Model-Driven Comparison of State-Machine-based and Deferred-Update Replication Schemes

Paweł T. Wojciechowski, Tadeusz Kobus, and Maciej Kokociński
Poznań University of Technology, Poland
Email: {Pawel.T.Wojciechowski,Tadeusz.Kobus,Maciej.Kokocinski}@cs.put.poznan.pl

***Abstract*—In this paper, we analyze and experimentally compare state-machine-based and deferred-update (or transactional) replication, both relying on atomic broadcast. We define a model that describes the upper and lower bounds on the execution of concurrent requests by a service replicated using either scheme. The model is parametrized by the degree of parallelism in either scheme, the number of processor cores, and the type of requests. We use our model to make a comparison with a non-replicated service, considering separately abcast- and request-execution-dominant workloads. To evaluate transactional replication experimentally, we developed Paxos STM—a novel fault-tolerant distributed software transactional memory with programming constructs for transaction creation, abort, and retry. We used JPaxos for state-machine-based replication. Both systems share the same implementation of atomic broadcast built on the Paxos algorithm. We present the results of performance evaluation of both replication schemes, and a non-replicated (thus prone to failures) service, considering various workloads. The key result of our theoretical and experimental work is that neither system is superior in all cases. We discuss these results in the paper.**

***Index Terms*—state machine replication; deferred update replication;**

I. Introduction

Replication is an important method to increase service reliability and accessibility. It means deployment of a service on several server machines, each of which may fail independently, and coordination of service replicas so that they maintain a consistent state. Each replica maintains service state in memory (and optionally in its local nonvolatile store).

We analytically and experimentally analyze two approaches to fault-tolerant replication of services (or objects). The first approach is a *replicated state machine (SM)* [1] in which a client request is executed on every server. Services must be *deterministic*: any service replica being in the same state always produces the same effect upon the same request. SM coordinates all servers, so that all requests are delivered and processed by every replica in the same order. Thus concurrent object accesses are consistent. The second approach is *transactional replication (TR)* based on *deferred update* [10] (also known as multi-primary passive replication). Programmers use atomic transactions to access critical objects; such objects are replicated on every server. The transaction's atomicity and serializability guarantee that the concurrent modifications of object replicas are propagated consistently to every server. In the paper, we consider deferred update relying on *atomic broadcast (abcast)*. This technique prevents deadlocks and allows better scalability than using two-phase commitment since transactions are never blocked [2], [3].

The SM scheme is more general than TR since it can also be used for replication of services that require linearizability [4] while transactions in TR normally only guarantee serializability. Thus, it is important to remember that some services can be replicated using SM but not TR. Thus, our comparison is valid only for replicated services, in which concurrent requests satisfy one-copy serializability [5]. We also assume the *crash-recovery* model of failure, in which $\lceil N/2 \rceil - 1$ server crashes can be tolerated, where N is the number of servers. A server can rejoin the system any time after crash.

The contributions of the paper are twofold. We define a model of SM and TR, and use it to analyze both replication schemes. Our model describes the upper and lower bounds on the execution of concurrent requests parametrized by the degree of parallelism, the number of processor cores, and the types of requests. Next, we present and discuss the results of the SM and TR performance evaluation, obtained using two popular microbenchmarks. To our best knowledge, this is the first experimental comparison of the two replication schemes and a non-replicated service in a uniform environment (same abcast), under varying contention loads and requests sizes.

To facilitate experimentation, we developed *Paxos STM—Distributed Software Transactional Memory (DSTM)* for TR replication that has programming constructs for transaction creation, abort, retry, and for annotating Java objects as transactional. DSTM systems are a follow-up of the work on STM—a concurrent programming mechanism intended to replace lock-based synchronization (see [6], [7] among others). In STM, an atomic transaction is any piece of code containing memory reads and writes that must be executed atomically. To avoid blocking, STM systems use optimistic concurrency control—a transaction that conflicts with another concurrent transaction is rolled back and retried. DSTM is essentially like STM but transactional memory is replicated on other network nodes. Contrary to the majority of existing DSTM systems, Paxos STM implements the deferred update replication scheme using a transaction certification protocol which is based on atomic broadcast.

In the example code below, which has been taken verbatim from the executable source, a transaction is created that accesses two objects `accA` and `accB` atomically:

```
@TransactionObject
class Account { ... }
new Transaction() {
```

```
  public void atomic() {
    float amount = 100;
    if (accA.balance() >= amount)
      accA.withdraw(amount);
    else
      retry();
    accB.deposit(amount)
  }
};
```

For objects to be accessed atomically, their class `Account` has been annotated as transactional. Paxos STM replicates such objects on other servers, coordinates the execution of concurrent transactions preserving isolation (one-copy serializability) and maintains a consistent view of object replicas on every server despite server failures. If `retry` is executed, a transaction is rolled back and reexecuted.

To experimentally evaluate SM replication, we used JPaxos [8]—a state-machine-based replication library, implementing Paxos [9] for replica (server) coordination. Paxos STM reuses this protocol code for agreement coordination. Thus, we are able to fairly compare the results of benchmarks that we implemented using both tools. The analytical model helped us to understand and interpret our experimental results precisely.

A. Motivations and results

The motivations to do this research were twofold. Firstly, to our best knowledge there was no prior work on rigorous evaluation and comparison of the SM and TR replication schemes, including estimation of the upper and lower time bounds. Secondly, reasoning about advantages, limitations and possible optimization paths of both replication schemes is difficult without a performance model that abstracts away from any uninteresting details. Although the *modus operandi* of SM and TR may appear simple, concurrency, transaction conflicts, and dealing separately with read-only and read-write (update) requests make the model quite subtle.

The main contributions of our work are the following:

- We designed and implemented Paxos STM, a programming tool for TR replication of services;
- We defined a model of SM and TR that describes the upper and lower bounds on processing a set of concurrent requests (assuming no delays); the lower bound is given for unoptimized and optimized schemes, where the latter recognizes read-only requests and treats them differently;
- Our model shows precisely the potential benefits of various means to increase parallelism in SM and TR and so also to increase throughput, such as optimized abcast, dealing with read-only requests differently, detecting conflicts earlier and fully using multi-core CPUs;
- We have shown when concurrent processing of requests by SM and TR can be faster than their sequential execution, considering (in TR) upper bounds on the number of conflicts that cause transactions to be reexecuted;
- We examined throughput and scalability using two microbenchmarks (Hashmap and Bank) and compared SM and TR; the comparison is fair since JPaxos and Paxos STM share the same implementation of abcast.

Both JPaxos and Paxos STM support the crash-recovery model of failure, which means that a server replica can recover after a crash and catch up on the current state automatically (from a local disk and/or other replicas). However, we do not show evaluation results under faulty behavior scenarios in this paper since our focus is on modeling and comparing normal behaviour of both systems when no failures occur. We leave examining the faulty behaviour for future work.

