A Library for Services Transparent Replication

Paola Martins Pereira
Universidade Federal do Rio Grande
paolapereira@furg.br

Fernando Luís Dotti
Pontifícia Universidade Católica do Rio Grande do Sul
fernando.dotti@pucrs.br

Cristina Meinhardt
Universidade Federal de Santa Catarina
cristina.meinhardt@ufsc.br

Odorico Machado Mendizabal
Universidade Federal do Rio Grande
odoricomendizabal@furg.br

ABSTRACT

State Machine Replication is a well-known approach to develop fault-tolerant application. Although it seems conceptually simple, building replicated state machines is not a trivial task. The developer has to be acquainted with aspects of the inner working of the specific agreement protocol to correctly develop and deploy the replicated service (and auxiliary processes – e.g. Paxos roles), instead of focusing on the specific service. In this work we propose a replication library that facilitates the development and deployment of fault-tolerant services, and provides replication transparency to service builders. This library allows to deploy a base SMR on top of which new services can be registered at runtime. A service builder focuses on service implementation and registers the service with the base SMR to enjoy the benefits of replication. Besides separating the complexity of providing a replicated infrastructure from service implementation, multiple services share the same consensus and replication infrastructure, allowing cost amortization. According to our evaluation, this approach leads to higher overall throughput compared to the separate deployment of different SMRs over the same resources.

CCS CONCEPTS

• **Computer systems organization** → Dependable and fault-tolerant systems and networks; • **Software and its engineering** → Software fault tolerance; Software usability;

KEYWORDS

reliability, transparent replication, state machine replication, consensus protocol

ACM Reference Format:
Paola Martins Pereira, Fernando Luís Dotti, Cristina Meinhardt, and Odorico Machado Mendizabal. 2019. A Library for Services Transparent Replication. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3297280.3297308

1 INTRODUCTION

High availability has become a typical requirement in software development and replication plays a key role to keep applications running despite the occurrence of partial failures. A successful approach for services[1] replication is State Machine Replication (SMR) [18, 26]. According to SMR, every service replica receives and executes the same sequence of commands. Since replicas start with the same initial state and command execution is deterministic, every replica will traverse the same sequence of states and produce the same outputs.

State Machine Replication (SMR) has been widely used by both the industry and research community. Some notable examples of practical implementations of SMR are Google's Chubby [6] and Apache Zookeeper [13]. Chubby is used by Borg [28], Google's cluster manager, Google File System (GFS) [11], and Bigtable [8], a distributed storage system. Apache's Zookeeper is a popular service, offering a simple interface to support group messaging and distributed locking. It is used by HDFS [27], a Facebook file system [5], to implement a key-value store service, server replication, and concurrency control. Cassandra [17], a distributed data store, relies on Zookeeper for leader election and metadata management.

Although the SMR model seems conceptually simple, developing a SMR application is not a trivial task. To ensure that replicas will receive the same sequence of commands, atomic broadcast [10] or a consensus protocol is required (e.g. Paxos [18] and its variations [19, 21, 23], Raft [25], among others). The integration between a consensus protocol and the application logic usually requires knowledge about fault-tolerance, the specific consensus protocol and its implementation under use. Furthermore, converting fault-tolerant algorithms into a practical, production-ready system involves implementation of many features and optimization[7, 15].

To address the complexity of developing and deploying SMR, in this paper we propose a library that separates service logic from replication techniques and deployment aspects. Our first goal is to allow programmers to be concerned with the application while it is transparently replicated following the SMR approach. A second objective is to improve resources usage by allowing multiple applications to share a single consensus protocol. Besides sharing the same agreement protocol, service applications can be registered or unregistered at runtime. Thus, the deployment of new services does not require system restarts, and there is no need of previous knowledge on service deployment configuration.

Service programming is made replication transparent using modules, that abstract specific consensus protocol and implementation

[1]The words 'service' and 'application' are used with the same meaning in this paper.

details, and annotations to declare/export the metadata needed to register the service with a replication service. This replication service provided with the library is responsible to serve registration, operation, update and removal of user services, which are the ones of interest. The replication service is itself implemented as a SMR: its state is composed by the set of specific user services registered and their states. Managing the replication service one would take care of the aggregate of services registered.

This paper is organized as follows. Section 2 describes the system model. Section 3 discusses SMR. Section 4 presents our library for transparent replication. Section 5 demonstrates the experimental evaluation. Related work is presented in Section 6 and Section 7 concludes the paper.

2 SYSTEM MODEL

We assume a distributed system composed of interconnected processes. There is an unbounded set $C = \{c_1, c_2, ...\}$ of client processes and a bounded set $S = \{s_1, s_2, ..., s_n\}$ of server processes. We assume the crash failure model and exclude malicious and arbitrary behavior (e.g., no Byzantine failures). A process is *correct* if it does not crash or *faulty* otherwise. We assume f faulty servers, out of $n = 2f + 1$ servers, i.e., f is the maximum number of server failures that can be tolerated.

