelized, as at this stage requests are put on their final sequential order. The only obvious way to improve this stage is by optimizing its single-thread performance.

C. Effect of number of ClientIO threads in throughput

As mentioned previously, the number of ClientIO threads is an important factor in the scalability of JPaxos. In Section VI-A we have shown the results obtained with the optimal number of ClientIO threads for each data point (number of cores). In this section, we look in more detail at the effect of this parameter in the throughput.

Figure 9 shows the results of an experiment using all 24 cores of the parapluie while varying the number of ClientIO threads, using the same experimental settings as in Section VI-A. The results show clearly that handling client connections offers a major opportunity for parallel execution: the throughput goes from 40K requests per second with one ClientIO thread to over 100K with four threads (Figure 9a), a 2.5x improvement with just three additional threads. However, the performance degrades slightly with more than eight threads, dropping to 80K requests per second. Figure 9b shows that the CPU utilization mirrors the behavior of the throughput, reaching a maximum of 550% with four ClientIO threads, then decreasing slightly.

We were not able to determine precisely the cause of the degradation in performance with more than eight threads, although we ruled out some possibilities. It is not caused by contention on locks inside the JVM, as even with 24 ClientIO threads the total blocked time of all threads is under 10% of the run time, even less than with the optimal of four threads (See Section VI-B). There may be other sources of contention that are not reported by the JVM, like the reply cache kept in a `ConcurrentHashMap` (See Section V). As this data-structure uses non-blocking primitives like test-and-set, contention would manifest itself as higher CPU utilization. However, this hypothesis is not supported by the results, which show lower CPU utilization with 24 threads than with eight threads. In Section VI-D, we show that the cause of contention is likely in the Linux TCP stack, which as mentioned previously suffers from scalability bottlenecks in the networking subsystem in the kernel version used in the tests [14].

These results illustrate the importance of carefully choosing the level of parallelism, as often there is a narrow range in which the performance is optimal.

D. Determining the system bottleneck

In the experiments in the `parapluie` cluster, the throughput peaked at around 100K requests per second with eight cores, remaining at this level as the number of cores was increased further. The results did not show any clear indication of a bottleneck. From Figure 8b we can conclude that threads are not spending a significant amount of time blocking on locks, there is no single-thread bottleneck (*i.e.*, the busy time of every thread is under 60%), and the total CPU utilization is well under the maximum. Therefore, it remains to identify the bottleneck limiting the performance. In this section, we show that at this performance level the system is limited by the network subsystem of the leader replica. This confirms that we have not hit the scalability limits of the threading architecture of JPaxos, even in a 24 core cluster with high-end network equipment.

In the rest of this section we use WND to denote maximum number of parallel ballots the leader is allowed to execute, and BSZ to denote the maximum batch size. Note that both are limits defined by configuration parameters. The actual number of parallel ballots in execution at any given time and the actual size of batches created by the leader, vary during a run according to the workload.

1) Internal queues: We start by looking at what happens inside JPaxos during a run, by looking at the average size of the internal queues that are used by threads to communicate with each other. Table I shows, for experiments using different values of WND, the average size of the three queues that play the most important roles, *i.e.*, *RequestQueue*, the *ProposalQueue* and the *DispatcherQueue* (See Section V). Additionally, it shows the average number of parallel instances during each run.

In average, the *RequestQueue* is always more than one-quarter full (maximum 1000) and the *ProposalQueue* more than half full (maximum 20). This shows that the bottleneck is not between the clients and the leader, as the leader has batches waiting to be ordered. On the other hand, the *DispatcherQueue* is in average empty, indicating that the Protocol thread is most of the time idle. The reason for this is clear from the average number of parallel instances, which is always very close to the limit. This indicates that the Protocol thread cannot start new instances because it is waiting for messages from other replicas in order to decide the current instances.

This shows that the bottleneck is in the network between the leader and the other replicas. There are several possible causes for this delay, the most obvious ones being limited bandwidth, high latency, and slow replicas. The bandwidth is not the limit because in the experiments above the maximum data rate reached by the leader[4], is 38MB/sec out and 22MB/sec in, which is far from the limit of 114MB/sec. We can also exclude the other replicas as the cause of the delay, as they were very lightly loaded in all experiments and showed no signs of contention. In the next section we investigate the remaining possibility, that is, that the latency is the cause of the delays.