B. Paper structure

The paper has the following structure. Firstly, we define the analytical model of state machine and deferred update replication in §II. Next, we show the results of our evaluation experiments in §III, comparing performance and scalability of the two replication schemes. Then, we discuss related work in §IV. Finally, we conclude in §V.

II. Analytical Model

In both SM and TR replication protocols, we can identify some of the following 5 phases [10], each of which takes the amount of time given in brackets:

1) Client request (q)
2) Server coordination (sc)
3) Local execution (e)
4) Agreement coordination (ac)
5) Client response or answer (a)

In our model, each client request consists of a sequence of operations to be executed atomically, that can read or modify the database (i.e. objects). We call a client request a "transaction" both in the TR and SM scheme. A replicated service (or database) satisfies *one-copy serializability (1SR)* [5]: transactions performed on the replicas have an ordering which is equivalent to an ordering obtained when the transactions are performed sequentially in a single centralized database.

We can identify three types of requests: *read-only*, *write-only* and *update*, denoted respectively, r, w and rw. The former two describe requests that contain respectively, only read or only write operations to be executed by a replicated service, while the last contains at least one write and read.

For the model to be tractable, we consider a replicated system in which the time e of processing a request locally by either TR or SM is the same for all requests; the same holds for q and a. Thus, we can estimate the total time T of processing n requests by a service replicated on N machines (servers) using SM or TR to be:

$$T = \mathcal{M}(q, sc, e, ac, a) \tag{1}$$

where function $\mathcal{M}$ depends on the semantics of SM and TR and the parallelism enabled by the underlying system. For each replication scheme we estimate the upper and lower bounds on time T, that correspond to respectively, the worst and best cases when computing n concurrent requests without any delay. We consider the number of available processors (or CPU cores) per server, denoted c, and abstract away other hardware restrictions. A mean time of processing n requests is an average of the lower and upper bounds: $T' = (T_{lower} + T_{upper})/2$.

So, we get the average throughput $P = n/T'$. We use the following symbols: N is the number of servers (replicas), n is the number of requests (equal to the number of transactions), t_{abc} is the execution time of a single atomic broadcast (abcast) with a superscript r (requests) for SM and o (object read-sets and write-sets) for TM (t^r_{abc} can differ from t^o_{abc} considerably, depending on the size of requests vs. the size of read/write-sets), β is the degree of the atomic broadcast optimization ($\beta \geq 1$), where the greater β the higher throughput of atomic broadcast; $\beta = 1$ means no optimization.

Optimizations of abcast are request batching and running concurrent rounds of abcast [11]. Batching means processing of several requests by a single round of abcast. The best results are when there is a continuous stream of requests so that the protocol does not need to wait for the batch to be filled in. Running concurrent rounds of abcast can further shorten the time required to deliver requests in a total order. In our model, broadcasting a continuous stream of n requests by the optimized abcast protocol takes $\frac{n}{\beta}t_{abc}$, where β is a small number (say < 5).

A. State machine replication (SM)

We first model a non-replicated state machine, and extend it to a replicated SM executing on N servers with c cores each.

Single processing ($N = n = 1$):

$$sc = t^r_{abc} \tag{2}$$

$$ac = 0 \tag{3}$$

$$T^1_{SM} = q + sc + e + ac + a = q + t^r_{abc} + e + a \tag{4}$$

Sequential processing ($N = 1$, $n > 1$):

$$T^{1 \to n}_{SM} = nT^1_{SM} \tag{5}$$

Parallel processing ($N > 1$, $n > 1$): In a system replicated on N servers using SM:

- all requests must be executed by all servers,
- all requests are executed by each server sequentially,
- server coordination (atomic broadcast) and execution of requests can occur in parallel,
- a client can submit request to any non-faulty server, and the current leader of the Paxos protocol will broadcast the message.

The above assumptions hold in JPaxos [8]—our reference implementation of SM. An optimized (rare in practice) variant of SM could recognize requests that are read-only and process them differently, replacing the first two assumptions by:

- all update requests must be executed by all servers,
- all update requests are executed by each server sequentially (i.e. processing is single-threaded),
- read-only requests can be executed by one server only, in parallel with any other requests, and $t^r_{abc} = 0$.

Consider a sequence of three client requests $r(o)$, $w(o)$, and $r(o)$, each containing only a single operation either reading (r) or updating (w) a replicated object o. The unoptimized SM guarantees that the last read will always see the object o modified by $w(o)$. In the optimized SM, read-only requests are not ordered, so the last read may not see the update of object o. To solve this problem additional machinery is required (see e.g., [9]), which we neglect in our model since without this the 1SR property is still guaranteed: the effect can be equivalent to a sequential execution $r(o)$, $r(o)$, and $w(o)$.

Below we compute the upper and lower bound on the total time of processing n requests by SM. The lower bound is computed both for the unoptimized and optimized SM.

1) Upper bound: In the worst case there is no concurrency, which means sequential execution. Thus, we have:

$$T^{1|..|n}_{SM_{upper}} = nT^1_{SM} \tag{6}$$

2) Lower bound (unopt. SM): In the best case there is as much concurrency as possible, i.e. server coordination (abcast) and the execution of requests occur in parallel. Concurrently the client requests and responses are communicated but these times are relatively short. Thus, the total time of processing n requests by an unoptimized SM has only two outcomes, depending on which of the parallel parts will take more time. We model this choice using a function max.

$$T^{1|..|n}_{SM_{lower}} = T^1_{SM} + max(\frac{n}{\beta}t^r_{abc}, ne) - \delta_{SM} \tag{7}$$

where

$$\delta_{SM} = \begin{cases} t^r_{abc} & \text{if } max(\frac{n}{\beta}t^r_{abc}, ne) = \frac{n}{\beta}t^r_{abc} \\ e & \text{if } max(\frac{n}{\beta}t^r_{abc}, ne) = ne \end{cases}$$

The interpretation of the above two cases is as follows. In the first case, abcast time is dominant, i.e. requests are short. In the second case, request processing time is dominant, i.e. requests are long. If $\frac{n}{\beta}t^r_{abc} = ne$, max returns either value.

3) Lower bound (opt. SM): In the optimized SM, read-only requests can be processed by any non-faulty replica in parallel with any other requests. However, parallelism is limited by the fact that each of the N servers (replicas) has only c processors (or processor cores), where $c \geq 1$.

