Dynamic Content Web Applications: Crash, Failover, and Recovery Analysis

Luiz E. Buzato[*]
IC, Unicamp
Campinas, Brasil
buzato@ic.unicamp.br

Gustavo M. D. Vieira[†]
IC, Unicamp
Campinas, Brasil
gdvieira@ic.unicamp.br

Willy Zwaenepoel[‡]
SCCS, EPFL
Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

This work assesses how crashes and recoveries affect the performance of a replicated dynamic content web application. RobustStore is the result of retrofitting TPC-W's on-line bookstore with Treplica, a middleware for building dependable applications. Implementations of Paxos and Fast Paxos are at the core of Treplica's efficient and programmer-friendly support for replication and recovery. The TPC-W benchmark, augmented with faultloads and dependability measures, is used to evaluate the behaviour of RobustStore. Experiments apply faultloads that cause sequential and concurrent replica crashes. RobustStore's performance drops by less than 13% during the recovery from two simultaneous replica crashes. When subject to an identical faultload and a shopping workload, a five-replicas RobustStore maintains an accuracy of 99.999%. Our results display not only good performance, total autonomy and uninterrupted availability, they also show that it is simple to develop efficient recovery-oriented applications using Treplica.

1 Introduction

In this work, we evaluate how *crashes, failovers, and recoveries* affect the performance and availability of RobustStore, a highly available dynamic content web application. RobustStore has been implemented by retrofitting the stand-alone on-line bookstore specified by TPC-W [6] with Treplica, a middleware for building dependable applications [22]. Thus, the assessment of RobustStore is, in fact, the assessment of the fitness of Treplica as a high-availability support for dynamic content web applications. The TPC-W benchmark, augmented with faultloads and dependability measures, is used to evaluate the behaviour of RobustStore.

The process of recovering failed replicas is a main concern for highly available applications because it has a negative impact on their availability and reliability. Recovery time is primarily a function of application state size, so a larger application state should have a larger negative impact on the application, leading to performance loss. One could expect even more pronounced performance oscillations in scenarios with multiple overlapping crashes followed by multiple recoveries. We show that this is not the case for RobustStore. In fact, even in the worst case failure scenarios performance stays close to the levels delivered before the failures occurred.

Experiments apply faultloads that cause sequential and concurrent replica crashes. For example, RobustStore's performance drops by less than 13% during the recovery from two simultaneous replica crashes. When subject to an identical faultload and a shopping workload, a five-replicas RobustStore maintains an accuracy of 99.999%. The good performance, total autonomy and uninterrupted availability displayed by RobustStore in the experiments indicate that Treplica offers an efficient support for the construction of highly available distributed applications.

The remainder of the paper is structured as follows. Section 2 describes Treplica, and its use of Paxos [15] and Fast Paxos [16]. Treplica has been designed with performance, modularity and ease-of-use as primary objectives. The toolkit offers two very simple programming abstractions for programmers: state machine and asynchronous persistent queue. Section 3 summarizes the features of TPC-W, a web application benchmark widely accepted by industry and academia. In Section 4 we show how we have dealt with non-determinism, randomness, and database substitution during the development of RobustStore. Section 5 measures how the performance and availability of RobustStore is affected by crashes, failovers, and recoveries. Section 6 brings a summary of research that is related to our work. Section 7 summarizes our results and contributions.

[*] Corresponding author. On a sabbatical leave at EPFL. Supported by CNPq grant 201934/2007-8. Address: Caixa Postal 6176, 13083-970, Campinas, São Paulo, Brasil. Phone: +55 19-3521-5876

[†] Supported by CNPq grant 142638/2005-6. Address: Caixa Postal 6176, 13083-970, Campinas, São Paulo, Brasil. Phone: +55 19-3521-0345

[‡] Address: Building BC 407, Station 14, CH-1015, Lausanne, Switzerland. Phone: +41 21 693 64 89

978-1-4244-4421-2/09/$25.00 ©2009 IEEE

2 Treplica

This Section describes the features of Treplica that are relevant to this work; additional information can be found in [22]. Treplica supports the construction of highly available applications through either the *asynchronous persistent queue* or the *state machine* programming interfaces. The main programming abstraction is the persistent queue, a totally ordered collection of objects with the usual `enqueue(Object)` and `Object dequeue()` methods. `Enqueue(Object)` is, for efficiency reasons, implemented as an asynchronous primitive. `Object dequeue()` has a synchronous (blocking) semantics, as usually provided by queue implementations available in programming libraries. Persistence means that a replica bound to a queue can crash, recover and bind again to its queue, certain that the queue has preserved its state and that it has not missed any additional enqueues made by any other active replicas. Thus, by relying on the total order guaranteed by the queue and the fact that queues are persistent, individual processes can become active replicas while remaining stateless; the persistence of their state has been delegated to the queue.

The asynchronous persistent queue is implemented using the Paxos [15] and Fast Paxos [16] algorithms. These algorithms were chosen because they were designed to provide continuous operation of the application under the occurrence of partial failures, without requiring the programmer to use reconfiguration protocols. As a consequence of our choice, Treplica transparently transfers to the application the resiliency qualities of these algorithms. In particular, for N processes the configuration of Treplica used in this work uses Fast Paxos as long as $\lceil 3N/4 \rceil$ processes are working. If fewer processes than $\lceil 3N/4 \rceil$ but at least $\lfloor N/2 \rfloor + 1$ are available, Treplica falls back on Paxos. If fewer than $\lfloor N/2 \rfloor + 1$ processes are operational, the algorithm blocks until enough failed processes have recovered.