We follow a traditional SMR model [26]. Client processes invoke service commands, which are implemented by service replicas. Commands are executed in the same order by all replicas and commands are assumed to be deterministic. Therefore, if servers start from the same state and execute commands in the same order, they will produce the same state changes and results after the execution of each command.

Total order of commands across replicas is provided by an atomic broadcast protocol [10] which ensures that (i) if a correct process broadcasts a message m, then it eventually delivers m (*validity*); (ii) if a process delivers a message m, then all correct processes eventually deliver m (*uniform agreement*); (iii) for any message m, every process delivers m at most once, and only if m was previously broadcast by $sender(m)$ (*uniform integrity*); and (iv) if both processes p and q deliver messages m and m', then p delivers m before m', if and only if q delivers m before m' (*uniform total order*).

We implement atomic broadcast using Paxos [18], a consensus protocol. Paxos assumes a partially synchronous network, relying on eventual synchrony to guarantee progress. Messages can be lost or arbitrarily delayed. However, eventually all messages between correct processes are delivered within some bounded delay.

3 STATE MACHINE REPLICATION

State Machine Replication (SMR) is a general approach to implementing fault-tolerant services by replicating servers and coordinating client interactions with server replicas [18, 26]. The service is defined by a state machine and consists of *state variables* that encode the state machine state and a set of *commands* that change the state. The execution of a command may (i) read state variables, (ii) modify state variables, and (iii) produce a response for the command (i.e., the output). To ensure that the execution of a command will result in the same state changes and responses at different replicas, each command must be *deterministic*: the state change and response are a function of the state variables the command reads and the command itself. With the above, once replicas start in the same initial state, to keep replicas consistent they have to process the same commands in the same order. This is achieved by having clients atomically broadcast commands and replicas executing client commands sequentially. With this, SMR provides strong consistency (e.g. *linearizability* [2, 12]), an intuitive service behavior that hides from the clients the existence of multiple replicas.

The total order delivery needed by SMR is usually provided by an agreement layer, which can be implemented using Paxos or other consensus protocol, such as Raft [25] and Zab [14]. We assume Paxos for more detailed discussion. In a consensus protocol, a set of participants have to decide on exactly one proposed value. With Paxos, participants act as *proposers*, *acceptors* or *learners*. To get chosen, a value has to be proposed by a *proposer* and accepted by a majority of the *acceptors*. Only one value may be chosen and every *learner* learns the decided value. State machine replicas thus rely on Paxos to deliver client requests: for every new request, a new Paxos instance is started. Paxos instances carry a request number i, which indicates the i^{th} instance in a total order of requests. A replica executes instance i only if it is the next in the totally ordered sequence of requests and has learned the i^{th} via the Paxos protocol.

Figure 1 shows a SMR architecture. Communication is ensured by an agreement layer, which is responsible for implementing atomic broadcast (e.g. through a Paxos protocol). Both client application and replicated service are implemented atop a Paxos library. To use Paxos, developers must explicitly configure the quorum of acceptors, the logging strategy to keep instances decided by acceptors, log trimming policy, among other parameters [7]. APIs provided by Paxos libraries may broadly differ on how to proceed this.

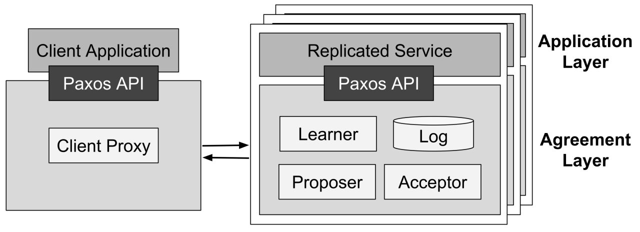

Figure 1: Replicated service using Paxos.

Besides the particularities of Paxos libraries regarding API and general settings, practical Paxos implementations can follow different architectural styles and communication patterns. As illustrated in Figure 2, in S-Paxos [4] and Ring Paxos [22], the *leader*[2] role is assumed by one of the *acceptors*. In OpenReplica and Libpaxos, a separate process is elected as the *leader*. In Ring Paxos [22], however, any node that receives requests from clients and forwards them to other processes are *proposers*. OpenReplica and RingPaxos implement a client proxy, which is a separate module in the client application. This module batches client requests and send the batches to the *leader*. In RingPaxos batches are sent to an

[2]The *leader* is a selected *proposer*, which is in charge of proposing values. In case of *leader* failure, another process among *proposers* is elected as the new *leader*. The *leader* role minimizes collisions among proposed values.

