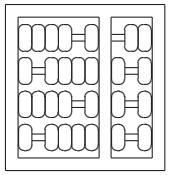


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Implementation of an Object-Oriented Specification for Active Replication Using Consensus

Gustavo M. D. Vieira *Luiz E. Buzato*

Technical Report - IC-10-26 - Relatório Técnico

August - 2010 - Agosto

Implementation of an Object-Oriented Specification for Active Replication Using Consensus

Gustavo M. D. Vieira* Luiz E. Buzato†

Instituto de Computação, Unicamp
Caixa Postal 6176
13083-970 Campinas, São Paulo, Brasil
{gdvieira,buzato}@ic.unicamp.br

Abstract

Most of the software tools created so far to aid in the construction of distributed applications addressed how to replicate data consistently in the presence of failures, but without offering much relief for the problem of building a dependable and long-running application. This paper describes our experience building Treplica, a replication toolkit offering application developers a very simple programming model based on an object-oriented specification for replication of durable applications. Treplica simplifies the development of high-available applications by making transparent the complexities of dealing with replication of data that must survive process crashes and recoveries. These complexities are not negligible, and we believe we have found a compelling way to address this problem under a simple to understand object-oriented interface. We have used Treplica successfully to add fault tolerance to a implementation of the TPC-W benchmark and we have obtained very good performance, even in the presence of failures.

1 Introduction

For more than three decades system developers have pursued the goal of connecting off-the-shelf computers together using standard network resources to obtain a system with better availability than any of its individual parts. The system obtained this way has greater availability because it contains sub-systems to spare. For example, each computer of the system can contain a copy of some critical process so that partial computer failures are guaranteed not to make the system or the application it hosts unavailable to its users. Unfortunately, the full potential of redundancy and replication can only be successfully harnessed if three main obstacles are overcome: (i) performance, (ii) availability despite component failures and (ii) programming simplicity.

*Partially supported by CNPq grant 142638/2005-6.

†Project partially supported by CNPq grants 47340/2009-7, 201934/2007-8 and FAPESP grant 2009/06859-8.

These three seemly simple goals are very hard to reach in practice due to asynchronous nature of distributed systems. Nonetheless, many solutions exist for replicating data and services with many different suppositions regarding system models, replication guarantees and application behavior [21]. However, with the exception of data intensive solutions for relational databases, few solutions tackle the problem of managing replication of applications whose services and data must be always available for long periods of time. The designer of this class of distributed applications faces the daunting task of maintaining consistency in the presence of unpredictable failures and concurrency.

This paper discusses our experience in building and using Treplica, a replication library that overcomes (i) by implementing consensus-based active replication and (ii) by offering the application developers a very simple programming model based on an object-oriented specification for replication. Treplica has been designed to be resilient, transparent and efficient. Resiliency means that Treplica implements at its core a replication protocol that gives applications the ability of tolerating crashes and recoveries of a subset of their replicated components without having to worry about the consistency of the replicated state. Treplica guarantees resiliency through consensus-based active replication, specifically the Paxos [26] algorithm. Transparency guarantees that programmers can develop replicated distributed applications without having to be concerned about how replication is actually implemented. In fact, application programmers can program their stateful applications as a set of stateless objects. Concerning efficiency, experimental results show that Treplica can provide the necessary processing capacity to guarantee very good application response times.

In summary, Treplica simplifies the development of high-available applications by making transparent many of the complexities related to consistent replication and recovery in the presence of failures. These complexities are not negligible; care must be taken to correctly implement the replication algorithms, detect and manage failure, perform recovery, among other issues [3, 13]. We believe we have found a compelling way of factoring out these concerns under a simple to understand programming interface. Treplica stands in the middle ground between the low level flexibility of message-based group communication toolkits and the extensive data processing capabilities of databases. The main contributions of this work are:

- The design and implementation of an object-oriented abstraction for replication as a way to simplify the construction of dependable and long-lived applications.
- The use of consensus as a foundation for the implementation of this modular abstraction.
- The description of the software architecture of Treplica accompanied by a detailed discussion of our design choices and the problems that motivated them.
- The experimental validation of Treplica's performance and availability. Treplica shows good performance and uninterrupted service, even with multiple failures.

This remaining of this paper is structured as follows. Section 2 gives an overview of Treplica, goes in more depth in the rationale for its creation and outlines its software

architecture. Section 3 describes the object-oriented specification for replication while Section 4 gives an example of how this specification is used in Treplica to build a complete application. Section 5 goes into more detail on the internal structure of Treplica, describing our implementation of the Paxos protocol. Section 6 briefly describes the typical profile of applications built with Treplica and Section 7 shows the performance attained by one such application, the TPC-W benchmark. Section 8 discusses related work and Section 9 makes some concluding remarks.

2 Treplica

2.1 Rationale

Replication is a crucial mechanism used in distributed systems to increase the system reliability and performance. Active replication is a general technique to replicate the internal state of processes that prioritizes consistency [33]. In active replication all processes sharing the same state, called *replicas*, behave as deterministic state machines. All replicas share the same source of events that trigger transitions in their underlying state machine. As a consequence, all replicas stay the same as long as they process the same sequence of events.

There are many forms of implementing active replication. One of the more common is to employ a total order broadcast primitive to propagate events orderly among the replicas [17]. As an example, take the usual situation of a set of servers providing service to a set of clients. In this scenario, a client can broadcast its request to the set of servers using the total order broadcast. All the servers will observe the request at the same position in its events sequence and will perform the same operation, yielding the same result. The client picks the first answer it receives. The conceptual simplicity of active replication over a total order broadcast primitive makes it a very used solution in practice.

However, it is necessary to consider the relationship between the properties of the total order broadcast primitive and of the application being developed. In particular, it is necessary to establish how the state of the total order broadcast relates to any local state maintained by the application, specially in the presence of failures. By definition persistent data survives failures, thus any information transmitted by the total order broadcast primitive that do not survive failures is a potential source of inconsistency for the application. To illustrate this point we take as an example the more mature way to implement total order broadcast: virtual synchrony-based group communication.

In the virtual synchrony model, message delivery is constrained by views of operational processes maintained by a group membership service [8]. This group membership service supports dynamic groups where processes join and leave the system at will, either explicitly or due to a failure. This service acts as the basic fault tolerance mechanism, hiding from the programmer the need to monitor the occurrence of failures. All messages sent during the lifetime of a view are received by all processes encompassed by it, and all message delivery guarantees such as total order are enforced. However, group membership in virtual synchrony assumes a crash-no-recovery failure model. If a process fails, it

can only rejoin the computation when a new view is instated. If the failure of a process is wrongly detected, the process is forced to shutdown and rejoin the group to guarantee view consistency [8].

Whenever a process joins a group, creating a new view, it is assumed this is the first time this process is seem by the group. That is, there is no explicitly defined rejoin operation. Processes are assumed to be stateless and they must catch up with the group state by means of a state transfer from another process in the group. This behavior directly affects the type of failures supported by the application. Take for example a distributed application where all replicas reside in the same cluster. If it is possible to guarantee the whole set of replicas never crashes completely, one can use the shared state maintained in the main-memory of these replicas to preserve the application state. However, if one must tolerate whole cluster failures, some stable storage must be used.

Specifically, each replica must store and update its complete state in disk to account for the situation it is the last one to fail in the cluster. During recovery of a failed replica, it runs a protocol to determine if it is joining an existing group or creating a new group. If it is joining, it should discard all its local state and restart from the state currently held by the replicas in the group. Thus, all processes use costly stable memory, but it is only necessary by a single process in the less likely event of a total crash, instead of the more common occurrence of a partial crash. One can circumvent this basic behavior by creating an application specific protocol that makes the state transfer more efficient. This can be accomplished by using as much as possible the local persistent state held by a replica to complete the state held by a view of processes. However, it rests on the programmer the hard task of designing and implementing this protocol. Moreover, if a replica is *wrongly* suspected of having failed, it still must restart its operation and discard all its local state.

Although we have used virtual synchrony as an example, the problem just described comes from the necessity of synchronizing total order delivery state and application state. In fact, all properties of the total order primitive being used and of the resulting application must be cautiously matched, considering consistency suppositions, failure models and other aspects. This way, the lower level details contaminates all the upper layers including the programming abstraction, making the task of the application developer much harder.

2.2 Overview

Treplica is a replication library designed to provide a simple and object-oriented way to build highly available applications. These applications can encompass the entire system or be restricted to crucial sub-systems where performance, consistency and reliability are central. To reach this goal, we decomposed the problem of implementing replication in components with simple and clearly defined interfaces. So, a developer who wants to implement a replicated distributed application does not reason in terms of messages, processes, failures or data items. Instead, he reasons about the execution of the application operations, transitions of a *replicated state machine*, that are triggered by events that are made available through an *asynchronous persistent queue*. Treplica is an implementation of this object-oriented specification for replication.

We decided to expose the state machine component to the developer as a programming tool using the reflection facilities of modern languages to encode and execute state and state transitions. Using state machines as a concrete programming interface is desirable because states and transitions are easily implemented as objects. Treplica is implemented in Java, and in this language the application state is represented by serializable objects and actions as runnable, serializable objects. The object-oriented specification embodied by Treplica can easily be implemented in any other dynamic language and, with some extra programming effort, in more traditional languages such as C.

The main design decision underlying Treplica is to allow the programmer to consider the application as being stateless, leaving the actual durability of the application to the library. This decision is supported by the observation that the same requirements of active replication can be used to provide a simple but powerful persistence mechanism. Active replication requires the application to perform actions that change its state in a deterministic way. These actions are then broadcast, in the same order, to all replicas that locally replay them. Within this same framework, we consider that the actions aren't only sent to the other replicas but logged to stable storage [9]; this way it is possible to recover from failures by replaying the log. Determinism ensures that after each recovery the application will restart in the same state it was before the failure. For efficiency and ease of implementation, we require that the application fit in main-memory, as we do not provide any means of selectively unloading parts of the application state to secondary memory. With the current size and cost of main-memory, we don't consider this limitation to be a problem for the class of applications that can benefit from using Treplica.

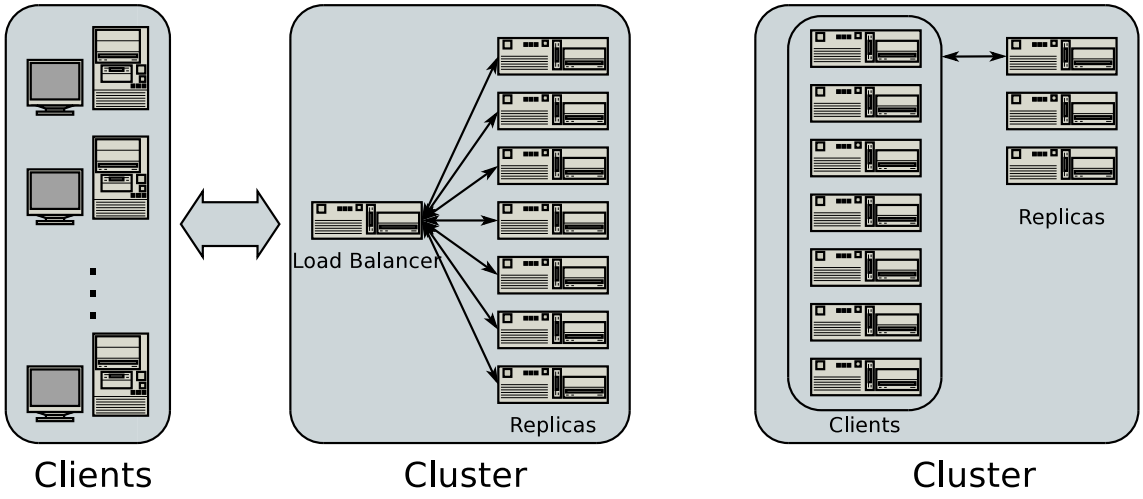
To support active replication in Treplica, we have decided to concentrate on consensus-based total order algorithms for the crash-recovery failure model. The Paxos algorithm and its variants are examples of specially suited algorithm of this class, as it was created with active replication in mind. These algorithms are particularly interesting because they provide the continuous delivery of messages to a replica even in the presence of failures and recoveries. This allows Treplica to have a simpler software architecture and increases its potential for good responsiveness in the presence of partial failures. Moreover, these extra guarantees allows Treplica to avoid expensive coordination of the local application state and the shared state during recovery. Obviously, relying on stronger guarantees implies a larger cost to deliver messages. However, for this class of algorithms, this cost is related to writes to stable memory and these writes are already required to ensure the application can survive catastrophic failures. By combining the stable memory requirements of the application and the total order primitive we were able to obtain a good failure-free performance that is minimally affected by the occurrence of faults.