Thus, the best case for servers with one processor core is:

$$\begin{aligned} T^{1|..|n}_{SM^{opt}_{lower}} &= max\left(\frac{n_{rw}}{\beta}t^r_{abc}, (n_{rw} + \left\lceil \frac{n_r}{Nc} \right\rceil)e - \pi\right) \\ &+ T^1_{SM} - \delta_{SM}, \text{ where } c = 1 \text{ and } \pi \in \langle 0, min(t^r_{abc}, e)\rangle \end{aligned} \tag{8}$$

The request processing is dominant if either requests are long or they are short but many read-only requests are among them. π describes any forward shift in a schedule caused by read-only requests ($\pi = 0$ if $n_r = 0$).

If servers have many processors (or processor cores) then:

$$\begin{aligned} T^{1|..|n}_{SM^{opt}_{lower}} &\approx max\left(\frac{n_{rw}}{\beta}t^r_{abc}, n_{rw}e, \left\lceil \frac{n_r}{N(c-\theta)} \right\rceil e\right) \\ &+ T^1_{SM} - \delta_{SM}, \text{ where } c > 1 \text{ and } \theta \in \{0, 1\} \end{aligned} \tag{9}$$

where n_{rw} and n_r denote correspondingly, the number of update and read-only requests, and δ_{SM} is equal respectively, t^r_{abc} (or 0) if the first (or second) compound of max is the largest, and e otherwise. When no parallel update requests are present, read-only requests can be executed on c cores instead

of $c-1$. We use θ to model this choice. More precisely, the third compound of max should be $\lceil \frac{n'_r}{Nc} + \frac{n''_r}{N(c-1)} \rceil e$, where $n'_r + n''_r = n_r$. However, we prefer the less precise approximation to avoid cluttering the model with additional parameters.

Below we give the main results for the optimized SM. The proofs of lemmas are available in the technical report [12].

Lemma 1: The speedup of processing requests by the optimized SM when compared to the unoptimized SM is proportional to the number of read-only requests, and—for abcast dominant processing—inversely proportional to β.

Lemma 2: In the best case, a service replicated on N single-core processor servers ($N > 1$) using the SM scheme is not slower than the non-replicated service if at least one request is read-only and

$$n_{rw} + \left\lceil \frac{n_r}{N} \right\rceil - \frac{\pi}{e} \leq \frac{n_{rw} t^r_{abc}}{\beta e} \leq n - 1 \tag{10}$$

when abcast is dominant, and

$$\left(\frac{n_{rw}}{\beta} + 1 \right) t^r_{abc} \leq \left\lceil \frac{n_r}{N} \right\rceil e - \pi \leq ne \tag{11}$$

when request execution is dominant.

Note that if $N = 1$ or $n_r = 0$, then the equation (10) is false, which is intuitively valid since the overhead of replication must slow down the replicated service if no parallelism is possible. Also, if $n_r = 0$, then the equation (11) is false ($t^r_{abc} > 0$).

B. Transactional replication (TR)

In TR, each request is processed as a single *atomic transaction* that can read and write a set of objects atomically. All transaction objects are replicated on every server. TR maintains one-copy serializability of distributed object accesses. Concurrent transactions are executed optimistically (objects are not locked) and may conflict. An update transaction (or request) x *conflicts* with some concurrent transaction y that is about to commit, resulting in x being rolled back and reexecuted, if x reads any object modified by y. We call the former transaction *conflicting* and the latter one *committing*.

We denote K to be the number of conflicts while processing n requests by TR ($K \geq 0$). Note that K is also the number of transaction (or request) reexecutions caused by conflicts. K depends on the type of requests and the intersection of objects modified by transactions (the more shared objects, the higher the probability of a conflict). By a conflict definition, write-only requests cannot conflict[1]. Since read-only requests are not causally ordered, they also do not cause conflicts, unless strict 1SR is required. K does not depend on the number of servers (replicas) N. K cannot be statically predicted since whether (or not) a conflict occurs depends on transaction interleaving at runtime. However, the upper bound can be estimated—if n update requests have been submitted by clients concurrently, the number of conflicts cannot be greater than $(n-1) + \ldots + 1 = \frac{(n-1)n}{2}$.

[1]However, in our object-oriented Paxos STM they are treated as regular rw requests, since a transaction normally modifies only a subset of object fields; the other object fields are then "read".

Instead of server coordination, TR requires agreement coordination, which is responsible for *transaction certification*: when a transaction has completed, the effects of its execution are sent to all servers (replicas) using atomic broadcast; if no conflicts with other concurrent transactions are detected locally, the effects are made permanent on every server and the transaction *commits*. Otherwise, the transaction is rolled back and reexecuted.

Thus, the server and agreement coordination times are:

$$sc = 0 \tag{12}$$

$$ac = t_{cer} + t^o_{abc} \tag{13}$$

where t_{cer} and t^o_{abc} are correspondingly, the local transaction certification time and the time of atomic broadcast, where the latter is executed on commit only. The broadcast data include object read-sets, write-sets and changes made to objects. For simplicity, we assume that all these messages have the same size for all update requests (transactions).

The agreement coordination phase also includes any other operations of the transaction processing protocol, such as creation of object shadow copies accessed by transactions. Some of these operations are executed in parallel with abcast, while the rest is assumed to be included in t_{cer}. Since in typical applications the network communication will be the bottleneck, we assume that $t_{cer} << t^o_{abc}$.

Single processing ($N = n = 1$):

$$T^1_{TR} = q+0+e+t^o_{abc}+t_{cer}+a = q+t^o_{abc}+e+a+t_{cer} \ . \tag{14}$$

Sequential processing ($N = 1, n > 1$):

$$T^{1 \to n}_{TR} = n T^1_{TR} \ . \tag{15}$$

Note that if $t^o_{abc} = t^r_{abc}$ then we obtain $T^1_{TR} = T^1_{SM} + t_{cer}$ and $T^{1 \to n}_{TR} = n T^1_{SM} + n t_{cer}$.

In single and sequential processing, there are obviously no concurrent transactions, so conflicts cannot occur.

Parallel processing ($N > 1, n > 1$): In a system replicated on N servers using TR:

- a client can submit request to any non-faulty server,
- each request (transaction) is executed by one server only and any object modifications are consistently applied to object replicas on all servers,
- requests can be executed in parallel,
- each server is multi-threaded and can execute its requests (transactions) concurrently under optimistic concurrency control scheme (no blocking).

Paxos STM that we developed for experimental validation allows an optimized variant of TR (which is common for TR):

- read-only requests do not need agreement coordination,
- conflicts are detected as soon as possible, so a conflicting transaction, can be aborted before completion, giving the execution time less than e and $t^o_{abc} = 0$ (but for simplicity, we use e to describe such cases),
- a conflict can also be detected after a transaction completes but before its effects are broadcast, causing an abort of the committing transaction; then t^o_{abc} is also 0.

We say that a conflict is detected *early* to describe one of the two cases above. Conflicts can be detected early no matter if the conflicting and committing transactions are executed on the same server or not.