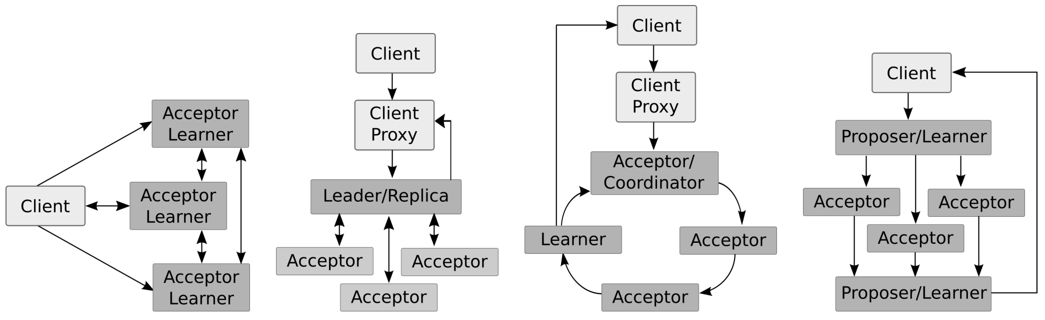

Figure 2: The communication architectural differences among Paxos libraries. This figure is adapted from [20].

acceptor that also plays the role of *proposer*. LibPaxos clients send requests to a single *proposer* while S-Paxos randomly chooses a replica for sending client requests.

Therefore, to build SMRs developers must be experienced with complex communication protocols [7], such as Paxos, different (under-specified) APIs and deal with a non trivial set of configuration parameters.

4 REPLICATION LIBRARY

This section presents our library for transparent replication. The library supports the client/server model. At client side, the library offers primitives to invoke the service. At server side, it supports transparent replication following the SMR approach.

Next, we introduce the key features of the library; followed by its architecture; and then the API (Application Programming Interface) is presented in order to instruct how to use the library.

4.1 Overview

The R3Lib[3] (i.e., Reliable Runtime Replication Library) provides replication transparency for system builders. It allows a simple deployment and easy to configure approach. Next, we outline the main design decisions and concepts adopted.

Decoupling between service and agreement protocol. In order to simplify the development of replicated services, our library decouples the agreement protocol from the application development. The replication library provides an interface between the application and the delivery module. This enables developers to focus on writing the service code without having to worry about replicas coordination or specific details of Paxos or any other reliable broadcast protocol. By separating the delivery of requests from the application logic, developers can choose which agreement protocol they want to use without affecting the application. Alternatively, developers can implement their own delivery module, favoring maintenance, extensibility and reuse.

Sharing of agreement protocol by multiple services. The replication library allows multiple applications to share the same delivery protocol under use. This strategy provides flexibility and better use of resources. While multiple applications execute, a delivery module is responsible for ordering client requests into a single totally ordered sequence of commands. To ensure consistency and service isolation, the library dispatches client requests only to the target applications and in the same order they were delivered. This approach avoids the instantiation of multiple consensus or atomic broadcast protocols. This design decision becomes important since practical consensus or atomic broadcast libraries may consume a non-negligible amount of resources. For instance, besides requiring a quorum of distributed processes, reliable broadcast libraries normally implement failure detection, reconfiguration, logging and specific optimization strategies (e.g. batching [1, 22] and parallelism [23]). As presented in Section 5, the cost of keeping multiple delivery protocol instances can be drastically reduced when a single protocol is shared by applications.

Register and unregister of services at runtime In order to allow the addition, removal or update of services without suspending other services, the replication system is designed as state machine replica where only the basic commands to *register* and *unregister* applications are initially available. New applications are registered into the state machine replicas through a SMR command invocation. Analogously, by invoking the unregister command, the application is removed from state machine replicas. This approach facilitates service changes, such as service maintenance or updates.

4.2 Architecture

The replication library implements the client/server model and its architecture is illustrated in Figure 3. The modules in the left of the figure represent the client while the set of modules to the right represents the structure of the replicas. Both client and replicas communicate through a consensus layer. In the following we describe each of the modules composing the solution:

Client: Represents the client program which makes use of one or more replicated services, sending command requests and receiving respective responses, according to the library API. Clients may also request the addition, update and removal of service instances using the same interface;

Consensus Proxy: This module is in charge offering transparency w.r.t. the underlying agreement protocol. It offers a standard consensus API while the implementation is agreement protocol specific. Besides bringing transparency, its function is to broadcast client requests sent through the standard interface and receive their respective responses, returning them to the client.

