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—The traditional architecture used by implementations of Replicated State Machines (RSM) does not fully exploit modern multi-core CPUs. This is increasingly the limiting factor in their performance, because network speeds are increasing much faster than the single-thread performance of CPUs. Thus, when deployed on Gigabit-class networks and exposed to a workload of small to medium size client requests, RSMs are often CPU-bound, as they are only able to leverage a few cores, even though many more may be available. In this work, we revisit the traditional architecture of a RSM implementation, showing how it can be parallelized so that its performance scales with the number of cores in the nodes. 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 six times improvement in throughput using eight cores. More generally, in all our experiments we have consistently reached the limits of the 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 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 in 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 level 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 key-value stores [1], lock servers [2], and coordination services [3], 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 instead of having every service implement its own ordering layer [4]. 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, [5] and [6] show how to achieve high-throughput with algorithmic improvements to the replication protocol. These protocols use the network efficiently, relying on techniques like ring topologies and IP multicast to achieve an efficiency of over 90% in a 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 ≥8KB. For smaller request sizes, they become CPU-bound and their efficiency drops significantly [5]. A similar CPU bottleneck with small request sizes has been identified in many other systems, including in production-quality implementations. For instance, in [3] the authors report that the atomic broadcast protocol at the core of ZooKeeper becomes CPU-bound 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 high throughput.

It is worth looking more carefully at the causes of this CPU bottleneck. In recent years the single-thread performance of CPUs improved only marginally, while the number of cores increased greatly. A modern server-class CPU contains anywhere between 4 and 16 cores, with even higher numbers on the horizon. Although each core 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, and Figure 1b which shows the per-thread profiling results with 24 cores. For each thread, we show the time spent executing (running), 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). Although ZooKeeper scales well up to four cores reaching a peak of 50K requests per second, its performance degrades to less than 30K as the number of cores increases to eight, after which it remains stable. We can make two main observations from the results. First, ZooKeeper can only use a limited number of cores (around eight), after which adding more cores has no effect.
In fact, Figure 1b shows that there are only seven threads with a non-trivial CPU utilization (threads with insignificant CPU utilization are not shown). This is a limitation of its design, which does not allow any change to the number of threads. Additionally, this makes ZooKeeper susceptible to single-thread bottlenecks because once a thread reaches its limit, it will hold back the full system (Figure 1b). Second, using more cores may in fact reduce performance, even when there are threads capable of making use of those cores, as seen by the sharp drop in performance from 4 to 8 cores. This is likely caused by a combination of increased contention on locks and cache trashing, because at any point in time there are more active threads contending for access to shared data. The profiling results show that contention is in fact a problem: With 24 cores available, threads spend a large fraction of time blocked (the profiling results are similar for all experiments from 8 to 24 cores).

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 the exception of I/O 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 revisited in order to reach the potential of modern Gigabit networks, multi-core CPUs and of the new generation of highly efficient replication protocols.

More recent implementations depart from the traditional architecture and use multiple threads. However if threads are introduced without making an effort to carefully partition the internal state of the RSM, they will end up sharing critical parts of the state. This requires the introduction of locks to ensure thread-safety, which in turn leads to high levels of contention. As a result, the improvement in parallelism is limited, as illustrated above with ZooKeeper.

In this paper we address the problem of parallelizing 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 independently of others, with minimal time wasted in coordination (e.g., contention). For the second point, it is essential to minimize shared state among threads.

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 sequential or because their state is complex and shared between several tasks. As mentioned previously, a naïve 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, 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:
- An analysis showing 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, both in academia [5], [6], [9] and in the industry. In the latter case, some of the noteworthy examples are the Chubby lock-server [2] and the ZooKeeper coordination service [3].

Recently, several works have focused on the performance of state machine replication. Mencius [9] proposes a rotating leader protocol designed for WANs, which improves throughput and latency by distributing over multiple replicas the load that is usually concentrated on the leader. In [4], the authors propose a protocol based on a similar idea but adapted to the LAN scenario. LCR [6] and Ring-Paxos [5] focus instead on achieving high network efficiency on fast Gigabit LANs. Zab [3], 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 the results presented in [3] and [5] show that 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. To the best of our knowledge, there is no previously published work addressing this particular problem.

