Treplica: Ubiquitous Replication

Gustavo M. D. Vieira[1]*, **Luiz E. Buzato**[1]

[1]Instituto de Computação, Unicamp
Caixa Postal 6176
13083-970 Campinas, São Paulo, Brasil

{gdvieira, buzato}@ic.unicamp.br

Abstract. *This paper describes Treplica, a tool designed for ubiquitous replication. Most of the software tools created so far to aid in the construction of distributed applications addressed solely how to maintain volatile data consistent in the presence of failures, but without offering any relief for the problem of managing persistence. Treplica simplifies the development of high-available applications by making transparent the complexities of dealing with replication and persistence. These complexities are not negligible, and we believe we have found a compelling way of factoring out these concerns in a simple to understand programming interface.*

1. Introduction

Consider the problem of implementing a highly available distributed application. It is a fact that one of the main development concerns is the management of persistent data, maintaining consistency in the presence of failures and concurrency. Thus, failures, consistency and performance are a concern even before replication has been considered as a mean to satisfy the high availability requirement. Most of the software tools created so far to aid in the construction of distributed applications addressed solely how to maintain volatile data consistent in the presence of failures, but without offering any relief for the problem of managing persistence [Birman and Joseph 1987b]. Usually, when developers opt for the use of such tools for replication, they have to cope with two problems: (i) the handling of replication itself and (ii) the explicit recovery of the state lost during a component failure.

This paper describes Treplica, a tool designed for ubiquitous replication. Treplica treats the solution of the two problems presented before as a unity, by offering to the programmer a simple way to deal not only with consistency but also with persistence. In the context of Treplica, ubiquitous means transparent, resilient and efficient. 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 statefull applications as if they were stateless applications. 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 data. Treplica guarantees resiliency through the implementation of Paxos [Lamport 1998] and Fast Paxos [Lamport 2006]. The unified handling of consistency and persistence means

*Financially supported by CNPq, under grant 142638/2005-6.

that applications developed atop Treplica are able to mask operational faults and support online modular maintenance. These in their turn can contribute to the lowering of the operational costs of running the application for very long periods of time. This is, in our opinion, a key factor for the adoption of any replication tool by developers of modern distributed applications. In terms of efficiency, our initial 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 the complexities of dealing with replication and persistence. These complexities are not negligible [Chandra et al. 2007], and we believe we have found a compelling way of factoring out these concerns in a simple to understand programming interface. The main contributions of this work are: we propose the idea of handling and presenting to the application programmer a unified programming abstraction for replication and persistence. We propose the use of consensus as a foundation for construction of one such unified replication toolkit. We describe our current work on Treplica, an efficient and generic implementation of these ideas, with preliminary performance data.

The rest of the paper is structured as follows. Section 2 describes Treplica, with emphasis on the design and implementation of its programming abstractions: asynchronous persistent queues and replicated state machines. Section 3 argues that Treplica can be used ubiquitously to support the development of a variety of distributed applications, ranging from distributed locking services to generic web services. Section 4 analyzes our experimental results and assesses the performance of Treplica. Section 5 is dedicated to reviewing and acknowledging research efforts related to Treplica. Finally, Section 6 concludes the paper by briefly summarizing our contributions.

2. Treplica

Treplica has been designed to provide a simple tool for the construction of replicated applications that reside in clusters[1]. These clusters of Unix boxes interconnected by a fast LAN are the hardware platform of choice for most of high-performance, high-throughput distributed applications. We assume nodes of the cluster behave accordingly to the crash-recovery failure model. Nodes can fail only by crashing, are serviced, and later reenter normal operation; any data not stored in persistent memory is lost during the crash. Communication is accomplished by message passing and messages can be lost, duplicated or arbitrarily delayed, but they cannot be corrupted.

Treplica considers an instance of the application component selected for replication as its unit of replication. Units of replication are assigned to nodes of the cluster in accordance with their availability requirements. The units of replication can have communication patterns that involve direct interaction with clients of the application or, alternatively, can have their communication restricted to components of the application that do not interact directly with clients. For example, the replicated components may be dedicated to wrapping around a legacy system, by converting and transfering data between the legacy system and the distributed application.

[1]Replication across clusters, over the Internet, isn't yet a concern of the project, it might be considered in the future.