[3]https://bitbucket.org/paolampereira/r3lib

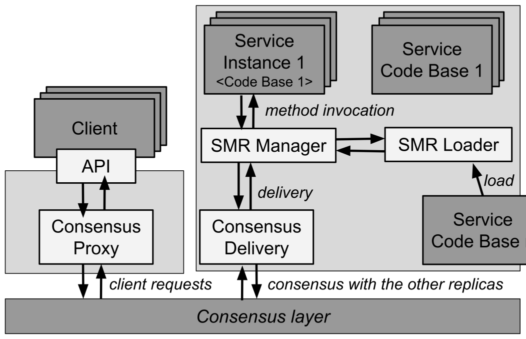

Figure 3: Replication Library Architecture.

Consensus Delivery: Just as Consensus Proxy communicates with the consensus layer on behalf of a client, Consensus Delivery performs this communication at replicas on behalf of the services. It receives the client commands ordered by the consensus layer and triggers a generic delivery event at server side. This event is forwarded to the SMR Manager (which is agreement protocol independent) that listens to those events, processes them, and returns their respective responses. Finally, this response is then forwarded by *Consensus Delivery* to the consensus layer.

Because specific agreement protocols do not follow a standard API, both the client-side Consensus Proxy and the Consensus Delivery on the replicas side are abstract and must be specialized. Figure 4 exemplifies three existing implementations for each of the abstract modules: Modules BFTProxy and BFTDelivery work with BFT-SMaRt[3], while SProxy and SDelivery with S-Paxos[4] and RingProxy and RingDelivery with RingPaxos[22]. Further consensus implementations may be plugged to the solution.

SMR Manager: Manages the SMR execution receiving and executing the events triggered by *Consensus Delivery*. Upon receiving a command, it is in charge of correctly invoke the respective *Service Instance* module or, if appropriate, invoke the SMR Loader to create a new replicated *Service Instance* module.

SMR Loader: In charge of loading, instantiating new services into the replica system, and starting their provision. In case of a request for loading an unknown service, the source code of the service is loaded (using a run-time class loader) and analyzed (using reflection), discovering which classes and methods it has, the signatures of such methods, and other relevant information (which will be detailed in the Section 4.3). From the analysis performed, the service definition is stored in the form of a *Service Code Base*, which stores the service code plus metadata. Be it a request for a new service or a known one, the next step is to create a new instance of the service, out of the *Service Code Base* definition, in the form of a *Service Instance* which is in charge of executing requests addressed to this new service.

Service Code Base: Every service subscribed to the state machine replica is represented as a *Service Code Base* module, as above.

Service Instance: After registering a service in the *Single State Machine*, it is possible to instantiate it several times, generating SMR applications completely independent of each other, that is, with identical initial states but distinct states evolution. As in the Object Oriented Programming paradigm, where an object is described following a set of rules and standards and can later be instantiated several times, this approach makes it possible to instantiate several distinct replicated applications from the same Service code base.

Consistency: The overall state of the SMR is composed by the state about services under provision (metadata) and the state of each service being provisioned.

SMR commands are ordered across replicas by the consensus protocol. This guarantees that the execution of commands follows the same total order in all replicas. The state of the SMRManager is replicated across replicas and is updated following the SMR rules. Therefore, replicas will have consistent states regarding the set of services under provisioning.

Since service provisioning commands are broadcast using the same consensus protocol as commands for user services, from the moment a new service starts operating to the moment it ceases operation the set of commands delivered specifically for that service is exactly the same and follows the same total order in all replicas. We assume that each command execution is atomic. Therefore a service request will only be processed after service deployment has completed.

According to the above, from the consistency properties of SMR, both the state about services being provided and the state of each service are consistently replicated, thus the overall consistency is assured.

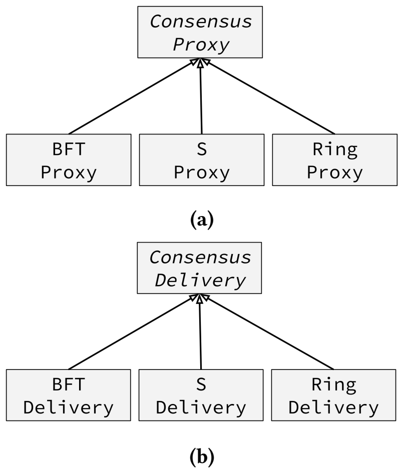

Figure 4: Generalization of specific consensus protocol functionality: (a) client side; (b) replica side.

4.3 Application Programming Interface

Our replication library is implemented in Java and is available through the package `r3lib`. The main resources to software developers, provided by library API, are a set of custom annotations, made available by the package `r3lib.annotations`, and methods for registering and excluding applications, made available by the package `r3lib.communication`.

The specification of methods that are visible to SMR clients (i.e., the state-machine commands) is done through annotations.

An annotation is a syntactic metadata that can be added to the source code. Annotations are embedded to the compiled code and interpreted by the virtual machine at run-time via reflection. This design decision aims to facilitate the implementation of replicated applications, since developers do not need to redesign the code structure. Thus, the application code remains unchanged, except by the inclusion of annotations.

