Hybrid Replication: State-Machine-based and Deferred-Update Replication Schemes Combined

Tadeusz Kobus, Maciej Kokociński, and Paweł T. Wojciechowski
Institute of Computing Science
Poznań University of Technology
60-965 Poznań, Poland
Email: {Tadeusz.Kobus,Maciej.Kokocinski,Pawel.T.Wojciechowski}@cs.put.edu.pl

***Abstract*—We propose a novel algorithm for *hybrid transactional replication (HTR)* of highly dependable services. It combines two schemes: a transaction is executed either optimistically by only one service replica in the deferred update mode (DU), or deterministically by all replicas in the state machine mode (SM); the choice is made by an oracle. The DU mode allows for parallelism and thus takes advantage of multicore hardware. In contrast to DU, the SM mode guarantees abort-free execution, so it is suitable for irrevocable operations and transactions generating high contention. For expressiveness, transactions can be discarded or retried on demand. We developed HTR-enabled Paxos STM, an object-based distributed transactional memory system, and evaluated it using several benchmarks: Bank, Distributed STMBench7, and Twitter Clone. We tested our system under various workloads and three oracle types: DU and SM, which execute all transactions in one mode, and Hybrid—tailored specifically for each benchmark—which selects a mode for each transaction dynamically based on various parameters. In all our tests, the Hybrid oracle is not worse than DU and SM and outperforms them when the number of replicas grows.**

***Index Terms*—state machine replication; transactional replication; deferred update; distributed transactional memory**

I. Introduction

Replication is an established method to increase service accessibility and dependability. It means deployment of a service on multiple machines and coordination of their actions so that a consistent state is maintained across all the service replicas. In case of a (partial) system failure operational replicas continue to provide the service.

We consider two basic models of service replication: *state-machine replication (SM)* and *transactional replication (TR)*. In the pessimistic SM approach [1], each client request is first ordered among all service replicas and then processed by each replica independently. Given that the service is deterministic and all requests are executed in the same order (sequentially) on all replicas, all the replicas are in a consistent state. The total order is achieved using fully distributed, fault-tolerant protocols for distributed agreement. TR corresponds to multi-primary passive replication (also known in the database community as multi-master replication) [2]. In this optimistic approach, no replica coordination is required prior or during request execution—each request is handled by only one replica using a fresh *atomic transaction*. The transaction can run in parallel with any other transactions. However, a transaction that *conflicts* with a concurrent transaction is forced to abort (revoke all the changes performed so far) and is restarted. A conflict occurs when a transaction reads data modified by an older transaction that runs concurrently. We consider TR that employs the *deferred-update (DU)* replication scheme which relies on the atomic broadcast (abcast) primitive to propagate transaction updates between replicas, and alleviates the need for atomic commitment (see [3], [4], [5]). Transaction commit is based on a certification test. The use of abcast has advantages compared to atomic commitment, such as deadlock avoidance (as explained e.g., in [5] among others).

The programming model of TR matches *distributed transactional memory (DTM)*—an extension of *transactional memory (TM)* [6] [7] to distributed systems. Programmers can use language constructs to declare the request processing code (or other concurrent code to be executed atomically) as an atomic transaction. Note that the code executed upon requests in SM may also be perceived as (serialized) transactions. However, SM guarantees sequential execution of transactions (requests), so isolation is provided trivially. Moreover, the SM model is more general than TR since it can also be used for replication of services that require linearizability [8]. TR can normally only guarantee serializability.

In our previous work [9], we analytically and experimentally compared the SM and TR replication models. Our results show that, surprisingly, there is no clear winner—each approach has its advantages and drawbacks, and various factors such as a workload type, parallelism on multicore CPUs, and network congestion have significant impact on performance of the SM and DU replication schemes. This insight has led us to an idea of combining SM and DU into a *hybrid transactional replication (HTR)* model, where both replication schemes may be used interchangeably on a per-request (or transaction) basis. In this way, we aim to achieve increased performance and more flexible semantics. Some requests are better performed in the state machine (SM) mode, especially if they access many objects, result in large updates, or cause many conflicts (e.g., resizing and rehashing a hashtable). On the other hand, other requests that can be easily executed concurrently benefit from execution in the deferred update (DU) mode.

In this paper, we introduce the *HTR algorithm* in which the DU and SM modes co-exist and are selected dynamically per transaction execution instance by an application-specific oracle. The oracle constantly monitors the system to determine which mode is optimal for a particular run of a transaction. Among the data gathered by oracles are the duration of transaction execution, the latency of atomic broadcast, the size of messages, network congestion, and the system load.

In addition to efficiency gains, HTR has another advantage. The fully-optimistic TR scheme imposes some limitations on transaction semantics. Since a transaction may be forced to restart in case of a conflict, it must not execute any irrevocable operations, that is, operations whose effects cannot be rolled back (e.g., local system calls). The problem of transactions which contain irrevocable operations has been extensively studied in the context of non-distributed TM systems (see e.g., [10] among others). However, none of the proposed solutions can be easily adapted to the distributed environment (and replication). To our best knowledge there was no prior work on irrevocable operations in the context of DTM. In the HTR algorithm, transactions with irrevocable operations are simply executed in the SM mode which ensures abort-free execution of transactions. Our algorithm satisfies opacity [11].

To evaluate our ideas, we extended Paxos STM [9], our optimistic distributed transactional memory system, with the HTR algorithm presented in this paper. Paxos STM replicates all *transactional objects* (objects shared by transactions) and maintains strong consistency of object replicas. Transactions are executed atomically and in isolation despite system failures, such as server crashes; the crashed servers can be recovered. For expressiveness, transactions can be rolled back or retried on demand. The former operation revokes all the changes performed so far by a transaction and resumes the code after the transaction, while the latter rolls back and reexecutes a transaction. The transaction *retry* construct can be used in programming idioms such as suspending the execution until a given condition is met.

In the paper, we describe the results of the experimental evaluation of the HTR algorithm using three benchmarks: Bank, a distributed version of STMBench7, and a real world application (a custom implementation of the Twitter social networking service). We examine various oracles and observe how they influence the system performance and network congestion under varying workloads. The results indicate that in all cases an application can benefit from the HTR scheme.

A. Motivations and contributions

The motivations to conduct this research were threefold. Firstly, as our previous work [9] showed that neither the state-machine-based nor deferred-update replication scheme was superior, we were eager to combine these two into a single algorithm to bring together the best of both worlds. Secondly, we are not aware of any prior research on applying transactional semantics to state machine replication for increased expressiveness. Contrary to pure SM replication, we achieve greater expressiveness by incorporating the *rollback* and *retry* constructs—they enable revoking changes performed by a request and restarting the execution of a transaction if required. Thirdly, to our best knowledge our research is the first on irrevocable actions in DTM.