2.3 System Specification

The target platform for Treplica are commodity clusters. The main characteristic of such clusters is that the nodes are connected by a high bandwidth and low latency interconnect network that supports broadcast. The replicas exchange messages through this network to keep the shared state consistent while potentially serving client requests. Throughout this paper, *client* is any process that does not have a copy of the replicated state. Only

replicas hold the replicated state and only them are able to use Treplica services to query and change this state. Clients depend on the replicas, that act as servers, to perform these actions in their behalf. In fact, clients often interact with a higher level abstraction provided by the replicas and are unaware of the existence of the replicated state. We call this higher level view of the set of replicas an *application*.

Treplica does not restrict how the clients access the application or how their access is load balanced among the replicas. The application is free to implement its service in many ways, as long as the guarantees provided by Treplica are sufficient. For example, it can serve remote clients using a RPC mechanism, it can implement a web service, it can serve local clients through sockets, etc. Treplica does not dictate or implement any such mechanism, leaving the designer free to choose the more appropriate solution for a particular application. Figure 1 shows two examples of possible cluster configurations. Figure 1(a) shows a setup where remote clients connect to replicas in a cluster mediated by a load balancer acting as a reverse proxy. Figure 1(b) shows a group of clients that share the cluster with the replicas and access an application elected master.



(a) Clients access a load balancer

(b) Clients access a master node

Figure 1: Cluster Configurations for Replication

The replicated application state is left under the control of Treplica. This way the application programmer should not be concerned with replica management or fault-tolerance implementation details, as shown in Figure 2. For simplicity of implementation and performance, the entire replicated state must fit in main-memory. However, this isn't an intrinsic property of the design, only a characteristic of the current implementation. More importantly, the application state must change only in a deterministic and controlled way to accommodate active replication.

The architectural restrictions imposed by Treplica affect only the replicated state. Usually, information kept by the application that only regards the local status of the connections with its clients are not replicated and are kept in local volatile memory. Moreover,

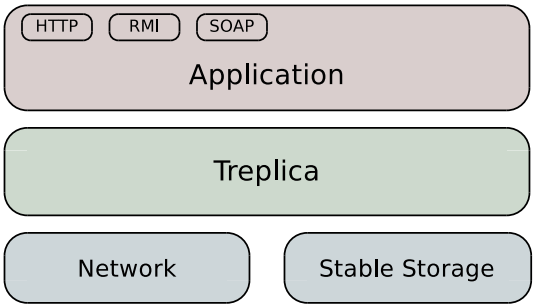


Figure 2: Software Architecture of an Application

any data kept by the application that does not require replication or persistence can be stored in any way required by the application designer.

3 An Object-Oriented Abstraction for Replication

Our proposal of an object-oriented abstraction for replication is based on two main components: *replicated state machine* and *asynchronous persistent queue*. Figure 3 shows the interface of these components and their relation to the application and to each other.

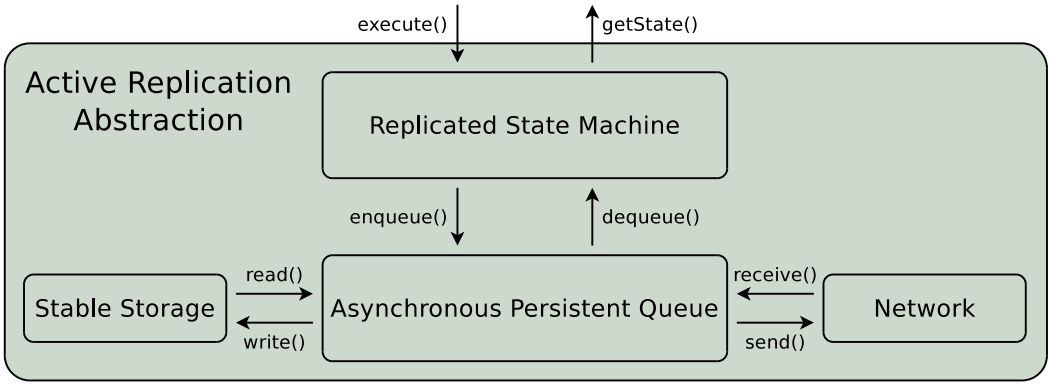


Figure 3: Active Replication Components

The replicated state machine provides an abstraction to the operation of any deterministic application. It allows the maintenance of application state by stipulating a simple interface for querying and modifying such state. This component is accessed directly by the application that uses its services to hold, replicate and persist its state. All these operations are performed transparently, and require no intervention from the user of this component. The asynchronous persistent queue is an abstraction for a persistent and fault tolerant object queue. It represents an ordered record of objects sent to a group of processes, that is guaranteed to be available even if all processes in this group fail. It can be used as a persistent log of events triggering transitions in the distributed state

machine. In fact, this component is more general and could be used in other settings as it represents an abstraction for a consensus service, useful in other contexts besides replication.

In the rest of this section we describe the specification of these two components. Our description follows a bottom-up approach, starting with the asynchronous persistent queue and then moving up to the replicated state machine. This way it is easier to isolate the services provided by each component, how these services can be used and the design decisions related to their provision.

3.1 Asynchronous Persistent Queue

Asynchronous persistent queues are a way for a group of processes to exchange objects. These objects are sent by any process connected to the queue and broadcast to the others, totally ordered and with guaranteed delivery regardless of failures. This behavior can be more precisely described by the following three properties:

- Objects are delivered in the same order to all processes.
- Objects are delivered to all processes, even if a process crashes and later recovers.
- Objects are persistent and survive crashes of all processes.

These properties are very similar to the properties of total order broadcast [17], but state explicitly that a failed process that eventually *recovers* must also receive all ordered objects. Each process that interacts with the queue component does so through a *queue endpoint*, bound to a specific queue. The primitives of the asynchronous persistent queue component are very simple:

`enqueue(object)`: Adds an object to the end of this queue, making it available to all other processes.

`dequeue()`: Removes the next object from this queue.

The `enqueue()` method changes the state shared by all processes, the queue itself. The contents of the queue should be consistently managed ensuring that all calls to `enqueue()` in every process generate only a single ordering of all objects. Correspondingly, every call to `dequeue()` made by the processes sharing a queue will reflect this same order. Each queue endpoint has associated with it the object delivery history. For instance, a new process joining a queue, using a new queue endpoint, will receive all objects ever sent to the queue. These objects are both local queued objects or objects queued by other processes, and they may be stored locally or fetched from the network. From the point of view of the client process, this distinction is irrelevant. Thus, by relying on the total order guaranteed by the queue and in the fact that queues are persistent, individual processes can become replicas of each other using active replication, while remaining in their perspective completely stateless.

To efficiently provide this high level abstraction to the client process, it is necessary to define some mechanism to limit the number of objects in the queue. Suppose a process fails after having executed for a considerable time and then recovers. It is the responsibility of the queue to provide it with its recovery state in the form of an object log that, in this case, can be very large. To reduce the size of this log one might periodically take snapshots of the queue, save them to stable storage and rollback the process to one of such snapshots when necessary. The problem with this approach is that the persistent queue abstraction promises the application it will receive all objects in a queue, regardless of failures. Our solution to this dilemma is what we call *queue controlled persistence*, where a snapshot of the application is stored alongside with a snapshot of the queue to stable storage. The queue handles the coordination of local snapshots among all replicas and guarantees that each replica always sees a sequence of objects *consistent with its state*. This means a process, remaining stateless, never misses a single object even when just a subset of the objects are re-delivered in the presence of failures and recoveries. This requires the process to put its state under control of the persistent queue, by being instrumented with get and set state procedures that are callable by the persistent queue implementation. Two extra primitives are added to the persistent queue component to bind it with the entity responsible for storing the application state and to control the checkpointing process:

`bind(stateManager)`: Binds a process state, represented by its state manager, to a queue endpoint. The state manager is any application component capable of implementing the `getState()` and `setState(state)` primitives.

`checkpoint()`: Instructs the queue to save a current snapshot of its state, including the process state.

At any time, but specially during the call to `bind()`, the state manager must guarantee that it is able to take a meaningful snapshot of the process state and it is able to replace the state with a snapshot provided by the queue. By correctly choosing an appropriate snapshot, the local state of the client always will be consistent with the next object to be received from the queue. This may require, if a process fails and falls behind the others, that upon recovery the queue replaces its local state with any suitable snapshot obtained from the other replicas. This snapshot can either be in the logical past or future of the state the process had when it crashed. Similarly, even if a process just falls behind the other but does not fail, its state still can be changed by the queue, but in this case only to a forward state. Thus, to support the strong guarantee of queue persistence the application not only can be stateless, but it is required to be stateless. Some control over the process of snapshot creation is provided by the `checkpoint()` operation. This method is provided so the client process of the persistent queue can influence the time a snapshot is taken, but the queue implementation is free to implement its own checkpointing policy.

3.2 Replicated State Machine

Using the guarantees provided by the asynchronous persistent queues it is straightforward to build a set of replicas using active replication. It is possible to use the ordered

sequence of objects provided by the queue component to implement the replicas. This would require the application programmer to build some type of deterministic state machine to use active replication, to convert operations on this state machine to data, to build a monitoring subsystem to service the client requests synchronously, and to create a state manager to handle the set and get state operations required by the queue component. These are exactly the functions performed by the replicated state machine component. This component provides a higher level abstraction that supports the construction of replicated state machines with minimum effort.

The state machine component is a very simplified version of a finite state machine. It does not concern itself with the definition of all states, transitions, conditions and actions. It just treats the set of all states as a black box, and routes all external generated events to this black box. The state machine component is simply a framework to event logging, where events generate changes in a deterministic state. If the application requires a more complete implementation of a state machine, it is free to do so by using the persistent queue component directly.

The replicated state machine component allows the state it manages to be changed only by executing *actions*. An action is a data item that represents an operation to be performed on the stored state and its parameters. The existence of an action represents the occurrence of an event that may trigger transitions in the underlying state machine. The component doesn't care how transitions are implemented, thus an action must encode the conditions and operations that should be performed. Locally, each replica stores all its state in the replicated state machine and only changes it using actions passed to the `execute()` method. The primitives provided by the replicated state machine component are listed below:

`create(initialState, queue)`: Creates a new state machine bound to a queue. An initial state should be provided, because the replica that calls this method can be the first one to bind to this queue.

`getState()`: Returns the current state of the state machine. A process can query this state at will, but cannot change it.

`execute(action)`: Executes an action on the distributed state, performing all necessary steps to coordinate this change with the other replicas.

Once a state machine is created with a template initial state, the actual state of the application is unknown and can change at any time. If the application wants to consult its state, it should first obtain a updated version by calling `getState()`. The state can be queried at will, but changes can only be performed by creating suitable actions and passing them to the `execute()` operation. Actions applied to the state machine by the local client are only performed by the state machine after they have been submitted to the asynchronous persistent queue component. The local client of the state machine perceives the execution of the action as a call to a blocking primitive. A successful return of the call guarantees that the action submitted has been performed by this replica and the effects of such execution are visible in the local state. As the underlying queue is asynchronous,

the fact an action was executed in one replica does not imply that it was performed in all replicas.