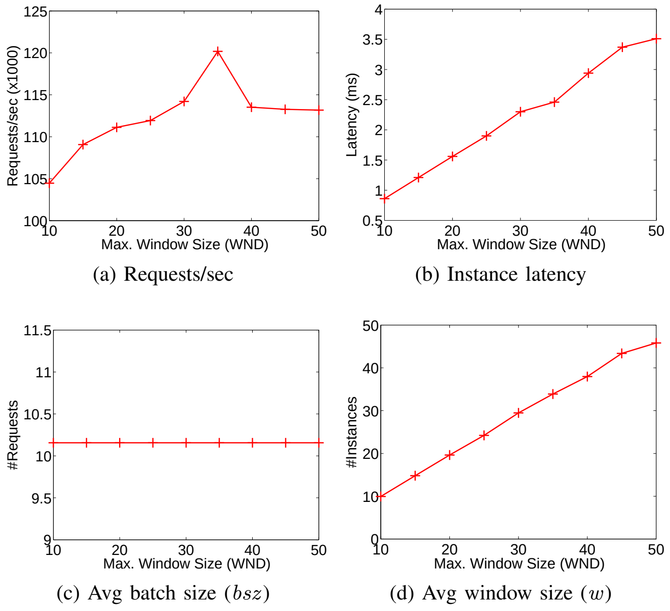


Fig. 10. Performance as a function of window size. Parapluie, 24 cores, n=3, $BSZ = 1300$.

2) Varying the window size: If the communication latency between the leader and the followers is the cause of the poor performance, then by increasing the limit WND the leader should be able to use its idle time to start more ballots while waiting for the messages of the previous ones, thereby improving throughput. Figure 10 shows the results of a set of experiments where WND varies from 10 to 50.

Increasing WND improves the throughput from 100K to a peak of 120K requests per second with $WND = 35$ (Figure 10a). But for higher values of WND, the throughput drops to 110K. The other plots explain the reason. Figure 10c shows that batches are always full and Figure 10d shows that the average number of parallel instances in execution at any given time is always very close to the limit. But in spite of executing more instances in parallel, the throughput does not increase after $WND = 35$. The reason can be inferred from the instance latency (Figure 10b), *i.e.*, the time the leader has to wait since it proposes a value for an instance until receiving at least one Phase 2b message from another replica, and thus deciding the instance. The instance latency grows steadily with the maximum number of parallel instances: up to 35, it grows slower than WND, resulting in a net gain in throughput, while after 35 it grows faster than WND, resulting in a decrease in throughput.

[4]These values were measured with the Ganglia monitoring interfaces of Grid 5000: https://helpdesk.grid5000.fr/ganglia/.

WND	*RequestQueue*	*ProposalQueue*	*DispatcherQueue*	Avg parallel ballots
10	629.70 ± 23.94	14.33 ± 0.36	2.14 ± 0.25	9.63 ± 0.12
35	550.29 ± 8.11	14.94 ± 0.29	1.26 ± 0.41	34.67 ± 0.20
40	440.30 ± 7.49	14.97 ± 0.30	1.47 ± 0.31	39.50 ± 0.26
45	406.52 ± 8.36	14.85 ± 0.33	1.54 ± 0.54	43.88 ± 0.47
50	255.91 ± 10.90	12.99 ± 0.46	4.51 ± 1.14	45.86 ± 0.85

TABLE I
AVERAGE SIZE DURING A RUN OF INTERNAL QUEUES AND OF THE NUMBER OF PARALLEL BALLOTS. PARAPLUIE CLUSTER, $n = 3$, $BSZ = 1300$.

		ping (ms)
idle	any ↔ any	0.06
experiment	other ↔ other	≈ 0.06
	follower ↔ other	≈ 0.06
	follower ↔ follower	≈ 0.06-0.08
	leader ↔ any	≈ 2.5

TABLE II
PING TIMES BETWEEN NODES OF THE PARAPLUIE CLUSTER, WHILE IDLE AND DURING AN EXPERIMENT ($WND = 35$, $BSZ = 1300$, $n = 3$). *other* DENOTE A NODE IN THE CLUSTER NOT INVOLVED IN THE EXPERIMENT, AND *follower* DENOTES NON-LEADER REPLICAS.