The main contributions of the paper are as follows:

- We proposed hybrid transactional replication (HTR) and designed a novel HTR algorithm which combines state-machine–based and deferred-update replication schemes for better performance, scalability, and improved code expressiveness; the algorithm leverages transactional semantics and provides opacity as a consistency criterion;
- We developed HTR-enabled Paxos STM, a tool for hybrid transactional replication of services;
- We evaluated throughput and scalability of HTR-enabled Paxos STM under various workloads using three benchmarks: Bank, Distributed STMBench7, and Twitter Clone—a custom implementation of the Twitter social networking service. We examined extreme cases where all updating requests were executed either in the SM or DU modes, and the combination of these two (Hybrid mode);
- We showed when a replicated service can benefit from HTR and discussed some techniques on how to configure the HTR algorithm for higher performance.

B. Paper structure

The paper has the following structure. Firstly, we present related work in Section II. Next, we discuss the SM and TR models in Section III. Then, in Section IV, we present the HTR algorithm and discuss its characteristics. Next, in Section V, we show the results of HTR-enabled Paxos STM evaluation by comparing its performance and scalability under diverse workloads and oracles. Finally, we conclude with Section VI.

II. Related Work

The replicated state machine (SM) was originally proposed in [12] (using the basic idea of logical clocks to order events), and later elaborated in [1] (see also [13]). For replica coordination, various fault-tolerant synchronization algorithms for totally ordering events were proposed (see e.g., [14], [15] among others). More recently, the atomic broadcast (abcast) primitive is often used to reproduce requests at every service replica and execute them sequentially (see [16] for a survey of abcast algorithms and [2] for further references).

The transactional replication (TR) uses atomic transactions which allows requests to be executed in parallel. The use of transactions in general-purpose programming was researched in the context of transactional memory (TM) [6], [7]. The programming model of TR corresponds to distributed TM (discussed below). The replication model of TR resembles multi-primary passive replication based on "deferred update" [2]. In this model, a transaction's updates can be consistently propagated to all service replicas using various synchronization protocols. However, some protocols suffer from deadlocks or instability [5]. In this paper, we consider TR that employs the deferred-update (DU) optimistic replication scheme which is based on abcast [4]. The advantages of abcast in database or service replication have been demonstrated by many authors (see e.g., [3], [4], [5], [17], [2] among others).

The concurrency control schemes in transactional systems can be divided into two main groups: optimistic and pessimistic. In the former, all transactions are executed optimistically, in isolation, and undergo a certification procedure before being allowed to commit [18]. "Deferred update" protocols fall into this category. In the pessimistic approach, transactions request an exclusive access to the shared data, which blocks concurrent transactions that attempt to access the same data. Such concurrency control schemes (e.g., two-phase-locking [19]) are not deadlock-free, so transactions may need to abort. However, the acquisition of all locks upfront guarantees abort-free execution (as in [20] or [21] or the SM mode of HTR).

Our HTR algorithm guarantees (one-copy) serializability [22] and opacity [11]—a stronger criterion which was specifically defined for TM. Many other correctness criteria have been defined, especially in the context of data replication (see Chapter 4 in [23] for a survey).

We added the HTR functionality to Paxos STM [9] which is an object-based DTM system that we developed to compare the SM and TR replication schemes. It builds on JPaxos [24]—a highly optimized implementation of the Paxos algorithm [25]. Several other DTM systems were developed so far (e.g., DiSTM [26], Anaconda [27], and Cluster-STM [28]). Notably, our system was designed from ground up as a fully distributed, fault-tolerant system in which crashed replicas can recover. Unlike DiSTM, there is no central coordinator which could become a bottleneck under high workload. The closest design to ours is the one represented by D2STM [29] which also employs full replication and transaction certification based on abcast. However, unlike us, D2STM does not allow replicas to be recovered after crash nor transactions to contain irrevocable operations (the latter feature will be discussed below).

Various protocol switching schemes were recently proposed in the context of DTM. For example, PolyCert [30] features three certification protocols that differ in the way the read-sets of updating transactions are handled. Online and offline machine learning techniques are applied for deciding which certification protocol to use. Contrary to PolyCert, our approach aims at the ability to execute transactions using different replication schemes. Additionally, our system considers a much wider set of parameters and can be tuned by the programmer for the application-specific characteristics. StarTM [31] uses static code analysis to select between the execution satisfying snapshot isolation (SI) and one-copy serializability (1SR). Serializability is ensured at all times, since the system executes transactions in the SI mode only when it is guaranteed that no write-skew anomalies can occur.

Approaches that combine locks and transactions are also relevant. In [32], Java monitors can dynamically switch between the lock-based and TM-based implementations. Similarly, adaptive locks [21] enable critical sections that are protected either by mutexes or executed as transactions. Like in HTR, the system state is monitored to choose the best execution mode. However, the above two approaches use a fixed policy. In our approach, the HTR oracles implement a switching policy that is application-specific, and gather statistics pertaining to the distributed environment.

To our best knowledge, HTR-enabled Paxos STM is the first distributed TM which enables transactions with irrevocable operations. This is possible by executing such transactions in the pessimistic SM mode (assuming that their code is deterministic). Notably, the system performance is not compromised since a single transaction in the SM mode can run in parallel with all transactions executed in the DU mode. The problem of irrevocable operations has been researched in the context of non-distributed TM (see e.g., [33], [34], [35], [10] among other). These operations are typically either forbidden, postponed until commit, or switched into an *ad hoc* pessimistic mode [21]. However, the problem remained untackled in the context of distributed TM where distribution (or replication), partial failure, and communication delays must also be considered. Some solutions for starved transactions are relevant here, e.g., based on a global lock [18] or leases [36] but the former is not optimal and the latter does not guarantee abort-free execution and requires a transaction to be first executed fully optimistically at least once.

In database systems, there exists work on allowing nondeterministic operations, so also irrevocable operations. In [20], a centralized preprocessor is used to split a transaction into a sequence of subtransactions that are guaranteed to commit. The next subtransaction to be executed is established after the previous one commits. This, however, requires one broadcast per subtransaction which significantly increases latency.

III. The Context of HTR

In this section, we define the context for the HTR algorithm. We begin with the description of the system model. Then, we present the SM and TR replication schemes. Finally, we briefly discuss strengths and weaknesses of both approaches.

A. System model

The model consists of N service processes (replicas) running on independent machines (nodes) connected via a network. The processes communicate only by means of messages. Requests are concurrently executed on all replicas. A *request* (also called a *transaction*) consists of a unique identifier `id`, `code` to be executed or methods to be invoked, and `data` necessary to execute the request. Some requests may be marked as *read-only (RO)*, i.e., they do not alter the system's state. The *read-write (RW)* requests usually modify the system's state but are not obligated to. We assume a crash-recovery failure model, where crash of at most $\lceil \frac{N}{2} \rceil - 1$ nodes is tolerated. After recovery a failed process can rejoin the system at any time. The discussed algorithms are memory model agnostic, i.e., they can be used in either the object– or memory-word–based environments. To match our implementation, we assume an object-oriented memory model.

