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

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

Technical Report - IC-08-34 - Relatório Técnico

December - 2008 - Dezembro

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

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

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

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

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 indicate that Treplica simplifies the programming of application recovery, a great benefit for programmers of highly available applications.

1 Introduction

In this work, we evaluate how *crashes and recoveries* affect the performance and availability of RobustStore, a highly available dynamic content web application. Experimental research on crash-recovery and failover for these applications has not yet received much attention for at least three reasons. First, performance measurements of highly available applications have so far prevailed over fault-tolerance measurements as they are important to evidence that middleware-based active replication is able to deliver satisfactory performance [6, 15]. Second, conducting experiments with applications whose components crash and later recover is intrinsically harder than experimenting with the same applications on failure-free executions [13, 25]. Third, recovery of replicated systems is harder then recovery of stand-alone systems because it involves bringing the state of the crashed replica back to a state synchronized with the current state of the active replicas.

[*]Supported by CNPq grant 142638/2005-6.
[†]On a sabbatical leave at EPFL. Supported by CNPq grant 201934/2007-8.

RobustStore shows continuous availability, a very high accuracy, and total autonomy in the presence of crashes and recoveries. Usually for highly available applications, the recovery process is a main concern because it has a negative impact on the availability and reliability of the application. 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 resilience is good and stays close to the levels delivered before the failures occurred. Our results (i) narrow the gap between performance and dependability characterizations of dynamic content web applications and (ii) show that uniformity (consensus and total order) is a key property for replication toolkits because it keeps the recovery process simple, and (iii) show that the functionality of Treplica's asynchronous persistent queue and state machine have eased the task of fault-tolerant programming.

The remaining of the paper is structured as follows. Section 2 describes Treplica, a replication toolkit that implements both Paxos and Fast Paxos. Treplica has been designed with modularity and ease-of-use as primary objectives; it offers two very simple programming abstractions for programmers: state machine and asynchronous persistent queue. Section 3 summarizes the features of TPC-W. Section 4 describes RobustStore, the highly available application created by retrofitting a stand-alone dynamic content web application with Treplica. We show how we have dealt with non-determinism, randomness, and database substitution. Section 5 discusses, with the support of experimental results, how the performance of RobustStore is affected by crashes, failovers, and recoveries. Section 6 brings a concise account of research on Paxos-based applications, recovery, failover, and dependability benchmarking. Section 7 summarizes our results and contributions.

2 Treplica

This Section explains the features of Treplica that are relevant to this work; additional information can be found in [32, 33]. Treplica supports the construction of highly available applications through either the asynchronous persistent queue or the state machine programming interfaces.

Asynchronous persistent queues are simple. A queue is a *totally ordered* collection of actions with the usual `enqueue(Object)` and `Object dequeue()` methods. The method `create(queueId)` creates a queue endpoint identified by the provided queue identifier. All processes of the replicated application use the *same* queue, but interact with it through local queue endpoints. Persistence means that a process 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 active replicas. Thus, an asynchronous persistent queue maintains a history of the actions it has ever held since its creation. This means, for example, that a replica that returns from a failure can join an already existing queue, using the queue's unique identifier, and it will be able to dequeue all actions ever enqueued. Thus, by relying on the total order guaranteed by the queue and in 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. To be able to use the services of a queue every application is supposed to implement a StateManager and register it with the queue by calling `bind(stateManager)`. The StateManager is a simple component that must be capable of taking and processing checkpoints of the application's state; it implements the `takeCheckpoint()` operation. Checkpoints are taken by the application but triggered via a callback to `takeCheckpoint()` by the asynchronous persistent queue and are used to speed recoveries, as we describe later.