To ease the task of creating replicated applications out of the objects (operations) held by the asynchronous persistent queue, Treplica provides a higher level abstraction that supports the construction of replicated state machines. The state machine programming interface does not contain explicit support for the definition of states, events (transitions), conditions, and actions. Instead, it considers an application a black-box component whose public methods (interface) implement the set of events, conditions, and actions of a deterministic state machine. The application programmer uses the state machine programming interface of Treplica to treat all events, conditions, and actions as generic *actions*—Java objects—that can be managed by the asynchronous persistent queue and delivered to the application for execution.

A newly (re-)activated state machine sets its state to a consistent state. After that, the only way to change the state of the replica is through the execution *actions* triggered at the replica by the `execute()` method of Treplica's state machine. At any moment it is possible to obtain a snapshot of the most recent consistent state of a state machine by invoking its `getState()` method.

Actions invoked at one replica are guaranteed to be performed by it only after they have been converted into a message and enqueued into the asynchronous persistent queue for delivery to the other replicas. The original invoker of the action sees its execution as a call to a (synchronous) blocking method. A successful return of the call guarantees that the action has been performed by the invoker's replica and that the effects of the execution are now visible in the local state.

Recovery: Suppose a replica crashes and some time later recovers. Initially, a stateless instance of the application is created and its constructor, in turn, instantiates a state machine and invokes its `getState()` method. The method `getState()` interacts with the replica's asynchronous persistent queue. It is the responsibility of the asynchronous persistent queue to provide the recovering replica with the state to which it must be reset, in the form of a locally obtained checkpoint and an associated suffix of the queue's history. After resetting its state to that of the checkpoint, the recovered replica rejoins the remaining replicas. The queue's suffix necessary to complete the re-synchronization of the recovered replica is learned from the active replicas using Paxos. As soon as the queue re-synchronization ends, the recovered replica is ready to proceed as if it had not crashed. From the point of view of the programmer, all that needs to be done is to call `getState()`, the rest is transparently handled by Treplica.

3 The TPC-W Benchmark

The TPC-W benchmark specifies all the functionality of an on-line bookstore, defining the layout of access web pages, application semantics and the database structure. The bookstore application is based on a standard three-tier software architecture. Enterprises [6] and Universities [10, 11, 19] have extensively used implementations of TPC-W to assess the performance of machines, operating systems, and databases as supports for web services. The TPC-W implementation created at University of Wisconsin-Madison [3] has been used as the basis for our experiments. Performance is measured in *web interactions per second* (WIPS), with *web interactions response time* (WIRT) as a complementary metric. TPC-W defines three workload profiles that differ from each other by varying the ratio of book browsing interactions (read access) to book ordering interactions (write access). The shopping workload profile specifies that 80% of the accesses are read-only and that

978-1-4244-4421-2/09/$25.00 ©2009 IEEE

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 defines a distribution where 50% of the accesses are read-only and 50% updates. TPC-W names each of these workload profiles differently to make clear from the metric name which workload has been used in every experiment. The unit name WIPS is assigned to the shopping workload profile, WIPSb is used for the browsing profile and WIPSo for the ordering profile.

During an experiment, workloads are generated by remote browser emulators (RBE). To emulate the behaviour of human interactions, the RBE specification includes a *think time*, defined by TPC-W as 7 seconds. Thus, the number of web interactions per second (WIPS) generated by a set of emulated RBEs is given by #RBEs/think time. TPC-W also has a very strict definition of database model (conceptual and physical) and of the type and amount of data generated to populate the database.

4 RobustStore

In this Section, we summarize the changes we made to the implementation of the TPC-W online bookstore [3] to implement RobustStore. The method described here is general enough to guide the retrofitting of any application with Treplica. The steps are the following: (I) determination of the application state to be replicated; (II) review of the application methods that change the state and their transformation into deterministic actions. In the case of RobustStore, we had to deal with the non-determinism generated by calls to date and time system functions, and random number generation. The retrofitted application is structured as shown in Figure 1.

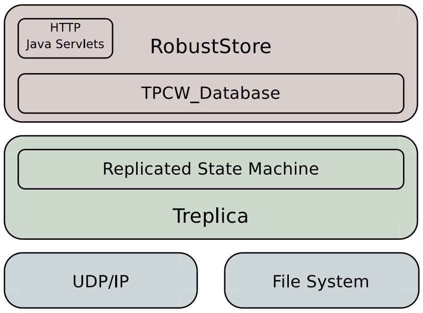

Figure 1. RobustStore components

Task (I) requires the design of an object model to represent the application objects that are going to be replicated. In the case of the online bookstore, we devised an object model composed by 9 classes that represent the entities and relations of TPC-W's online bookstore conceptual model. These classes and their instances represent the critical state of the bookstore and as such have to be programmed using the state machine abstraction provided by Treplica. The methods of these classes represent all the database functionality required by the bookstore. The original bookstore was structured as a set of web components (*servlets*) that accessed the database through a facade class (`TPCW_Database`) that served as a higher-level abstraction for the actual database. RobustStore has kept this structure intact, but the facade class now uses Treplica's state machine to execute operations equivalent to the original SQL transactions. The conversion of the facade class demanded 0.5 man-month. In total, about 2300 lines of code were changed. The final program had 3145 lines of code, 147 less than the original implementation. We did not have to change the code of the servlets, remote browser emulator or any other support program.