The main design decision that supports Treplica is the combination of persistence and replication requirements of the application under a single and simple programming abstraction. A well-accepted way to handle replication is using Lamport's *active replication* [Lamport 1978], where an application is modelled as a deterministic state machine, the actions of the application are modelled as transitions of this machine and the sequence of transitions is broadcast, in the same order, to all replicas. The determinism of the state machine guarantees that all replicas will be identical. Persistence in Treplica is built using the same principle: the application is a deterministic state machine, the operations are transitions of this machine and the sequence of transitions is logged to stable storage [Birrell et al. 1987]; 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.

A key part of active replication is the ordered broadcast of messages, essential to the consistency of the replicas. By design, Treplica already is responsible for the persistence of each replica and manages the stable state of the application. Thus, to avoid explicit coordination of the application state and the total order algorithm state in the presence of failures [Défago et al. 2004] we have decided to use uniform total order as the communication primitive. The combination of uniform total order and log-based fault tolerance is the key to avoid unnecessary duplication of persistent memory logging. This approach can be more efficient because it is possible to batch and serialize access to stable storage, using the disk in its more efficient mode. Treplica was designed to accommodate any uniform total order algorithm, but we have decided to concentrate on consensus based total order algorithms because they allow Treplica to have a simpler software architecture and increases its potential for good responsiveness in the presence of partial failures [Mena et al. 2003].

Finally, the resulting software architecture should be easy to program and useful for a large number of applications, adhering to different programming styles. To accommodate this, we have decided to offer a state machine abstraction as a programming tool, using the reflection facilities of modern languages to encode and execute state and state transitions. Using state machines as a programming tool 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 transitions as runnable, serializable objects. Treplica programming tools can easily be implemented in any other dynamic language and, with some more effort, in more traditional languages such as C/C++. So, a developer who wants to implement a replicated distributed application does not reason in terms of replicated and persistent objects, instead, it reasons about the execution of the application operations, transitions of a *replicated state machine*, that are triggered by events that are made available through a *asynchronous persistent queue*. In the remaining of this section we describe these two abstractions and how they are implemented in Treplica.

2.1. Asynchronous Persistent Queues

The search for a simple programming abstraction for Treplica led us to the notion of asynchronous persistent queues, an abstraction for a distributed persistent log of messages. Informally, asynchronous persistent queues are a way for a group of processes to exchange messages, with three important properties: (i) messages are delivered in the same order to all processes, (ii) messages are delivered to all processes, even if a process crashes and later recovers, and (iii) messages are persistent. These properties are very similar to the properties of total order broadcast in a group communication toolkit using view synchrony [Birman and Joseph 1987a], but delivery of messages is constrained by the state of the application and not by a view. Asynchronous persistent queues are also very similar to the publish/subscribe groups implemented in message oriented middleware [Banavar et al. 1999], but here processes are tight-coupled.

Each persistent queue is uniquely identified by a *queue identifier* (queue id). Each process that interacts with a queue does so through a *queue endpoint*, created using a queue id and bound to a specific queue. A process may access many queues at the same time, creating a separate endpoint for each one of them. The primitives of an asynchronous persistent queue are very simple:

`create(queueId):` Creates a queue endpoint identified by the provided queue id.
`getProcessId():` Returns the process id associated with this endpoint.
`put(message):` Sends a message to all other processes trough this queue. A message can be any serializable object.
`get():` Receives the next message of this queue.

Individual processes don't have to worry about making their queues persistent, all is taken care of by the endpoint. Each queue endpoint has associated with it the message delivery history. For instance, a new process joining a queue, using a new queue endpoint, will receive all messages ever sent to the queue. 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.