Asynchronous means that the enqueue operation is a non-blocking operation. Despite its simplicity, the non-blocking characteristic of enqueue requires concurrent programming; its disciplined use will probably induce a programmer to build a state machine to handle active replication, or at least a monitor to handle, on the one hand, the concurrent asynchronous events generated by the queue, and on the other, the synchronous calls made by its clients. To ease this task, Treplica provides a ready-to-use state machine programming abstraction. From Treplica's perspective, the state machine is a black box implemented as a component of the retrofitted application. Treplica only manages the actions (events) that drive the state machine, which are implemented as Java objects. The programming interface of the state machine contains only three methods. A state machine is created and bound to an asynchronous persistent queue by calling the method `create(initialState, queue)`. An initial state must be provided, because the replica that calls the method can be the first one binding to the queue. The method `getState()` returns the current state of the application. Once bound to a queue, an application replica can invoke this method at any time. The method `Object execute(Action)` is a blocking method that executes a totally ordered action on the local state machine, performing all necessary steps to coordinate its execution with the other replicas, through the asynchronous persistent queue. It returns an object that contains the results of the action executed.

Applications built atop Treplica must adhere to the state machine approach [21, 31] regardless of whether they use the state machine or directly the asynchronous persistent queue interface. If necessary, the application can use multiple threads to service its clients, but the use of Treplica's state machine guarantees that only one thread at a time executes an action on the state machine. It is important to note that application determinism does not reduce the internal parallelism of Treplica; the deterministic state machine does not condition performance.

Recovery: Suppose a replica crashes and some time later recovers. Initially, a stateless instance of the application is created and its constructor, in its turn, eventually instantiates a state machine and invokes `getState()` on it. 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 checkpoint and an associated suffix of the queue's history. The checkpoint is obtained locally. After resetting its state to the state of the checkpoint, the application 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 it has to do is to call `getState()`, the rest is transparently handled by Treplica.

The queue's suffix that must be recovered can be very large. In fact, in the absence of a checkpointing mechanism it would only keep growing while the application executed. In Treplica, state managers play a key role in the collection of the queue's stale suffix and application checkpoints. Every time a new checkpoint is induced by the queue its previous checkpoint and the history prefix that precedes the checkpoint can be garbage collected. Internally, the queue manages the checkpoints and guarantees that when needed a replica receives the most recent checkpoint and is able to learn the queue's suffix necessary re-synchronize itself with the active replicas.

3 The TPC-W Benchmark

The TPC-W benchmark specifies all the functionality of an on-line bookstore, defining the access pages, their layout, and image thumbnails, the database structure. The bookstore application is based on a standard three-tier software architecture. Enterprises [10] and University research groups [1, 14, 15, 26] have extensively used TPC-W's bookstore and workloads to assess the performance of machines, operating systems, and databases as supports for web services. TPC-W defines three workloads that are differentiated from each other by varying the ratio of browsing (read access) to ordering (write access) in the web interactions. Performance is measured in *web interactions per second* (WIPS). An example of a read-only interaction is the search for books by a given author, while an example of an update interaction is the placement of an order. The shopping profile specifies that 80% of the accesses are read-only and that 20% generate updates, measured in WIPS. The browsing profile specifies that 95% of the accesses are read-only and that only 5% generate updates (WIPSb). Finally, the ordering profile fixes a distribution where 50% of the accesses are read-only and 50% generate updates (WIPSo). The implementation of these profiles is made by the remote browser emulator (RBE). To emulate the behaviour of human interactions, that inherently include inactivity periods, the workload definition includes a *think time*, defined by TPC-W as 7 seconds. The total number of web interactions generated per second by a set of emulated RBEs can be calculated as the #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 have made to the original implementation of the TPC-W online bookstore [5] 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.

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 have created an object model composed by 9 classes that represent the entities and relations of TPC-W's online bookstore conceptual model. These classes and its 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 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 state machine uses an asynchronous persistent queue to totally order the operations and make them available to each replica; recall that each replica accesses the asynchronous persistent queue through its local endpoint. In total, the conversion process took the equivalent of 2 work weeks of a programmer familiar with Treplica. The creation of the object model from the database conceptual model was easy because much of the structure of the model could be recovered by direct inspection of the SQL queries and indexes employed by the facade class to interact with the database. 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. Use of random numbers, dates and times is not a problem for a centralized system, but it is a problem for a replicated system. For example, whenever a new order is created the order creation time is set to the current time. If each replica used its local clock to determine the order creation time inside the create order method, then each of the replicas would most likely end up with a different order creation time. To avoid this, the code in the facade responsible for creating new order actions in the state machine consults its local clock *before* the action is created and the resulting time is passed as an action argument. Now, every replica is going to have exactly the same value for the order creation time. Calls to random number generators, for example, to generate the discount for a newly created customer, were also pulled out of the method that would otherwise have generated different values at each replica. In our solution one of two mechanisms has been adopted. Either, call the random number generator once to obtain a value, so that the same value was used by all replicas, or pass the same seed to the pseudo-random number generators of all replicas.