B. State machine replication

In the replicated *state machine (SM)* [1], [37] all requests are executed sequentially by all replicas. A pseudocode is given in Algorithm 1. We assume that each request is handled by a

Algorithm 1 State Machine Replication

Thread T on request t (executed on one replica)

```
 1: upon REQUEST
 2:     package $p \leftarrow (t.id,\ t.code,\ t.data)$
 3:     abcast $p$
 4:     wait for $outcome$
 5:     stop executing request $t$ and return
```

The main thread of SM (executed on all replicas)

```
 6: upon ADELIVER(package $p$)
 7:     execute $p.code$ with $p.data$
 8:     if thread $T$ executes on this node then
 9:         $outcome \leftarrow success$
10:         pass $outcome$ to thread $T$
```

designated thread. Upon a new request, it is distributed to all replicas using the total order broadcast primitive (line 3) and handled by each replica sequentially with other requests (lines 6-10). Since we demand the handler code to be deterministic, the state of all replicas is consistent given the same initial state. After the request's code is processed (line 7), the replica which broadcast this request returns the result of the request's execution (line 5).

C. Transactional replication

Transactional replication (TR) leverages the semantics of atomic transactions to enable easy development of replicated services. A shared state that must be consistent among replicas should be accessed only by transactions (declared by the programmer). Any updates of this state performed by transactions are consistently propagated to all replicas. We allow many concurrent transactions on every replica. Concurrent execution of transactions in a replicated system is one-copy-serializable [22] (some stronger properties can also be supported, e.g., Paxos STM ensures opacity [11]). Programmers can use the *commit*, *rollback* and *retry* constructs to respectively, commit, rollback, or rollback and retry uncommitted transactions.

We consider TR based on deferred update (DU) relying on abcast which avoids blocking and limits the number of costly network communication steps [4]. The pseudocode for the DU-based TR scheme is given in Algorithm 2. Each replica maintains logical clock LC (line 1), a variable representing the system's current logical time. It is incremented every time after a transaction is committed successfully, i.e., its updates are applied (line 47). It means that each successfully committed transaction has a unique value of LC that represents the end moment in time of its whole execution; for a transaction t this value is stored in the end variable (line 44). Additionally, each transaction stores in the $start$ variable the start moment of its execution (line 25). The $start$ and end values of a transaction allow the system to reason about the precedence order between transactions. We say that transaction t_1 precedes transaction t_2 (denoted $t_1 \rightarrow t_2$) iff $t1.end < t2.start$.

The TR system traces the accesses to objects independently for each transaction. On every read or write, an object's ID is added to respectively, the $readset$ (line 16) or $writeset$ (line 22) of the transaction that accessed the object. Moreover, object modifications are recorded in the transaction's $updates$ set (line 23). All the above sets are distributed to other replicas on transaction commit.

To manage control flow of transactions, the TR system can execute the following operations:

- $rollback$ (line 36) stops the transaction's execution and revokes all the changes it performed so far;
- $retry$ (line 38) forces a transaction to rollback and restart;
- $commit$ (line 28) starts the certification phase for a transaction which either finishes the transaction (with all object modifications applied) or retries the transaction.

To commit a transaction t, t's readsets, writesets and updates are broadcast to all replicas with a global order (lines 29-30). Then, the replicas independently certify the received sets against already-committed transactions which do not precede t (lines 9-14). If a conflict is detected, i.e., t reads data modified by a concurrent but already-committed transaction t', and neither $t \rightarrow t'$ nor $t' \rightarrow t$ holds, then t is aborted and restarted (line 33). Otherwise, the system applies the updates performed by transaction t and commits it (lines 44-47). The consistency of replicated data is preserved since the process of certifying transactions and updating the system's state is atomic, deterministic, and performed in the same order on each replica by relying on the total order broadcast.

Note that the operations on LC (lines 25, 44, 47), log (lines 10 and 45) and the accesses to transactional objects (lines 7 and 46) have to be synchronized. For simplicity, in the pseudocode a `lock` statement is used. For better performance, the implementation can rely on fine-grained locks or it can avoid locks altogether. In Paxos STM, we implemented a lock-free version of this algorithm that builds on multiversioning (explained in Section III-D).

The certification test (lines 9-14) detects if a transaction does not conflict with other (older) transactions. In the pseudocode, this test is executed on every read. However, our implementation uses an optimized algorithm in which objects are associated with version numbers (value of LC at the time they were last modified) that help to avoid repeating the certification test for concurrent, already-committed transactions (details are omitted due to lack of space).

D. SM vs TR comparison

The SM scheme in many cases proves to be highly efficient although no parallelism in the request execution is allowed. In the optimized version of SM, which we modeled in [9], read-only requests can be executed in parallel but it is not straightforward to implement this optimization in Paxos [38]. Since the service code is inherently single-threaded the SM scheme is also relatively easy to implement since there is no inter-thread synchronization. All the complexity is hidden behind atomic broadcast (see [16] for a survey of algorithms).

Unfortunately, the replicated state machine model has numerous drawbacks. In order to preserve consistency, a replicated service has to be deterministic. Moreover, since requests are executed sequentially on each node, the system does not scale with the increasing number of nodes, nor can it take

Algorithm 2 Transactional Replication using Deferred Update

```
1: integer $LC \leftarrow 0$
2: set $log \leftarrow \emptyset$

3: function GETOBJECT(transaction $t$, objectId $oid$)
4:     if $(oid, v) \in t.updates$ then
5:         $value \leftarrow v$
6:     else
7:         lock { $value \leftarrow$ retrieve object $oid$ }
8:     return $value$

9: function CERTIFY(integer $start$, set of ids $readset$)
10:    lock { $L \leftarrow \{t \in log \mid t.end > start\}$ }
11:    for all $t \in L$ do
12:        if $readset \cap t.writeset \neq \emptyset$ then
13:            return $failure$
14:    return $success$

15: function READ(transaction $t$, objectId $oid$)
16:    $t.readset \leftarrow t.readset \cup \{oid\}$
17:    if CERTIFY$(t.start, t.readset) = failure$ then
18:        raise RETRY
19:    else
20:        return GETOBJECT($t$, $oid$)

21: procedure WRITE(transaction $t$, objectId $oid$, object $v$)
22:    $t.writeset \leftarrow t.writeset \cup \{oid\}$
23:    $t.updates \leftarrow t.updates \cup \{(oid,\ v)\}$
```