By its use of the asynchronous persistent queue all actions are made persistent and the state held by this abstraction is under the management of the queue. The `create()` operation can either start operating with the provided state or it can recover some state from the queue. Once a suitable state is found and installed, all pending actions in the queue are replayed and the state machine is ready to resume operations. This means that, from the point of view of the client of the state machine component, recovery is completely transparent. However, the client must be aware of this fact and avoid keeping local state associated with the replicated object, that is, it not only may be but it required to be stateless.

The replicated state machine component has only three simple primitives that implement a well-defined and easy to use programming abstraction. Thus, the major task a programmer will have to perform to use this abstraction is the definition of the application state and of the actions that modify the state, regardless of state persistence, state replication, checkpointing and recovery concerns. It is worth to note that this step is usually carried out even for applications that do not have replicated state, so it does not add complexity to the development process.

4 Treplica by Example

We now describe a simple application using Treplica to make clear the service provided by the abstract components. We focus in how these services can be used to create a replicated application and how this application can be programmed using Treplica. To this end, we develop a simple hash table application that maps a string key to a value. This application exports its service to remote clients through a SOAP interface composed of two simple methods: `get(key)` and `put(key, value)`.

The software architecture of the complete application is very similar to the one depicted in Figure 1(a). The application is replicated using Treplica, thus providing dependable operation to its clients. The clients only know a single SOAP descriptor and are unaware of the fact the application is replicated. Each replica is organized as shown in Figure 4. The application interacts with Treplica using the replicated state machine component described in the last section. The replicated data is managed by Treplica, stored in the state machine component.

We start the creation of the application by implementing its most basic data structure: the hash table. Using Treplica it is possible to create a distributed hash table by simply extending the hash table implementation found in the Java standard library (`HashMap`) and making it comply with the Treplica replication abstraction. The initial step of this process is to define what constitutes the application state and how it can be changed: the events and transitions. This example shows that we do not have to explicitly define states and transitions. We simply define that the state is held by the Java hash table as a black box and that its state is changed by method calls. These calls are the events and their implementation encode the transitions.

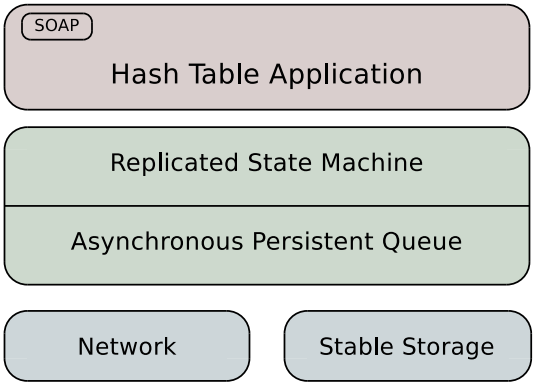


Figure 4: The Hash Table Application

Next, we must assert whether this component behaves as a deterministic state machine. This is done by studying the contract of the object, by analyzing its methods and, if available, by inspecting the source code. In general, objects that do not perform I/O, don't generate random numbers and don't employ date and time are safe. This is the case of the Java implementation of a hash table. If this was not the case, a simple strategy for non-determinism removal can be used. All non-repeatable operations are performed only once by a single replica and the results of these operations are encoded as constant data in the actions.

Finally, we create a proxy class (`ReplicatedMap`) that holds an instance of the original object as its state and uses the Treplica state machine to replicate and persist it, while at the same time presenting the same hash table interface. Example 1 shows a fragment of the proxy class, with its constructor and the two most important methods. This proxy architecture illustrates a common pattern of development using Treplica: we first start with an existing object that encodes the data and functionality we want to have replicated and wrap it in a Treplica aware layer. In this case, the object to be replicated is an of-the-shelf component that is used as a black box, but this pattern is applicable and recommended even if we have access to the object source code.

The `ReplicatedMap` class has as only attribute an instance of a replicated state machine (Line 2). This object holds the state of the application, replicating it and making it persistent. The constructor of `ReplicatedMap` initiates the state machine using one of the factory methods provided by Treplica (Lines 4–8). In this particular instance, a Paxos-based persistent queue is created and the state machine is bound to this queue. The Paxos persistent queue is described in Section 5. An empty `HashMap` is used as initial state and a path to a local directory is set as the stable memory repository. The initial state provided will most certainly be replaced if a previous instance of the `ReplicatedMap` class was created in the same local directory or if it binds to an already existing queue. Other arguments of the factory method instruct the queue to be created considering a specified maximum number of processes, an expected round-trip time and if Paxos or Fast Paxos should be used.

The state stored in the state machine can change continually and, sometimes, the

Example 1 Proxy Class

```
01 public class ReplicatedMap<K, V> implements Map<K, V> {
02    private StateMachine stateMachine;
03
04    public ReplicatedMap(int nProcesses, String stableMedia)
05          throws TreplicaException {
06       stateMachine = StateMachine.createPaxosSM(new HashMap(), 50,
07             nProcesses, true, stableMedia);
08    }
09
10    public V get(Object key) {
11       return ((HashMap<K, V>) stateMachine.getState()).get(key);
12    }
13
14    public V put(K key, V value) {
15       try {
16          return stateMachine.execute(new PutAction<K, V>(key, value));
17       } catch (TreplicaException e) { throw new RuntimeException(e); }
18    }
19 }
```

actual object holding the state (the `HashMap`) may also change. So, before accessing the state it is necessary to get hold of a current object reference by calling `getState()`. The method `get(key)` is implemented by simply obtaining a reference to the current state and executing this operation directly, as it is just a query and does not change the stored data (Lines 10–12). The `put(key,value)` method is much more interesting (Lines 14–18). To implement this method we have created an object that holds the equivalent action. This includes all data required for calling the method and the definition of which particular method should be called. Example 2 shows the action object for the put action of the hash table. This class holds the method parameters as its attributes (Line 3), initialized by the object constructor (Lines 5–7). By the contract of the `Action` interface, it implements the `executeOn(state)` method (Lines 9–12). The caller of this method provides the current application state as argument and the implementation performs the action using the data held in the attributes.

The implementation of the other methods of the hash table are similar and are not shown here. If they only query the values, they are implemented like `get(key)`. If the state is changed, the methods are implemented by means of an action object as in `put(key,value)`. The complete implementation of the proxy will yield a class that can be used in place of any other map implementation in Java. The client of such class doesn't necessarily need to be aware that the object is replicated or persisted as long as its semantics match that of the state machine abstraction, as described in the last section.

Hidden in the `ReplicatedMap` lies all the integration between the application and

Example 2 Action Class

```
01 protected class PutAction<K, V> implements Action, Serializable {
02
03    private K key; private V value;
04
05    public PutAction(K key, V value) {
06       this.key = key; this.value = value;
07    }
08
09    public Object executeOn(Object state) {
10       Map map = (Map<K, V>) state;
11       return map.put(key, value);
12    }
13 }
```

Treplica. The client facing part of the application can be built using any tool desired. In this example, it is done using SOAP. A class implementing the functionality exported by SOAP is shown in Example 3. This class is just another wrapper over the hash table, restricted to the methods we want to export to the clients. The SOAP management is done by an external tool (Axis[1]), setup with the service description shown in Example 4.

The SOAP framework works as an application server hosting the application and insulating it from particulars of the SOAP protocol, such as connection establishment and arguments marshaling. Thus, it keeps its internal data out of the reach of the application and of Treplica. As this bookkeeping data is constant (interface description, etc.) or volatile (connection status, etc.) this arrangement is permissible and desirable. This shows how Treplica allows the application to freely organize the aspects of its software architecture that are not related to replication.

5 Treplica Implementation

The asynchronous persistent queue and replicated state machine components form the base of the replication service provided by Treplica. These two abstractions are implemented as objects in the Java language, whose methods are very close to the primitives defined by the components. We have taken advantage of the object-oriented features of the language to simplify the interfaces as much as possible. The most noticeable strategy is that the objects transported by the persistent queue are serializable objects. That is, any object that can have its state automatically extracted by the Java Virtual Machine can be transported by the queue. Also, actions of the replicated state machine are simple serializable Java objects, modeled after the `Command` design pattern [18]. The methods encoded

[1] `http://ws.apache.org/axis/`

Example 3 A Hash Table Application Using SOAP

```
01 public class HashTableApplication {
02    private ReplicatedMap<String, String> table;
03
04    public HashTableApplication(int maxProcesses, String stableMedia)
05          throws TreplicaException {
06       table =
07         new ReplicatedMap<String, String>(maxProcesses, stableMedia);
08    }
09
10    public String get(String key) throws TreplicaException {
11       return table.get(key);
12    }
13
14    public String put(String key, String value) {
15       return table.put(key, value);
16    }
17 }
```

Example 4 Fragment of the SOAP Service Description

```
01 <service name="HashApp" provider="java:RPC" xmlns:hash="HashApp">
02  <parameter name="allowedMethods" value="*"/>
03  <parameter name="className" value="br.unicamp.HashTableApplication"/>
04  <parameter name="scope" value="application"/>
05 </service>
```

in the actions are built to act on the state held by the state machine using the data carried by the action as arguments.

5.1 Replicated State Machine Implementation

The replicated state machine component expects the higher level services provided by an asynchronous persistent queue. As described in Section 3.2 the replicated state machine doesn't care about explicit transitions, it just executes actions on the stored state. It is the responsibility of the client to implement, with the appropriate set of actions, meaningful states and transitions.

To support these actions the state machine provides two main services: it manages the binding of the state with the queue and it dispatches and executes actions. To keep the local view of the replicated state bound with the persistent queue it is necessary to implement a state manager. The state machine stores the replicated data for the application, providing a state manager with the required semantics. The creation of a state machine object automatically initiates the binding process, triggering any necessary recovery steps in the queue.

To change the local state, applications create actions and call the `execute()` method of the state machine. Once the action is ordered and executed on the local state, the `execute()` operation returns values or throws exceptions just like a direct invocation of the action. To effectively implement the replication, every action is sent to the queue before it is executed on the internal state. After the queue orders the actions, they are applied on the stored data by the state machine. As we assume the actions change the state deterministically, all replicas will evolve in the same way as actions are dequeued. Also, return values and exceptions are captured and routed to the corresponding calling replica.

However, the persistent queue is asynchronous. This means that the queuing of an object does not guarantee it will be dequeued next. An arbitrary number of objects may be dequeued before a just queued object is retrieved from the queue. Actually, even objects queued locally can be dequeued in a distinct order than the one they were queued. To provide a monotonic increasing view of the stored state, the `execute()` operation of the state machine is a blocking operation. Once an application thread call this operation it is blocked until the action created by this operation is received on the queue, it is executed and return values or exceptions are captured. The call then returns as if these operations were performed atomically.

To support this method of operation the replicated state machine has one internal thread dedicated to constantly receiving objects from the queue and to executing the associated actions on the local state. This thread consults a local data structure with action ids from all locally queued actions and decides if any local thread is blocked waiting the just executed action. If a suitable thread is found, it is woken up and return values or exceptions related to the execution of the action are routed to it. This arrangement allows the application to use multiple threads to service its clients, but Treplica guarantees that only one thread at a time executes operations on the state machine and that locally competing threads will see a consistent order of action execution.

5.2 Paxos Persistent Queue

The asynchronous persistent queue component depends on lower level abstractions: read and write to stable storage and send and receive messages. The service provided by these low level abstractions is not defined in the specification and building a fault tolerant *object* delivery system using them is far from trivial. Thus, Treplica does not assume a single implementation for the persistent queue component. It is defined as a generic interface that can be implemented in many ways that satisfy the properties outlined in Section 3.1.

Despite Treplica generality, we propose the use of consensus-based implementation for the persistent queue component. Specifically, our implementation in Treplica uses the Paxos algorithm. This solution to the consensus problem requires the use of stable memory in a way that allows it to be easily combined with the persistence requirements of the application. Nonetheless, it is possible to implement a persistent queue using other strategies. Besides Paxos, we have implement prototype queues using a virtual synchrony based group communication toolkit (JGroups[2]) and using no replication at all for testing. Actually, the network and stable storage blocks in Figure 2 only reflect our current Paxos-based implementation, as queues may have their own structural requirements.