Once we were sure that all critical state was encapsulated by the classes of the model and that all methods complied with the state machine's requirement of determinism, the retrofitting had ended. A very important observation is that during the whole retrofitting process we did not have to concern ourselves with any aspect of replication, persistence or the 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? What level of human intervention is necessary to maintain RobustStore operational? We have 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 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.

The cluster has been divided into three disjoint sets of nodes, each node executed only the software necessary for the experiments. The first set contains the server replicas that run the bookstore application, and contains from 4 to 12 servers. 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 network traffic generated during the experiments result from Treplica's total-order broadcasts. The TPCW_Populate program generates RobustStore's initial data as a function of two parameters: number of RBEs and number of objects (books). These two parameters determine the number of customers, number of address profiles, number of authors, and number of orders. Parameters were chosen to generate initial application state sizes of 300MB, 500MB, and 700MB. During the execution the growth rate of the state size of a replica is a function of the workload applied. For the ordering workloads the average state size at the end of the measurement interval is approximately 550MB, 750MB, and 950MB; all states 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.

The second set contains only one node and runs the reverse proxy HAProxy, that has a load balancing module. The load balancer listens on a TCP port for HTTP requests from the browser emulators. These requests are forwarded to the application servers on the server replicas. The HAProxy configuration implements key failover mechanisms. 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. The HAProxy uses the hash to distribute the load uniformly among the active replicas as long as the hash keys are distributed uniformly; this is the case for the clients unique ids. If a server fails during the execution of a client request, HAProxy will terminate the connection and the client will observe an error.

The third set is composed by 5 client nodes that run the RBEs. Each client node instantiates the number of RBEs required to generate a specified workload, every node executes the same number of RBEs. 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 the workloads generated by the RBEs of the 5 client nodes were not sufficient to stress RobustStore. It is important to note that shorter think times do not change neither the read to write ratios nor the probabilistic characteristic of the workloads. Performance metrics are written by the RBEs into log files stored in the local disk. Except for the speedup and scaleup, where failure-free runs were used, the workload generated by the RBEs has always been calibrated to 1000 WIPS, the maximum workload sustainable by the given replica configurations to guarantee that the errors counted during the crash-recovery essays resulted strictly from the faults injected. The ramp-up and ramp-down periods follow TPC-W's specification; they were set to 30 seconds each. The measurement interval was set to 9 minutes; a period long enough to guarantee that all crashes and recoveries occur inside the period were the system is in steady operation.

RobustStore does not rely on a database, but the changes we have made to the application do not affect the generation of test data or the transactional semantics of the interactions. The TPC-W function that determines the number of objects and amount of data generated as a function of the number of clients (RBEs) has not been changed, to maintain the workload as originally implemented.

In brief, a dependability benchmark consists of a system specification, a faultload, a workload and a metric [19]. The TPC-W benchmark consists of a system specification, a workload and a metric. Thus, to turn TPC-W into a dependability benchmark we added to it a faultload and metric specifications [13]. 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 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 are explained later, during the discussion of the experimental results.

5.2 Speedup

Speedup experiments evaluate the maximum possible increase in performance obtained when RobustStore's scale goes from 4 to 12 replicas. Figure 1 shows the speedup values obtained for the three workloads and an initial state size of 500MB. Treplica's performance is a function of the costs associated with Paxos and Fast Paxos, that are, 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 superlinearity 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. 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 superlinearity can still be explained by the same factors used to explain it for the browsing workload, despite the fact that this load has four times the number of updates of the browsing workload. Here, the replicas can no longer be considered independent of each other because of their shared use of the asynchronous persistent queue (Paxos). Each replica added yelds 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, Treplica does well here. Certainly, there is a threshold after which the cost of uniform total-ordering impedes superlinearity. Figure 1 shows that the ordering workload has by far crossed this threshold. In this case, RobustStore's speedup is sublinear, and 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 yelds a performance gain of ≈5.35%, at the expense of a ≈37% increase in the average response time.