Outside the field of state machine replication, there is a rich body of work on tools and techniques to exploit the parallelism available in 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 encapsulate threading within a module using message passing, but deviated from this rule when it would significantly harm 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 guarantees that the state of the replicas remains consistent and available, in spite (of a limited number) of failures.

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 [10]. 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 instances of the ordering protocol (sometimes also called ballots). In each instance, 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 corresponding sequence number.

Two optimizations that greatly improve the performance of Paxos are batching and pipelining. When using batching, the leader groups several client requests in the same instance, while when using pipelining, the leader executes several instances 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 adds the request 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 according to this order, and the replies are 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 [11] for an example). However, the threading architecture and the underlying guiding principles that we propose are general and can be applied to other designs.

An implementation of a RSM consists roughly of four modules: ClientIO, ReplicaIO, ReplicationCore, and ServiceManager (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 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 module 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 module 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 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, 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 with respect to 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 to scale the performance of RSMs with the number of cores while keeping complexity under control: 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 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 tasks (sequential or event-handlers). The choice of the appropriate design involves a series of trade-offs between these factors.

As the CPU-intensive tasks have the greatest potential for parallelism, they should be implemented as multi-threaded modules, ideally with a configurable number of threads. This is easy in some cases, like with 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 tasks that have little or no potential for parallelism (e.g., retransmission and failure detection), we chose a simple blocking, single-threaded design.

V. THREADING ARCHITECTURE

To simplify the presentation, we do not distinguish the cases where a process is acting as leader or as non-leader replica, 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.

![Figure 3. Scalable threading architecture. Dashed arrows represent asynchronous calls (putting a message in the message queue of the module), thin solid arrows represent synchronous calls, and thick solid arrows represent network communication.]

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 simultaneous 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. 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 RequestQueue. 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. We use a configurable number of ClientIO threads, which allows this module to easily take advantage of the cores available in the system.

The optimal number of ClientIO threads depends on the number of cores and of clients. Based on our experiments,
the most important rule to follow is to not exceed the number of cores left unused by the other threads in the RSM (or choose only one thread if all cores are used), so that ClientIO threads are not forced to do context switches. Another useful rule, although with a smaller impact, is to set a number proportional to the expected number of clients, aiming to keep each thread running between 60% and 80% of the time. In our tests the optimal value was typically between 3 and 6.

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 in 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 or have crashed without closing the TCP connection. 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 of 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 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.

C. ReplicationCore Module

This module contains four threads: Batcher, Protocol, FailureDetector and Retransmitter. The Protocol thread has the central role, because it executes 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. Instead, this module does not use any lock explicitly: 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 allow shared variables if accessing them can be done without locks. Several conditions must be met for this to be true, mainly the variable cannot be used in a condition variable (which usually requires locking) and it must not be part of an invariant that includes other variables. If these conditions are true, then we can rely on atomic operations or on the memory model guarantees of the language/runtime to access the variables. An example of the second case are volatile variables in Java, which ensure a global order on the reads and writes to the variable. In the following description, we will use volatile as defined in Java, although other languages have equivalent mechanisms to ensure the same properties.

Batcher thread: This thread takes requests from the Request queue, forms batches according to the batching policy, and puts them in the ProposalQueue. It accesses directly the state owned by the Protocol thread to read the number of instances 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 design reduces latency of request ordering because when the Protocol thread needs a batch to start a new instance, it can simply take one from the ProposalQueue, which is faster than generating a new batch from a list of requests. It also improves parallelism because, as shown by the experiments in Section VI-B, Figure 6, 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.

Protocol thread: This thread executes the core replication protocol, implementing an event-loop that takes events from the DispatcherQueue. These events include 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 ReplicationCore module state, 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.
Equally important, it is a simple design for ensuring thread safety.

**FailureDetector thread:** Depending on the role of the replica, this thread either sends heartbeats to the other replicas (leader role) or waits for heartbeats from the leader. When the leader is suspected, it enqueues a suspect event in the **DecisionQueue**. It also receives notifications from the Protocol thread whenever the view changes. For every other replica, the FailureDetector thread keeps timestamps of the last message received or sent. These timestamps are updated directly by the ReplicaIO threads when they read or write the corresponding messages. 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.