The Paxos algorithm [26] is, at the same time, a solution to the consensus problem and a mechanism for the delivery of ordered messages with the purpose of supporting active replication [33]. As such, it is a perfect semantic fit for implementing the asynchronous persistent queue component. Paxos implements uniform consensus in the crash-recovery failure model, thus it guarantees that any process that fails and later recovers will still be able to reach consensus. A consequence of this guarantee is that all processes must persist in stable memory information pertaining to the progress of the consensus instances. It is possible to directly derive all state pertaining to the queue from this state. This offers a great advantage, as the cost required to ensure strong consistency by using Paxos is the same that would be required to make the asynchronous queue persistent.

Fast Paxos [27] is an optimistic variant of Paxos that saves a communication round by assuming messages will be naturally ordered by the communication medium. Fast Paxos exhibits the characteristics that make Paxos a good choice for the implementation of a persistent queue. Treplica uses Paxos and Fast Paxos in the main persistent queue implementation (`PaxosPersistentQueue`) in such a way to selectively support both variants. In the remaining of this section we describe the Paxos and Fast Paxos algorithms, their suitability for the replication of persistent data and the implementation of a Paxos persistent queue.

5.3 The Paxos Algorithm

A full description of Paxos and Fast Paxos is beyond the scope of this paper, but we offer here a simple description of their main properties as they relate directly to the implementation. Full descriptions of both algorithms can be found in [27], including the computational and failure models assumed.

[2] `http://www.jgroups.org/`

Processes in the system are reactive agents that can perform multiple roles: a *proposer* that can propose values, an *acceptor* that chooses a single value, or a *learner* that learns what value has been chosen. To solve consensus, Paxos agents execute multiple *rounds*, each round has a *coordinator* and is uniquely identified by a positive integer, the *round number*. Proposers send their *proposal* to the coordinator that tries to reach consensus on it in a round. The coordinator is responsible for that round and is able to decide, by applying a local rule, if other rounds were successful or not. The local rule of the coordinator is based on quorums of acceptors and requires that at least $\lfloor N/2 \rfloor + 1$ acceptors take part in a round, where N is the total number of acceptors in the system [27]. Each round progresses through two phases with two steps each:

- In Phase 1a the coordinator sends a message requesting every acceptor to participate in a round.
- In Phase 1b every acceptor that has accepted the invitation answers to the coordinator with the value of the last vote it has cast.
- In Phase 2a, if the coordinator has received answers from a quorum of acceptors, it asks the acceptors to cast a vote for a suitable proposal.
- In Phase 2b, after receiving a request to cast a vote from the coordinator, acceptors cast their vote for the proposal.
- Finally, a learner learns that the proposal has been chosen if it receives Phase 2b messages from a quorum of acceptors.

Fast Paxos changes Paxos by allowing the proposers to send proposals directly to the acceptors. To achieve this, rounds are separated in *fast* rounds and *classic* rounds. The quorums used by Fast Paxos are larger than the ones used by Paxos and can assume many values that satisfy the requirements of the local rule. In particular, it is possible to minimize the number of processes in a fast quorum ensuring that both a fast and classic quorums contain $\lfloor 2N/3 \rfloor + 1$ processes [27, 34]. A Fast Paxos round progresses similarly to a Paxos round, except that Phase 2 is changed:

- In Phase 2a, if the coordinator has received answers from a fast quorum of acceptors indicating none of them has voted yet, it instructs the proposers to ask the acceptors directly to cast a vote for a proposal of their choice.
- In Phase 2b, after receiving a request to cast a vote from one of the proposers, acceptors cast a vote for a proposal.

This description of both algorithms considers only a single instance of consensus. However, Paxos also defines a way to deliver a set of totally ordered messages. The order of delivery of these messages is determined by a sequence of positive integers, such as each integer maps to a consensus *instance*. Each instance i eventually decides a proposed value, which is the message (or ordered set of messages) to be delivered as the ith message of the sequence. Each consensus instance is independent from the others and

many instances can be in progress at the same time. To support the crash-recovery failure model, both algorithms require the agents to store state in stable memory [27]. The state is comprised of a record of the instances initiated, the round numbers used and proposals made or voted, among other data.

In Paxos and Fast Paxos, any process can act as the coordinator as long as it follows the rule for choosing a suitable proposal in Phase 2a. The choice of coordinator and the decision to start a new round of consensus are made relying in some timeout mechanism, as Paxos assumes a partially synchronous computational model to ensure liveness. Specifically, there can be only one active coordinator at any given time to ensure progress. If two or more processes start coordinator agents, the algorithm can stall as the multiple coordinators compete for the attention of the acceptors with fast increasing round numbers. For this reason, liveness of the algorithm resides on a coordinator selection procedure. This procedure doesn't need to be perfect. Safety is never compromised if zero or more coordinators are active at any time. However, the coordinator selection needs to be robust enough to guarantee that only a single coordinator will be active most of the time. We call the creation of a coordinator agent by a process, guided by the coordinator selection procedure, *coordinator validation*.

Considering the concurrent nature of the consensus instances, a common optimization is done during coordinator validation. Phase 1 and Phase 2a are run once for all the unused consensus instances at that time. In fact, it is always guaranteed that an infinite number of instances are in this situation. The coordinator in Paxos "saves" these instances for future use or, in Fast Paxos, it frees the proposers to use these instances. The improvement brought about by this factorization allows Paxos to achieve consensus in three communication rounds and Fast Paxos in only two communication rounds. Unfortunately, Fast Paxos cannot always be fast. Proposers can propose two different values concurrently, in this case their proposals may collide. Also, process and communication failures may block a round from succeeding. Different recovery mechanisms can be implemented to deal with collisions and failures, but eventually the coordinator intervention may be necessary to start a new classic round [27].

5.4 Paxos And Replication

The Paxos algorithm possesses many useful properties when used as a total order mechanism for active replication. It adheres to the crash-recovery failure model, ensuring that replicas that fail by crashing can later recover and return to the computation. Moreover, it implements uniform consensus, ensuring that even faulty replicas will see the same global order of messages. To see why these properties are invaluable, we will consider the progress of the system from the point of view of a failed replica.

In the event of a failure Paxos ensures us that the information in stable memory is sufficient for a process to recover immediately, without any coordination with the other process. This is possible because the local stable storage of a process includes always consistent status of all consensus instances. Thus, a process that experiences a brief failure, such as a system reboot, just resumes operation normally after restoring its local state. Neither the recovering process or the other processes notice anything unusual. This be-

havior is specially interesting if we consider that the process has not failed at all, but was temporarily disconnected from the remaining of the replicas. As expected, this situation isn't distinguishable from a real failure and doesn't require any type of special action or coordination from the other replicas.

If the process is unable to recover or resume communication immediately, the system will continue operation uninterrupted as long as the minimum number of corrects processes is maintained. The failed process still can recover without coordination, but it may have missed a large number of messages and must catch up with the other processes. This amounts to a type of coordination with the notable exception that the system never blocks while the recovering process brings its state up to date. The process may resort to some type of state transfer with a more up to date replica, but during the execution of this process only the recovering process remains blocked. The remaining replicas operate unaffected.

This happens because Paxos does not rely on a group membership service or timeouts to decide if a process has failed. Actually, Paxos does not care about the state of any specific process to function correctly and just requires a stable coordinator and a possibly anonymous majority of working acceptors to progress. An optional group membership module may run on top of Paxos [26], but the criteria it employs to decide a process is to be excluded from the group can and should be distinct from the criteria the underlying total order algorithm uses to decide if a message is or isn't to be expected from a suspect process. This observation is fundamental to understand the high resilience to failures observed in the execution of Paxos (Section 7).

To support this level of resilience in the crash-recovery failure model, Paxos needs to keep information in stable memory. Usually implemented with magnetic disks, stable memory is very slow compared to volatile memory and adds significantly to the latency of operations. This time penalty could be considered against the use of Paxos for active replication. However, in Treplica we are interested in applications where the replicated state must be stored persistently no matter the type of failure. In particular, application state must survive a complete crash of the entire replica set. Consequently, applications would need to access stable memory anyway to keep their own state persistent. This stable memory requirement is completely independent from the requirements of the total order algorithm used. Depending on the guarantees provided by this underlying algorithm, the application would have the additional work of reconciling its persistent local state with the state of the message delivery in case of failure. Fortunately, Paxos allows us to take a different approach. Instead of reconciling total order algorithm and application state, we tie them together and manage them as a unity. This is possible because Paxos must remember the state of any consensus instances, and this state includes the messages proposed and decided. From the contents of these messages it is trivial to obtain the application state. Thus, Paxos is a very desirable algorithm to implement a persistent queue because its properties combine performance that is resilient to failures and a unified view of replication and persistence.

5.5 Treplica Software Architecture

The software architecture of the Paxos-based asynchronous persistent queue follows very closely the agent decomposition used in the description of the algorithm, achieving a very modular design. The queue implementation is composed by internal classes performing the functionality of the four agents, assisted by generic support modules. Figure 5 shows the main modules of the Paxos persistent queue.

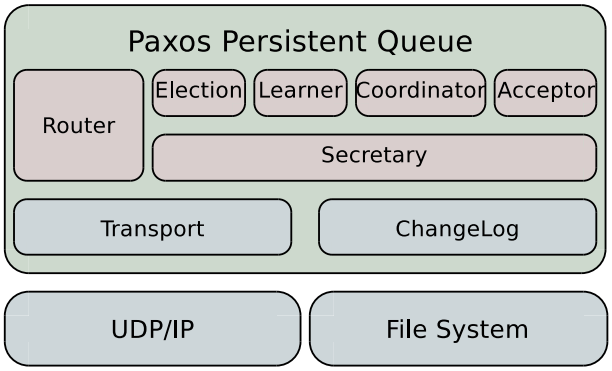


Figure 5: Paxos Persistent Queue

The four Paxos agents are implemented in the following classes:

`Learner` Combines the proposer and learner agents in a single class responsible by monitoring the flow of Paxos instances, converting them into objects and delivering them to the queue.

`Acceptor` Acts as an acceptor.

`Coordinator` Acts as a coordinator.

`Election` Handles the leader election algorithm used to select a single coordinator.

These classes were designed to execute independently, making it possible to create a Paxos process with only a subset of agents. Specifically, a process containing only a `Learner` module could propose and learn values, effectively running a complete queue, without taking part in the consensus procedure. A process configured this way could be used to increase the scalability of the system. However, this functionality isn't supported by the queue yet.

The main classes behave in a similar way to the agents they implement as they are strictly reactive modules. They operate by processing messages addressed to them by the `Router` class. As a consequence of this processing, these classes may send new messages to the network, store information in stable memory or deliver ordered objects to the application. These tasks are handled by the `Secretary` class, that offers a uniform interface to all I/O required by the agent classes. The abstraction provided by the `Secretary` class gives the agent class a way to send messages encoded in a `Message` class and to

access stable memory wrapped in a `Ledger` class. The `Secretary` on its turn relies on the services of the `Transport` and `ChangeLog` classes to access the underlying network and stable storage. These two classes provide abstractions that shield the other modules of the persistent queue from implementation details. We have implemented a `Transport` based on multicast over UDP/IP networks and a `ChangeLog` based on a simple file system.

In the following sections we describe these modules in more detail. They are presented in a bottom-up manner, starting with the support modules and then describing the Paxos agents. For each module we show its main function, how it interacts with the other modules and the implications of its structure on the Paxos implementation. To simplify this description, we name modules that depend on the a module being described as its *clients*. The original Paxos specification leaves many details unspecified, specially regarding liveness constraints and optimizations. In the following text we strive to make clear how we have implemented important open aspects of Paxos.