Task (II) has to do with non-determinism removal. The use of random numbers, dates and time is not a problem for a centralized system, but it is a problem for a replicated system. For example, whenever a new book order is created the order creation time is set to the current time. If each replica read its local clock inside the create order method to obtain the timestamp of the order, then each of the replicas would very likely stamp its order with a different timestamp. To avoid this, the code in the facade responsible for the creation of actions in the state machine reads its local clock *before* the action is created, and passes the resulting timestamp as an argument to the action's constructor. This simple procedure guarantees that every replica receives an order with exactly the same timestamp. Calls to random number generators are handled in the same way. For example, to generate the value of the discount applied to orders of a new customer, the random number generator is called before the action that creates a customer is instantiated and the value is passed as a parameter to the action.

It is important to note that the retrofit of TPC-W's bookstore with Treplica—execution of tasks (I) and (II)—did not require the programmer to think about replication, persistence, or the replica recovery process.

5 Evaluation

In this Section, we seek answers to four questions. First, how long can RobustStore be expected to run without interruption? Second, how much service can RobustStore be expected to deliver during failure-free and failure-prone operation periods? Third, what accuracy can be expected of RobustStore in the presence of crashes, failovers, and recoveries? Fourth, what level of human intervention is necessary to maintain RobustStore operational? We devised four sets of experiments to gather results associated with these questions. The first set contains speedup and scaleup experiments that show how RobustStore behaves in deployments of different scales. The other sets assess the dependability of RobustStore using the three TPC-W workloads and three different faultloads.

5.1 Method

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.

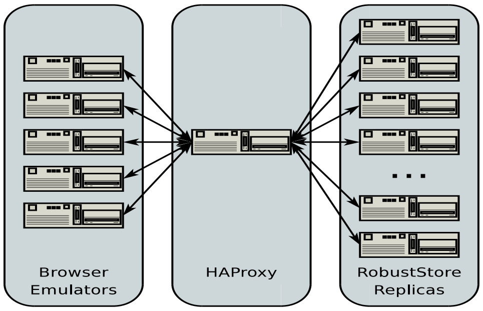

Figure 2. Experimental setup

The cluster has been divided into three disjoint sets of nodes as shown in Figure 2. The first set is composed by 5 client nodes that run the RBEs. Each client node holds the same number of RBEs. Instantiation and finalization of RBEs is done by a user initiated script, that computes and starts the exact number of RBEs necessary to generate the desired workload. Performance metrics are written by the RBEs into log files stored in the local disk. The second set contains from 4 to 12 server replicas that run the bookstore application. Each node of this set runs a copy of Tomcat that serves both static and dynamic web content. The application itself uses Treplica, as described in Section 4 and is configured to write only to the local disk. The final set contains only one node and runs the reverse proxy HAProxy, that has a load balancing module. The HAProxy is responsible for the failover mechanism. First, it actively queries the state of all of the server replicas using an HTTP probe. If it senses a replica is down (after 4 unsuccessful tries), it removes it from its servers list until it is probed active again. Second, requests are balanced among the server replicas using a hash mechanism based on unique client identifiers that are included in all interactions. If a server fails during the execution of a client request, HAProxy will close the connection and the client will observe an error.

RobustStore does not rely on a database, but the changes we have made to the application do not affect the data stored or the transactional semantics of the original application interactions. As a consequence, our experiments maintain the value of all experimental parameters as recommended by TPC-W, with one minor exception. To reduce the number of RBEs effectively required to provide a given load we changed the default 7s think time of the TPC-W specification to 1s. With a 7s think time the workloads generated by the RBEs of the 5 client nodes were not sufficient to saturate RobustStore. It is important to note that shorter think times do not change either the read to write ratios nor the probabilistic characteristic of the workloads. Even with the reduced think time, we still had to set aside 5 nodes only to generate load. This left a maximum of 12 nodes to hold replicas, but this number is sufficient to emulate most commercial deployments of replicated application servers. Thus, the real systems that TPC-W is expected to assess are faithfully represented by our experimental setup.

The replicas were populated using the standard TPC-W population procedure, with 10,000 items and 30, 50 and 70 emulated browsers, even tough we instantiated a larger number of RBEs. The parameter *number of browsers* was chosen to generate initial application state sizes of 300MB, 500MB, and 700MB, respectively. For the most write intensive profile (ordering) the average state size at the end of the measurement interval was approximately 550MB, 750MB, and 950MB, respectively. This respects the experimental requirement that all state must fit into main-memory. This is important to guarantee as much as possible that the performance variations observed are solely related with Treplica's activity on the network and on the disk. For all experiments the ramp-up, measurement interval and ramp-down periods follow TPC-W's specification; they were set to 30 seconds, 9 minutes and 30 seconds, respectively.