This increase in latency is not due to delays in processing the messages inside the replicas JVM, because as seen before both the leader and the non-leader replicas are far from being overload. To determine the cause, we have instead to look at the network latency between replicas during an experiment. Table II shows the round-trip-time (RTT) between nodes in the cluster before and during an experiment, measured using the `ping` command. While idle, the RTT is consistently around 0.06ms between any node. During an experiment, it is also 0.06ms between nodes not involved in the experiment. Between nodes involved in the experiment, but excluding the leader, the RTT becomes more irregular, varying between 0.06 and 0.08, a marginal increase. But between the leader and any other node in the cluster it increases to 2.5ms. This value matches the instance latency in this experiment (Figure 10b, which is also around 2.5ms. Since the RTT only increases when the leader node is involved, this leaves only the network subsystem of the leader as the cause of the delays. Also note that the `ping` command uses directly the low-level networking system of the kernel, which further confirms that the source of the delay is within the kernel and not in the JVMs or even in the TCP stack. This conclusion is validated by results published in [14], which show that the network subsystem of the Linux kernel in versions prior to 2.6.35 (so including version 2.6.26 used in our experiments) suffered from poor scalability in multi-core machines when processing a large number of packets.[5]

3) Varying the batch size: The limiting factor in the experiments above was the number of packets processed by the network subsystem of the leader. We now investigate if increasing the maximum batch size improves the throughput beyond the maximum of 120K requests per second achieved in the previous section. The rationale is that for the same throughput, larger batches may decrease the total number of individual network packets handled by the leader, because the larger messages sent by the leader to the other replicas will be split into packets of an higher average size. Figure 11 shows the results.

[5]Since the initial version of this article was written, we have determined that the bottleneck is caused by the way the Linux kernel handles interrupts from the network card. By default, it directs all the network interrupts to a single queue that is served by only one core, which creates a single-thread bottleneck as the number of packets increases. However, it is possible to distributed the load among cores using mechanisms like Receive Side Scaling (RSS) and Receive Packet Steering (RPS) [15]. We have repeated some experiments with these settings enabled and in most cases the throughput doubled.

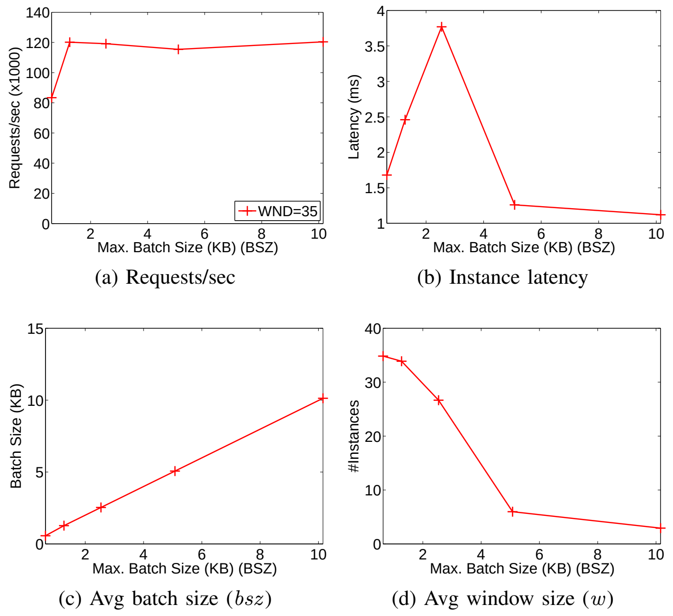


Fig. 11. Performance as a function of batch size. Parapluie, 24 cores, $n = 3$, $WND = 35$.

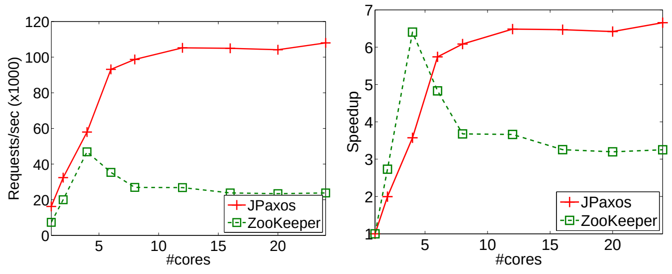
As BSZ increases from 1300 bytes to a little over 10KB, the throughput remains at 120K requests/sec (Figure 11a). Therefore, higher batches do not improve the throughput.