5.6 Support Modules

The support modules give an abstraction of the underlying system with clearly defined interface and service guarantees. These guarantees are very simple and can be directly mapped to many types of networks and stable storage semantics. The major motivation was to simplify the API used by the Paxos agents, hiding details about process addressing, multicast and unicast message passing, stable storage allocation and deallocation, main memory management, I/O management and error detection.

Transport The transport abstraction is defined by the `Transport` interface and represents a generic multicast transport. It allows its clients to send and receive unicast and multicast messages, closely matching the network properties expected in an asynchronous system. The messages are exchanged in an unreliable manner, and may be delivered out of order, duplicated or may be lost. We have made the choice of defining the transport abstraction with so few guarantees motivated by two reasons: it matches the network requirements of many consensus algorithms for asynchronous systems, including Paxos, and it closely reflects the guarantees effectively provided by our chosen network transport, UDP/IP.

Matching the algorithm requirements is important because we avoid duplication of functionality. For example, as Paxos does not require reliable message delivery, it includes a mechanism for message buffering and retransmission. Using a reliable mechanism would duplicate this mechanism. Moreover, the reliable delivery provided by a transport such as TCP/IP only works for the crash-fail failure model. In the crash-recovery failure model, the consensus algorithm still needs to check if messages were delivered even when using TCP. We understand the benefits of using reliable transports, specially regarding point-to-point bulk transfers of data [1], but we believe that algorithms like Paxos must evolve and be tuned to support multipoint delivery of ordered data with the same efficiency, but respecting the chosen failure models.

Besides message delivery properties, the transport abstraction defines a unified view of unicast and multicast messages, with a supporting addressing scheme. In brief, the

same transport implementation is able to send, and more importantly, receive messages sent both to a process and to the multicast group comprised of all processes in the system. Unicast addressing is done using a opaque transport id created and managed by the transport implementation. Each process obtains its id by a call to the `getId()` method of the transport. A process may exchange unicast messages with any other process in the system as long as it knows its id, using the `sendMessage(Serializable, TransportId)` method. Ids are simply data, and may be exchanged inside regular messages. Thus, a process may announce its existence by simply multicasting its id. A message can be multicast to all processes in the system with the `sendMessage(Serializable)` method. The `receiveMessage(int)` method is used to receive a message. It is a blocking primitive that blocks until a message is received or a timeout expires.

The group of processes reached by multicasts is implicitly defined by the underlying multicast primitive used by the transport implementation. This means that the transport is required only to identify and send messages to a suitable set of processes eligible to be part of the system, this being considered the set of "all processes". As an example of how this can be easily done, consider the UDP transport implementation. Its definition of all processes in the system is given by all processes that are listening to a predefined multicast IP and port. This IP and port are configurable parameters of the UDP transport. IP multicast routing infrastructure defines whose processes are effectively in the system. For instance, all processes linked to the same local area network, without any intervening level 3 routers or firewalls, are in the same system. This allows to trivially consider all processes in a cluster to be in the same system and, with more elaborate routing configurations, to setup a system spanning many distinct sites.

Note that this is completely different from a group membership service that defines which processes are part of the system. If such service exists, it sits above the transport and somehow filters messages received from processes not considered to be in the group. Once again, this reflects the expected network behavior of protocols like Paxos. They do not require the precise identity of processes, but only that a minimum number of them be correct at any time. This also lays ground to build more sophisticated architectures like the one proposed in [31], where the group membership is built on top of consensus and not the other way around.

Change Log The change log abstraction shields the Paxos agents from the details regarding stable storage. Basically, the service provided is that of a persistent log of changes to an object, with support for checkpointing. In fact, the interface presented to the programmer is very similar to a simple append-only file, but with explicit support for recovery. Changes to an object can be persistently appended to the end of the log and the object can be later reconstructed by replaying these changes. Checkpointing is used to improve the performance of reconstruction by storing the changes interspersed with full copies of the object. As a file, the abstraction provides `open()` and `close()` methods to prepare a change log for use. Once open, individual changes are written with the `writeChange(Serializable)` method and checkpoints are written with the `writeCheckpoint(Serializable)` method. The similitude with a file is just an ap-

proximation to a common API for stable storage, but there is no need for an explicit backing file to support actual implementations. Our main implementation (`DiskChangeLog`) uses the local file system to implement the change log, keeping its data in several files to speed the recovery.

The change log abstraction further deviates from a simple file as it provides active support for recovery. Whenever a change log is open, all the information required to reconstruct the underlying object is transferred to the module opening the change log. This action is triggered by a call to the `open(ChangeLogClient)` method, that requires a reference to a recovery client. The recovery is very simple. The most recent checkpoint is read and passed to the recovery client. Once the client has loaded this checkpoint, all subsequent changes are passed in turn to the client that must be able to apply such changes. This way, the recovery client implements all recovery policy while the recovery mechanism is driven by the change log. Recovery consistency is guaranteed by the change log: all changes and checkpoints writes are atomic and a failure automatically closes the change log. If a client wants to keep using the change log, it must re-open it and, at its choice, perform recovery to the point of the last successful change or detect the change whose write failed and rewrite it.

The reason we have chosen to abstract stable storage in the form of a change log is simplicity and performance, but also the desire to experiment with alternative forms of persistent storage. The append-only operation of the change log allows the use of an underlying magnetic and solid state (flash) disk in its optimal sequential access mode. Appending operations sequentially at the end of a file also simplifies recovery, as it is never necessary to reconstruct a log in case of write failure. We just create a new file and, as necessary for recovery, read the old file until the failure point. Moreover, a simpler abstraction allows multiple implementations. We already explored this possibility with a change log implementation that actually stores data in the volatile memory of other processes in the system. In this ongoing research project we are investigating the feasibility and reliability of such scheme.

Ledger The ledger is an abstraction to the stable state of the Paxos implementation. It is a common data structure, shared by all Paxos agents. The agents see main-memory oriented methods defined in the `Ledger` interface, while a concrete implementation has support for efficiently storing this information in stable storage. As described in Section 5.3, it is possible to derive the state of the replicated process from the state of the consensus instances stored in stable memory. Thus, the ledger abstraction concentrates all data that is to be held in stable memory, making it easily accessible from main-memory. The `LoggingLedger` is the object effectively made stable by the change log. To simplify the use of the change log, this implementation has support for detecting and isolating changes made to its internal state. It can export these changes and later recover its state by reapplying a set of previously exported changes. The ledger stores the complete state of each individual consensus instance, holding all data required by all types of agents. This way, it is possible for a process to create new agents, such as a coordinator, without reloading the data structure.

Secretary The secretary abstraction presents a unified view of I/O for the Paxos agents. It handles stable storage implementation using the change log and the ledger, it handles message passing using the transport, and it handles the object queue used to deliver objects to the application. The main reason this abstraction was created wasn't to isolate the agents from the underlying building blocks but to remove costly I/O operations from the thread executing the agents. Disk I/O in particular has a great potential to reduce the throughput of any Paxos implementation because of two reasons. First, all changes written to stable storage must be flushed from any intermediary caches before algorithm execution continues, to guarantee consistency. Second, some steps of the algorithm can generate many stable memory writes. Considering that each flush operation takes around 1ms to complete and that a Paxos round demands at least two stable memory writes, we add at least a 2ms latency to all consensus instances. Moreover, all rounds must compete for access to the disk and a round must add to its latency the time required to flush the data of rounds executed before it. The secretary abstraction creates a way to solve this problem by removing from the agents the task of effectively performing I/O.

Once I/O is handled only by the secretary, it is now possible to solve the problems caused by many stable memory writes in a single operation and the lack of parallelism among multiple rounds. This is done by queuing and grouping distinct logical writes in a single physical write. This approach is advantageous because the size of the data in a complete disk write, usually performed by a `sync()` system call, has little impact on the operation latency. Making use of this observation, the secretary implementation keeps continuously writing and flushing data to the disk, in a separate thread, as long as there are requests waiting to be written. Requests that arrive in the midst of a write are queued and wait for the next flush.

This approach streamlines access to the disk, but it doesn't change the fact that Paxos correctness is rooted in the stability of the information written to stable storage. What this means is that a thread executing an agent must block until the I/O operations it has requested to the secretary are complete. To obtain parallelism in the agent execution with this blocking behavior, we could manage many threads executing concurrently the agents. This is possible, but it requires complex concurrency control to the common data structures and is prone to lock contention. Instead, we have decided to run agents in a single thread with total control of the data structures, simplifying concurrency control. However, this approach has the problem that it serializes the execution of rounds if we were to maintain the simple blocking behavior. The solution comes from the observation that rounds are independent in Paxos. Thus, a single thread is capable of managing many rounds at a time if it can avoid to be blocked for I/O, but instead changes rounds. The secretary allows exactly this behavior by implementing asynchronous I/O operations.

These asynchronous operations work by creating a virtual barrier between the actions an agent has performed and the actions that the other agents observe it to have performed. An agent has three types of interaction with the outside world: it sends messages to the network, it delivers objects to the application and it writes to stable storage. In the Paxos algorithm messages can be lost, so a simple message send isn't binding. However, the stable write done before the message send is binding, to allow the message to be recreated later. For example, in Phase 1b an acceptor, before sending the coordinator its last vote,

must record its participation in the round chosen by the coordinator (Section 5.3). One way of ensuring that messages are only sent after the write to stable memory is committed is to hold the messages sent by the agent until the write is stable.

We say that a message, sent to the network or delivered to the application, is *dependent* on the last write made to stable storage but not yet actually written by the secretary. From the point of view of the other agents, the write never happened until its dependent messages are delivered. That is, a message can only be delivered after all stable memory writes that precede it causally [25] are flushed to disk. Whenever a write completes, the secretary unblocks the dependent messages and send them to the other agents. Meanwhile, the agent that created the dependent messages is free to keep processing new messages, making further changes to the stable storage and sending additional messages as long as it has work to do. If the writes are held back indefinitely, the system will eventually stop. However, there will be a steady flow of concurrent rounds to be processed to keep the non I/O bound thread of the agents busy most of the time. Put in another way, in our solution the agents do not block for I/O, but external effects of their actions do.

Another function of the secretary is to manage the creation of checkpoints for recovery. As the secretary concentrates all I/O, it is able to freeze all operations of a queue and, as a consequence, the application. This way, it can obtain a snapshot of all relevant data structures and of the application. As explained in Section 3, the application is accessed through its state manager, the other data structures are under control of the secretary. The secretary also generates and handles a Paxos id, uniquely identifying this process.

Router The router is a simple but vital module of the Paxos persistent queue. It binds all agents together and provides them with their main thread. Its function is to run the main loop of the Paxos implementation, receiving messages from the underlying transport and, according to their type, routing them to the appropriate agent for processing. This way, agent execution is sequential and shared data structures such as the ledger do not need concurrency control. Also, this single thread monitors a central timer and generates timer events to the agents that need it. As explained previously, the message processing code of the agents is free of long running or blocking operations. This way, agents are programmed as simple event handlers in a asynchronous event-based processing architecture. Additionally, the router is responsible for instantiating the agents and the appropriate supporting modules, handling initial configuration of the Paxos persistent queue.

5.7 Paxos Agents Modules

Paxos agents effectively implement the Paxos algorithm. They implement behaviors described in the algorithm specification and are responsible for its correct operation. They use the support modules described in the previous sections, adhering to the processing model of asynchronous event-based message handlers that create stable memory dependent external events. This section describes their functionality and also documents our solutions for the gaps found in the Paxos specification.

5.7.1 Election

This agent is responsible for the leader election protocol required by Paxos to make progress. It exposes to its clients the interface of a Ω failure detector. Briefly, this failure detector requires that any election agent trusts one process in the system as correct and that there is a time after witch all election agents trust the same process [14]. If we make this trusted process run a coordinator agent, we eventually have only a single coordinator agent running as required to ensure Paxos liveness. The election agent doesn't require the clients to poll its service to notice leadership changes. Specifically, it detects when the process running the agent is the elected leader and initiates a coordinator agent in response to this event. Conversely, it detects when the process stops being a leader and stops the coordinator agent.