As before, we consider 1SR as the criterion of correctness. If consistency guarantees should reflect any causal relations between requests issued by various clients, they have to be ensured by the replicated service itself.

Below we compute the upper and lower bounds on the total time of processing n requests by TR.

1) Upper bound: We consider almost sequential execution but allow a bit of concurrency, so before a transaction commits with its effects stored, a new transaction can commence that may conflict with the committing transaction. Then the total time of processing n requests is

$$\begin{aligned} T^{1|..|n}_{TR_{upper}} &\approx n(q+e+t^o_{abc}+t_{cer}+a)+K(e+t_{cer}+t^o_{abc}) \\ &= T^{1\to n}_{TR}+K(e+t_{cer}+t^o_{abc}) \quad \text{where } 0\le K<n \end{aligned} \tag{16}$$

We approximated the almost sequential execution to sequential but admitted K conflicts ($0 \le K < n$). We assumed the worst case: a conflicting transaction x has completed and broadcast its effects to all servers as part of transaction certification. Not till then did the servers detect that there is a conflict with some other transaction that completed soon after x had commenced but not committed yet. The conflicting transaction must be rolled back and reexecuted, so the execution time is $2(e+t_{cer}+t^o_{abc})$.

2) Lower bound (unoptimized TR): In the best case, all transactions are executed by TR in parallel but in case of any conflicts, as before, the conflicting transactions must be rolled back and reexecuted. The first naîve version of the lower bound is

$$\begin{aligned} T^{1|..|n}_{TR_{lower}} \approx T^1_{TR} + max\Big(f(n')(e+t_{cer})+\frac{K'}{\beta}t^o_{abc}, \\ \frac{n+K'}{\beta}t^o_{abc}\Big)-\delta_{TR} \end{aligned} \tag{17}$$

where $f()$ is a *conflict function* which for a given number of conflicting transactions n' executed in parallel ($0 \le n' < n$) returns a factor that when multiplied by a time of processing one transaction gives the time of committing all n' transactions. Note that some of the conflicting transactions can be executed several times until they commit, so the number of conflicts K can be larger than n'. K' is the number of conflicts that are *not* detected early, so transaction effects are abcast to all servers. Once a conflict is detected the conflicting transaction is aborted but for simplicity we still use the complete times e and t_{cer} to describe its execution.

As before, *max* describes the parallel execution of abcast and local processing on nodes. However, we must also reflect the order imposed by the TR phases. Thus, the first compound of *max* describes a *sequence* of the 'execute', 'certify' (locally), and 'abcast' operations of a subset of transactions that run into conflicts, while the second compound describes the total of 'abcast' operations of all transactions; abcasts of non-conflicting transactions may occur in parallel with the local execution of retried transactions. We reduce (17) to

$$T^{1|..|n}_{TR_{lower}} \approx T^1_{TR} + max(f(n')e', \frac{n}{\beta}t^o_{abc}) + \frac{K'}{\beta}t^o_{abc} - \delta_{TR} \tag{18}$$

where $e' = e + t_{cer}$

$$\delta_{TR} = \begin{cases} 0 & \text{if } max(f(n')e', \frac{n}{\beta}t^o_{abc}) = f(n')e' \\ t^o_{abc} & \text{if } max(f(n')e', \frac{n}{\beta}t^o_{abc}) = \frac{n}{\beta}t^o_{abc} \end{cases}$$

The interpretation of the above conditional is as follows. In the first case, request execution is dominant, i.e. either transactions are relatively long or many conflicts occur that are detected early (which means no abcast is required). In the second case, atomic broadcast is dominant, i.e. transactions are relatively short and there are few conflicts.

The more concurrent accesses of shared objects occur and the more parallelism is allowed, the more probable the conflicts are. The actual number of conflicts K and a value $f(n')$ depend on the intersection of objects shared by transactions and their runtime interleaving. In the report [12], we defined a conflict function $f()$ for a few special cases, and computed the upper/lower bounds for the worst case ($f(n') = n'$).

3) Lower bound (optimized TR): In our estimation of TR's lower bound, we neglected hardware restrictions and assumed an unlimited number of processors. Below we approximate the lower time bound of processing n concurrent requests by a service replicated in a system of N servers with c processor cores each, and consider the optimized TR in which read-only requests do not require the agreement coordination phase:

$$\begin{aligned} T^{1|..|n}_{TR^{opt}_{lower}} &\approx T^1_{TR} + max\Big(\Big\lceil\frac{n+K}{Nc}\Big\rceil e' + \Sigma, \frac{n_{rw}+K'}{\beta}t^o_{abc}\Big) + \\ &- \delta_{TR} \text{ where } \Sigma = \sigma e' + \Big\lfloor\frac{\sigma}{x}\Big\rfloor t^o_{abc} \text{ and } \sigma \in \langle 0, f_H(Nc-1)\rangle \end{aligned} \tag{19}$$

where δ_{TR} is equal to e' if the first compound of max is the larger, or t^o_{abc} otherwise. The conflict function $f_H()$ is defined as before but the function domain is $\langle 0, f_H(Nc)\rangle$, which means that at the same time there cannot be more than Nc concurrent conflicting transactions. $\Sigma = \sigma e' + \lfloor\frac{\sigma}{x}\rfloor t^o_{abc}$ describes the offset caused by the conflicting transactions that have not managed to be executed in parallel with non-conflicting transactions (see Fig. 1), where the upper bound on σ is equal $f_H(Nc-1)$ and x ($0 < x \le \sigma+1$) reflects the fact that some of the conflicts can be detected early and no abcast is used.

Below we give the main results for the optimized TR:

Lemma 3: A service replicated on N servers, each having c-processor cores, using the optimized TR runs $n_r\frac{t^o_{abc}}{\beta}$ faster than when using the unoptimized TR.

Lemma 4: In the best case, a service replicated on N single-core processor servers ($N > 1$) using the TR scheme can be faster than the non-replicated service if

$$\Big\lceil\frac{n+K}{N}\Big\rceil e' + \Sigma \le \frac{n_{rw}+K'}{\beta}t^o_{abc} \le ne - e' \tag{20}$$

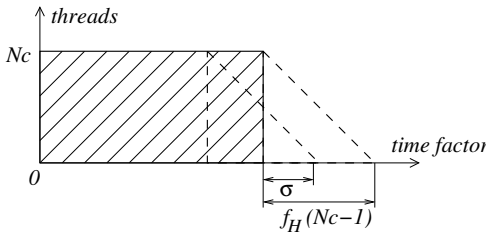

Fig. 1. Processing $n + K$ requests on Nc cores, where $K \geq Nc - 1$

when abcast is dominant, and

$$\frac{n_{rw} + K'}{\beta} t^o_{abc} \leq \left\lceil \frac{n+K}{N} \right\rceil e' + \Sigma \leq ne - t^o_{abc} \qquad (21)$$

when request execution is dominant.
The proofs of lemmas are available in technical report [12]. Note that in the equation (20) n_{rw} (and so t^o_{abc}) cannot equal 0 since we assumed abcast dominance.