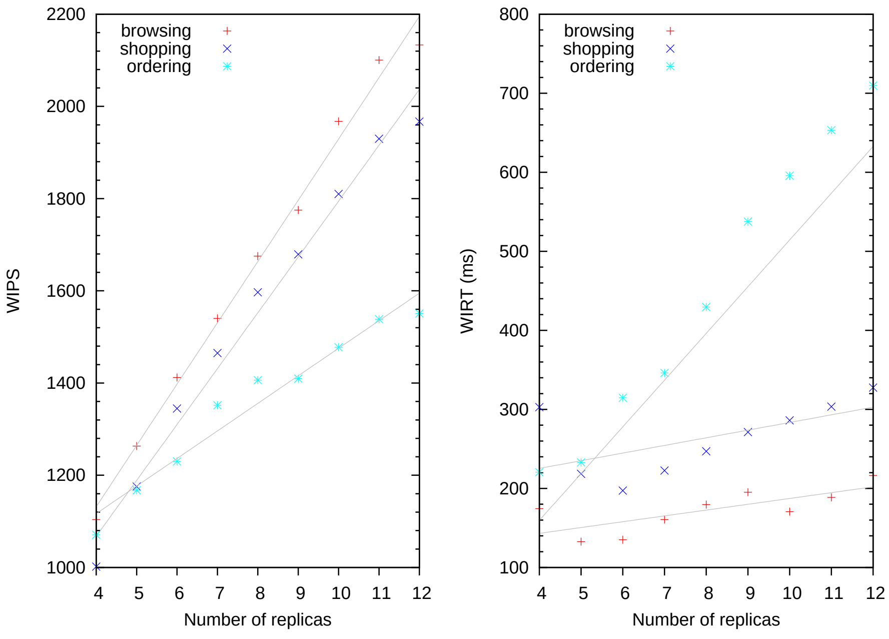

Figure 1: Speedup

5.3 Scaleup

Figure 2 show how the system scales for a fixed workload of 1000 WIPS and increasing replica state sizes. 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 have used regression analysis. The best fit for every set of points is given by a line (confidence coefficients ommitted), plotted as gray lines along the scaleup values (Figure 2). 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 $P < 0.0001$ and $r^2 = 0.8788$ for browsing, $P < 0.0001$ and $r^2 = 0.9976$ for shopping, and $P < 0.0001$ and $r^2 = 0.9958$ for ordering. Thus, we have a system that scales linearly and has performance linearly related to response time. These facts help the analysis of RobustStore's behaviour

RobustStore shows a superlinear scalability for the browsing workload, for the same reasons RobustStore has a superlinear speedup for browsing. For the shopping profile, RobustStore's scaleup is sublinear but with a gradual *linear* decrease in performance; ≈0.85% per replica added, with a correspondent average increase of WIRT of ≈27.3% (Figure 2). 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 2).