Thread T on request t (executed on one replica)

```
24: upon TRANSACTION
25:    lock { $t.start \leftarrow LC$ }
26:    execute $t$
27:    raise COMMIT

28: upon COMMIT
29:    package $p \leftarrow (t.id,\ t.start,\ t.readset,\ t.writeset,\ t.updates)$
30:    abcast $p$
31:    wait for $outcome$
32:    if $outcome = failure$ then
33:        raise RETRY
34:    else                                    // $outcome = success$
35:        stop executing transaction $t$ and return

36: upon ROLLBACK
37:    stop executing transaction $t$ and return

38: upon RETRY
39:    $t.readset \leftarrow \emptyset$, $t.writeset \leftarrow \emptyset$, $t.updates \leftarrow \emptyset$
40:    raise TRANSACTION
```

The main thread of TR (executed on all replicas)

```
41: upon ADELIVER(package $p$)
42:    $outcome \leftarrow$ CERTIFY$(p.start, p.readset)$
43:    if $outcome = success$ then
44:        lock { $p.end \leftarrow LC$
45:               $log \leftarrow log \cup \{p\}$
46:               apply $p.updates$
47:               $LC \leftarrow LC + 1$ }
48:    if thread $T$ executes on this node then
49:        pass $outcome$ to thread $T$
```

advantage of multicore architectures to parallelize requests' execution. Therefore this scheme is very susceptible to CPU intensive workloads, as we showed in [9]. This can be partially alleviated by executing read-only requests on one node only (the *optimized SM* replication scheme in [9]).

Contrary to SM, in TR parallelism is supported by default since transactions are executed (on the same node or on different nodes) concurrently and in isolation. However, workloads generating high contention may force transactions to be aborted numerous times before committing, thus limiting the performance. Aborting transactions in the execution state as soon as they are known to be in conflict with a transaction that had just recently committed may help to some degree, i.e., after a transaction t had been successfully certified, every concurrent transaction t' is aborted when $t.writeset \cap t'.readset \neq \emptyset$ (see Algorithm 2, lines 17-18).

In TR, no synchronization (no communication step) is required among replicas in case of RO transactions since they do not change the system's state. Additionally, they can be provided with abort-free execution guarantee by introducing multiversioning scheme [22]. Multiversioning allows multiple versions of all transactional objects to be stored while being transparent to the programmer, i.e., at any moment only one version of any transactional object is accessible by a transaction. Paxos STM implements both early conflict detection as well as the multiversioning scheme.

Usually, there is a significant difference in the size of network messages communicated between replicas in TR and SM. In TR, abcast messages contain transaction readsets, writesets and updates. The size of these messages can be significant even for a medium sized transaction. Large messages cause strain on the abcast mechanism and increase certification overhead. On the other hand, the propagated requests in SM consist only of an identifier of a method to be executed and data required for its execution; these messages are often as small as 100B.

TR supports concurrency on multicore architectures. Concurrent programming is error-prone but atomic transactions greatly help to write correct programs. Firstly, operations defined within a transaction appear as a single logical operation whose results are seen entirely or not at all. Secondly, concurrent execution of transactions is deadlock-free which guarantees progress. Moreover, the *commit*, *rollback* and *retry* constructs enhance expressiveness. However, *irrevocable operations*—operations that cause side effects which cannot be rolled back (such as system calls) are not permitted by fully optimistic TR schemes where a transaction may be forced to abort and restart due to conflicts with other transactions.

IV. The HTR Algorithm

In this section, we define the hybrid transactional replication (HTR) algorithm which extends transactional replication based on deferred update (DU) with the state-machine-based (SM) replication scheme. First, we discuss the transaction oracle—the key new component of our algorithm. Next, we explain the HTR algorithm by presenting its pseudocode, and also sketch the proof of correctness. Finally, we briefly discuss the strengths of HTR and tuning the HTR oracle.

A. Transaction oracle

Our aim was to seamlessly merge the SM and DU replication schemes, so that requests (transactions) can be executed in either scheme depending on the desired performance considerations and execution guarantees (e.g., support for irrevocable operations). *Transaction oracle* (or *oracle*, in short) is a mechanism that is able to assess the best execution mode for a given transaction's run, and dynamically switch between SM and DU. It may rely on hints declared by the programmer as well as on dynamically-collected *statistics*, i.e., data regarding various aspects of system's performance, such as:

- duration of various phases of transaction processing e.g., the execution time of a transaction code, abcast latency, and the duration of transaction certification,
- the transaction abort rate, i.e., the ratio of the transaction's aborted runs to all execution attempts,
- sizes of abcast messages, readsets and writesets,
- system load,
- performance of garbage collector,
- saturation of the network.

Read-only transactions are always executed locally since they do not alter the system's state and thus do not require distributed certification. Hence decisions made by the oracle only regard updating transactions.

Since the hardware and the workload can vary between the replicas the system can use different oracles at different nodes and independently change them at runtime when desired. For brevity, in the description of the algorithm we abstract away the details of the oracle implementation and treat it as a black box with only two functions: `feed(data)` (used to update the oracle with data collected over the last transaction's run, regardless of the outcome) and `query` (used to decide in which mode a new transaction is to be executed). The problem of constructing an oracle that matches the characteristics of the application is discussed in Section IV-D. The oracles used in the experimental evaluation of the HTR algorithm are described in Section V-A.

B. The algorithm

Below we describe the HTR algorithm (see Algorithm 3). Our algorithm is essentially TR (Algorithm 2), extended with the SM replication scheme and the UPDATEORACLESTATISTICS procedure (line 56) that feeds the oracle with the statistics collected in a particular run of a transaction before the transaction is committed, rolled back, or retried.

When the DU mode is chosen (lines 26-29), a transaction, called *DU transaction*, is executed under the deferred-update replication scheme. When the SM mode is chosen, a transaction, called *SM transaction*, is broadcast (lines 31-32), and executed on all replicas by their main thread (lines 67-72) which is the thread that also performs the certification test for DU transactions and applies the updates. It means that at most one SM transaction can be executed by a replica at a time. However, the algorithm does not prevent concurrent execution of a SM transaction and multiple DU transactions, but the certification test and state update operations may be delayed till the SM transaction is completed. Since the SM transaction is executed by all replicas, its code has to be deterministic. Note that for any SM transaction, the `certify` function (lines 9-14) always returns *success*. This is correct since no other transaction can commit at the same time.

Lemma 1: The HTR algorithm ensures opacity.

Proof (sketch): Following the authors in [11], we need to show that for every history H produced by HTR there exists a sequential history S, such that (1) S is equivalent to some history in the set $Complete(H)$, and S preserves the real-time order of H, and (2) every transaction $t \in S$ is legal in S.

Informally, we have to prove that for every complete history H produced by HTR, H is strictly serializable and every transaction (either in executing, committing or committed state) always observes a consistent state of the system, i.e., every read operation on each object x returns the value stored by the most recent preceding write operation on the object x of a locally committed transaction.