The `SmrMethod` annotation identifies which methods are *SMR methods*. A SMR method can be invoked through a unique identifier, the *commandID*. Only these methods can be viewed and invoked by clients. Classes that have at least one *SMR method* are called *SMR Classes*. Descriptions of *SMR methods* and their parameters can be added by developers through `SmrArg` annotation. The `SmrArg` can be used to detail *SMR method* parameters, naming and describing them. Although descriptions are optional, they can be used to document method's details and instructions for use. The *commandID* field associates a command name to a given method. If the *commandID* is not explicitly defined by the developer, a unique command name is generated by the library.

A simple annotation example is given in Listing 1. A *helloWorld* method is annotated with `@SmrMethod` and a command name `hello` is associated to it. By simply adding this annotation, *helloWorld* method becomes available for remote invocation from clients through the command identifier *hello*.

```
@SmrMethod(commandID = "hello")
public String helloWorld() {
    return new String("Hello world");
}
```

Listing 1: Annotation Example

Once application methods are annotated, applications should be registered and instantiated on the replicated service. Firstly, the application code is registered on the replicated system as a new *Service Code Base* through the method `newService`. This method will pack the whole application into a JAR file and associate it to a unique identifier. Once a new application is registered in the replicated system, it can be instantiated one or several times through the method `newService`. Every time `newServiceInstance` is invoked, it generates a new *Service instance*, with its own state. Thus, each service instance has a unique identifier, a set of classes provided by a *Service code base*, and its own state.

Listing 2 describes the methods implemented by the `ConsensusProxy` class. Every proxy is associated to a single client, then the constructor receives the client identifier as argument. This class implements methods for creating new libraries with a JAR file and the library name as parameters, and new applications with the library and application names as parameters. Clients invoke methods through the method `invokeMethod`, passing the command identifier and a list of *Objects* as parameters. A directory service is offered through methods `getAvailableServices`, `getServiceDoc`, `getAvailableInstances`, and `getAvailableCommands`. By executing these methods, clients can discover which services or service instances are available, check the documentation of a service, and get references to the commands made available by a given service instance. Finally, methods `updateService` and `removeService` are responsible for updating or removing services, and method `removeInstance` for removing service instances. Because the library does not support state retrieval yet, the `updateService` primitive has not yet been implemented.

```
public class ConsensusProxy {
    public ConsensusProxy(int clientId);
    public void newService(java.nio.file.Path jarFilePath, String servName);
    public void newServiceInstance(String servName, String instName);
    public Object invokeMethod(String commandID, Object... params);
    public List<String> getAvailableServices();
    public String getServiceDoc(String servName);
    public List<String> getAvailableInstances();
    public List<String> getAvailableCommands(String instName);
    public void updateService(java.nio.file.Path jarFilePath, String
          servName);
    public void removeService(String servName);
    public void removeInstance(String instName);
}
```

Listing 2: Methods implemented by ConsensusProxy class

As discussed in Section 4.2, *Consensus Proxies* and *Consensus Deliveries* are concrete implementations of agreement protocols. Then, application developers can opt by different protocols by choosing a given *consensus proxy*. The consensus proxy and the addresses of replicas that make part of the system are indicated by system parameters presented in a setup file.

5 EVALUATION

In this section we evaluate the impact of using our replication library. Our first goal is to identify the overhead caused by the library. Despite the ease of use and high abstraction provided by our library, we expect a reduction in the throughput and a higher latency to the overall performance. A second aspect to consider is the performance of multiple replicated applications executing in the same infrastructure. Since our library allows multiple applications to share the same Paxos instance, we conjecture that with better use of resources, a better performance in each individual application will be experienced. Finally, we calculate the impact caused by registration of new applications at runtime. The goal of this experiment is to observe how the addition of new services affects the performance of other applications under execution.

We empirically compare the performance of a key-value store application implemented with our library against key-value store applications implemented directly using both BFT-SMaRt and S-Paxos libraries. Load is generated by client nodes. In each node, multiple threads simulate clients accessing the application. Every client thread invokes methods repeatedly, but a new invocation is executed only after receiving the reply to the previous one.

5.1 Test environment

The environment used in our experiments was the Emulab[4] [29], a collaborative infrastructure with scientific purposes. Our experimental setup was composed by 45 computer nodes with 3 GHz 64-bits Pentium Xeon and 2GB of main memory, 1 core and network interface gigabit, interconnected by a 1Gb switch. Computers run CentOS 6.9 operating system. Each replica executed in a dedicated

[4]https://www.emulab.net/

node and clients were uniformly distributed among the remaining nodes. In the experiments the replication factor was set to one, i.e., a total of three replicas were executed. In each replica heap size of Java virtual machine was set to 512MB.