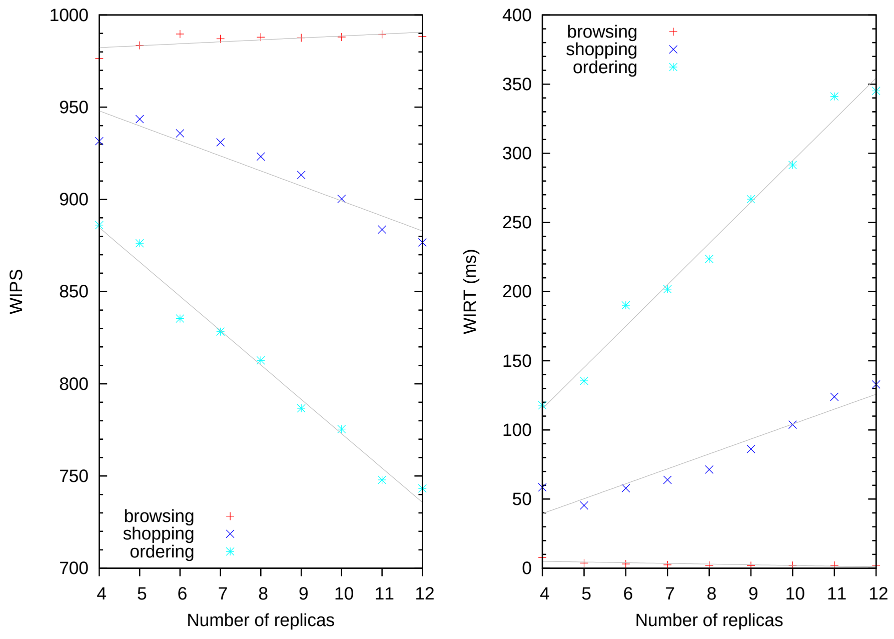

Figure 2: Scaleup for 1000 WIPS

The speedup and scaleup results have tried to characterize the behaviour of RobustStore in the absence of failures, but our main focus is not 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 3 shows the behaviour of a five replica 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 scaleup (Section 5.3) show its utility here. Due to the linearity, a good estimate of the **worst case WIRT** can be obtained by simply considering WIPS as inversely proportional to WIRT. For example, in Figure 3, to estimate the latency at t=275s (the bottom of deepest the 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 2) 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 3 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 allows the comparison of average performances (AWIPS) during failure free runs with the average performances during the period of recovery. 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 variance (COV). 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 WIPS browsing (WIPSb) with a COV 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 COV 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 COV 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 3). There, it is possible 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 COVs (Table 1, line 5/o, column PV).

As expected, the recovery times grow as the replica size grows. Figure 4 shows the recovery times all one-failure experiments for three initial sizes of replica state (300MB,

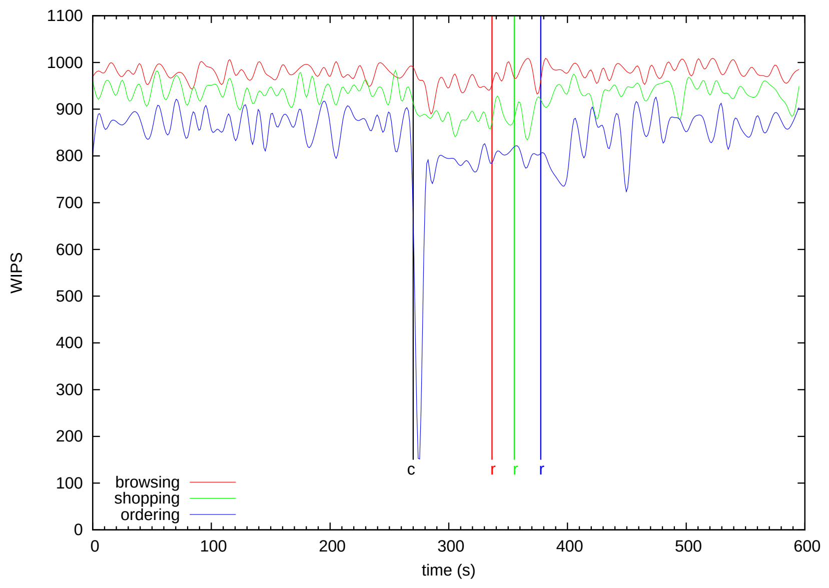

Figure 3: One failure: 5 replicas

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 pattern emerged from the results obtained for experiments with two-failures.

Table 2 shows the error rate in the presence of faults, accuracy [13], for RobustStore in the presence of one crash. Clearly, RobustStore produces very few erroneous outputs when subject to failures.