The TPC-W benchmark consists of a system specification, a workload and a metric. A dependability benchmark consists of a system specification, a faultload, a workload and a metric. Thus, to turn TPC-W into a dependability benchmark we added to it a faultload and metric specifications [9]. The faultload consists of environment or operator generated faults injected at precise times; all machines had their clock synchronized using NTP with clock skew smaller than 100ms. The time of failure was chosen to guarantee that full recovery of all failed replicas was observed within the experiment measurement interval. The abrupt server shutdown (crash) has been emulated by killing the application server at the operating system level. The abrupt server reboot (initiates a recovery) has been emulated by re-instantiating the application server. Re-instantiation of application servers is carried out automatically by a simple watchdog process that monitors the application server and re-instantiates it as soon as it detects the crash.

The dependability measures used in the experiments are availability, performability, accuracy, and autonomy [9].

The system under test is available when it is able to provide the service requested by the workload. **Availability** is defined as the ratio between the time the application is operational and the total duration of the run. **Performability** gives an idea of the impact of failures on the performance of the application. It is defined as as the ratio between the average performance (AWIPS) during the failure free period of the measurement interval and the average performance during the period of recovery. **Accuracy** is defined as the ratio between the number of requests with error and the total number of requests of the experiment. **Autonomy** is defined as the ratio between the number of human interventions required to restart a failed replica and the number of faults injected.

5.2 Speedup

Speedup experiments evaluate the maximum possible increase in performance obtained when RobustStore's scale goes from 4 (baseline system) to 12 replicas. The relative speedup for a k-replicated RobustStore is defined by $S_k = \pi_k/\pi_4$, where π_k is the performance of a k-replicated application. Figure 3 shows the speedup values obtained for the three workloads and an initial state size of 500MB. For example, for the browsing workload, $S_8 \approx 1.59$, $S_{10} \approx 1.81$, and $S_{12} \approx 1.97$; the addition of four replicas to the baseline system increases its performance by nearly 60%. Treplica's sublinear speedups are a function of the costs associated with Paxos and Fast Paxos: the message complexity, latency complexity and the latency derived from writing data to stable storage. Thus, the different read/write ratios defined by the workloads pose increasing demands on Treplica's efficiency in terms of network and stable storage. Web interactions that only read values can be executed without resorting to the total order broadcast. This is the case of browsing workload that has only 5% of updates, so 95% of requests (reads) can be fulfilled locally. Also, the small proportion of updates reduces access to disk. So, in this case the good speedup observed (Figure 3 browsing) can be explained by (i) the read-bound workload; (ii) the main-memory residence of the state; and (iii) the light use of the asynchronous persistent queue (total order).

The shopping workload generates 20% of updates, meaning that total order is going to be invoked for at least 20% of operations. In this scenario, the speedup is practically identical to the speedup obtained with the browsing workload. The maintenance of the good speedup for shopping can be explained by the same factors used to explain it for the browsing workload, despite the fact that the shopping load has *four* times the number of updates of the browsing workload. Here, the replicas can no longer be considered independent of each other due to their heavier use of the asynchronous persistent queue (Paxos). Each replica added produces a performance gain of ≈11.3%, with an associated increase in response time of ≈4.29%. The shopping workload is TPC-W's *reference workload*. So, Treplica continues to speed up well when subject to TPC-W's reference workload, but there must be a workload threshold after which the cost of uniform total-ordering impedes the maintenance of the good speedups observed so far. Figure 3 shows that the ordering workload has by far crossed the threshold. In this case, RobustStore's S_8 has dropped to ≈1.29. The change can be explained by the growth in the costs related to Treplica that now has to totally order half of the requests. Each replica added yields a performance gain of ≈5.35%, at the expense of a ≈37% increase in the average response time.

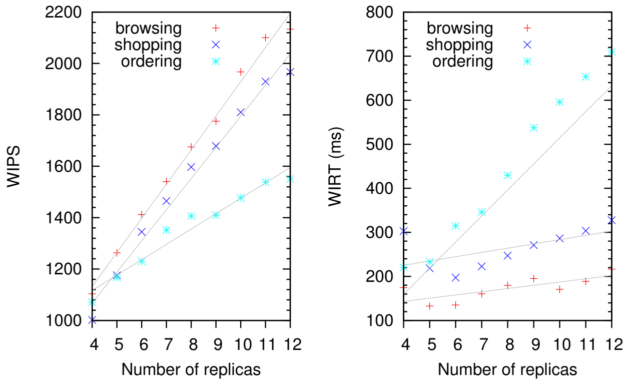

Figure 3. Speedup

5.3 Scaleup

Figure 4 shows how the system scales for a fixed workload of 1000 WIPS and increasing number of replicas. This measurements serve as a baseline to later assess the behaviour of Treplica in the presence of partial failures. An initial replica size of 300MB is used; this size has been chosen to minimize as much as possible interferences caused by swapping. A perfectly scalable system should show an horizontal scaleup line. The determination of the scaleup curves shown by RobustStore for each workload is important as it characterizes its behaviour when the scale is changed. To determine the curves we used regression analysis. The best fit for every set of points is given by a straight line, plotted in gray (confidence coefficients omitted) along the scaleup values (Figure 4). Additionally, we can ask ourselves how performance (WIPS) is related to response time (WIRT). Correlation analysis of the two variables for each workload reveals that they are linearly correlated, with correlation coefficients: $r^2 = 0.8788$ for browsing, $r^2 = 0.9976$ for shopping, and $r^2 = 0.9958$ for ordering. The case $r^2 = 1.0$

corresponds to the maximum possible linear association between WIPS and WIRT, meaning that all data points will lie exactly on a straight line. Thus, we have a system that has performance linearly correlated to response time and that scales up linearly. In Section 5.4 we use these observations to explain the behaviour of RobustStore after a crash.