To understand the reason, we show in Table III the average number of packets per second sent and received by the leader and the total outgoing and incoming bandwidth, as reported by the monitoring interfaces of Grid 5000.

For all batch sizes displayed, the number of packets/sec sent by the leader is at 150K, which suggests that this is the limit of the network subsystem of the leader. The leader uses the same network interface to communicate with the clients and the replicas. Therefore, for a throughput of X requests/sec, the leader has to received/send X packets/sec from/to the clients (each request is sent in a single packet), regardless of BSZ. The leader must also use a number Y of packets/sec to order

BSZ	Throughput	Packets/s (out/in)	Bandwidth (out/in)
650	83K	150/145 K	38/22 MB/s
1300	114K	150/145 K	44/25 MB/s
2600	119K	150/140 K	44/25 MB/s
5200	120K	150/135 K	44/25 MB/s

TABLE III
THROUGHPUT AND NETWORK UTILIZATION FOR VARYING SIZES OF BSZ. PARAPLUIE CLUSTER, $n = 3$, $WND = 35$.



(a) Throughput (paraplue) (b) Speedup (paraplue)

Fig. 12. JPaxos vs ZooKeeper with increasing number of cores. Parapluie cluster, $n = 3$

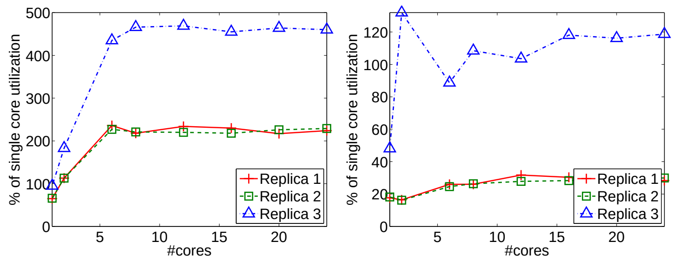
these requests, but this number will depend on BSZ: higher BSZ decrease the number of packets required to order the same number of requests. With this in mind, we can interpret the results.

With $BSZ = 650$, the leader sends 83K packets/sec to the clients and 67K to the replicas. This is an inefficient configuration, because an Ethernet frame can be up to 1500 bytes in a typical deployment, while the leader is sending batches of 650 bytes to the other replicas. With $BSZ = 1300$, the leader packs the double number of requests in a single Ethernet frame, therefore halving the number of frames exchanged with other replicas for the same throughput. The spare budget of network frames allows the leader to process more client requests, resulting in a higher throughput. Further increases in BSZ do not translate in a significant increase in the average size of each Ethernet frame sent by the client, because with $BSZ = 1300$ bytes the packets were already close to the limit. Therefore, the increases in throughput are small for $BSZ > 1300$.

The slight decrease in the number of packets received by the leader is because replicas send a single Phase 2b message to the leader in response to each batch, therefore with bigger batches the leader receives fewer Phase 2b messages for the same throughput.

E. Comparison with ZooKeeper

In this section we compare the results of our threading architecture with ZooKeeper, to show how it improves over a production-quality implementation of RSM. The tests in this Section were performed with ZooKeeper 3.3.3. We used the default behavior for the connections between the clients and replicas: the clients connect via TCP to a replica chosen randomly. As recommended in the ZooKeeper's documentation for achieving better performance, we changed the default configuration to force the leader to refuse client connections.



(a) CPU Usage (b) Contention

Fig. 13. ZooKeeper cpu usage and contention. Parapluie cluster, $n = 3$

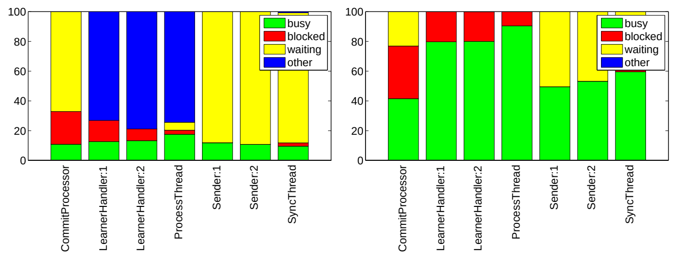
Each ZooKeeper client creates an ephemeral node at startup and then enters a loop issuing write requests (`setData()`) in that node. We used write requests to force ZooKeeper to execute the full ordering protocol, thereby making the results comparable with JPaxos which does not allow local reads and orders all requests. Otherwise, the workload settings were similar to the ones used by JPaxos. As we are interested on the multi-core scalability of the system, we did not use stable storage in our tests. However, ZooKeeper does not allow disabling stable storage. Instead, we have used a ramdisk (`/dev/shm`) for its transaction log and for saving snapshots, which effectively removes most of the cost of the disk writes.