The TR-replicated service is faster than a non-replicated service if the equation (21) holds. In particular, if $n_{rw} = 0$, there are no conflicts ($K = K' = \Sigma = 0$) and no abcast ($t^o_{abc} = 0$). So we have

$$0 \leq \left\lceil \frac{n_r}{N} \right\rceil e' + \Sigma \leq n_r e \qquad (22)$$

Since normally $t_{cer} << e$, the above equation is mostly true, which agrees with the intuition that a TR-replicated service is faster than a non-replicated service if there are many read-only requests that can be processed in parallel. Note that if $n_r = 1$ or $N = 1$ then the above equation is false. If $n_r \leq N$ then the equation is true only if $n_r \geq \frac{e'+\Sigma}{e}$.

III. Experimental Evaluation

In this section, we empirically evaluate and compare performance and scalability of SM- and TR-based replicated, and a non-replicated service, modelled by two benchmarks.

A. Programming tools

In order to evaluate SM-based replication, we used JPaxos [8]—an efficient implementation of the Paxos [9] algorithm, with the support of the *crash-recovery* model of failure. To boost performance, JPaxos supports concurrent rounds of consensus and request batching. Request types are not recognized so it implements the unoptimized SM model (see II-A).

To evaluate TR-based replication, we designed and implemented *Paxos STM*—an object-oriented fault-tolerant DSTM system, which replicates every shared object on each node for increased availability and minimal access latency. Paxos STM supports multi-primary passive replication (similar to multi-master replication in databases) and relies on the optimistic concurrency control scheme. Paxos STM's certification protocol is built on top of JPaxos, with each replica able to propose new updates to the distributed state by abcasting them. Paxos STM supports both the crash-stop and the crash-recovery failure models. Our system also takes advantage of modern multicore hardware by allowing multithreaded processing of transactions. It distinguishes between read-only and updating transactions, thus supporting the optimized version of TR replication (see II-B). By the use of the multiversioning scheme read-only transactions are guaranteed to commit successfully.

Operation (Transaction)	Default	Prolonged	High-Contention
`get` (RO)	100	100	100
`get` (RW)	8	8	40
`put/remove` (RW)	2	2	10
"active wait" (RO+RW)		1ms	

Fig. 2. Input parameters for the Hashtable benchmark

B. Benchmarks

To evaluate SM and TR replication schemes under different workloads and compare with a non-replicated run, we implemented two popular microbenchmarks: Hashtable and Bank.

1) Hashtable microbenchmark: The hashtable of size n stores key-value integer elements and manages them through *get/put/remove* operations. It is prepopulated with $n/2$ random elements from a defined range, thus giving the saturation of 50%. A single run consists of a series of requests issued to the hashtable. There are two types of requests (or transactions): *read-only (RO)* and *read-write (RW)*, which correspond to requests r and rw in Section II. The RO request atomically performs a given number of *get* operations with a randomly chosen set of keys. The RW request executes a defined number of *get* operations followed by updating operations (either *put* or *remove*). To keep the hashtable 50% saturated, the decision whether to insert a new object to the hashtable or remove an existing one depends on the previous *get* operations.

The benchmark parameters in Fig. 2 reflect different kinds of workload: Default, Prolonged, and High-Contention. The RO transactions scan through a vast amount of data using many *get* operations. In contrary, RW transactions involve much fewer operations (two to ten times less, depending on the test), 20% of which are modifying ones. Different levels of contention can be generated by manipulating the size (or the number of involved operations) of the RW transactions and the relative number of RO and RW transactions. In the evaluation we use the same sizes of RO transactions for all tests and two different sizes of RW transactions. For each test, we examine three scenarios consisting of a different mix of RW and RO transactions: 10/90, 50/50, 90/10, denoted respectively: 10%, 50%, and 90% of RWs. In the Prolonged workload each request additionally performs the "active wait" for a given amount of time (1 ms) to simulate computation-heavy workloads.

2) Bank benchmark: Operations are performed on an array of accounts shared between nodes. We have two types of transactions. An RW transaction performs *transfer* of funds from one account to another, thus executing in total two read and two write operations on two distinct accounts. An RO transaction computes *balance*, which requires reading all of the accounts and summing up the funds. In our tests, we evaluate three scenarios with different percentage of RW transactions,

Benchmark service	10% RW	50% RW	90% RW
a) Default Hashtable	110398	119135	161415
b) Prolonged Hashtable	666	667	667
c) High-Contention Hashtable	103666	106608	120830
d) Bank	123183	155615	206042

Fig. 3. The results of the non-replicated benchmark execution

namely 10%, 50%, and 90%. In each scenario, the number of accounts is 10000.

C. Evaluation Environment

We used a cluster of eight nodes, each equipped with a Xeon Quad-core X3230 2.66GHz, L2 cache 2x4MB CPU, 4GB RAM ECC DDR2, 800MHz, running OpenSUSE 10.3 (kernel 2.6.22.19) with Sun JRE 1.6.0. The nodes are connected via a private 1Gb Ethernet network.

JPaxos was configured to have at most two concurrent instances of consensus, the maximum batch size 64KB, and no batching delay. All available CPU cores were utilized, so the c parameter that reflects the number of physical on-board cores (see II) is four. We experimentally established an optimal number of worker threads in Paxos STM to be 20 for the Hashtable benchmark and 80 for the Bank benchmark (these values were used in all of our tests). Such a high number of threads (far exceeding the number of physical cores) is necessary to fully exercise the hardware potential due to threads blocking on network I/O operations.

D. Evaluation results and analysis

Below we discuss the results of benchmark tests. In Fig. 4, we present *throughput* obtained using JPaxos and Paxos STM, i.e. the number of transactions committed per second. We also present the transaction *abort rate* (in Paxos STM), i.e. the percentage of transactions aborted due to conflicts and reexecuted (equal $\frac{K}{n+K}100\%$ in our model in §II); the abort rate gives useful insight into the level of contention.

1) Default Hashtable: Hashtable with default configuration executed under JPaxos on two nodes, touches the score of 25000 requests per second (req/sec) (see Fig. 4-a). With the number of nodes increasing the performance gradually decreases stabilising at the level of around 17000 req/sec. This drop results from higher coordination costs of maintaining a higher number of replicas that the Paxos leader replica must handle. The differences among scenarios including a various mix of request types are minimal. However, slightly (7-11%) better results are obtained in scenarios including more read-write (RW) requests. The larger size of read-only (RO) requests adds to the execution time, as well to the abcast time since more data needs to be exchanged between nodes.