RobustStore shows an ideal scaleup for the browsing workload, for the same reasons RobustStore shows a good speedup for browsing. For the shopping profile, RobustStore's scaleup is sublinear but with a gradual *linear* decrease in performance, approximately 0.85% per replica added, with a correspondent average increase of WIRT of ≈27.3% (Figure 4). This is a good characteristic, showing that the expected impact of Treplica on the performance is *constant* as the system scales up. In fact, the actual cost of Treplica is smaller than 0.85% for this workload, because the costs inherent to RobustStore and its execution environment (JVM and Tomcat) were not subtracted from the 0.85%. For the ordering profile, each replica added to the configuration causes a constant performance drop of ≈2.1%, with an expected increase in WIRT of ≈25.9% per replica added (Figure 4).

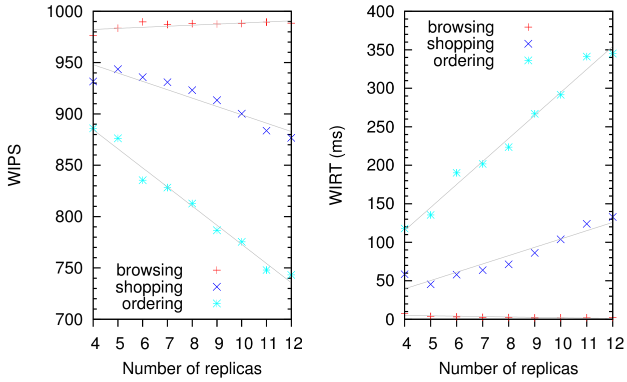

Figure 4. Scaleup for 1000 WIPS

The speedup and scaleup results characterize the behaviour of RobustStore in the absence of failures, but our main focus is not on raw performance but on what happens to performance and other important dependability indicators when RobustStore is subject to crashes.

5.4 One crash, one autonomous recovery

For the first experiment, one crash was injected at t = 270s, followed by the automatic triggering of recovery by the local replica watchdog. Figure 5 shows the behaviour of a five-replicas RobustStore for the three workload profiles. As expected, all curves show a performance drop. Let us start with the curve for the ordering workload. There is a short (≈14s) and sharp (≈700 WIPS) drop in performance. This load surge is caused by the HTTP proxy redistribution of the excess load among the active replicas. What is interesting to note is that after this short period, the recovery is still going to last for another 113s, but the average performance is already close to the performance before the failure. RobustStore's linear correlation between WIPS and WIRT (Section 5.3) can be used to analyse what happens in this scenario. Due to the correlation, a good estimate of the **worst case WIRT** can be obtained by simply considering WIPS as inversely proportional to WIRT. For example, in Figure 5, to estimate the latency at t=275s (the bottom of the deepest valley for the ordering workload) we can subtract ≈140WIPS from 841.4 average WIPS (Table 1, line 5/o, column failure-free AWIPS) to obtain the magnitude of the performance drop: ≈700WIPS. Thus, in the worst case, the latency at t=275s is estimated as ≈700ms. Before the crash it was ≈50ms, as estimated by the regression line in the scaleup WIRT (Figure 4) for 5 replicas. The value sampled by the RBE for the interval of 5s that includes the valley shows a latency of ≈613ms. In Figure 5 it is possible to observe that the browsing and shopping workloads have much lower variability, so do, in the same proportion, the response times associated with them.

Table 1 contains the performability measurements for this experiment. Column R/P shows the replication degree and workload profile. For example, 5/b means five replicas, browsing workload. The variability of the load is characterized by the coefficient of variation (CV): the ratio of the standard deviation of the workload to its mean. The column PV shows the Performance Variation as a percentage of the failure-free AWIPS. Line 5/b shows that RobustStore delivers an average 977.4 WIPSb with a CV of 0.01, almost no variation, during a failure-free run. It also shows that during the recovery period the performance drops to 898.28 WIPSb (-8.1%); a small drop. For the shopping profile PV is smaller than 4% during recovery; performance remains practically stable during recovery. The CV values show that the browsing and shopping workloads have low variability, meaning that the PVs can be trusted to have been caused by the recovery. This is not the case for the ordering workload, with a CV of ≈0.20 for 5/o, and ≈0.33 for 8/o, they render the average WIPS useless as indicators of performance variation. The only resource available in this case is the WIPS histogram (Figure 5). There, it is possible to confirm that there was a performance drop during recovery, and that performance went back to its pre-crash level after the end of the recovery, but the estimated magnitude of performance drop during recovery, ≈13%, cannot be trusted due to the high CVs (Table 1, line 5/o, column PV).