The correctness proof is constructed on a number of important observations: The first one is that (a) the same abcast mechanism is used to broadcast with the total order a package containing information about the execution of a DU transaction (the `DUpackage`, lines 48-49) as well as a transaction to be executed in the SM mode (the `SMpackage`, lines 31-32). Additionally, each replica has one designated thread (the *main HTR thread* in Algorithm 3), which sequentially executes all DU- and SM-packages abcast by any replica. For that reason (b) execution of any SM transaction t_{SM} cannot be interleaved with processing of any other package. This means that no two SM transactions can be executed at the same time on one node. Moreover, (c) updates to the system's state are performed atomically in the DU/SM package arrival order without any reordering possible (lines 61-64 for DU and 74-77 for SM). One can also notice that (d) all transactions execute updating operations on copies of objects, i.e., they do not modify the system's state unless they are marked as committed (line 23). Finally, (e) an executing DU transaction that is known to conflict with other just committed transaction, is aborted as soon as it performs a read operation on a shared object (it never observes inconsistent state) (lines 17-18).

The proof is based on the above observations and uses the formalism of [39]. Since SM transactions have to be deterministic, the system's state is consistent across all the replicas and opacity is preserved. ■

The transactions in the executing state that are known to conflict with the just-committed transactions have to be aborted. This is necessary not only to provide opacity but also to save on unnecessary computation. Multiversioning brings the same benefits to HTR as in the case of TR—the RO transactions are guaranteed abort-free execution.

C. The algorithm's strengths

Below we present the advantages of the HTR algorithm compared to the exclusive use of the schemes discussed in Section III. We also discuss potential performance benefits that will be evaluated experimentally in Section V.

1) Expressiveness: Implementing services using the original SM replication scheme is straightforward since it does not involve any changes to the service code. However, the

Algorithm 3 The Hybrid Transactional Replication (HTR) Algorithm

```
1: integer $LC \leftarrow 0$
2: set $log \leftarrow \emptyset$

3: function GETOBJECT(transaction $t$, objectId $oid$)
4:     if $(oid, v) \in t.updates$ then
5:         $value \leftarrow v$
6:     else
7:         lock { $value \leftarrow$ retrieve object $oid$ }
8:     return $value$

9: function CERTIFY(integer $start$, set of ids $readset$)
10:    lock { $L \leftarrow \{t \in log | t.end > start\}$ }
11:    for all $t \in L$ do
12:        if $readset \cap t.writeset \neq \emptyset$ then
13:            return $failure$
14:    return $success$

15: function READ(transaction $t$, objectId $oid$)
16:    $t.readset \leftarrow t.readset \cup \{oid\}$
17:    if CERTIFY($t.start, t.readset$) $= failure$ then
18:        raise RETRY
19:    else
20:        return GETOBJECT($t$, $oid$)

21: procedure WRITE(transaction $t$, objectId $oid$, object $v$)
22:    $t.writeset \leftarrow t.writeset \cup \{oid\}$
23:    $t.updates \leftarrow t.updates \cup \{(oid,\ v)\}$
```

Thread T on request t (executed on one replica)

```
24: upon TRANSACTION
25:    $t.mode \leftarrow TransactionOracle.query()$
26:    if $t.mode = DU$ then
27:        lock { $t.start \leftarrow LC$ }
28:        execute $t$
29:        raise COMMIT
30:    else                                          // $t.mode = SM$
31:        SMpackage $p \leftarrow (t.id,\ t.start,\ t.code,\ t.data)$
32:        abcast $p$
33:        wait for $outcome$
34:        UPDATEORACLESTATISTICS($t$)
35:        if $outcome = retry$ then
36:            $t.readset \leftarrow t.writeset \leftarrow t.updates \leftarrow \emptyset$, $t.stats \leftarrow \perp$
37:            raise TRANSACTION
38:        else
39:            stop executing transaction $t$ and return

40: upon RETRY                                       // for DU transactions
41:    UPDATEORACLESTATISTICS($t$)
42:    $t.readset \leftarrow t.writeset \leftarrow t.updates \leftarrow \emptyset$, $t.stats \leftarrow \perp$
43:    raise TRANSACTION

44: upon ROLLBACK                                    // for DU transactions
45:    UPDATEORACLESTATISTICS($t$)
46:    stop executing transaction $t$ and return

47: upon COMMIT                                      // for DU transactions
48:    DUpackage $p \leftarrow (t.id,\ t.start,\ t.readset,\ t.writeset,\ t.updates)$
49:    abcast $p$
50:    wait for $outcome$
51:    if $outcome = failure$ then
52:        raise RETRY
53:    else                                          // $outcome = success$
54:        UPDATEORACLESTATISTICS($t$)
55:        stop executing transaction $t$ and return

56: procedure UPDATEORACLESTATISTICS(transaction $t$)
57:    $TransactionOracle.feed(t.stats)$
```

The main thread of HTR

```
58: upon ADELIVER(DUpackage $p$)
59:    $outcome \leftarrow$ CERTIFY($p.start, p.readset$)
60:    if $outcome = success$ then
61:        lock { $p.end \leftarrow LC$
62:                $log \leftarrow log \cup \{p\}$
63:                apply $p.updates$
64:                $LC \leftarrow LC + 1$ }
65:    if thread $T$ executes on this node then
66:        pass $outcome$ to thread $T$

67: upon ADELIVER(SMpackage $p$)
68:    $p.start \leftarrow LC$
69:    execute $p.code$ with $p.data$
70:    raise COMMIT
71:    if thread $T$ executes on this node then
72:        pass $outcome$ to thread $T$

73: upon COMMIT                                      // for SM transactions
74:    lock { $p.end \leftarrow LC$
75:            $log \leftarrow log \cup \{p\}$
76:            apply $p.updates$
77:            $LC \leftarrow LC + 1$ }
78:    $outcome \leftarrow committed$
79:    goto line 71

80: upon ROLLBACK                                    // for SM transactions
81:    $outcome \leftarrow rolledback$
82:    goto line 71

83: upon RETRY                                       // for SM transactions
84:    $outcome \leftarrow retry$
85:    goto line 71
```

programmer does not have any constructs to express control-flow other than the execution of a request in its entirety. In our HTR replication scheme, the programmer can use expressive transactional primitives `rollback` and `retry` to withdraw any changes made by transactions and to retry transactions (possibly in a different replication mode). Upon retry, the SM transaction is not immediately reexecuted on each node. Instead, the control-flow returns to the thread which is responsible for handling the original request. The oracle is then queried again to determine in which mode the transaction should be reexecuted. Similarly, reexecution of DU transactions is also controlled by the oracle.

In the original SM scheme, suspending a request's execution until some event occurs or a condition is met is not advisable since this would effectively block the whole system. This is because all requests are executed serially in the order they are received. On the contrary, when the `retry` is called from within a SM transaction, the HTR algorithm rollbacks the transaction and allows it to be restarted when the condition is met.