5.2 Performance results

Our first set of experiments aims to analyze the overall performance and saturation point of applications implemented with and without the replication library. We start experiments by generating load from a single client node and gradually increasing the number of clients up to a maximum of 42 nodes.

Figure 5 presents a latency versus throughput graph for the four versions of key-value store application: using BFT-SMaRt, S-Paxos, the replication library with BFT-SMaRt consensus delivery implementation, and the replication library with S-Paxos consensus delivery implementation. Throughput was measured in the server side, while latency was computed by clients. Latency is given by the 90^{th} percentile. This way, outliers are discarded.

As expected, a performance decrease is observed when using the replication library. Both BFT-SMaRt and S-Paxos versions can process more than 6000 commands/s before saturating, while versions using our library process between 2700 and 3500 commands/s depending on the consensus delivery implemented. When light workloads are applied to service versions using the library, the throughput reduction is imperceptible. Latency is influenced by the consensus protocol in use. S-Paxos and library with S-Paxos consensus delivery versions latency stays around 0.015s before their saturation point, and BFT-SMaRt and and library with BFT-SMaRt consensus delivery versions latency stays around 0.005s before saturating.

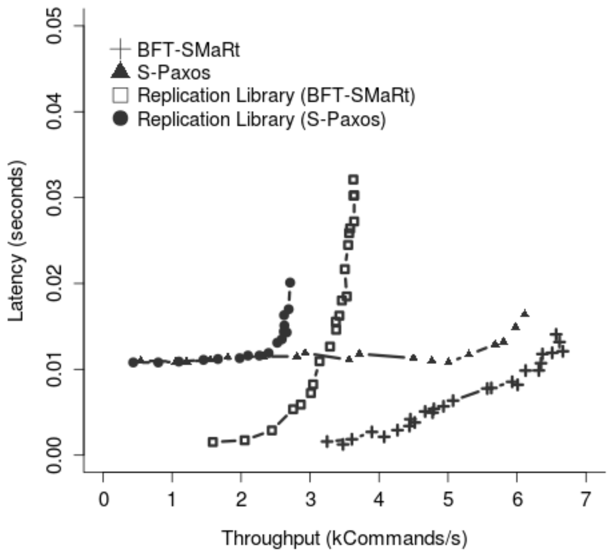

Figure 5: Latency versus throughput for key-value store implementations.

This overhead is caused by the level of abstraction and transparency provided by our library. The execution of a general purpose SMR results in larger messages (extra control information is attached to client requests), additional local invocations to coordinate execution of library components, and additional cost for loading and execute dynamic code. Since reflection involves types that are dynamically resolved, certain Java virtual machine optimization cannot be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts.

As future work, we intend to reduce the costs with serialization and reflection by using alternative implementations. For instance, Kryo[5] and Skyway [24] implement serialization libraries. Studies performed in [24] indicated a considerable performance increase of these libraries over traditional Java mechanisms.

Although our library presents a lower performance for a single application, it makes better use of resources when multiple applications share the same computational nodes. Therefore, we investigated the performance impact caused by the addition of applications under execution in the same infrastructure. We devised scenarios with one, two and three instances of key-value store application. The number of clients per application instance was equally distributed and it corresponds approximately to 80% of the maximum load.

Figure 6 shows the throughput obtained by key-value store implementations with 1, 2 and 3 running instances. In the test scenario with a single instance, the higher throughput is achieved without using our replication library, as expected. Application running the BFT-SMaRt version processes approximately 5400 commands/s, while the version implemented with our library and BFT-SMaRt consensus proxy processes 3500 commands/s. However, as application instances are added, the throughput achieved by application without our library reduces drastically while the throughput achieved by instances using our library remains unchanged. In both scenarios with 2 and 3 running instances of key-value store we can observe a better performance with our library. For 3 instances, for example, our library using BFT-SMaRt consensus proxy remains around 3400 commands/s while BFT-SMaRt throughput drops to 2400 commands/s and S-Paxos drops to 1300 commands/s.

The reason for keeping performance levels despite the number of running applications when using our library is the sharing of the same consensus instance. Replicas do not need to instantiate additional Paxos processes. This is advantageous, since Paxos components demand processing power and especially storage access for logging accepted values. Although less impacting, the management of resources in the system level is also alleviated. For instance, a smaller amount of memory is allocated, fewer context switches and interruptions are handled.

The next experiment aims to identify how the registration of new applications impacts other applications under execution. Towards this end, a key-value store application was registered and subjected to a constant load of clients. Then, 3 extra applications were registered at different instants. The new applications were composed by (i) 277 classes and 3801 methods; (ii) 230 classes and 3299 methods; and (iii) 629 classes and 12354 methods.