Stable Leader Election Any unreliable election procedure is appropriate for Paxos correctness as long as it implements Ω. This procedure must combine a mechanism to elect a single process with some type of heartbeat-based failure detector [15] as processes and communication links may fail. Moreover, as many Paxos instances are to be executed in sequence, it makes sense to avoid arbitrary leader changes and keep the same coordinator instance elected at all times. Thus, stability is important requirement in the implementation of the election service. We have implement an election procedure that is a variant of the algorithm proposed by Larrea et. al. [29]. To function properly, our protocol requires all links incident to a non-faulty processes to be eventually timely in both directions. This effectively makes link failures to be equivalent to process failures, the most common failure situation in clusters. Besides being simple to implement, this protocol has the advantage of only requiring the regular sending of a constant size broadcast message to maintain a single leader once it is elected.

Our leader election algorithm modifies the algorithm of Larrea et. al. in two important ways: it supports an unknown set of processes and it implements leader stability. Consistently with the transport abstraction of the network provided to the Paxos agents, the election agent assumes a completely interconnected network of anonymous participants. Anonymous means a single process does not know beforehand how many other processes there are in the system or their identity, but processes do have a unique identifier and they can discover each other by exchanging broadcast messages. Leader stability guarantees that once the system behaves synchronously long enough to elect a leader process, this process won't be demoted during synchronous operation and will always be (re)elected leader after periods of asynchronous operation as long as it does not fail.

The algorithm works as follows. All processes listen for election messages and keep a local timer. Whenever a process receives an election message that indicates a process with higher priority is requesting leadership, the process records the sending process as leader and behaves as a follower. If the timer expires and the process has not received any message from a higher priority process, it assumes it is the leader and starts sending election messages advertising its leadership. The leader process does not expect confirmation from the followers, the absence of competition indicates an implicit success of its leadership bid. In our particular implementation, the local timer is configured to ex-

pire in a time sufficient for two election messages to be received. If the network behaves synchronously and no messages are lost, after a round of election messages are sent by all contenting leaders only the process with the higher priority will still consider itself a leader and only this process will keep on sending election messages.

A careful choice of process priority is required for the election protocol to function. At a minimum, it is necessary for all priorities to be unique to allow only one process to possess the higher priority from all contenting processes. To this end, the process priority contains a unique Paxos id provided by the secretary. However, uniqueness isn't enough to ensure stability as a process with higher Paxos id can demote a elected leader. To achieve stability we have defined the priority to be a pair (uptime, id), where uptime is an integer counter incremented every time a leader process tries to renew its leadership. The uptime counter is initialized to 0 every time a process starts or recovers. The process with the highest counter, that is, the process that stayed most time as leader without crashing, is the process with the highest priority. In case of identical uptime values, Paxos id is used to break the tie.

5.7.2 Learner

The learner agent in Treplica implements the functionality of the learner and proposer agents in the Paxos algorithm. It is responsible for processing requests from the persistent queue client, creating suitable proposals to order these requests, monitoring the proposals until they are ordered and delivering ordered proposals as objects to the persistent queue client. To understand why we have combined the functionality of these two agents in the same module, it suffices to observe the activities performed by this agent. It is possible to classify the first two tasks as pertaining to the proposer agent only and the last task to fall under the activities of the learner. Nonetheless, the third task is fundamental to the correct operation of our implementation of the Paxos persistent queue and it requires knowledge held by both proposer and learner. This happens because we support both Paxos and Fast Paxos in the same implementation and Fast Paxos removes from the coordinator agent the sole responsibility of proposing consensus values.

Stateless Proposals In Fast Paxos, at a minimum, it falls on the proposer the selection of an unused consensus instance, the proposal of a client request in this instance to the acceptors and the detection that a proposal has completed or has failed. The last step is necessary because the proposer must be able to forward a failed proposal to the coordinator, which restarts the consensus instance with a classic round number. Moreover, even in Paxos, the proposer faces the problem of making sure every request made by a client translates into exactly one ordered proposal, without repetitions. It could rely on the coordinator to ensure this, relaying to it not proposals but client requests. However, the coordinator can fail before sending a proposal in a suitable consensus instance or the message containing the request may never arrive. In this case, the proposer must resend the request to the coordinator until it is ordered. This only shifts part of the problem to the coordinator, that now must check every request it has received to see if it wasn't

already ordered, without actually relieving the proposer from the task of monitoring the sequence of ordered proposals looking for its pending requests.

We solve this problem by completely shifting from the coordinator to the learners not only the task of creating a proposal from a client request and monitoring it, but also the selection of an appropriate consensus instance. A learner receives requests to be ordered from its client and queues them until it is ready to create a new proposal. When this happens, it selects from its local view of the consensus instances a non-started instance. In Fast Paxos, this means that the learner can submit the proposal to be voted by the acceptors immediately. When running Paxos, it forwards the proposal to the coordinator to be decided in the consensus instance it has selected. Either way, the acceptors broadcast their votes directly to the learners and it is their responsibility now to check if the created proposal is decided in the position specified by the selected consensus instance in a timely fashion.

This proposal monitoring is easier now, as a learner only has to observe the specific consensus instance it has selected. If another proposal is decided in this instance, the learner selects a new non-started consensus instance and tries again. Meanwhile, the coordinator has not to monitor the proposals it manages, it just tries to decide proposals in the indicated consensus instances. Obviously, it won't violate Paxos consistency to satisfy the learner request, and informs the learner when the selected consensus instance isn't actually free for use. This behavior of the coordinator is exactly the same in Fast Paxos, but the coordinator is only called to action when a collision or timeout occurs. This way, both learner and coordinator can manage the flow of proposal requests in a stateless way. Consensus instance consistency still requires stable storage, but a request is managed only in main-memory.

Gap Detection The learner monitors only the proposals it has created, but it receives votes and computes the consensus decision for all instances. These values are ordered according to the predetermined consensus instances numbers and delivered sequentially to the client as soon as they are decided. This delivery is done by adding values to a queue, while the client removes elements from this same queue. Actually, from the point of view of the client, this queue is the asynchronous persistent queue itself. Consensus instances, however, are not decided sequentially. Due to lost messages or collisions, a gap of undecided instances may appear before a decided instance. These gap instances prevent values decided in subsequent instances of being delivered to the client. Thus, to force the decision of gaps a learner tries to pass a *null* proposal, sending it to the coordinator as usual. If these gaps are already decided but the learner is unaware of it, the coordinator will tell the learner so. If the decision isn't known to the coordinator, it will start a new round. This round, according to the consistency guarantees of Paxos, will discover if a sufficient number of acceptors have decided any proposal in this instance. In the unlikely event that nothing was decided yet, the *null* proposal will be decided. However, this proposal will be ignored by all learners and not delivered to the client.

A learner must ensure that some value is eventually decided in all consensus instances it knows to be active, be it a gap or not. This is controlled by a timeout mechanism that

resends proposals to the coordinator if the particular consensus instance where it was originally sent isn't decided. As the learner is responsible for selection of the intended consensus instance for a proposal, the operation performed by coordinator upon the receipt of this message is idempotent. As an optimization the coordinator checks to see if this consensus instance is already in progress or decided and notifies the learner. This behavior is identical in Paxos and Fast Paxos, that is, the learner can always ask the coordinator to pass a proposal in a specific consensus instance if it detects a problem. This covers a special case in Fast Paxos: a collision.

Collisions In Treplica, learners detect collisions as they tally the votes cast by the acceptors. These votes can be to distinct proposals if two or more learners have concurrently initiated the same consensus instance. If the number of acceptors that have not cast a vote yet isn't enough to win a majority for the most voted proposal, a learner detects a collision. It then restarts immediately this consensus instance by sending a request to the coordinator. Lamport describes several other strategies to resolve collisions [27], but we have decided to use this simple restart procedure because of the low overhead associated with running single classic Paxos rounds and the fact that the number of collisions observed in practice is very low [35].

Congestion and Flow Control Since the learner is responsible for starting consensus rounds directly or through the coordinator, it is also responsible for any type of congestion or flow control. Intuitively, congestion and flow control have the objective of avoiding the saturation of the transmission capacity of a link or the processing capacity of a CPU, respectively. When saturated these resources tend to provide less service than when just below their maximum nominal capacity. The general mechanism to attaining this type of control for network applications is to limit the rate messages are generated by individual processes. This rate limiting can be done using explicit readings of the load on the network or CPU, or be based on indirect measures like the latency of messages or message loss. However, due to its distributed and fault-tolerant nature, Paxos present some subtle difficulties to both flow and congestion control.

By design, a subset of all processes are allowed to fail in Paxos, and they can take an arbitrarily long time to recover. This has a direct impact to flow control as it is impossible for a process to distinguish if any of the other processes are slow or failed. This means that a process should never wait a slow process to avoid being blocked indefinitely by a crashed one. Congestion control seems to be immune to this effect as a process can observe and control the rate it creates its new proposals. Nevertheless, it is still necessary to ensure that the rate chosen by any single process will provide a "fair" allocation of the available link bandwidth. Again we return to the same problem: a fast process can never be sure a slow process hasn't actually failed and it can't wield to it, because a failed process may never request the released resource.

We consider adaptive congestion and flow control mechanisms that solve these problems to be very interesting research areas. We have not had the chance to research on suitable policies for congestion and flow control, but we have implemented in Treplica a

rich mechanism capable of supporting many interesting policies. The mechanism defines a maximum number of pending proposals per learner, a maximum size for a proposal and a maximum number of queries for gaps in the instance sequence. The maximum number of pending proposals is the basic tool for congestion control. It limits the rate new proposals are created and sent for vote by the learners, by forcing them to wait the complete approval of a proposal to create a new one once its local maximum was reached. This mechanism takes full advantage of the fact that learners are responsible for proposal creation. Each learner is able to control its maximum number of proposals independently and the coordinator doesn't need to monitor this quantity.

When a learner reaches its maximum number of pending proposals, subsequent client requests are queued waiting for a proposal to complete. When this happens, the learner creates a proposal containing more than one client request, ordered by arrival time. Under high client load, the size of the request queue can grow very fast, so the learner limits the number of requests that are packed in a proposal. This is controlled by the proposal maximum size. A leaner creates a new proposal by concatenating client requests until the maximum size is reached or the request queue empties.

The final congestion and flow control mechanism has to do with filling gaps in the consensus instance sequence. A single learner cannot deliver the ordered requests to its client if there are undecided consensus instances before decided instances. This is particularly relevant when a crashed process tries to recover and discovers a large subset of instances whose outcome is unknown. A naive learner might try and decide *null* on all instances at once, overloading the network and the coordinator. To avoid this we define a maximum number of queries sent to the coordinator for filling gaps in the consensus instance sequence. This is similar to the maximum number of pending proposals, but can usually be larger because the gap filling process is faster than a full Paxos round for decided instances.

Currently the parameters that control maximum number of pending proposals, maximum proposal size and maximum number of gap filling queries are fixed. The values used were experimentally obtained; the current values are given in Table 1.

Pending Proposals	2
Proposal Size	10kB
Gap Queries	100

Table 1: Parameters for congestion and flow control

5.7.3 Coordinator

The coordinator is a Paxos agent responsible for ensuring the prompt conclusion of consensus instances. It receives messages from the learners asking to pass a proposal in a selected consensus instance, starts or resumes an appropriate round for this instance and monitors its conclusion. As explained in the previous section, the coordinator does not monitor individual *proposals*, but instead it ensures that a *consensus instance* reaches

a single decided value. The actions of the coordinator when starting a new round for a consensus instance are central to the safety guarantees of Paxos and the timeliness of these actions is central do the liveness properties of Paxos. Thus, this agent is a very important part of any Paxos implementation.