The persistence of the queues cannot be implemented efficiently unless careful design and implementation steps are taken. Suppose a process fails and returns after having executed for a considerable time. It is the responsibility of the queue to provide it with its recovery state in the form of a message log that, in this case, can be very large. There is no upper limit for the size of the recovery log and as a consequence stability of state has to be designed to allow obsolete queue and application state to be adequately collected and discarded; this is the function of the queue stable state managers. The state managers of the asynchronous persistent queue provide what we call *queue controlled persistence*, where periodically a snapshot of the queue and application state is taken and the log is garbage collected. The queue handles the coordination of local snapshots among all replicas and guarantees that each replica always sees a sequence of messages consistent with its state. This may require, if a replica fails and falls behind the others, that upon recovery the queue replaces its local state with an updated snapshot obtained from the other replicas. To support this mode of persistence the application should be instrumented with a save state (take checkpoint) procedure that is callable by the queue

state manager. An extra primitive is available to bind the queue with the entity responsible for storing the application state:

`bind(stateHolder)`: Binds a process, represented by its state holder, to a queue endpoint. The state holder is any application component capable of implementing the `takeCheckpoint()` primitive. Returns a checkpoint if recovery was necessary. The process must reset its state to the returned checkpoint.

2.2. Replicated State Machine

Treplica can support the construction of replicated applications in many ways. A straightforward approach would be to access the asynchronous persistent queues API directly and use the ordered sequence of messages to implement the replicas. The direct use of asynchronous persistent queues would require the application programmer to build some type of state machine to use active replication, to build a message monitoring subsystem to service the client requests synchronously, and to handle the bind and take checkpoint operations of the queue. To make this task easier, Treplica provides a higher level abstraction that implements this replicated state machine.

The replicated state machine component is an implementation of a state machine that has its state changed only by executing *operations*. Operations are implemented as Java objects that contain methods built to act on the state held by the state machine. Locally, each replica stores all its state in the replicated state machine and only changes it using operations passed to the `execute()` method. The programming interface of the replicated state machine is listed below:

`create(initialState, queue)`: Creates a new state machine bound to a queue. An initial state should be provided, because the process 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(operation)`: Executes an operation on the distributed state, performing all necessary steps to coordinate this change with the other replicas. This is a blocking method.

A replicated state machine has only three simple methods in its programming interface that implement a well-defined, well-known 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 operations 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. Operations applied to the state machine by the local client can only be actually performed by the state machine after they have been converted into a message and submitted to the asynchronous persistent queue. The local client of the state machine perceives the execution of the operation as a call to a blocking primitive. A successful return of the call guarantees that the operation submitted has been performed in the same order by all replicas.

2.3. Software Architecture

The software architecture of an application built on top of Treplica is shown in Figure 1. The main architectural components are the application itself, replicated state machine and asynchronous persistent queue, the total order service and the state manager.

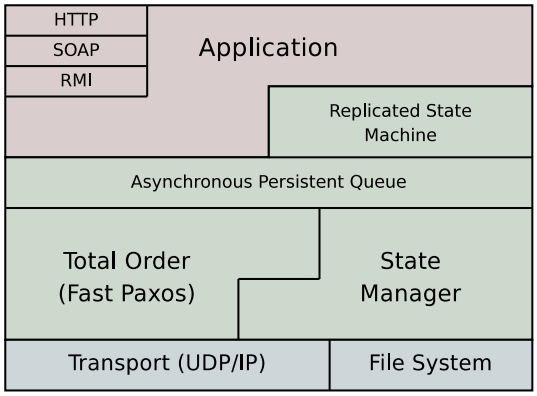

Figure 1. Software architecture of Treplica.

The client application can interact with its clients in any way possible. For example, it can serve remote clients using RMI, it can implement a web service, it can serve local clients through sockets, etc. The only architectural constraints imposed by Treplica on the application are the ones described in Section 2; the application replicated data fits in main-memory and that the application that handles the replicated data is deterministic. Section 3 lists some examples of applications that can benefit from Treplica. If necessary, the application can use multiple threads to service its clients, but Treplica guarantees that only one thread at a time executes operations on the state machine.

2.4. Implementing Asynchronous Persistent Queues

Usually, the total order service provided by group communication toolkits isn't uniform. Thus, we decided to employ a total order algorithm based on consensus, as they provide a complete solution and usually implement uniform total order [Défago et al. 2004]. In practice, Paxos [Lamport 1998] is one of the most sucessful and used consensus algorithms [Chandra et al. 2007, Elnikety et al. 2006, MacCormick et al. 2004], and fits perfectly the Treplica design because of its adherence to the crash-recovery failure model and the way it structures its persistent memory. Specifically, the adoption of Paxos has allowed us to delegate to the consensus component the management of stable storage for Treplica.