Results of Paxos STM evaluation show more differences between various scenarios. Conversely to JPaxos, better results are obtained with a higher percentage of read-only requests, which do not require the server agreement phase and therefore the costly abcast operation. The results in scenarios including 90% and 50% of read-write requests resemble the ones obtained with JPaxos, but scaled up by a certain factor. Paxos STM 50% scenario performance is roughly twice the JPaxos'. One can observe that the best performance is obtained with a small number of nodes and it falls with the increasing number of nodes. The 90% scenario exhibits different characteristics. It scales up with the number of nodes. It is the result of Paxos STM's ability to process read-only requests (which are almost an order of magnitude more frequent than in other scenarios) in a fully parallel manner with no communication overhead. The higher number of nodes the more requests can be processed. The top performance obtained for a maximum number of nodes reaches almost 150000 requests per second. In the other scenarios the predominant cost of abcast does not allow to fully exercise the potential level of parallelism. The abort rate in all scenarios is moderate and ranges from 0 to 12%, depending on the scenario.

The results obtained using a non-replicated Default Hashtable greatly surpass the results of both JPaxos and Paxos STM (see Fig. 3). This immense throughput of the latter is, however, achieved at the cost of absolutely no fault tolerance. Note that in the 10% RW scenario, Default Hash-table replicated using Paxos STM outperforms its non-replicated, non-fault-torerant variant if there are more than 4 replicas.

2) Prolonged Hashtable: Contrary to the first test, where the cost of abcast in SM as well as TR was predominant, the second test aims at mimicking a computation-heavy workload (which corresponds to the request processing time dominance in §II). The parameters of this test differ only in one aspect compared to the Default Hashtable configuration—the execution of each request is prolonged by 1 ms.

JPaxos's evaluation (see Fig. 4-b) show stunningly uniform performance of 675 requests per second regardless of the number of nodes involved in computation. It indicates that a high execution time of requests entirely covers up any cost of replica coordination. The system throughput is directly limited by the time needed by replicas to actually execute the requests.

On the other hand, Paxos STM exhibits excellent scaling capabilities. Performance increases with the number of nodes. For the 10% RW scenario, it does so almost linearly. In other scenarios the agreement coordination phase required by (more frequent in these cases) RW requests introduces a slight overhead. The fall of performance in case of 50% RW and 90% RW scenarios is small up to 3-4 replicas and slightly raises for a higher number of nodes. Performance achieved for the minimal number of nodes is almost four times higher than in case of JPaxos since Paxos STM takes advantage of the multicore hardware architecture. The abort rate is nearly identical to the one from the previous test.

The throughput of a non-replicated Prolonged Hashtable is similar to JPaxos and limited by the request execution time. Now, the performance of the non-replicated service cannot be matched to Paxos STM which is superior this time (see Fig. 3). This result is justified by Lemma 4-(21), showing precisely when TR can be faster for the execution-dominant workload.

3) High-Contention Hashtable: This benchmark test aims at examining both replication approaches under high contention. For this, the number of read and write operations in

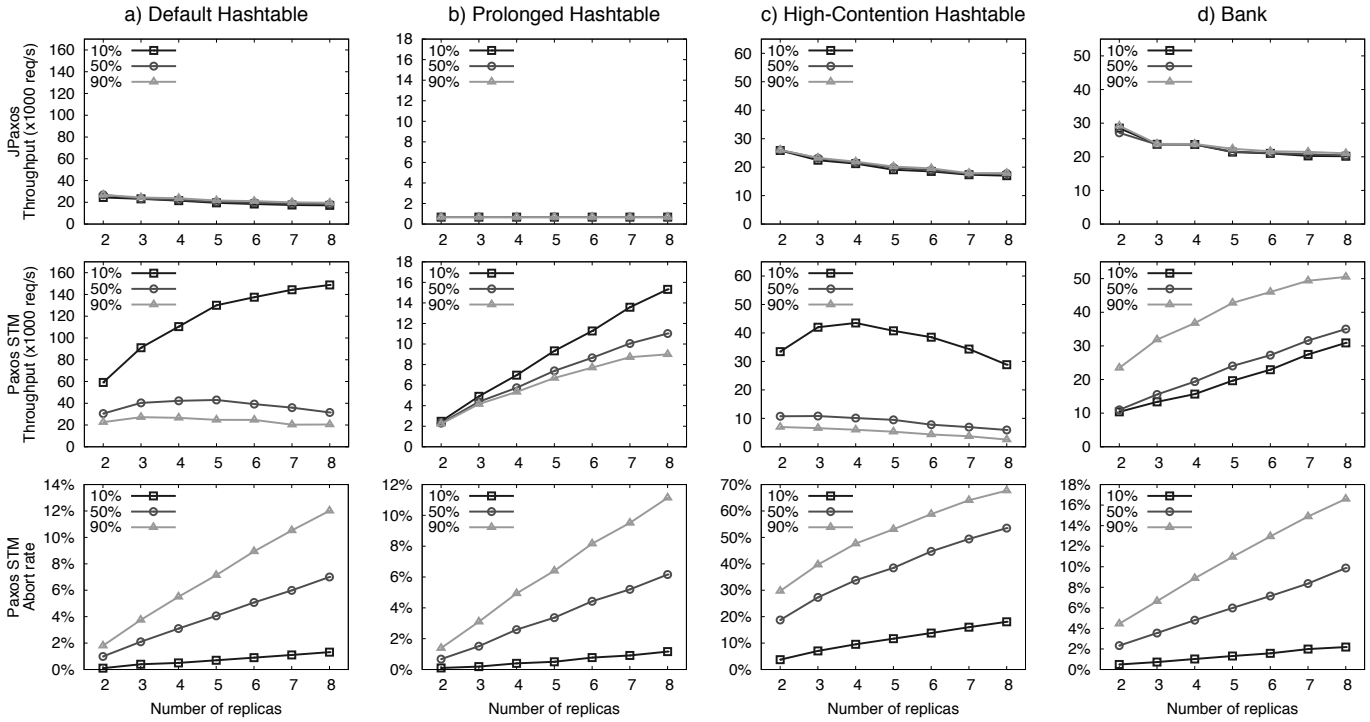

Fig. 4. Benchmarks, where 10%, 50%, and 90% denote the percentage of read-write (RW) requests (or transactions).

RW requests grew 5 times compared to the Default Hashtable benchmark configuration. The performance of JPaxos is very similar to the one obtained from the first test (see Fig. 4-c). It is due to the requests being executed by JPaxos sequentially, thus not conflicting with each other. The main difference is the lower performance of scenarios involving more RW requests. Their performance is now closer to the performance of the read-only dominated scenario, because the lengths of RO and RW requests are now comparable.