As expected, the recovery times grow as the replica size grows. Figure 6 shows the recovery times for all one-failure experiments for three initial sizes of replica state (300MB,

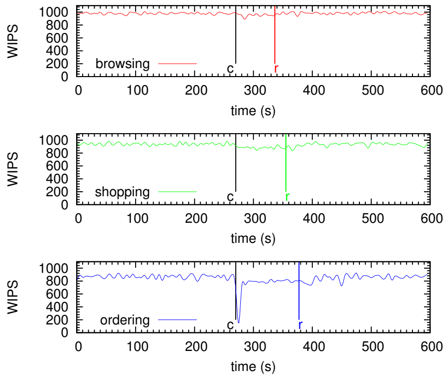

Figure 5. One failure: 5 replicas

	failure free		recovery		
R/P	AWIPS	CV	AWIPS	CV	PV (%)
5/b	977.4	0.01	898.28	0.01	-8.1
5/s	928.1	0.06	884.46	0.07	-4.7
5/o	841.4	0.20	732.33	0.24	-12.9
8/b	985.3	0.01	980.4	0.01	-0.5
8/s	916.8	0.01	903.88	0.09	-1.4
8/o	790.8	0.33	761.74	0.34	-3.7

Table 1. One failure: performability

500MB, and 700MB). For any replication degree, it is clear that recovery times grow faster for the browsing and shopping profiles, than they do for the ordering profile. This can be explained by the way recovery is handled by Treplica. Once a replica is rebooted, the application rebinds to its asynchronous persistent queue and requests the loading of the most recent checkpoint from stable memory. In parallel, the asynchronous persistent queue starts the recovery of the operations that have been enqueued by the remaining replicas since its failure, its backlog. For the browsing and shopping profiles the cost of queue resynchronization is relatively smaller than the cost of loading the most recent checkpoint from disk, so parallelization helps but still the time to recover is dominated by the loading of the checkpoint from disk. For the ordering profile, both state transfers become larger. In this case, the parallelization of the tasks contributes to a noticeable reduction of the total time of recovery, leveling the recovery times as we move across different state sizes, and reducing the impact of Treplica on RobustStore's performance during recovery. For the next experiments we have omitted the recovery times to save space, but the same recovery pattern was observed.

Table 2 shows the accuracy of RobustStore in the presence of one crash. Clearly, RobustStore produces very few erroneous outputs when subject to one crash-recover failure.

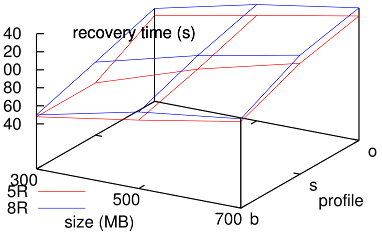

Figure 6. One failure: recovery times

replicas	browsing	shopping	ordering
5	99.999	99.999	99.985
8	99.999	99.999	99.986

Table 2. One failure: accuracy

5.5 Two crashes, autonomous recoveries

In this set of experiments RobustStore is subject to two concurrent crashes, followed by autonomous recoveries of the crashed replicas. The replicas to be crashed were chosen at random and crashed at t=240s and t=270s. The WIPS histogram (Figure 7) shows small performance losses during recovery for all three workloads. For the browsing profile, the first replica crashed becomes operational at t = 303s, approximately 63s after the crash. The second replica rejoins RobustStore at t=336.8s, 66.8s after it crashed. In a little more than a minute the two replicas, with state sizes greater than 500MB, had already rejoined RobustStore. The shopping and ordering profiles also show that RobustStore recovers gracefully from the concurrent crashes even when exposed to increasingly write-intensive workloads. Table 3 shows that the largest PV is inferior to 5%, a drop that can be considered small given the adverse crash scenario generated by the faultload. The CVs for the ordering profile are high and similar to the ones observed before for one crash. Table 4 shows that RobustStore has maintained a high accuracy when submitted to concurrent crashes. From the point of view of maintainability and autonomy, RobustStore has so far shown that it can recover fully automatically, to a good extent due to its reliance on the simple recovery mechanism offered by Treplica (Section 2).

5.6 Two crashes, one autonomous, one delayed recovery

The last experiment has been designed to show how Treplica influences the performance of RobustStore in a

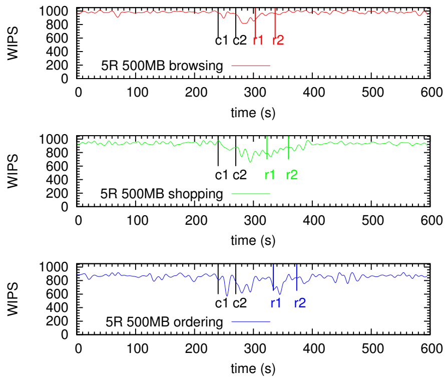

Figure 7. Two overlapped crashes

	failure free		recovery		
R/P	AWIPS	CV	AWIPS	CV	PV (%)
5/b	971.5	0.02	942.24	0.02	-3.0
5/s	910.4	0.09	876.58	0.09	-3.7
5/o	841.5	0.21	801.96	0.22	-4.7
8/b	982.8	0.01	962.6	0.01	-2.0
8/s	907.9	0.01	891.32	0.01	-1.8
8/o	787.1	0.33	763.96	0.34	-2.9

Table 3. Two overl. crashes: performability

replicas	browsing	shopping	ordering
5	99.998	99.993	99.978
8	99.999	99.998	99.978