A full description of Paxos is beyond the scope of this paper, but we offer a simple description of the algorithms main components. Total order using Paxos consists of a sequence of individual consensus instances, each one corresponding to a deterministically ordered list of messages. The instances are totally ordered by definition and this order drives the ordering of messages, as all processes must reach consensus and select a single message list for each slot. Each consensus instance requires actions from processes performing the following roles: proposer, acceptor, coordinator and learner [Lamport 2006]. Each process can perform more than one role, but at least a proposer, a coordinator and a majority of acceptors must be present for the algorithm to progress. In Treplica all

process perform all roles, except the coordinator, that must be unique to ensure progress of the algorithm. Our implementation shares a similar architecture with the proposal for group communication over consensus of Mena et al. [Mena et al. 2003], but we do not yet implement group membership functions or any other delivery semantics besides total order.

We have implemented the Fast Paxos [Lamport 2006] generalization of Paxos, but with support for the classic Paxos algorithm if desired. Fast Paxos optimizes the number of communications delays associated with a classic Paxos instance from three to two, but it requires a larger number of correct acceptors to ensure progress. To support both algorithms interchangeably, without reconfigurations, we shift the initial ordering of a proposal from the leader to the proposers. Each proposer selects a locally consistent position for its next message list and sends it directly to the acceptors, if this is a Fast Paxos instance, or to the leader, if this is a classic Paxos instance. Either way, it is the responsibility of the proposer to check if the message list was ordered as requested, or retry the request with a different Paxos instance if unsuccessful. Due to this change, the leader becomes unnecessary in Fast Paxos, except to coordinate failure recovery. This is very interesting as it removes the possible performance bottleneck represented by the leader.

Another interesting property of Fast Paxos that affects the implementation of Treplica is the fact that different proposers can try and order distinct messages at the same position at the same time in a fast instance. In this case, it is possible that none of the proposals will succeed, in what is called a collision [Lamport 2006]. Lamport describes several strategies to resolve this collision [Lamport 2006] and we have implemented the simplest: whenever a collision is detected a new instance of classic Paxos is initiated, with the participation of the leader to solve the conflict. We have decided not to use any of the more elaborated collision recovery techniques because of the low overhead associated with running single classic Paxos instance and the relative rarity of collisions in the target architecture [Pedone and Schiper 2003].

Treplica is built on top of UDP/IP and uses multicast IP addresses as queue identifiers. Processes use their IP address and port number as unique identifiers and use the multicast IP addresses as references to their peers. There isn't an explicit group membership procedure, not even a static hard-coded one. The Paxos implementation of Treplica requires only a correct definition of a majority to work. For a system with n acceptors, a majority is defined as $\lfloor n/2 \rfloor + 1$ acceptors for Paxos and $\lceil 3n/4 \rceil$ acceptors for Fast Paxos. At any time, there can be no more than n active acceptors, but the current implementation does not enforce this limit. We plan to extend Treplica to support group reconfigurations, bringing it closer to the architecture proposed in [Mena et al. 2003]. However, the current implementation allows for a considerable degree of flexibility as it only requires that the *maximum* number of processes in the system to be fixed, not their identity.

To ensure liveness, Paxos requires a leader election component, which includes the failure detector module. We have implemented a very simple algorithm where a process makes a bid for or announces its leadership by broadcasting its process identifier; the algorithm is similar in its design principles to the algorithm proposed by Korach et al. [Korach et al. 1984]. In practice, this simple leader election algorithm is rather limited as it requires all links to be timely to function properly and it isn't stable. Stability is

a very desirable property of any leader election algorithm used by Paxos [Malkhi et al. 2005] and we will improve the leader election algorithm as the code matures.

The state manager takes care of the persistent storage of the Paxos component. By handling the persistence of all votes cast by acceptors and all elections started by the coordinator, all state of the total order algorithm and of the application are stored in a persistent log. This log is represented by the *ledger* abstraction of the Paxos algorithm [Lamport 1998]. Treplica state manager is a careful implementation of this ledger, optimized for efficient access to disk. For example, write access is sequential, minimizing disk head movement and increasing throughput. Also, non critical writes are batched and only flushed to disk when a synchronous write is necessary. Unfortunately, the state manager is currently the less mature component of the system and has not been implemented in full yet.