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 5) shows small

	failure free		recovery		
R/P	AWIPS	COV	AWIPS	COV	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

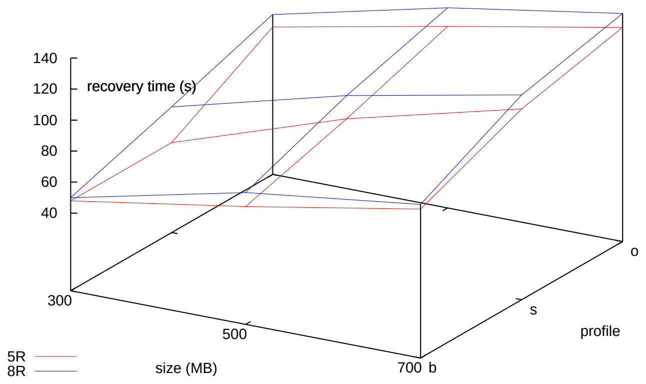

Figure 4: One failure: recovery times

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 re-joins 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-intense 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 COVs 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 showed that it can recover fully automatically, in the event of one crash, previous experiment, and two concurrent crashes.

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

Table 2: One failure: accuracy

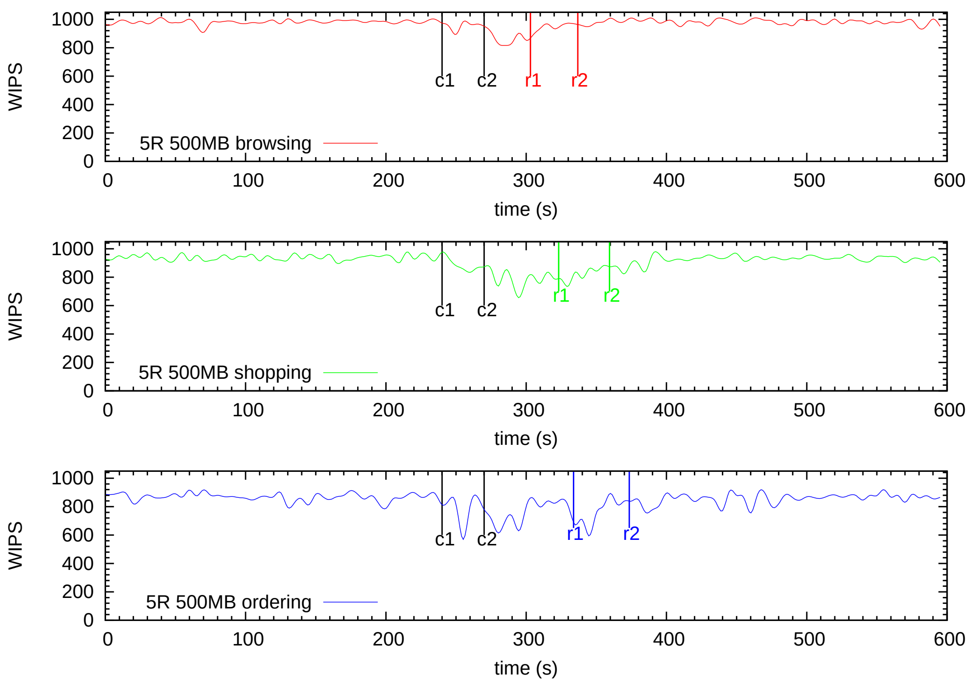

Figure 5: Two overlapped crashes

5.6 Two crashes, one autonomous, one delayed recovery

The last experiment has been designed to show how Treplica affects performance in a 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 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 6), 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

	failure free		recovery		
R/P	AWIPS	COV	AWIPS	COV	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

contain the COV values because they are very similar to the COV 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 replica RobustStore should perform an average 8% better. Recall that this reasoning is only valid because of the very low COVs shown by the shopping workload. The AWIPS during the period from r_1 to u_2 is 902.78 WIPS. The 4 replica 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 worload has a COV = 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 6). 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 the experiments, RobustStore's accuracy (Table 6) remained high and consistent with the accuracies found in the previous experiments.