Seamless Validation To perform these vital coordination tasks effectively, any coordinator agent created goes through a validation process. During this validation, the coordinator starts all infinite consensus instances and completes the Phase 1 of the algorithm for them. This way, in Paxos all instances that have never progressed beyond Phase 1 in any *previous round* can be started directly in Phase 2 as soon as the coordinator receives a proposal from a learner. This activation process is even more important in Fast Paxos, in which rounds can only be decided in a fast way if they have their Phase 1 previously completed by the coordinator.

The traditional specification of the validation process requires the coordinator to be brought up to date with the state of the consensus instances held by of a quorum of processes. As a consequence, the coordinator blocks as it performs the required state transfer and the execution of consensus instances is interrupted. We have implemented an optimized version of the validation process that avoids tying up the coordinator more than the minimum necessary. Our method works by splitting validation in two concurrent activities: activation and recovery proper. It turns out, only activation is strictly required for a coordinator to be able to start Phase 1 of all uninitiated consensus instances. Our improved procedure provides seamless coordinator validations, with reduced coordinator blocking and less disruptive performance oscillations. A more complete description of this optimization can be found in [36].

Fail Fast Rounds Another optimization found in Treplica implementation of Paxos reduces the number of rounds required for a coordinator to successfully validate. Whenever a validating coordinator fails to receive a quorum of responses to its validation request before a timer expires, it assumes the validation has failed and creates a new round with a larger round number. However, if this failure was motivated by a previous coordinator that has created a much larger round number before being demoted, it may take a long time before the new coordinator can produce a round large enough if it naively increases the round number. A simple solution to this problem is for the acceptors to send the coordinator a special message indicating they are unable to reply to the round number just received because they have replied to a larger round number. This way, the validating coordinator knows how large its round number must be and, if no two contending coordinators are active at the same time, uses this number to successfully validate. This approach has the added benefit that the acceptor can use this message to inform the coordinator of any situation where a round number cannot be acted upon because of another larger round number. This way, a coordinator can actually discover if its status as single coordinator is being contended before a timeout.

Instance Management Once activated, a coordinator can begin processing messages from the proposers asking it to pass a proposal. Even in Fast Paxos, where the proposers act independently, proposal requests are still sent to the coordinator if they fail to be decided in a fast round. When a proposal arrives, the coordinator first checks to see if the selected instance is decided or is in progress. If it is in progress the coordinator does nothing. If the instance is decided the coordinator informs the proposer of the instance outcome using a special message. If the instance is neither decided or in progress, the coordinator starts Phase 2 of the Paxos algorithm or Phase 1 of the Fast Paxos algorithm. That is, for Paxos the coordinator continues the round started during the validation. For Fast Paxos, the coordinator assumes the proposal has failed and immediately starts a new round.

After deciding how to process a new proposal, the coordinator receives and processes the remaining Paxos messages exactly as described in Section 5.3. As described above, the coordinator won't restart a consensus instance in progress if it receives any proposal request for the same instance. This happens because this instance is now under the coordinator control, and it will monitor and keep retrying it until it decides any value. The coordinator maintains a timer for each instance, and initiates a new round, with a greater round number, each time this timer expires. The proposers should not be allowed to influence this timing. They will maintain local timers for their proposals, but the coordinator refusal in restarting an instance in progress allows both timers to be integrated.

Coordinator Self-Stabilization During normal operation of the Paxos algorithm there should be only a single coordinator. However, the leader election procedure can make mistakes and another process can briefly believe itself to be a coordinator. This process j will then start a coordinator that will compete with the correct coordinator in process i, generating increasing round numbers. This is a momentary situation that is caused by election procedure mistakes, not by any real process or network failure. As such, it is important for the system to self-stabilize after these instability periods. This is specially relevant if we consider that such unstable periods are quite frequent under heavy load [36].

If the leader election module in process i effectively detects it has momentarily lost the leadership and deactivates the coordinator, this recovery is automatic. This is equivalent to an actual failure. Once the system stabilizes, i will re-validate its coordinator agent and a large enough round number will be promptly discovery. However, if the leader election module in i doesn't recognize any contending coordinator, the coordinator in j could have finished validation with a larger round number. After the election module stabilization, the coordinator in i continues processing unaware of the fact it has effectively lost the coordination position and that no acceptor will reply to its current round number. If this happens, Paxos consistency isn't violated, but all proposals with the round number from the original validation of the coordinator in i will fail. As a consequence, after a costly timeout a new round will have to be enacted in full. This has a severe impact on the throughput of the system. To avoid this problem, and ensure proper stabilization after failure detector mistakes, we adopt a simple solution. Whenever an acceptor receives a

Phase 2a message and ignores it because of a larger round number, it sends the coordinator a special message indicating this fact. When this message is received, the coordinator restarts, acquiring per the rules of validation a new large enough round number.

In Fast Paxos the problem is more subtle. A possible result of a contended coordination is a set of proposers that assume they can send proposals directly to the acceptors using a round number from the coordinator in i. Because of the competition from the coordinator in j, proposals coming directly from the proposers with this round number will timeout. This happens because there won't be enough acceptors that agree to vote on these proposals. Only after a timeout the coordinator intervention is requested, and it will eventually decide the instance. However, fast rounds are impossible and the coordinator is oblivious to this fact. To fix this, the coordinator resends at regular intervals the round number it has associated with its last validation. This effectively renews the authorization it has made for this round number to be used directly by the proposers. If an acceptor notices one of these notifications with a smaller round number it informs the coordinator. As in Paxos, when this message is received, the coordinator restarts, acquiring a new large enough round number.

5.7.4 Acceptor

The acceptor is an agent responsible for voting in consensus instances according to the Paxos algorithm. This agent reflects very closely the behavior of Paxos described in Section 5.3. It concerns itself mostly with the safety of the algorithm. The acceptor waits for Phase 1a messages starting a new consensus round and answers them, if appropriate, with an account of the vote it has cast last. This enables a round to proceed and the acceptor casts a vote in this round when it receives a suitable Phase 2a message. In Treplica this behavior has only two small changes that increase the performance of the system: the acceptor reduces a vote to small constant size message and it actively warns the coordinator of decided consensus instances.

Constant Size Votes Usually, a Phase 2b message carries the round number and the proposal being voted. This way, any learner listening to broadcast votes can compute, independently, the outcome of any round. However, this is wasteful of network resources as the same proposal is sent once in the Phase 2a message and n additional times, once for each of the n acceptors in a quorum. A solution to this problem implemented in Treplica is to configure the acceptor to send only an unique identifier of the proposal in its Phase 2b messages. This identifier is generated by the learner agent that created and submitted the proposal. This way, proposals are uniquely identified in the votes and it is possible to discover the outcome of the round as long as it is possible to map this identifier to the actual proposal. To this end, learners are also changed to monitor Phase 2a messages and keep a local cache of proposals presented for voting, indexed by proposal id. The end result is that the Phase 2b messages, that account for the larger number of messages send in a Paxos round, are reduced to two integers: the round number and the proposal identifier.

Succeed Fast Rounds Another improvement is the reduction of the number of messages required for a coordinator to finish an election for an already decided consensus instance. A coordinator can always enact a complete Paxos round with all its phases, regardless of the state of a consensus instance. If this consensus instance is already decided, the coordinator will always compute a majority of votes selecting the already decided proposal. This happens frequently in Fast Paxos whenever a process hosting the coordinator looses many messages and its local learner has to catch up on the decided consensus instances. To short circuit this process, an acceptor agent warns the coordinator of a decided consensus instance by broadcasting the decided value. Thus, the coordinator and any learner that may have missed the voting can now discover the decision for this consensus instance without requiring a complete election.

6 Applications

Virtually all distributed applications require replication. The amount and importance of replicated state to the application varies, depending on consistency requirements and overall cost. Keeping all information replicated consistently is potentially very costly and may limit the system efficiency or availability [20]. Thus, it is extremely important to be able to replicate data consistently and integrate this data with the rest of the components of the system. Due to its modularity, Treplica can be used as a tool to assist in the construction of practically all types of distributed applications.

As described in Section 2, Treplica can help in creating complete applications, programmed as if they were centralized. For example, good candidates for replication using Treplica are web applications that use a database for data storage and sharing. In general, these applications do not use all properties of a relational database and only use it as a convenient way of persisting data. As a consequence, these applications end up needlessly tied to a centralized point of failure. Treplica can offer a more direct programming abstraction to persistence while providing replication and fault tolerance. We expect Treplica to be a viable option to build a enterprise wide application or a small scale Internet shop. We analyze the performance of Treplica in one of these applications in Section 7.

Another way Treplica can be used is in the construction of a central coordination point for more loosely coupled components. A simple way to coordinate the access of a shared resource by several independent agents is through the use of locks and leases [26, 28]. The replicated state machine is the perfect abstraction to build a cluster of reliable lock servers that can be accessed through a RPC interface. For example, distributed file systems maintain large amounts of data stored on stable storage, replicated for fault tolerance and reliable access. Due to the amount of data and to the performance requirements, this involves only two or three replicas with a primary-backup scheme. Nonetheless, the state of these replicas can be controlled by a replicated state machine, such as the identity and status of the replicas are always consistently updated and made available to both file system replicas and clients. The Google File System [19] and Chubby [13] are systems built with this software architecture. Apart from a small prototype, we have not extensively

tested Treplica in a similar environment. Nonetheless, we believe Treplica would fit nicely in a similar architecture because of its simple design and performance. More data about Treplica performance can be found in [35, 36]. While these works do not put Treplica performance in the perspective of a standard benchmark, they give an idea of the expected Treplica performance.

7 Performance

We have made an extensive study of Treplica performance, in the context of a small scale Internet shop. This application, called RobustStore, is an implementation of the TPC-W benchmark [16] using Treplica. We use TPC-W as a benchmark for a complete application running on Treplica, in such way that the real systems that TPC-W is expected to assess are faithfully represented by our experimental setup. The complete performance analysis, including details of the experimental setting, can be found in [12]. We reproduce in this section a brief description of TPC-W metrics and key Treplica performance charts.

The TPC-W benchmark specifies all the functionality of an on-line bookstore, defining the access pages, their layout, and image thumbnails, the database structure. The bookstore application is based on a standard three-tier software architecture. TPC-W defines three workloads that are differentiated from each other by varying the ratio of browsing (read access) to ordering (write access) in the web interactions. Performance is measured in *web interactions per second* (WIPS), with *web interactions response time* (WIRT) as a complementary metric. An example of a read-only interaction is the search for books by a given author, while an example of an update interaction is the placement of an order. The shopping profile specifies that 80% of the accesses are read-only and that 20% generate updates. The browsing profile specifies that 95% of the accesses are read-only and that only 5% generate updates. Finally, the ordering profile fixes a distribution where 50% of the accesses are read-only and 50% generate updates.

In the remaining of this section we show two of the experiments we designed using this benchmark. The first experiment (Speedup) characterizes the scalability of the application, increasing the number of replicas in the system and the load handled by it. The second experiment (One Failure) shows the resiliency to failures of the application, injecting a failure that disables a replica. The experiments were carried out in a cluster with 18 nodes interconnected through the same 1Gbps Ethernet switch. Each node has a single Xeon 2.4GHz processor, 1GB of RAM, and a 40GB disk (7200 rpm). The software platform used is organized with Fedora Linux 9, OpenJDK Java 1.6.0 virtual machine, Apache Tomcat 5.5.27 and HAProxy 1.3.15.6.

Speedup The speedup experiment evaluates the maximum possible increase in performance obtained when RobustStore's scale goes from 4 to 12 replicas. Figure 6 shows the speedup values obtained for the three workloads and an initial state size of 500MB. It is possible to observe that RobustStore scales well, specially with a large proportion of reads. Write performance is still good and it indicates the application can handle gracefully varying load profiles.