3. Treplica Applications

This section lists examples of systems where Treplica can be employed. In some of these domains it is already possible to find other implementations using mechanisms similar to the ones used by Treplica. We comment more on the similarities and differences between Treplica and these other systems in Section 5.

3.1. Lock Service

Some large distributed applications do not and cannot require that all data are replicated in a consistent way. However, the data must be accessed in a controlled way. A simple way to coordinate the access of a shared resource by several independent agents is through the use of locks and leases [Lamport 1998, Lampson 1996]. The replicated state machine is the perfect abstraction to build a cluster of reliable lock servers that can be accessed through a RPC interface.

The Chubby lock service [Burrows 2006] is an example of a lock server implemented using a very similar approach to Treplica. Internally, Chubby uses a replicated and persistent log of operations component implemented using Paxos [Chandra et al. 2007]. This persistent log is very similar to the asynchronous persistent queues and the creators of Chubby argue that this is a very helpful abstraction that could be used in other distributed applications [Chandra et al. 2007].

3.2. Distributed File System

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 Boxwood framework [MacCormick et al. 2004] constructs its RLDev (Replicated, Logical Device) abstraction using two replicas, a primary and a backup, where writes can only be performed in the primary and reads can be performed on both. The location of each replica, the identity of the primary and recovery information are kept in a component of Boxwood called the Paxos service. This service offers very similar semantics to the replicated state machine and could be implemented using Treplica.

3.3. Database Transaction Certifier

Tashkent [Elnikety et al. 2006] is a distributed database that uses generalized snapshot isolation to manage concurrency and consistency among database replicas. The system is organized as a transaction certifier that coordinates a number of database replicas that run off-the-shelf database servers interfaced to clients through local proxies. Read operations are executed locally, but write operations are first ordered by the certifier before being applied.

The certifier component is more than a simple ordering mechanism, it also handles the durability of the write operations, increasing the overall performance of the cluster as it relieves local replicas from costly local I/O. The certifier runs in a cluster of replicated machines independent from the cluster of database replicas and uses Paxos to guarantee consistency among its members. Thus, not only Treplica can be used to implement the certifier replication for Tashkent, Treplica internal checkpoint handling is completely compatible with the Tashkent handling of replica recovery [Elnikety et al. 2006].

3.4. Web Services

Supply chains are being deployed as a composition of web services [Alonso et al. 2004, pp. 123-134]. Integration of services mean, on the one hand, that companies gain the capability of reacting faster to their clients needs, potentially raising revenue. On the other hand, companies and clients become dependent on the continuous provision of the services. Thus, reliability is very important and Treplica provides a very simple infrastructure to implement replication. Using the abstractions of replicated state machines and asynchronous persistent queues it is very simple to implement actual web services. The way this abstraction works is completely compatible with the way web service requests are handled, including concurrent requests. As a proof of concept, we have sucessfully implemented two simple but representative web services to assess the ease of use and test the replication properties of Treplica. One of the applications emulates an Internet banking system, the other implements an auction service.

A software architecture for replication of web services must deliver satisfactory dependability and performance, while maintaining compatibility with all web services standards. Compatibility is an easy task, considering the modular software architecture commonly found in web services middleware, however by using active replication performance can become an issue as there is a relatively tight association among the replicas and the provided consistency can be more than the minimum required by some applications. The ease of use provided by Treplica and the fact that main-memory capacities keep growing and networks get faster may compensate this extra cost. We estimate Treplica performance can be enough to accommodate an enterprise wide application or a small scale Internet shop as shown in the next section, but there is still much work to do on validating these claims.

4. Preliminary Performance

In this section we present some preliminary data on the performance of Treplica. The experiments performed were not designed to be a comprehensive study of Treplica, but only to assess the feasibility of using it as the replication engine for the applications described in Section 3. The current Treplica prototype is not yet properly optimized and

some functionality is missing, but the data presented here shows that it delivers satisfactory performance, at least for the small subset of configurations tested.

To validate the API and have an actual platform for testing, we have developed a simple Internet auction application using Treplica. The application is very simple and allows a client to put items for sale, list all auctions, list the recent k auctions, consult the status of any auction and place a bid on any item, but only the creation of a new auction and the placement of bids change the application state and need to be processed by Treplica. The unit of replication is an auction agent, accessed through a façade that exports a simple interface for the users. Remote clients access this interface through SOAP, local clients can call the façade methods directly.