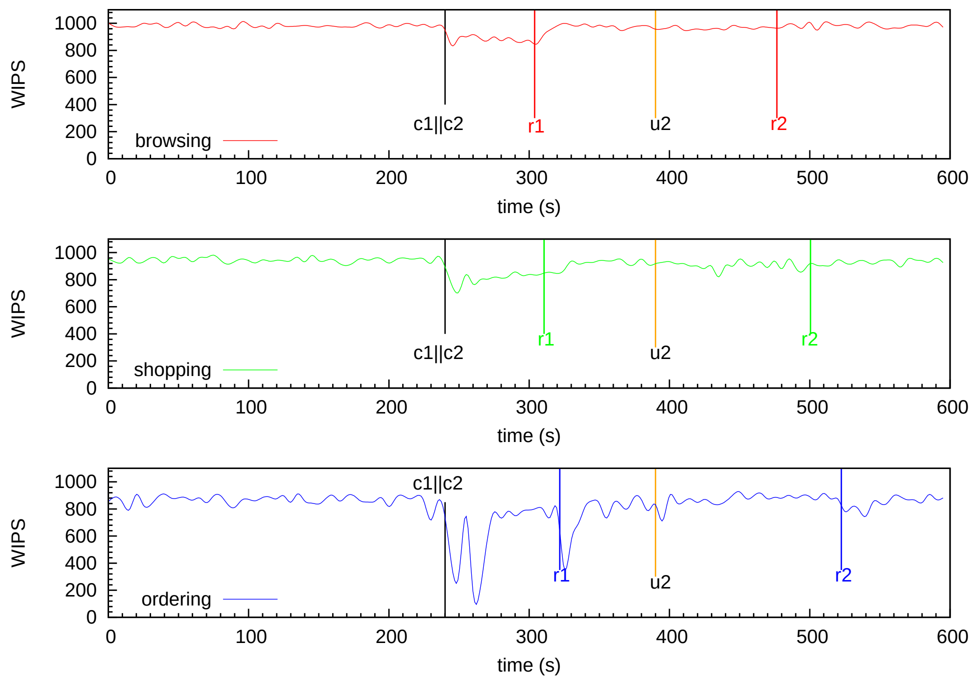

Figure 6: Delayed recovery

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, 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 looses less than 13% of its average performance during recovery in the worst case, that 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

	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

maintain RobustStore operational? None, when subject to the faultloads presented here, RobustStore has showed 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 works whose applications were built upon middleware that uses uniform repeated consensus (total order broadcast) [12]. Specifically, we are interested in toolkits that implement Paxos [22]. Examples of applications that satisfy this criteria include a lock service [4], data center management [16], data storage systems [24, 30], database replication [14, 6], a distributed hash table system [17], and dynamic content web services [11, 27]. The projects listed in Table 7 have successfully employed the state machine approach [21, 31, 28] 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., [8, 27]) to database-based implementations (e.g. [14, 15, 6, 2]). 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	
HP	FAB [30]	•		2004
Microsoft	Boxwood [24]	•		2004
EPFL/USI	Tashkent [14]	•		2006
Microsoft	Autopilot [16]	•		2007
Google	Chubby [4]	•		2007
USI	Sprint [6]	•		2007
UNICAMP	Treplica [33]	•	•	2008

Table 7: Paxos and Application Availability.

Replicated databases and recovery. Liang and Kemme [23] 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. View-synchrony group communication is used by the middleware, leading to the specification of a recovery protocol that has to take into account view changes; this requires blocking and unblocking group communication during the time a replica is actually updating its state. Manassiev [26] 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. Chen [9] study performance as a function of autonomic provisioning of replicas. A resource manager allocates replicas on-demand during load surges. The authors focus on the comparison of reactive and proactive policies for the provisioning of resources. This work only considers planed reconfiguration in failure free runs. Wu and Kemme [36] address recovery. Their work considers different recovery strategies depending on the failure scenario: (i) an active replica fails is reactivated and re-enters the replica group, (ii) a new replica joins the group, and (iii) after a total failure, the replicated system is brought back to operation. Ganymed [29] is a master-slave replication middleware, total order broadcast is not employed. The experiments with Ganymed show the effect of crashes without recoveries of both a slave replica and the master replica. Jímenez-Peris and colleagues [18] specify a reasonably complex to implement recovery protocol for replicated applications built atop a group communication toolkit that offers strong view synchrony. Our solution does not require any complex recovery protocol. Kemme [20] present algorithms based on virtual synchrony [3] that allow the reconfiguration of replicated databases in the presence of failures. It is a theoretical work that shows how recovery can be accomplished by extending virtual synchrony model to ease the specification of recovery protocols. In summary, the papers reviewed show that: (i) all prototypes have based replica communication upon group communication toolkits that implement view synchrony. Recovery using virtual synchrony group communication toolkits seem to require recovery protocols more complex than the one presented here; (ii) research on autonomous failover and recovery represents an important problem that requires further research, specially when we consider the use of replicated systems in the context of modern data centers, with thousands of machines.