**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 fails and later is 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 in 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, the *Replica thread*, which receives the batches put by the Protocol thread in the **DecisionQueue** once they are ordered. 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 request execution, 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. 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 **ConcurrentHashMap** from java.util.concurrent, which eliminated any signs of contention on 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 is usually unable to order batches as quickly as the Batcher thread generates them. The **ProposalQueue** therefore fills up, which in turn stops the Batcher thread from taking requests from the **RequestQueue**. This is detected by the ClientIO threads, which in turn temporarily stops 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 under high load, the resource usage at the replicas remains bounded, without any noticeable performance degradation.

**VI. PERFORMANCE EVALUATION**

To evaluate the multi-core scalability of the architecture described above we have implemented it in JPaxos, which is a library for state machine replication based on the Paxos protocol (more precisely, MultiPaxos). JPaxos is based on the generic design described in Section III, and includes the optimizations of batching and pipelining.

The experiments were run on the Grid5000 testbed\(^2\). We used the *parapluie* cluster, which consists of nodes with two 12-core CPUs (AMD Opteron 6164 HE) running at 1.7Ghz, for a total of 24 cores. The network was Gigabit Ethernet 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.

We restricted the number of cores used by the JVM by setting the process affinity with the GNU command *taskset*. In choosing the cores from the two CPUs, we tried to colocate 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 the 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

\(^2\)https://www.grid5000.fr
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 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 it would introduce an additional bottleneck making it harder to test the multi-core scalability.

We used a total of 1800 clients distributed over 6 machines. For the pipelining optimization we set the maximum number of parallel instances to 10, and for batching we used a maximum batch size to 1300 bytes. With these settings the system is CPU-bounded, thereby allowing us to better observe the gains from parallel execution.

We use as metrics the throughput, the speedup, the 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 one core. The CPU utilization of a replica is measured using the GNU time command and is shown as a percentage of one core, i.e., 100% is equivalent to one core being fully utilized. These values are the average over the full run, including the warm up period.

Figures 5b and 5d show the sum across all threads of the time spent blocked trying to acquire a lock. 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, which 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 we show the best results.

**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 24. Figures 4 and 5 show the results.

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.

![Figure 4. JPaxos performance with increasing number of cores.](image)

![Figure 5. JPaxos CPU usage and contention.](image)

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 a 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 since it is 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.

The bottleneck limiting the performance to 100K requests/second is in the network subsystem of the leader, as we explain now. 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. We have performed experiments with different parameters including smaller requests, larger batch sizes, and larger maximum number of parallel instances, 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. 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 [12].

Note that although by using a larger batch size it is possible to pack more requests in 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.

**B. Discussion of scalability limits**

The results above confirm that the threading architecture scales efficiently with the number of cores up to the limits of the node’s networking subsystem. 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 with a faster networking subsystem. For instance, the aggregate CPU utilization is far from the maximum (500% for a maximum of 2400%), and the contention does not increase with the number of cores, suggesting that the performance would 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 6 shows the CPU usage of the main threads in JPaxos. For each thread, we show the time spent executing (running), 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 (i.e., calling 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.

In the tests with one core (Figure 6a), the ClientIO thread has the highest running time, of over 50%, while the other threads are under 20%. JPaxos is CPU-bound in this test, as the sum of the running time of all threads is close to 100%, which is the maximum with one core. With all cores enabled (Figure 6b) all threads are running between 30 and 60% of the time. This shows that the workload is well balanced between threads, reducing the likelihood of any single thread becoming the bottleneck when the load increases.

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 running less than 40% of the time, suggesting that there is room for more than doubling the throughput given a faster network.

The running 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 load balancing among Batcher threads and can potentially increase latency as each batch takes longer to be formed. The work done by the Protocol thread cannot be easily parallelized because of the 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 in their final sequential order. The only obvious way to improve this stage is by optimizing its single-thread performance.

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 complexity and its potential for parallelism. 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, with additional cores and in absence of other bottlenecks, the performance would continue scaling.

ACKNOWLEDGMENT

The authors would like to thank Paweł T. Wojciechowski, Jan Kończak and Tomasz Żurkowski for their work on JPaxos.

REFERENCES