To separate load factors related to SOAP from the load generated by Treplica, we conducted our tests using only the local interface. Moreover, our generated load consists only of auction creations. This way, we can be sure that load factors unrelated to the core Treplica have been ruled out and we are able to analyse the data as if Treplica were the only possible bottleneck. The load was defined as a sequence of create auction operations, generated with a fixed rate. This load is generated in the same hosts running the replicas, but care was taken to ensure that the load generation wasn't competing with the application processing and that the specified load rate was being generated.

The tests were performed on a cluster of six machines, but we only evaluated a system with three replicas. Each of the hosts has four Intel Xeon 2.4GHz processors and 1GB RAM, all interconnected by a switched 100Mbps ethernet link. We tested two different configurations: *single* and *multi*. In the *single* configuration we have only a single working thread generating load and a new state machine operation is dispatched only after the previous one has been completed. This configuration measures the response time expected by a single synchronous client and also the maximum throughput in terms of Paxos instances per second. Due to the synchronicity of this configuration, all load is generated by one of the replicas and the final load rate is directly derived from the average response time. In the *multi* configuration, we have as many working threads as necessary to guarantee a constant load of state machine operations. This configuration shows the throughput expected by a group of unrelated clients, and measures the effect of bundling many messages in a single Paxos instance. In this configuration, load is generated independently by all replicas, limited by the selected operation rate and sustained during the whole experiment duration. We increased the selected operations rate until request queues started to grow and the average response time exceeded an arbitrary threshold of 100 milliseconds. In both configurations, the response times observed displayed an exponential distribution and we present the response time considering a cut point of 85% of the distribution, that is 85% of the requests were serviced in a time equal or inferior to the figure provided.

Configuration	Throughput (op/s)	Resp. Time (ms)
single	90.7	4
multi	1685.1	34

Table 1. Treplica operation throughput and response times.

Table 1 shows the data collected for this simple experiment. Taking into account the consistency guarantees provided by Treplica, we consider its performance to be satisfactory and in line with the performance of similar systems [Abdellatif et al. 2004, Chandra et al. 2007]. We expect this figures to improve as we optimize Treplica. Also, these results reflect only write operations, while read operations were intentionally left out. Considering an application with 80% of read only operations, the data presented suggest a potential limit of about 8000 operations per second.

5. 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. [Birrell et al. 1987]. The current API of Treplica was influenced by the Prevayler [Wuestefeld 2003] 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.

A common abstraction for replication is to use traditional databases as replicated data stores and access the data through conventional query mechanisms, such as SQL. In this case replication isn't offered as a service, but as a mean to attain greater availability or performance for the replicated database systems. Recent research systems in this area are Postgres-R [Kemme and Alonso 2000], Sequoia/C-JDBC [Cecchet et al. 2004], Tashkent [Elnikety et al. 2006] and Tashkent+ [Elnikety et al. 2007]. Differently from Treplica, these systems offer a relatively heavy-weight solution to the problem of replication, not very applicable as a building block to general distributed applications. Tashkent and Tashkent+ are built using a light-weight replication module and shift the burden of persistence (durability) from the database to this replication module, with a positive performance impact. This approach is very similar to the change from a persistence-based programming interface to a replication-based one proposed by Treplica.

Closer in essence to Treplica is the Boxwood framework for the construction of distributed storage applications [MacCormick et al. 2004]. Boxwood creators advocate the use of generic data structures as a foundation where to build more complex distributed systems. However, Boxwood is focused in one domain of application (file systems and databases) and provides a more low level interface to its services, while Treplica offers a higher level programming API. Another similar system is the Chubby locking service that is used to power a myriad of distributed applications at Google [Burrows 2006]. 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, used as basic unit of replication. 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. Both Boxwood and Chubby use the Classic Paxos algorithm to implement replication, while Treplica uses both the Classic and Fast Paxos variants.

The asynchronous persistent queues abstraction is very similar to the publish/subscribe pattern of communication for process groups implemented in message oriented middleware (MOM) [Banavar et al. 1999]. The message exchange in MOM is asynchro-

nous 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[2] and Apache ActiveMQ[3] 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 do not provide explicit message flow and message format conversions.