2) Irrevocable operations: In the DU replication scheme, transactions may be aborted and afterwards restarted due to conflicts with other older transactions. Thus, they are forbidden to perform *irrevocable operations* whose side effects cannot be rolled back (such as local system calls). *Irrevocable (or inevitable) transactions* are transactions that contain irrevocable operations. Support for such transactions is problematic and has been subject of extensive research in the context of non-distributed TM (see Section II). However, the proposed methods and algorithms are not directly transferable to distributed TM systems where problems caused by distribution, partial failures, and communication must also be considered. Below we explain how the HTR algorithm deals with irrevocable transactions which, to the best of our knowledge, represents the first attempt to provide this capability in distributed TM.

In the HTR algorithm, irrevocable transactions are executed exclusively in the SM mode, thus guaranteeing abort-free execution which is necessary for correctness. It also means that only one irrevocable transaction is executed at a time. This is also recommended since (in general) the kind of side effects that the irrevocable (non-memory) operations may cause are unknown prior to execution of the transaction. However, our scheme does not prevent DU transactions to be executed in parallel—just only certification and the subsequent process of applying updates of DU transactions (in case of successful certification) must be serialized with execution of SM transactions. Since a SM transaction runs on every replica, we only consider deterministic irrevocable transactions. Non-deterministic transactions would require acquisition of a global lock or a token to be executed exclusively on a single replica. Alternatively, some partially centralized approaches could be employed, as in [20]. However, they introduce additional communication steps, increase latency, and may force concurrent transactions to wait a significant amount of time to commit.

We forbid the *rollback* and *retry* primitives in irrevocable transactions (as in [10] and other TM systems) since they may leave the system in an inconsistent state.

3) Performance: As mentioned in Section III-D, it is not straightforward to optimize the original state-machine-based replication scheme to handle the read-only (RO) requests (or transactions) in parallel with other (RO or RW) transactions. However, in the HTR algorithm, RO transactions are executed only by one replica, in parallel with any RW transactions—there is no need for synchronization among replicas to handle the RO transactions. Moreover, the RO transactions are guaranteed abort-free execution thanks to multiversioning.

Unless a RW transaction is irrevocable (thus executed in the SM mode) or non-deterministic (thus executed in the DU mode), it can be handled by HTR in either mode for increased performance. The choice is made by the HTR oracle that constantly gathers statistics during system execution and can dynamically adapt to the changing workload (which may vary between the replicas). In Section IV-D, we discuss the tuning of the oracle.

D. Tuning the oracle

As pointed out in [40], DTM workloads are usually highly diversified in regard to the execution times and to the number of objects accessed by each transaction (this is also reflected in our benchmark tests in Section V). However, the execution times of the majority of transactions are way under 1 ms. Therefore, the mechanisms that add to transaction execution time should be lightweight, e.g., the oracle should be as lightweight as possible in order to gain benefits of dynamic adaptation.

In the HTR algorithm, the oracle is defined by only two methods that have to be provided by the programmer. Combined with multiple parameters collected by the system at runtime, the oracle allows for a flexible solution that can be tuned for a particular application. Our experience with testing HTR-enabled Paxos STM using multiple benchmarks shows that there are the two most important factors that should be considered when implementing an oracle (in brackets we give references to the evaluation section where we employ these techniques):

- keeping abort rate low—a high abort rate means that many transactions executed in the DU mode are rolled back (multiple times) before they finally commit; this undesirable behaviour can be prevented by executing some (or all) of them in the SM mode (V-C1). The SM mode can also be chosen for transactions consisting of operations that are known to generate a lot of conflicts, such as resizing a hashtable (V-C2,V-C3). On the contrary, the DU mode is good for transactions that do not cause high contention, so can be executed in parallel taking advantage of modern multicore hardware;
- choosing the SM mode for transactions that are known to generate large messages when executed optimistically in the DU mode. Large messages increase network congestion and put strain on the abcast mechanism, thus decreasing its performance. The execution of a SM transaction usually only requires broadcasting the name of the method to be invoked; such messages are often shorter than 100B (V-C2).

Note also that since the SM transactions are guaranteed to commit, they do not require certification which eliminates the certification overhead. This overhead (in the DU mode) is proportional to the size of the readsets, writesets, and updates.

Devising lightweight oracles that adapt to changing workload constitutes an interesting research path that we plan to investigate in the future. To increase the oracle's accuracy one may also consider offline methods such as static analysis. In database systems, where transaction processing times are significantly higher compared to DTM, oracles with higher overhead are suitable for consideration. Such advanced solutions could be based on the SQL query optimizer which is already in place with most of the database engines.

V. Evaluation

In this section, we present the results of the empirical study of the HTR algorithm with various oracles. In our tests, we used several benchmarks which produce varied workloads.

NumAtomicPerComp	100
NumConnPerAtomic	3
DocumentSize	20000
ManualSize	1000000
NumCompPerModule	500
NumAssmPerAssm	3
NumAssmLevels	5
NumCompPerAssm	3
NumModules	1

Fig. 1. The input parameters of Distributed STMBench7.

A. Benchmarks

We adopted three benchmarks that are commonly used to evaluate TM systems: *Bank*, a distributed version of STMBench7 (called *Distributed STMBench7*), and an implementation of a Twitter-like social networking service as an example of a real-world application (called *Twitter Clone*).

1) Bank: A replicated array of accounts is concurrently processed by several worker threads that execute two types of transactions. The update transaction (RW) performs money transfer between two accounts, and consists of two read and two write operations executed on two distinct accounts. The read-only transaction (RO) calculates the total balance, i.e., sums up the funds from all accounts. For the tests, we used a scenario consisting of 95% of RW and 5% of RO transactions. The number of accounts is 10000. The tests are performed with 160 threads running on each node. In Section V-C1, we explain precisely why such a high number of threads is used.

2) Distributed STMBench7: The STMBench7 benchmark [41], proposed to simulate realistic workloads for TM, was adapted to run in a distributed environment. The benchmark consists of a large, tree-like data structure (containing roughly 1.5 million elements) which is queried or modified by a set of operations of varying complexity. Each operation defined in STMBench7 is executed as a separate transaction. The values of the input parameters are given in Figure 1. The transactions are performed by 4 worker threads running on each node (the same as the number of the CPU cores).

3) Twitter Clone: Our implementation of the Twitter social networking service offers the same core functionality as the original Twitter service. A user can: (a) post 140-character-long messages which are seen by the users following him (i.e., subscribed to the user's posts feed), (b) reply to somebody's message, and (c) repost ("retweet") it. With certain probabilities specified by the input parameters, the users create tweets that mention their friend, a star, or a hashtag. All mentions are randomly chosen from a pool of values of an adjustable size. In order to simulate various workloads, the system users are divided in two classes: the ordinary users and the stars, differentiated by the number of followers. Ordinary users tend to create small subgroups called cliques in which all users are connected to each other. Stars also have similar small groups of friends, but additionally they are followed by a large number of ordinary users. Scanning through the user's home page as well as browsing tweets posted by a given user is performed as a RO transaction. Posting a tweet or retweeting other user's post is a lightweight RW transaction. The relative number of executed RW and RO transactions is defined by an input parameter. Conflicts between transactions may arise while: (a) adding replies to somebody's post (most noticeable in the case of replies to a star's tweet), (b) reposting, and (c) mentioning the same hashtag or the same user. To limit the number of tweets kept in the memory, the older ones are discarded after a defined period of time. As in the case of the previous benchmark, the number of worker threads matches the number of CPU cores on each node.