Figures 12 and 13 compare JPaxos and ZooKeeper in terms of performance, CPU usage, and contention.

ZooKeeper scales super-linearly up to four cores where it reaches a speedup of six (Figure 12b). However, for higher number of cores its performance degrades substantially, finishing at only four when all 24 cores are used. This degradation is caused by contention.

Figure 13b shows that the leader, Replica 3, suffers from very high levels of contention, with the aggregate blocking time of its threads exceeding 100% of the run time. By comparison, the blocking time in JPaxos does not exceed 20% (Figure 5b). ZooKeeper's CPU utilization (Figure 13a) is another sign of the problems with its architecture. Although the throughput drops when using more than four cores, the CPU utilization continues increasing up to ten cores, when it finally stabilizes; this means that the additional CPU utilization is spent on contention. With JPaxos, on the other hand, the CPU utilization (Figure 5a) closely follows the throughput, indicating minimal or no CPU wasted on contention. Looking in more detail at how threads spend their time (Figure 14) further confirms the high level of contention. Even when using a single core, several threads spend between 10 and 30% of their blocked. The situation gets worse when 24 cores are used, with one thread, the `CommitProcessor`, spending around 40% of its time blocked.

ZooKeeper also suffers from several single-thread bottlenecks, because the workload is poorly balanced among its threads. Figure 14b shows that when 24 cores are used, three of its main threads are busy or blocked 100% of their time, which limits the overall performance in spite of having more cores available.



(a) One core (b) 24 cores

Fig. 14. ZooKeeper: per-thread CPU utilization of the leader process.

VII. Conclusion

As replication protocols improve and the network infrastructure becomes faster, implementations of replicated state machines are increasingly limited by the single-thread performance of the system. This is because most of these implementations, including research projects and production-quality implementations like ZooKeeper, are not designed to take full advantage of multi-core systems. In this paper we have shown how to parallelize a generic implementation of a replicated state machine, so that its performance scales with the number of cores in the nodes.

The proposed threading architecture divides the internal state and tasks into a set of modules, with well-defined boundaries. At the core, there is a pipeline of event-driven stages that handle requests, with several satellite modules providing auxiliary services. As these modules differ substantially in complexity we used a variety of techniques, choosing the implementation of each module based on its potential for parallelism and complexity. We believe that the architecture proposed is general enough to be applied in a variety of implementations of state machine replication, with only minor adaptations needed.

The experiments show that in a 24 cores system, the architecture scales with the number of cores, until reaching the limits of the network subsystem. The results also suggest that in the absence of other bottlenecks, the performance would continue scaling with additional cores.

References

[1] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[3] M. Kapritsos and F. P. Junqueira, "Scalable agreement: toward ordering as a service," in *Proceedings of the Sixth international conference on Hot topics in system dependability*, ser. HotDep'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[4] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *Dependable Systems and Networks (DSN'10)*, Jun. 2010.

[5] R. Levy, "The complexity of reliable distributed storage," Ph.D. dissertation, EPFL, 2008.

[6] N. Santos and A. Schiper, "Achieving high-throughput state machine replication in multi-core systems," EPFL, Tech. Rep. (to appear), Nov. 2011.

[7] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP. New York, NY, USA: ACM, 2001, pp. 230–243.

[8] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, 2008.

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

[10] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 369–384.

[11] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, May 1998.

[12] 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.

[13] Y. Amir and J. Kirsch, "Paxos for system builders," Johns Hopkins University, Tech. Rep. CNDS-2008-2, 2008.

[14] 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[15] T. Herbert and W. de Bruijn, "Scaling in the linux networking stack," Nov. 2011. [Online]. Available: http://www.mjmwired.net/kernel/Documentation/networking/scaling.txt