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 [Birman 1993], to JGroups [Ban 1998], Spread [Amir et al. 2000] and Appia [Miranda et al. 2001], to list a few. The central idea behind these toolkits is the virtual synchrony [Birman and Joseph 1987a,b] application programming model. Treplica shares some similarity to these systems but does not implement the virtual synchrony model, nor does it support many message ordering guarantees, only a totally ordered message sequence. In virtual synchrony, processes are responsible for their own persistence, and their local stable state can be inconsistent with the messages being delivered for the group, even when using total ordering of messages. If necessary, the only way to synchronize a local process persistent state and the group state is by means of a state transfer from a process in the group, unless the application takes additional steps to create some other mechanism of recovery, such as a message log or a state delta. Another limitation of view synchrony based systems is the fact that if a process is suspected of having failed, even if it is wrongly so, it must restart its operation and discard all its state, resorting again to a costly state transfer. 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.

6. Conclusion

We have presented Treplica, a tool designed for ubiquitous replication. The design of Treplica was motivated by the limited support for handling the replication of nonvolatile data found in the tools currently used for the construction of distributed applications. Treplica handles this limitation by making transparent the complexities of dealing with replication and persistence. Treplica is built on the Paxos algorithm for consensus, and have much in common with the architecture for consensus-based group communication proposed in [Mena et al. 2003].

We have described our current work implementing Treplica, and why we believe the proposed programming abstraction of asynchronous persistent queues and replicated state machines is an easier way of handling replication in distributed applications. Ease of programming is a difficult point to argue, but our experience so far and some research reports [Burrows 2006, MacCormick et al. 2004] indicate this is a useful abstraction. We

[2] `http://www-306.ibm.com/software/integration/wmq/`
[3] `http://activemq.apache.org/`

have also presented some preliminary data that shows satisfactory performance, even before Treplica is functionally complete and fully optimized.

References

Abdellatif, T., Cecchet, E., and Lachaize, R. (2004). 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.

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). *Web Services: Concepts, Architectures and Applications*. Springer Verlag.

Amir, Y., Danilov, C., and Stanton, J. R. (2000). 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. IEEE Computer Society.

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

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

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

Birman, K. P. and Joseph, T. A. (1987a). 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. ACM Press.

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

Birrell, A. D., Jones, M. B., and Wobber, E. P. (1987). 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. ACM Press.

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

Cecchet, E., Marguerite, J., and Zwaenepoel, W. (2004). C-JDBC: a middleware framework for database clustering. In *USENIX 2004 Annual Technical Conference, FREENIX Track*, pages 9–18.

Chandra, T. D., Griesemer, R., and Redstone, J. (2007). 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. ACM Press.

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

Elnikety, S., Dropsho, S., and Pedone, F. (2006). Tashkent: uniting durability with trans-

action ordering for high-performance scalable database replication. In *EuroSys 2006: Proceedings of the 1st European Conference on Computer Systems*, pages 117–130, New York, NY, USA. ACM Press.

Elnikety, S., Dropsho, S., and Zwaenepoel, W. (2007). Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *EuroSys 2007: Proceedings of the 2nd European Conference on Computer Systems*.

Kemme, B. and Alonso, G. (2000). Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 134–143, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Korach, E., Moran, S., and Zaks, S. (1984). Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 199–207, New York, NY, USA. ACM Press.

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

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

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

Lampson, B. W. (1996). 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. Springer-Verlag.

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

Malkhi, D., Oprea, F., and Zhou, L. (2005). Ω meets Paxos: Leader election and stability without eventual timely links. In *DISC '05: Proceedings of the 19th International Conference on Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 199–213. Springer.

Mena, S., Schiper, A., and Wojciechowski, P. (2003). A step towards a new generation of group communication systems. In *Middleware 2003*, pages 414–432.

Miranda, H., Pinto, A., and Rodrigues, L. (2001). 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. IEEE Computer Society.

Pedone, F. and Schiper, A. (2003). Optimistic atomic broadcast: a pragmatic viewpoint. *Theor. Comput. Sci.*, 291(1):79–101.

Wuestefeld, K. (2003). 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. ACM Press.