The change of contention level has a much more visible impact on Paxos STM. The abort rate now reaches up to 20% for the 10% RW scenario (20 times more), 50% for 50% RW scenario and, 70% for 90% RW scenario. This means that in case of the highest contention level almost every RW transaction is aborted at least once. The impact of such a high abort rate can be easily observed in the throughput diagram. The 50% RW and 90% RW scenarios, which are the most affected by the contention increase, demonstrate the max. four times throughput downfall. The throughput is diminished even more for a higher number of nodes, where this decrease is even larger—up to eight times. In the 10% RW case, the throughput is roughly halved for a small number of nodes, and then falls drastically when the number of nodes increases. Note that only RW transactions may be aborted since RO transactions are guaranteed to commit successfully. If 90% of transactions are guaranteed to succeed then, in case of the 20% abort rate, the rest of transactions (all RW transactions) are aborted twice on average. The higher is the number of RW transactions performed (including the aborted re-runs), the more this scenario resembles scenarios with a higher base percentage of RW transactions, so its performance is decreased.

Higher contention also has an impact on the execution of a non-replicated High-Contention Hashtable service (see Fig. 3). The lower performance can be attributed to higher packet processing overhead and the execution time of RW requests.

4) Bank Benchmark: In this test (see Fig. 4-d), the JPaxos throughput ranges from nearly 30000 to slightly above 20000, similarly as in the case of Hashtable. There is no observable difference between various scenarios. Even though RO requests consist of a huge number of operations the increased execution time has no significant impact on the overall throughput. The factor that has the biggest impact on the throughput is the abcast cost, which is similar for both types of requests. Note that although the RO request requires a large number of operations to be performed (all the accounts are scanned), the amount of data being sent is limited to a single word which depicts the type of the request.

In Paxos STM, the observed tendencies are different than before. Contrary to the previous tests, where the scenarios including more RO requests performed better, now the opposite is true. The best results were obtained for 90% RW scenario. This result can be explained by a higher execution time of RO requests that is even more important in the TR approach due to transactional processing overhead. Although RO requests have higher execution times, they scale well with the number of nodes, allowing Paxos STM to improve its performance from 10000 requests per second for two nodes up to 30000 for eight nodes in case of 10% RW scenario, and up to 35000 in case of 50% RW scenario. While the performance of Paxos STM

increases with the number of nodes, the performance of JPaxos lowers slightly. In the 10% RW and 50% RW scenarios, both tools obtain roughly equal throughput on 5 nodes. With the lower number of nodes, JPaxos exhibits better performance, while with 5 and more nodes Paxos STM is superior.

In case of 90% RW scenario one can notice exceptionally good behaviour of Paxos STM. Usually in an abcast dominated workload the speed of Paxos STM is limited by the speed of abcast. Clearly, in this case, it is the opposite. It is the result of a number of effects occurring simultaneously: very small requests (each 76 bytes), extremely short transaction execution times, and a high number of updating transactions performed concurrently. In such circumstances replicas might have a lot of transactions ready to commit at the same moment. In this case, the abcast protocol may broadcast them all at once using a single message. Thus, in practice, by optimizing abcast, we will be able to considerably improve performance of TR.

The non-replicated Bank service again tops all the approaches with the best throughput, ranging from 120000 to over 200000 (see Fig. 3). Yet again, this performance gain is at the expense of no support for fault-tolerance.

E. Evaluation Summary

In most cases, a non-replicated service outperforms its the SM- and TR-based replicated variant. It is however done at the expense of providing absolutely no fault tolerance. In some scenarios, however, the TR-based replicated service is the clear winner. This is justified by our theory: Lemmas 2 and 4 show precisely when SM- and TR-based replicated services can outperform its non-replicated variant, considering abcast or request dominant workloads. As expected (see II-A2), JPaxos does not scale at all and is insensitive to high-contention workloads. It performs poorly when the workload is execution dominated since all requests have to be processed sequentially. It is not the case with Paxos STM which takes the advantage of multicore hardware and allows for concurrent distributed processing on several nodes (see II-B3). This is especially visible in the case of read-only transactions. However, TR performance suffers under high contention. Abcast overhead should be reduced as much as possible since otherwise it may overshadow gains of parallelism and reduce scalability (e.g., 10% RW scenario in Fig. 4-c). On the other hand, SM outperforms TR in High-Contention Hashtable for 50% and 90% RW. Therefore one can see that no single solution would fit all purposes. However, TR-based replication holds a lot of promise.

IV. Related Work

A lot of work was done on replication in distributed systems in the past years (see [10] for a survey), and different models and replication techniques have emerged. Unfortunately, various authors often use different terms to name similar abstractions. Some models (such as state-machine replication, originally proposed in [1]) evolved considerably since their initial formulation. Below we briefly describe some of the work most closely related to ours.

Our SM model can be used to describe performance of the state-machine [1] as well as quorum-based [13] approach to replication, each relying on a distributed agreement protocol. The key idea of the former approach is processing all requests in the same order by all replicas. On the other hand, the fundamental idea of the quorum-based replication is that a transaction is executed if the majority of sites vote to execute it. Our TR model describes a variant of the primary-copy replication [14] that allows many concurrent master replicas. It is called multi-primary passive replication [10] or, in the database community, deferred update or multi-master replication; in the classification of [3] (see Chapter 12), it is an eager, update everywhere approach. As in the primary-backup replication, update transactions can only be processed on a master replica, with the updates propagated eagerly or lazily to slave replicas [3], but many concurrent master replicas are allowed.

The deferred update systems often employ pessimistic concurrency control based on the strict two-phase locking (S2PL) [15]. Contrary, our TR model describes deferred update based on atomic broadcast, which allows transactions to be executed without blocking. Several authors demonstrated advantages of using this technique to replicate databases and make them tolerant to machine crashes (see e.g., [16], [17], [18], [19]). Various optimizations of the basic scheme are possible, e.g. readsets of update transactions do not need to be broadcast if an additional communication phase is introduced to broadcast the decision regarding committing or restarting a transaction [20]. More recently, deferred update protocols tolerating Byzantine faults are also investigated (see e.g., [21]).

There exists work on analytical performance evaluation of transactional and replicated systems. But there is relatively little work on formalization of replication schemes similar to ours. Yu [22] defines an analytical model of various concurrency control schemes used in transactional processing. Ciciani *et al.* [23] describe an analytical model designed to study the tradeoff between replicating data in database systems using various pessimistic, optimistic, and semi-optimistic concurrency control schemes. However, this study does not include approaches based on group communication. Nicola and Jarke [24] propose a 2D analytical queueing model of replication for performance evaluation of distributed and replicated database systems. Jiménez-Peris *et al.* [25], analytically and experimentally compare various quorum-based data replication schemes. The authors conclude that in most cases the read-only-write-all-available approach outperforms quorum replication.