Dependability benchmarks. Here we briefly review two other efforts in this area. The DBench project [19] has produced a number of dependability benchmarks for enterprise (online transaction processing [35, 34], operating systems), space, and automotive applications. The results obtained in this project highlight the benefits of dependability benchmarking for computing systems. The recovery-oriented computing project maintained by Stanford and Berkeley has also produced dependability benchmarks. Candea et al [7] have proposed action-weighted throughput as the metric for the availability of Internet services. Action-weighted throughput is measured in terms of actions (responses) per second, where an action is a group of HTTP requests that form an operation of interest to the end user.

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 variance (COV) 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. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proc. of 4th USENIX Symp. on Internet Techn. and Syst.*, 2003.

[2] 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.

[3] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

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

[5] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko H. Lipasti. An architectural evaluation of Java TPC-W. In *Proc. of 7th Int. Symp. on High-Performance Computer Architecture*, 2001.

[6] Lásaro Camargos, Fernando Pedone, and Marcin Weiloch. Sprint: a middleware for high-performance transaction processing. In *Proc. of 2nd European Conf. on Computer Systems*, 2007.

[7] George Candea, Aaron B. Brown, Armando Fox, and David Patterson. Recovery-oriented computing: Building multitier dependability. *Computer*, 37(11):60–67, 2004.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[9] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *Proc. of 3rd IEEE Int. Conf. on Autonomic Computing*, 2006.

[10] Trans. Proces. Council. *TPC-W Specification*, February 2002.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of 21st ACM SIGOPS Symp. on Operating Systems Principles*, pages 205–220, 2007.

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

[13] João Durães, Marco Vieira, and Henrique Madeira. Dependability benchmarking of web-servers. In *Proc. of 23rd Computer Safety, Reliability, and Security Int. Conf.*, pages 297–310, 2004.

[14] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proc. of 1st European Conference on Computer Systems*, 2006.

[15] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: memory-aware load balancing and update filtering in replicated databases. *Oper. Syst. Rev.*, 41(3):399–412, 2007.

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

[17] Yi Jiang, Guangtao Xue, and Jinyuan 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.

[18] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *Reliable Distributed Systems, IEEE Symposium on*, page 150, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[19] K. Kanoun, H. Madeira, and J. Arlat. A Framework for Dependability Benchmarking. In *DSN 2002: Proceedings of the Workshop on Dependability Benchmarking*, 2002.

[20] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks (DSN2001)*, 2001.

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

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

[23] WeiBin Liang and Bettina 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.

[24] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of 6th USENIX Symp. on Operating Systems Design and Implementation*, 2004.

[25] Henrique Madeira and Philip Koopman. Dependability benchmarking: making choices in an n-dimensional problem space. In *Proc. of the 1st Workshop on Evaluating and Architecting System dependability*, 2001.

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

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

[28] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, 2003.

[29] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[30] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGPLAN Not.*, 39(11):48–58, 2004.

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

[32] Gustavo M. D. Vieira and Luiz E. Buzato. On the coordinator's rule for fast paxos. *Inf. Process. Lett.*, 107:183–187, 2008.

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

[34] Marco Vieira and Henrique Madeira. Benchmarking the dependability of different oltp systems. In *Dependable Systems and Networks, Proc. of 2003 the International Conference on*, volume 0, page 305, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[35] Marco Vieira and Henrique Madeira. A dependability benchmark for oltp application environments. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 742–753. VLDB Endowment, 2003.

[36] 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.