Figure 7 shows the throughput of the key-value store application while the new applications are registered. As observed, there is a throughput drop when applications are being registered (around 110s, 210s, and 260s). This effect is caused by the creation of class

[5]https://github.com/EsotericSoftware/kryo

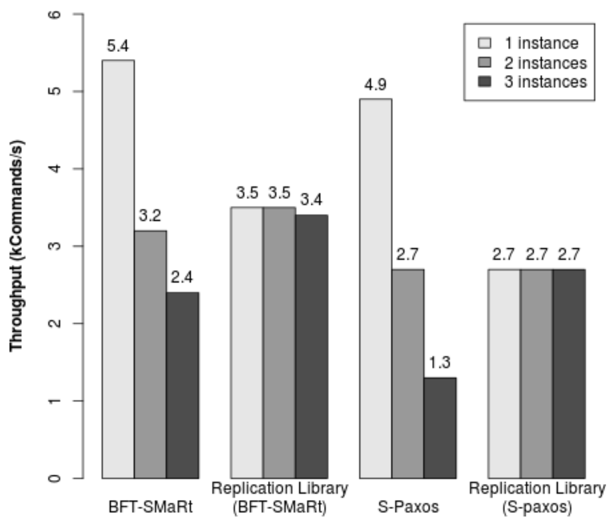

Figure 6: Multiple instances throughput

loaders for each new application. The class loader is responsible for updating the directory service with new methods and load the application classes into the Java virtual machine. Notice, though, that the throughput drop lasts for a very short period, which represents a minor prejudice to the application.

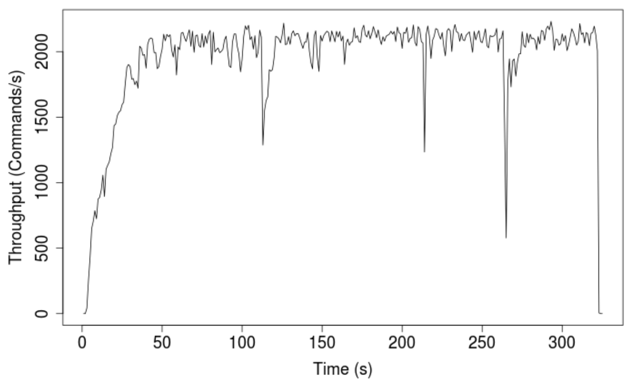

Figure 7: Performance impact caused by registration of new application.

6 RELATED WORK

There is a substantial body of work on State Machine Replication approach. Associated to these works, there is a long literature about consensus protocols, such as Paxos. However, as observed in a recent literature, building practical systems using SMR concepts normally requires experience and a solid knowledge about fault-tolerance and distributed systems [1, 7, 9, 16].

In [7] authors implement a fault-tolerant database using the SMR approach. They mention it was significantly harder to build the database. They attribute the difficulties encountered to gaps between the description of the consensus algorithm and the needs of a real-world system. According to authors report, converting the algorithm into a practical, production-ready system involved implementing many features and optimizations. Some of them published in the literature and some not. In addition, they emphasize that changes in specification during software development is common. Then, an implementation should be malleable. However, promoting changes to intricate code designed for particular fault-tolerant applications is cumbersome. In another work [16], authors mention that technical details, which are usually looked at as engineering considerations, actually have large liveness implications and can dramatically impact performance.

In order to reduce the complexity of developing and deploying replicated systems, different strategies have been proposed. Next we discuss other approaches for transparent replication and discuss how they differ from our replication library.

CRANE [9] is a parallel SMR system that transparently replicates general multi-threaded programs. Within each replica, CRANE intercepts POSIX socket and the Pthreads synchronization interface and implement deterministic versions of such synchronizing operations. To ensure total order delivery of synchronization commands across replicas, for each incoming socket call (e.g., accept() or recv()), CRANE runs a distributed consensus protocol, so that correct replicas see exactly the same sequence of calls. CRANE schedules synchronization commands using deterministic multithreading (DMT). This technique maintains a logical time that advances deterministically on each thread's synchronization.

In [30] authors propose a middleware for fault-tolerance in cloud computing environments. Similarly to CRANE, applications programmed using TCP socket API can be replicated by it. Fault-tolerance is achieved by implementing a leader/follower replication approach.

OpenReplica [1] provides transparent object replication, then clients need not be aware that certain components have been replicated. The OpenReplica implementation is based on the Paxos protocol. For performance reasons, they implement a light-weight version of Paxos built on asynchronous events. Binary rewriting is used to guarantee that clients invoke their replicated objects in the same way they invoke local objects. In OpenReplica it is possible to change the location and number of replicas at runtime. To ensure consistency upon reconfiguration, a view change protocol is implemented.