Table 4. Two overlapped crashes: accuracy

scenario where a replica recovers long after it crashed. This is an important issue for Treplica because of how Paxos and Fast Paxos work. During the downtime of the crashed replica, the active replicas have delivered a large number of operations to the application. This means that the recovering replica is going to have to load the checkpoint from stable memory and spend a larger period learning (state transfer) from the other replicas, before it re-synchronizes itself and can resume normal operation. In this scenario (Figure 8), both replicas are crashed at t=240s. The recovery of one of the crashed replicas is triggered automatically. The recovery of the second replica is triggered manually at t=390s. Consider the shopping profile. At this moment, the first failed replica has already ended its recovery process, that took ≈70s. The throughput curve shows that the recovery process implemented by Treplica has a small impact on performance of RobustStore for all workloads. Table 5 does not contain the CV values because they are very similar to the CV values obtained for the other two faultloads. Consider, for example, the shopping workload and five replicas. The impact on performance for the first failure is similar to the one verified in the case of two concurrent crashes. During a period of time RobustStore operates with 3 replicas, then the first failed replica recovers, taking RobustStore to 4 replicas. In the scaleup experiments using failure-free runs, we have observed that the addition of a replica causes an average performance drop of ≈8%. So a four-replicas RobustStore should perform an average 8% better. Recall that this reasoning is only valid because of the very low CVs shown by the shopping workload. The AWIPS during the period from r_1 to u_2 is 902.78 WIPS. The four-replicas RobustStore does not perform better because it is still processing the backlog of operations created by the two simultaneous failures, but it has recovered to a performance level that is only 1.4% below the performance before the crashes; the shopping workload has a CV = 0.09. The second recovery affects even less the performance of RobustStore, because the extra broadcasts demanded by the recovering replica to re-synchronize itself with the active replicas are processed concurrently by Treplica (Paxos). The consequence of this characteristic of Treplica is a reduced impact on performance stability, at the expense of a longer recovery time. (Figure 8). The same reasoning is valid for the other workloads, but, as stated before, the values of PV for the ordering profile are not valid because of the high variability of this workload. During these experiments, RobustStore's accuracy (Table 6) remained high and consistent with the accuracies found in the previous experiments.

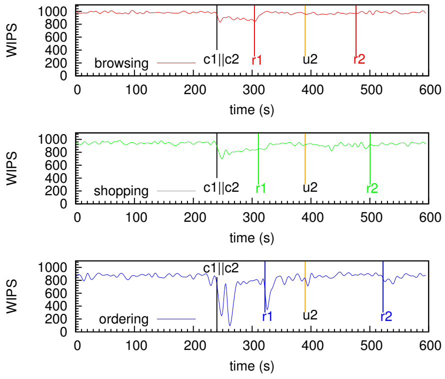

Figure 8. Delayed recovery

5.7 Discussion

Four questions were posed at the beginning of this Section. **1.** How long can RobustStore be expected to run without interruption? In the presence of only benign crashes,

	no failures	recovery R1		recovery R2	
R/P	AWIPS	AWIPS	PV (%)	AWIPS	PV (%)
5/b	966.6	858.49	-11.1	919.58	-4.8
5/s	915.3	813.09	-11.2	905.89	-1.0
5/o	821.2	603.31	-26.5	852.12	+3.8
8/b	985.1	949.3	-3.63	948.65	-3.7
8/s	915.0	864.94	-5.5	906.01	-1.0
8/o	785.6	686.67	-12.6	802.08	+2.1

Table 5. Delayed recovery: performability

replicas	browsing	shopping	ordering
5	99.990	99.988	99.957
8	99.998	99.995	99.974

Table 6. Delayed recovery: accuracy

as assumed, RobustStore will remain operational forever. **2.** How much service can RobustStore be expected to deliver during failure-free and failure-prone operation periods? RobustStore's throughput can be characterized as very resilient, and stable in the presence of the crashes, failover, and recoveries used in the experiments. We have carried out 18 dependability experiments, 6 for each faultload specified. For each replication factor (5 or 8) three initial sizes of RobustStore replicas were instantiated (300, 500, and 700MB). All these experiments have shown that RobustStore loses less than 13% of its average performance during recovery in the worst case, which occurs with the faultload that injects two concurrent crashes, later followed by autonomous recoveries. The longest recovery occurred in the experiment with two crashes and delayed recovery of one replica. It took the second recovering replica about 180s to become operational in a setting with 8 replicas, ordering profile, and a 700MB state size. During the 180s recovery the average throughput practically remained at the same level displayed during the failure-free period. For the shopping profile, the profile considered by TPC-W as the one that best approximates the behaviour of a dynamic content web service, RobustStore worst average performance loss is inferior to $\approx$4.0%. **3.** What accuracy can be expected of RobustStore in failure-prone executions? Very high, three 9s, in the worst case. **4.** What level of human intervention is necessary to maintain RobustStore operational? None, when subject to the faultloads presented here, RobustStore has shown total autonomy. The combined effect of high accuracy, throughput resilience, and full autonomy allows the conclusion that RobustStore is indeed a highly available dynamic content web application.