To show gains provided by the HTR algorithm, each benchmark is tested with three different oracles:

- DU – all updating (RW) transactions are executed in the deferred update mode,
- SM – all RW transactions are executed in the replicated state machine mode,
- Hybrid – transactions are executed either in the DU or the SM mode, according to a policy defined to match a given benchmark; for each benchmark we describe how the Hybrid oracle is constructed.

All RO transactions are always executed locally in the DU mode and they never abort. Therefore, the differences in performance for various oracles can only be attributed to the way the RW transactions are handled.

B. Evaluation environment

In our tests, we used a cluster consisting of nine nodes connected via a private 1 Gb Ethernet network. Each node is 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.

C. Results and analysis

1) Bank: This benchmark is characterized by very short RW transactions (money transfer between two accounts) and relatively long, time consuming RO transactions (the total balance of all accounts). Since RW transactions dominate the workload (95% RW transactions), one may expect the SM oracle to exhibit very limited scalability, since all the RW transactions have to be executed sequentially. On the contrary, the system in this configuration scales all the way up to 9 nodes (see Figure 2). In the tested scenario, performance of each node is not limited by its CPU power (the main HTR thread which executes the SM transactions is underutilized), but by the ability of each node to handle a high number of incoming requests. To better utilize the main HTR thread more concurrent transactions (requests) have to be allowed in. One way to achieve it is to further increase the number of threads running on each replica. However, this approach proved unsuccessful due to very high inter-thread synchronization overhead when more than 160 threads ran on each node. However, with the higher number of replicas it is possible to handle more concurrent requests, what explains why the system scales. On the other hand, in the DU mode, short execution times limit the possible performance gain due to parallel execution of RW transactions. Short execution times of RW transactions

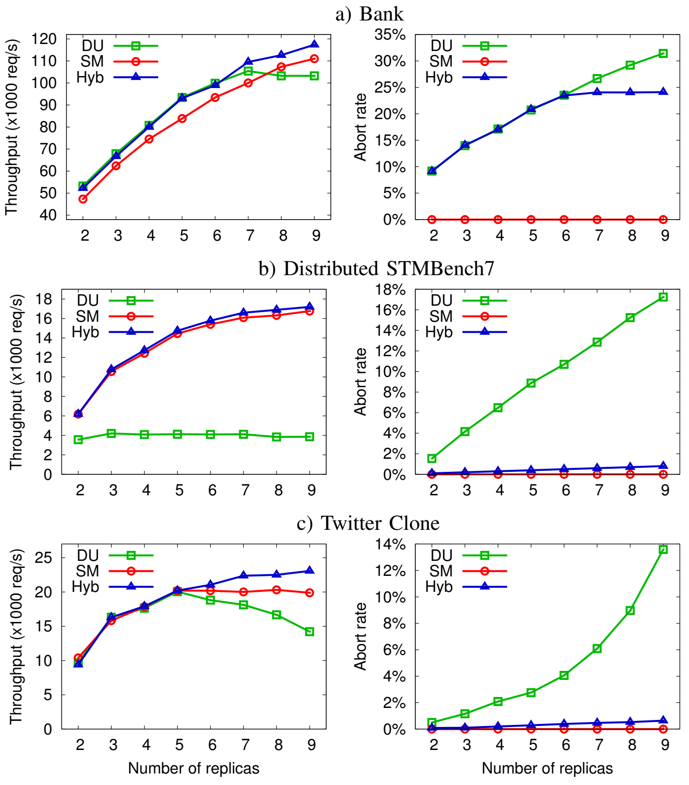

Fig. 2. Throughput and abort rate, where *DU*, *SM*, *Hyb* denote the oracle.

and similar sizes of messages for the DU and the SM modes (63B and 47B, respectively) result in very similar performance with both the oracles. Up to 6 nodes, the DU mode provides slightly better performance compared to the SM oracle (10% better on average). However, for the high number of nodes, the performance in the DU oracle is diminished due to high contention (the abort rate is as high as 34% for 9 nodes).

To counter the decrease of performance under high contention in the case of the DU oracle, we used a simple Hybrid oracle that does not allow the abort rate to exceed 25% threshold. Before each transaction's run the oracle checks whether the current value of abort rate (updated after each transaction's commit, rollback or retry) is greater than 25%. If so, the SM mode is chosen. The SM mode guarantees abort free execution, thus promises reduction of the abort rate. If the current value of abort rate is below the threshold, the system executes the transaction in the DU mode. The threshold value of 25% was established empirically to give the best results. The Hybrid oracle gives as good results as the DU oracle up to 6 nodes and allows the system to scale further, giving better results than when using either the DU or SM oracle alone.

2) Distributed STMBench7: STMBench7 was designed to test non-distributed TM systems under complex workloads. In the distributed environment, where network transmission time is often an order of magnitude larger than the transaction's execution time, STMBench7 shows unstable performance when operations such as structural modifications and long traversals are enabled. They heavily modify (distort) the tree-like data structure in a random manner and are never able to restore its intended properties. Regardless of the duration of the tests, the data structure is never balanced and highly varies in size, when comparing results from many rounds of tests. For this reason, we decided to conduct tests with long traversals and structural modifications disabled (an option in the original STMBench7).

The results, given in Figure 2b, show that the performance of Paxos STM in the DU mode is limited by high contention (up to 18% for 9 nodes). Detailed results reveal that for some transactions, such as the one executing Operation11, the abort rate is as high as 96% (on average, the transaction is aborted 24 times before finally committing). The SM oracle outperforms the DU oracle in all cases and scales due to no contention, giving the overall throughput over 4 times as high as the DU oracle for 9 nodes.

With such a significant difference in performances of the DU and SM oracles, it is very difficult to find a Hybrid oracle which performs better than the SM oracle. The Hybrid oracle used in this test always executes ShortTraversal8, Operation9, Operation13 and Operation14 optimistically in the DU mode. Other operations are always executed in the SM mode for low contention. This way the abort rate does not exceed 1%. The Hybrid oracle performs very similarly to the SM oracle, outperforming it by up to 3.5% for higher number of nodes.