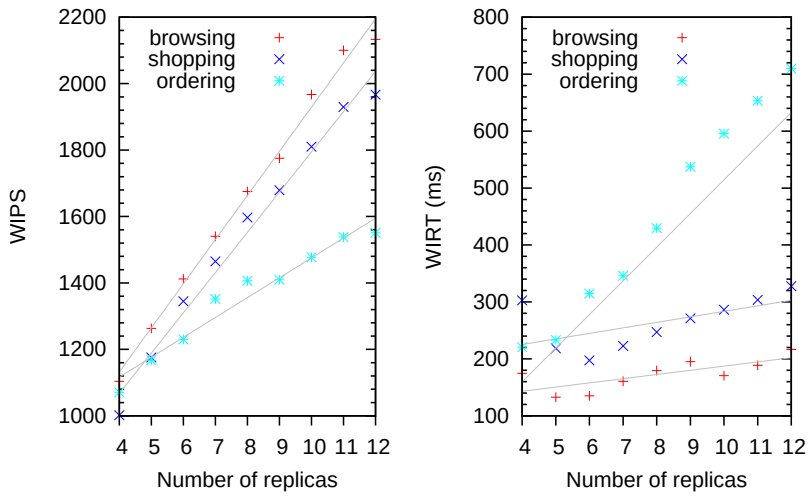


Figure 6: Speedup

One Failure In the one failure experiment one process is crashed 270 seconds in a 600 seconds run. The crash is immediately followed by the automatic triggering of recovery by a local replica watchdog. Figure 7 shows the behavior of a five-replicas RobustStore for the three workload profiles. The application throughput can be characterized as very resilient and stable in the presence of the crashes, failover, and recoveries used in the experiments. In this and other dependability experiments we have performed RobustStore loses less than 13% of its average performance during recovery in the worst case [12].

8 Related Work

The idea of main-memory storage, with a persistent operations log used as a fault tolerance mechanism, is described by Birrell et al. [9]. The current API of Treplica was influenced by the Prevayler [37] persistence layer, specifically in its use of features of modern dynamic languages like Java and C# to simplify implementation and provide a more straightforward API. Compared to these centralized systems, Treplica goes a step further as it uses this operation-based persistence approach as a basis for replication.

Data replication with strong consistency has been frequently used as basis for control mechanisms in large scale distributed systems [11, 24, 23]. These systems require some mechanism to control the operations of a large number of processors and services as autonomously as possible. These mechanisms present a lock based programming abstraction coupled with a configuration data repository. Similarly to Treplica, many of these systems use the Paxos algorithm to implement replication.

The Chubby locking service is used to power a myriad of distributed applications at Google [11]. Although a locking system is a different type of abstraction, Chubby shares many architectural features with Treplica, including a “persistent log”, very similar to a persistent queue. The designers of Chubby state the intention of using this queue ab-

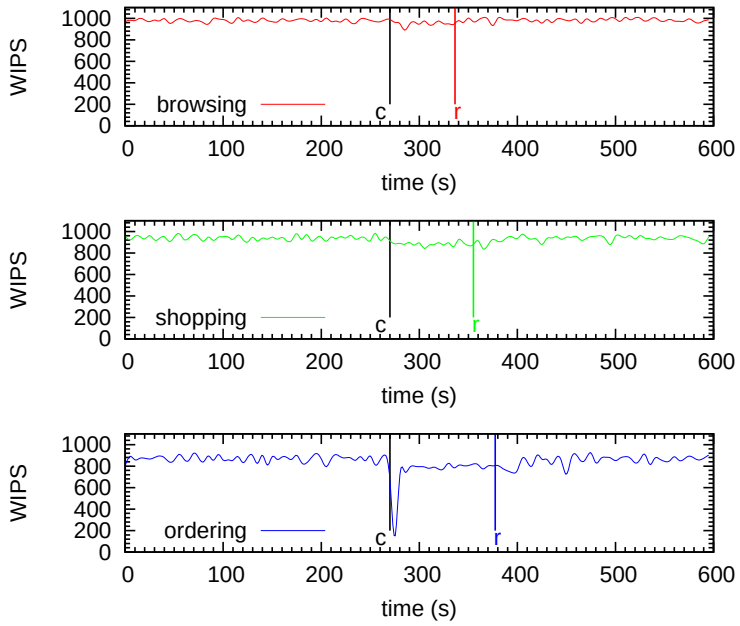


Figure 7: One Failure: 5 Replicas

straction for the construction of other distributed applications [13]. Chubby is a special purpose application used to provide lock services and doesn't export its internal replicated state service. In comparison Treplica exports only the replicated state service, a base where locking primitives can be build upon. Chubby uses the Classic Paxos algorithm to implement replication, while Treplica uses both the Classic and Fast Paxos variants.

Other distributed locking and distributed control systems are FaTLease [23] (part of the XtreemFS object-oriented file system [22]), Zookeeper[3] and Autopilot [24]. All these systems are used as control centers for distributed applications operating in a cluster. As such, they are replicated to ensure uninterrupted operation of these applications in the presence of failures. These systems provide a more specialized service than Treplica. Nonetheless, it is interesting to observe that the designers of these systems decided to employ a consensus-based mechanism to replicate vital data. Moreover, this data is kept in main memory, using stable memory as an accessory for fault tolerance. These are design decisions similar to the ones we have made in Treplica. We expect Treplica to be an excellent tool for the constructions of such systems.

Boxwood is a framework for the construction of distributed storage applications [30]. Boxwood creators advocate the use of generic data structures as a foundation where to build more complex distributed systems. One of the proposed abstractions is a generic consensus service based on Paxos. This module is used by several other Boxwood components, including its distributed locks manager. Boxwood is focused in one domain of application (file systems and databases) and provides a more low level interface to its

[3] `http://hadoop.apache.org/zookeeper/`

services, while Treplica offers a higher level programming API.

It is common in the literature the acknowledgment that Paxos, despite its simplicity, is full of subtleties that increase the complexity of an actual implementation [10, 13, 28]. A work that deals exclusively with a detailed description of a Paxos implementation can be found in [3]. This paper describes all aspects of a complete state machine replication using Paxos. Also, it presents a fairly complete study of the performance of the described implementation. The authors describe mechanisms for flow and congestion control, leader election and other implementation aspects not usually detailed. They observe that these mechanisms are generally considered to be implementation details, but that in fact they are central to the maintenance of many algorithm properties, specifically its liveness. With Treplica, we have proposed and implemented a modular abstraction for active replication including requirements for persistence. This way, Paxos is central for the development of Treplica but it encompasses a broader theme: object-oriented replication.

Group communication toolkits provide a service of message diffusion to a group of processes according to diverse ordering guarantees. Many of these systems exist, from the original Isis [6], to JGroups [4], Spread [2] and Appia [32], to list a few. The central idea behind these toolkits is the virtual synchrony [7, 8] application programming model. Treplica shares similar goals with these systems but does not implement the virtual synchrony model, nor does it support many message ordering guarantees, only a totally ordered message sequence. Treplica is designed to offer a simpler programming abstraction with built in support for persistence, thus the application programmer is free from the difficult task of guaranteeing state consistency. In a way, Treplica can be seen as a higher-level abstraction than group communication, and these toolkits could be used to create an implementation of the Treplica API.

The asynchronous persistent queues abstraction is very similar to the publish / subscribe pattern of communication for process groups implemented in message oriented middleware (MOM) [5]. The message exchange in MOM is asynchronous and even a failed or inoperative processes can expect to be delivered all messages sent, in the same order seen by all the other processes. Besides message diffusion, MOM allows the construction of elaborate message flow graphs and may perform message format conversion as messages are transported through this graph. Examples of such systems are the IBM WebSphere MQ[4] and Apache ActiveMQ[5] products. These systems are heavy-weight compared to Treplica and are usually implemented on top of a centralized relational database, inheriting the failure behavior of these systems. Also, Treplica is designed for more tightly coupled processes and transports application objects. As a consequence, it does not provide explicit message flow and message format conversions.

9 Conclusion

Correct, efficient and resilient replication of applications is a hard problem faced by many programmers of distributed applications. Unfortunately, there is limited support for com-

[4] `http://www-306.ibm.com/software/integration/wmq/`

[5] `http://activemq.apache.org/`

pletely handling replication in face of process failures and recoveries in the tools currently used for the construction of such applications. To address this problem we have created an object-oriented specification for replication and implemented it in the Treplica library. This paper described the object-oriented specification proposed and the software architecture of its implementation.

The advantages of using the proposed object-oriented specification for replication are twofold. First, it makes transparent to the programmer much of the complexity of dealing with a highly-available application. Second, it allows the middleware effectively implementing the replication to optimize many factors now outside of the programmer reach. In Treplica we use extensively this property to employ consensus-based active replication to effectively get application durability for free, after paying the cost of replication. The observed performance of the final system is a good indication of the success of this approach.

References

[1] T. Abdellatif, E. Cecchet, and R. Lachaize. Evaluation of a group communication middleware for clustered J2EE application servers. In *DOA 2004: Proceedings of the 2004 International Symposium on Distributed Objects and Applications*, pages 1571–1589, Agia Napa, Cyprus, Oct. 2004.

[2] Y. Amir, C. Danilov, and J. R. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.

[3] Y. Amir and J. Kirsch. Paxos for system builders. In *LADIS '08: Proceedings of Large-Scale Distributed Systems and Middleware*, New York, Sept. 2008.

[4] B. Ban. Design and implementation of a reliable group communication toolkit for java. Technical report, Cornell University, 1998.

[5] G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18, London, UK, 1999. Springer-Verlag.

[6] K. P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, 1993.

[7] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[8] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[9] A. D. Birrell, M. B. Jones, and E. P. Wobber. A simple and efficient implementation of a small database. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 149–154, New York, NY, USA, 1987. ACM Press.

[10] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003.

[11] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[12] L. E. Buzato, G. M. D. Vieira, and W. Zwaenepoel. Dynamic content web applications: Crash, failover, and recovery analysis. In *DSN 2009: 39th International Conference on Dependable Systems and Networks*, pages 229–238, Estoril, Lisbon, Portugal, June 2009.

[13] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[14] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[15] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, 2002.

[16] T. P. Council. *TPC-W Specification*, Feb. 2002.

[17] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Inc., 1995.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[20] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[21] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Ada-Europe '96: Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*, pages 38–57, London, UK, 1996. Springer-Verlag.

[22] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The xtreemfs architecture—a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, 2008.

[23] F. Hupfeld, B. Kolbeck, J. Stender, M. Högqvist, T. Cortes, J. Marti, and J. Malo. FaTLease: scalable fault-tolerant lease negotiation with paxos. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 1–10, New York, NY, USA, 2008. ACM.

[24] M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, 2007.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[26] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[27] L. Lamport. Fast Paxos. *Distrib. Comput.*, 19(2):79–103, Oct. 2006.

[28] B. W. Lampson. How to build a highly available system using consensus. In *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 1–17, London, UK, 1996. Springer-Verlag.

[29] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, page 52, Washington, DC, USA, 2000. IEEE Computer Society.

[30] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI '04: 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[31] S. Mena, A. Schiper, and P. Wojciechowski. A step towards a new generation of group communication systems. In *Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, pages 414–432, Rio de Janeiro, Brazil, 2003. Springer.

[32] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 707–710, Washington, DC, USA, Apr. 2001. IEEE Computer Society.

[33] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[34] G. M. D. Vieira and L. E. Buzato. On the coordinator's rule for Fast Paxos. *Information Processing Letters*, 107:183–187, Aug. 2008.

[35] G. M. D. Vieira and L. E. Buzato. The performance of Paxos and Fast Paxos. In *SBRC '09: Proc. of the 27th Brazilian Symposium on Computer Networks and Distributed Systems*, pages 291–304, Recife, Brasil, May 2009.

[36] G. M. D. Vieira, I. C. Garcia, and L. E. Buzato. Seamless paxos coordinators. Technical Report IC-10-13, Institute of Computing, University of Campinas, Apr. 2010.

[37] K. Wuestefeld. Do you still use a database? In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 101–101, New York, NY, USA, 2003. ACM Press.