Paxos STM that we developed is a Distributed Software Transactional Memory (DSTM) system. Most of these systems extend the implementations of some non-distributed STMs with replication protocols, which are often designed *ad-hoc*, providing no fault-tolerance and depending on a central coordinator. In contrast to such systems, we designed Paxos STM from the ground up as a fault-tolerant distributed STM.

DiSTM [26] is an object-level DSTM implementing several coherence protocols. Serialization of concurrent transactions is ensured either by a distributed mutual exclusion algorithm, or by a lease mechanism. Leases are managed by a designated

machine, which can be a bottleneck under high load. Anaconda [27] alleviates some of the DiSTM shortcomings by extending it with distributed object replication, caching mechanisms, and a new three-phase pessimistic concurrency control protocol. However, neither DiSTM nor Anaconda provide fault tolerance. The closest system to Paxos STM is D2STM [28], which also implements an optimistic transaction certification based on atomic broadcast and multiversioning. All objects are replicated on each node, thus eliminating the problem of fetching objects from remote locations. However, D2STM is built as a local STM, extended to support replication. More recently, D2STM has been equipped with the lease-based mechanism to limit abort rate under high contention [29].

V. Conclusions and Future Work

We analyzed and experimentally compared two approaches for replication of services (or databases), both based on atomic broadcast: replicated state machine (SM) and transactional (deferred update) replication (TR). The key corollary one can draw from our analytical model is that neither solution is superior in all cases. This is due to the differences between the two approaches in sensitivity to various workloads. Execution dominated workloads are handled much better when using TR since this approach can (inherently) execute multiple requests concurrently, contrary to classical SM. In particular, TR allows higher throughput than SM for read-write requests with a majority of read operations that do not cause conflicts (which is a typical workload of web services). However, performance gains from parallel request execution may be overshadowed by high costs of atomic broadcast, which is especially visible in the abcast-dominated workloads. The predictions given by our model are supported by the results of evaluation. For our experimental evaluation, we have used JPaxos (SM) and developed Paxos STM (TR). The tools are based on the same implementation of the MultiPaxos algorithm, thus ensuring fairness of the comparison. Since only TR exercises the ability to scale, one would expect it to perform better than SM. However, the results show that sometimes the overhead of transactional machinery makes SM a better choice. One can also observe the high footprint of using either replication scheme compared to the performance of a non-replicated (thus prone to failures) variant. However, the fault-tolerance is worth the price. Moreover, the costs of expensive inter-node communication can be partially compensated by parallel request execution in TR. In workloads that exhibit high request execution times this may even result in much higher performance of TR compared to a non-replicated service. To conclude, when considering replication as a mean of providing fault tolerance one should carefully choose one or the other solution based on the expected workload. In the future, we would like to compare TR and SM under faulty scenarios, using different protocols for recovery after crashes which are already supported by JPaxos and Paxos STM. Our comparison of SM and TR schemes is valid for services that require 1SR only. It may also be interesting to design TR with support of linearizability, and repeat the comparison.

Acknowledgments The authors would like to thank Jan Kończak, Nuno Santos, Tomasz Żurkowski, and André Schiper for their work on the implementation of JPaxos.

References

[1] F. B. Schneider, *Replication management using the state-machine approach*. ACM Press/Addison-Wesley, 1993, pp. 169–197.
[2] A. Schiper and M. Raynal, "From group communication to transactions in distributed systems," *Communications of the ACM*, vol. 39, Apr. 1996.
[3] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. of SIGMOD '96*, 1996.
[4] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM TOPLAS*, vol. 12, no. 3, 1990.
[5] P. A. Bernstein and N. Goodman, "Serializability theory for replicated databases," *J. Comput. Syst. Sci.*, vol. 31, pp. 355–374, Dec. 1985.
[6] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. of PODCS '95*, Aug. 1995.
[7] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proc. of OOPSLA '03*, Oct. 2003.
[8] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, "JPaxos: State machine replication based on the Paxos protocol," Faculté I&C, EPFL, Tech. Rep. 167765, Jul. 2011.
[9] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.
[10] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*, ser. LNCS 5959. Springer, 2010.
[11] N. Santos and A. Schiper, "Tuning paxos for high-throughput with batching and pipelining," in *Proc. of ICDCN '12*, 2012.
[12] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "Model-driven comparison of state-machine-based and deferred-update replication schemes," Poznań Univ. of Technology, Tech. Rep. RA-05/12, Apr. 2012, available from http://www.cs.put.poznan.pl/pawelw/pub/TR-05/12.pdf.
[13] D. K. Gifford, "Weighted voting for replicated data," in *Proc. of SOSP '79*, Dec. 1979, pp. 150–162.
[14] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed ingres," *IEEE TSE*, vol. 5, 1979.
[15] P. A., Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.
[16] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases," in *Proc. of EuroPar '97*, Aug. 1997.
[17] B. Kemme, F. Pedone, G. Alonso, and A. Schiper, "Processing transactions over optimistic atomic broadcast protocols," in *Proc. ICDCS '99*.
[18] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-JDBC: flexible database clustering middleware," in *Proc. USENIX ATEC '04*, Jun. 2004.
[19] F. Pedone, R. Guerraoui, and A.Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, Jul. 2003.
[20] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proc. VLDB '00*, 2000.
[21] F. Pedone, N. Schiper, and J. E. Armendáriz-Iñigo, "Byzantine fault-tolerant deferred update replication," in *Proc. of LADC '11*, Dec. 2011.
[22] P. S. Yu, "Modeling and analysis of transaction processing systems," in *Performance Evaluation of Computer and Communication Systems*, ser. LNCS 729. Springer-Verlag, 1993.
[23] B. Ciciani, D. M. Dias, and P. S. Yu, "Analysis of replication in distributed database systems," *IEEE Trans. on Knowl. and Data Eng.*, vol. 2, pp. 247–261, Jun. 1990.
[24] M. Nicola and M. Jarke, "Increasing the expressiveness of analytical performance models for replicated databases," in *ICDT '99*, Jan. 1999.
[25] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme, "Are quorums an alternative for data replication?" *ACM Trans. Database Syst.*, vol. 28, pp. 257–294, Sep. 2003.
[26] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson, "DiSTM: A software transactional memory framework for clusters," in *Proc. of ICPP '08*, Sep. 2008.
[27] C. Kotselidis, M. Lujan, M. Ansari, K. Malakasis, B. Kahn, C. Kirkham, and I. Watson, "Clustering JVMs with software transactional memory support," in *Proc. IPDPS '10*, 2010.
[28] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable Distributed Software Transactional Memory," in *Proc. of PRDC '09*, Nov. 2009.
[29] N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous lease-based replication of software transactional memory," in *Proc. of Middleware '10*, ser. LNCS 6452, 2010.