3) Twitter Clone: The benchmark's parameters allow us to simulate a wide range of real workloads. In the benchmark's default configuration, the obtained results were surprisingly similar for both the DU and SM oracles. Compared to the SM mode, Paxos STM in the DU mode executes the transactions faster. However, this gain is mitigated by larger messages because readsets and writesets need to be broadcast. Therefore, for the test discussed here, we have chosen workload that has a nontrivial characteristics. The results of the test are given in Figure 2c. Up to 5 nodes the performance of the DU and the SM oracles is the same, but later all oracles give throughput that varies significantly. In case of the DU oracle, contention raises rapidly up to 14% for 9 nodes, thus decreasing system performance. On the other hand, the SM oracle does not scale at all. This behavior is caused by the multitude of the RW transactions, overloading the main thread. Since they have to be executed sequentially, there is only a limited number of transactions that can be handled in a given amount of time.

The Hybrid oracle counters the negative aspects of the DU and SM oracles. By default, it chooses the DU mode, thus allowing for scalable behavior. The most conflicting transactions (all those referring to a star) are executed conservatively in the SM mode. This results in low contention (less than 1% for 9 nodes). When using the Hybrid oracle, the system scales, reaching 23000 transactions per second for 9 nodes (which is over 16% faster than when using the SM oracle).

D. Summary

The results of the experimental evaluation confirmed our predictions—the Hybrid oracle was never worse than the DU and SM oracles, and even outperforms them when the number of replicas grows. Paxos STM with the SM oracle performed really well in all our tests, even though the SM mode prevents parallel execution of RW transactions. The good performance can be attributed to the lack of contention (all RW transactions are executed sequentially) and small messages (no read/write-sets or updates need to be broadcast).

VI. Conclusions

In this paper, we presented and evaluated the hybrid transactional replication (HTR), a novel model and algorithm for replication of services. The two transaction execution modes that are used in HTR (deferred update (DU) and state machine (SM)) complement each other. The DU mode allows for parallelism in transaction execution, while the SM mode provides abort-free transactions which are useful to deal with irrevocable operations and transactions generating high contention. Dynamic switching between the modes enables HTR to perform well under a wide range of workloads, which is not possible for either of the schemes independently. In the future, we plan to investigate automatic machine learning techniques to make the oracle mechanism more robust.

Surprisingly, in all tests that we conducted, the SM oracle performs very well (all updating transactions are executed on all nodes sequentially, and all read-only transactions are executed locally without any replica coordination). This proves viability of the state-machine-based replication scheme. However, combining the SM with the DU scheme and transactional primitives, not only increases performance but also greatly improves expressiveness, making the HTR approach a promising, versatile solution.

Acknowledgements We thank anonymous reviewers for their useful comments and suggestions. T. Kobus and M. Kokociński were supported by the Polish Ministry of Science and Higher Education within the NCN Grant No. 2011/01/N/ST6/06762.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[2] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*, ser. LNCS, vol. 5959. Springer, 2010.

[3] F. Pedone, R. Guerraoui, and A. Schiper, "Exploiting atomic broadcast in replicated databases," in *Proc. of Euro-Par '98*, Sep. 1998.

[4] F. Pedone, R. Guerraoui, and A.Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, Jul. 2003.

[5] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases," in *Proc. of Euro-Par '97*, Aug. 1997.

[6] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proc. of ISCA'93*, 1993.

[7] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. of PODCS '95*, Aug. 1995.

[8] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM TOPLAS*, vol. 12, no. 3, 1990.

[9] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "Model-driven comparison of state-machine-based and deferred-update replication schemes," in *Proc. of SRDS '12*, Oct. 2012.

[10] A. Welc, B. Saha, and A.-R. Adl-Tabatabai, "Irrevocable transactions and their applications," in *Proc. of SPAA '08*, Jun. 2008.

[11] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proc. of PPoPP '08*, Feb. 2008.

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[13] F. B. Schneider, "Synchronization in distributed programs," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 2, pp. 125–148, Apr. 1982.

[14] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, pp. 95–114, 1978.

[15] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM TOPLAS*, vol. 6, no. 2, pp. 254–280, Apr. 1984.

[16] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[17] B. Kemme, F. Pedone, G. Alonso, and A. Schiper, "Processing transactions over optimistic atomic broadcast protocols," in *Proc. ICDCS'99*, 1999.

[18] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM TODS*, vol. 6, no. 2, pp. 213–226, Jun. 1981.

[19] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in *Proc. of IFIP Working Conference on Modelling in Data Base Management Systems*, Jan. 1976, pp. 365–394.

[20] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 70–80, Sep. 2010.

[21] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis, "Adaptive locks: Combining transactions and locks for efficient concurrency," *J. Parallel Distrib. Comput.*, vol. 70, no. 10, pp. 1009–1023, 2010.

[22] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM TODS*, vol. 8, no. 4, pp. 465–483, 1983.

[23] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D., MIT, MA, USA, Mar. 1999, also as Technical Report MIT/LCS/TR-786.

[24] 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.

[25] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[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. of IPDPS '10*, Apr. 2010.

[28] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain, "Software transactional memory for large scale clusters," in *Proc. PPoPP '08*, Feb. 2008.

[29] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *Proc. of PRDC '09*, Nov. 2009.

[30] M. Couceiro, P. Romano, and L. Rodrigues, "Polycert: polymorphic self-optimizing replication for in-memory transactional grids," in *Proc. of ACM/IFIP/USENIX Middleware '11*, 2011, pp. 309–328.

[31] R. J. Dias, D. Distefano, J. C. Seco, and J. Lourenço, "Verification of snapshot isolation in transactional memory Java programs," in *Proc. of ECOOP '12*, Jun. 2012.

[32] A. Welc, A. L. Hosking, and S. Jagannathan, "Transparently reconciling transactions with locking for Java synchronization," in *In ECOOP'06 (2006*, 2006, pp. 148–173.

[33] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *Proc. of ISCA '07*, 2007.

[34] M. Olszewski, J. Cutler, and J. G. Steffan, "JudoSTM: A dynamic binary-rewriting approach to software transactional memory," in *Proc. of PACT '07*, Sep. 2007.

[35] M. F. Spear, M. Michael, and M. L. Scott, "Inevitability mechanisms for software transactional memory," in *Proc. of TRANSACT '08*, feb 2008.

[36] N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous lease-based replication of software transactional memory," in *Proc. of Middleware '10*, ser. LNCS, vol. 6452, 2010.

[37] F. B. Schneider, *Replication management using the state-machine approach*. ACM Press/Addison-Wesley, 1993, pp. 169–197.

[38] R. van Renesse, "Paxos made moderately complex," 2012, online: http://www.cs.cornell.edu/courses/CS6452/2012sp/papers/paxos-complex.pdf.

[39] P. A., Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[40] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," in *Proc. of LADIS '08*, Sep. 2008.

[41] R. Guerraoui, M. Kapałka, and J. Vitek, "STMBench7: A benchmark for software transactional memory," in *Proc. of EuroSys '07*, Mar. 2007.