Similar to our work, CRANE and OpenReplica provide high transparency when developing fault-tolerant services. CRANE rewrites the standard APIs for process communication and threads synchronization. Thus, distributed applications developed with such basic primitives can easily migrate from standard libraries to CRANE without changing the application logic. One disadvantage, however, is that developers have to master distributed systems programming. OpenReplica provides transparency by replicating objects. This level of abstraction does not require explicit implementation of sockets or threads to establish processes communication.

The operation mode of OpenReplica is very similar to the one proposed in this paper. Both provide a very simple way to implement methods that can be replicated and made available to SMR clients. However, every new application initialized by OpenReplica launches a new set of replicas. In our approach, new applications can share the same consensus protocol, which has demonstrated a great advantage for applications running in shared infrastructure. In addition, OpenReplica replicates objects, while our library allows replication of large projects, composed by many classes organized in a given name space.

7 FINAL REMARKS

This paper presents a library that supports the development of fault-tolerant applications by making replication transparent to software developers. By using the library, developers can focus mainly in the application logic, with no need to configure the underlying replication system. Applications deployed with our library run into general purpose state machine replicas, which allow the addition or exclusion of service commands at runtime. New application instances can also be added to the state machine at runtime. This way, different applications can share the same replication environment.

This paper describes the overall architecture of the replication environment provided by our library. From a programming perspective, the greatest advantage of this library is its simple and compact API. By just using annotations, software developers can add class methods to the state machine and use them as SMR commands. This simplicity does not restrict flexibility. To emphasize this, we developed a key-value store application adopting design patterns, exploiting the use of non native data types and organized the software structure in different packages.

We experimentally assessed the performance of our library. While additional costs are inevitably added by the replication library, results have demonstrated that the efficient use of resources translates into higher throughput rates when multiple applications execute in a shared infrastructure.

Future steps in this work include the optimization of commands execution. To allow execution of commands loaded at runtime, our library uses reflection. The cost with reflection is the main cause for performance degradation in our library. Thus, alternative implementations of serialization and reflection should be investigated.

The extension of our replication library to the Byzantine domain should be challenging. To enforce isolation between applications running in a shared replication environment a security module would be needed. In addition, durability strategies, such as logging and checkpointing would require further investigation.

ACKNOWLEDGEMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior – Brasil (CAPES) – Finance Code 001, and Fundação de Amparo à Pesquisa do Estado do RS (FAPERGS) – Project PRONEX SISCHIP2/FAPERGS (16/2551-0000 497-8).

REFERENCES

[1] Deniz Altinbuken and Emin Gun Sirer. 2012. *Commodifying replicated state machines with openreplica.* Technical Report.

[2] H. Attiya and J. Welch. 2004. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* Wiley-Interscience.

[3] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. 2014. State machine replication for the masses with BFT-SMaRt. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on.* IEEE, 355–362.

[4] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. 2012. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on.* IEEE, 111–120.

[5] Dhruba Borthakur, Jonathan Gray, and Joydeep Sen et al. Sarma. 2011. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11).* 1071–1080. https://doi.org/10.1145/1989323.1989438

[6] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06).* 335–350.

[7] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07).* 398–407.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 1–26.

[9] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. P axos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles.* ACM, 105–120.

[10] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.* 36, 4 (2004), 372–421.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. 2003. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles.* 29–43.

[12] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[13] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *ATC*, Vol. 8.

[14] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on.* IEEE, 245–256.

[15] J. Kirsch and Y. Amir. 2008. Paxos for System Builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS).* 1–6.

[16] Jonathan Kirsch and Yair Amir. 2008. Paxos for System Builders: An Overview. In *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08).* ACM, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/1529974.1529979

[17] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[18] L. Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169.

[19] Leslie Lamport. 2005. *Generalized Consensus and Paxos.* Technical Report MSR-TR-2005-33. Microsoft Research (MSR).

[20] Parisa Jalili Marandi, Samuel Benz, Fernando Pedone, and Kenneth P. Birman. 2014. Practical Experience Report: The Performance of Paxos in the Cloud. *CoRR* abs/1404.6719 (2014).

[21] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2012. Multi-Ring Paxos. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on.* IEEE, 1–12.

[22] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. 2010. Ring Paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on.* IEEE, 527–536.

[23] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13).* 358–372. https://doi.org/10.1145/2517349.2517350

[24] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. SKYWAY: Connecting managed heaps in distributed big data systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 56–69.

[25] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm.. In *USENIX Annual Technical Conference.* 305–319.

[26] F. B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[27] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on.* Ieee, 1–10.

[28] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems.* ACM, 18.

[29] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 255–270.

[30] Wenbing Zhao, PM Melliar-Smith, and Louise E Moser. 2010. Fault tolerance middleware for cloud computing. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on.* IEEE, 67–74.