6 Related Work

Paxos and recovery. Here we comment on work whose applications were built upon middleware that uses uniform repeated consensus (total order broadcast) [8]. Specifically, we are interested in toolkits that implement Paxos [15]. Examples of applications that satisfy this criteria include a lock service [2], data center management [12], data storage systems [18, 21], database replication [11], a distributed hash table system [13], and dynamic content web services [7, 20]. The projects listed in Table 7 have successfully employed the state machine approach [14] and *uniform* total order broadcast based on Paxos to replicate *critical* application state, with systems often combining different replication mechanisms to obtain the required degree of reliability and performance. A key aspect of all papers listed in Table 7 is that their experiments were primarily designed to assess performance, not dependability, with the exception of FAB that shows fault-tolerance results for disk arrays. Most of the systems opted for the traditional message passing interface to expose Paxos, with the exception of Chubby. By contrast, we have opted to present uniform total order using a queue abstraction; queues are simple and widely-used objects.

There is much research on mechanisms to make dynamic content web applications highly available with emphasis on their performance improvement. Various reliable data management solutions have already been used, from file-based implementations (e.g., [5]) to database-based implementations (e.g. [4, 11, 1]). Tashkent's experiments (Table 7) were carried out using a dynamic web content application. Sprint, FAB, and Chubby (Table 7) can be used as supports to build highly available dynamic content web applications.

Institution	Project Name	Paxos		1st Publ.
		Classic	Fast	Date
HP	FAB [21]	●		2004
Microsoft	Boxwood [18]	●		2004
EPFL/USI	Tashkent [10]	●		2006
Microsoft	Autopilot [12]	●		2007
Google	Chubby [2]	●		2007
USI	Sprint [4]	●		2007
UNICAMP	Treplica [22]	●	●	2008

Table 7. Paxos and Application Availability.

Replicated databases and recovery. Liang and Kemme [17] compare two recovery strategies: (i) total versus (ii) partial copy of the database. They assess the trade-offs of (i) and (ii) in runs where a single failure occurs. Manassiev [19] reports, using TPC-W and a faultload with a single crash, on the availability of a multiversion master-slave in-memory database that tolerates a single failure. They show that it is possible to reduce the impact of recoveries on the availability of the replicated database. Treplica offers a simpler recovery and failover solution that does not require the maintenance of hot backups for fast failover. Wu and Kemme [23] consider different recovery strategies depending on the failure scenario: (i) a single failed replica must be recovered or (ii) all replicas have to be recovered.

7 Conclusion

We have presented a dependability analysis of RobustStore, a highly available dynamic content web application built upon Treplica. Treplica's programming interface, based on only 8 methods, simplifies the programming tasks associated with the construction of highly available applications, relieving the programmer of important concerns related to the recovery. We like to consider Treplica as Paxos made simple *in practice*, a great benefit for developers of highly available applications. The experimental results show that RobustStore/Treplica performs well in the presence of crashes and recoveries, showing very good performance stability, continuous availability and high accuracy. They also contribute to a better understanding of the impact of Paxos and Fast Paxos when used as building blocks of a replication middleware.

From the point of view of dependability benchmarking, we have shown that not all workloads of TPC-W can be used as off-the-shelf indicators in dependability experiments. The coefficient of variation of the browsing and shopping workloads warrant them as good workloads for dependability assessment. Unfortunately, the same cannot be said about the ordering workload because of its high variability. This shortcoming of TPC-W can motivate further research on the development of dependability benchmarks for dynamic content web applications.

References

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, 2003.

[2] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symp. on Operating Systems Design and Implementation*, 2006.

[3] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *Proc. of 7th Int. Symp. on High-Performance Computer Architecture*, 2001.

[4] L. Camargos, F. Pedone, and M. Weiloch. Sprint: a middleware for high-performance transaction processing. In *Proc. of 2nd European Conf. on Computer Systems*, 2007.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

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

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of 21st ACM SIGOPS Symp. on Operating Systems Principles*, pages 205–220, 2007.

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

[9] J. Durães, M. Vieira, and H. Madeira. Dependability benchmarking of web-servers. In *Proc. of 23rd Computer Safety, Reliability, and Security Int. Conf.*, pages 297–310, 2004.

[10] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proc. of 1st European Conference on Computer Systems (EuroSys)*, 2006.

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

[12] M. Isard. Autopilot: automatic data center management. *Oper. Syst. Rev.*, 41:60–67, 2007.

[13] Y. Jiang, G. Xue, and J. You. Toward fault-tolerant atomic data access in mutable distributed hash tables. In *Proc. of First Int. Multi-Symp. on Computer and Computational Sciences*, 2006.

[14] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

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

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

[17] W. Liang and B. Kemme. Online recovery in cluster databases. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 121–132, New York, NY, USA, 2008. ACM.

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

[19] K. Manassiev and C. Amza. Scaling and continuous availability in database server clusters through multiversion replication. In *Int. Conf. on Dependable Systems and Networks*, 2007.

[20] J. Ostell. Databases of discovery. *Queue*, 3(3):40–48, 2005.

[21] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGPLAN Not.*, 39(11):48–58, 2004.

[22] G. M. D. Vieira and L. E. Buzato. Treplica: Ubiquitous replication. In *Proc. of 26th Brazilian Symp. on Computer Networks and Distributed Systems*, 2008.

[23] S. Wu and B. Kemme. Postgres-R (SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 422–433, 2005.