SCORe: a Scalable One-Copy Serializable Partial Replication Protocol*

Sebastiano Peluso[1,2], Paolo Romano[2], and Francesco Quaglia[1]

[1] Sapienza University, Rome, Italy
{peluso,quaglia}@dis.uniroma1.it
[2] IST/INESC-ID, Lisbon, Portugal
{peluso,romanop}@gsd.inesc-id.pt

Abstract. In this article we present SCORe, a scalable one-copy serializable partial replication protocol. Differently from any other literature proposal, SCORe jointly guarantees the following properties: (i) it is genuine, thus ensuring that only the replicas that maintain data accessed by a transaction are involved in its processing, and (ii) it guarantees that read operations always access consistent snapshots, thanks to a one-copy serializable multiversion scheme, which never aborts read-only transactions and spares them from any (distributed) validation phase. This makes SCORe particularly efficient in presence of read-intensive workloads, as typical of a wide range of real-world applications. We have integrated SCORe into a popular open source distributed data grid and performed a large scale experimental study with well-known benchmarks using both private and public cloud infrastructures. The experimental results demonstrate that SCORe provides stronger consistency guarantees (namely One-Copy Serializability) than existing multiversion partial replication protocols at no additional overhead.

Keywords: Distributed Transactional Systems, Partial Replication, Scalability, Multiversioning.

1 Introduction

In-memory, transactional data platforms, often referred to as NoSQL data grids, such as Cassandra, BigTable, or Infinispan, have become the reference data management technology for grid and cloud computing systems. For these platforms, data replication represents the key mechanism to ensure both adequate performance and fault-tolerance, since it allows (a) distributing the load across the different nodes within the platform, and (b) ensuring data survival in the event of node failures.

* This work has been partially supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102496/2008 and PEst-OE/EEI/LA0021/2011, by the EU project Cloud-TM (contract no. 57784) and by COST Action IC1001 EuroTM.

A common design approach for these data platforms consists in the adoption of relaxed data-consistency models, such as eventual consistency [1] and non-serializable isolation levels [2], or restricted transactional semantics, such as single object transactions [3] and static transactions [4]. These schemes have been shown to yield significant performance advantages with respect to classic strongly consistent transactional paradigms. Unfortunately, these advantages come at the cost of additional complexity for the programmers, who have to reason on the correctness of complex applications in presence of weak consistency guarantees and/or may need to identify non-trivial work-around solutions to circumvent the limitations of constrained programming paradigms.

Some recent proposals have been targeted at more strict consistency models, such as One-Copy Serializability, and have been based on the usage of (optimistic) atomic broadcast protocols [5]. Unfortunately, these solutions have been tailored for the case of full replication (in which each node maintains a copy of the entire data-set), which is a clearly not viable option for large scale systems. Indeed, the adoption of partial data replication schemes appears to be an essential requirement for large scale systems. In this context, a key requirement to maximize scalability is to ensure *genuineness* [6, 7], namely to guarantee that only the sites that replicate the data items accessed within a transaction exchange messages to decide its final outcome (hence excluding solutions that rely on centralized components or that involve every site in the system). Unfortunately, several partial replication protocols, such as [8, 9], do not exhibit this property, thus again hampering scalability. On the other hand, some genuine protocols proposed in literature, such as [7], require read-only transactions to undergo a remote validation phase. This is also quite undesirable from a performance perspective, especially for geographically dispersed infrastructures, given the predominance of read-intensive workloads in typical applications [10].

Genuine protocols guaranteeing relatively strong consistency levels, while also avoiding the validation of read-only transactions have been recently presented in [11, 12]. However, these protocols do not ensure One-Copy Serializability (1CS). In particular, the protocol in [11] only ensures Extended Update-Serializability (EUS), which allows different client applications, during the execution of read-only transactions, to observe the commits of non-conflicting update transactions as serialized in different orders. Similar considerations can be made for the protocol in [12], with the additional note that the above anomalies can also involve the commits observed by update transactions.

In this paper we present SCORe, namely a scalable one-copy serializable replication protocol. SCORe overcomes the above drawbacks by employing a genuine partial replication scheme which guarantees that read-only transactions always observe a consistent snapshot of the data, hence avoiding to incur in expensive remote validation phases. This result is achieved by combining a local multiversion concurrency control algorithm with a highly scalable distributed logical-clock synchronization scheme that only requires the exchange of a scalar clock value among the nodes involved in the handling of a transaction. All the above features jointly allow SCORe to be performance effective, highly scalable,

and able to provide supports for a wider set of applications, including those that impose strict data consistency requirements.

We have implemented SCORe within Infinispan, a mainstream open source data grid framework developed by Red Hat [13]. We have assessed the effectiveness of SCORe via an extensive experimental study based on both the TPC-C [14] and YCSB [15] benchmarks, using as experimental testbeds a private cluster with up to 20 nodes, and a public cloud (FutureGrid) with up to 100 nodes. Major outcomes from the study entail demonstrations of linear scalability by SCORe across a wide range of workloads, and no overheads compared to state of the art genuine partial replication solutions [11] guaranteeing weaker consistency semantics (namely EUS).

The remainder of this paper is organized as follows. In Sect. 2 we discuss related work. The model of the system we are targeting is provided in Sect. 3. The SCORe protocol is presented in Sect. 4. The proof of correctness is provided in Sect. 5. The results of the experimental analysis are reported in Sect. 6.

2 Related Work

The issue of transactional systems replication has been thoroughly addressed in literature. Most of the existing proposals have been targeted at the case of full replication, where a copy of each data item is retained at each involved site. In this context, solutions have been provided coping with aspects such as protocol specification [16], and design of replication architectures based on middleware level approaches [17, 18] and/or on extensions of the inner logic of individual transactional systems [16]. Comparative studies [19] have demonstrated how the solutions that coordinate the replicas via total order group communication primitives, such as [20, 21], exhibit the potential for improved performance levels. Also, total order based protocols relying on speculative transaction processing schemes, such as [22, 23], have been shown to further reduce the impact of distributed synchronization on both latency and throughput. On the other hand, compared to all these proposals, in this paper we address performance and scalability of the replicated system from an orthogonal perspective since our focus is on architectures making use of partial data replication, as opposed to full replication.

When considering partial replication schemes, literature proposals can be grouped depending on (i) whether they can be considered genuine, and on (ii) the specific consistency guarantees they provide. The works in [8, 9] provide non-genuine protocols where the commitment of a transaction requires interactions with all the sites within the replicated system. Compared to these approaches, genuine partial replication schemes have been shown to achieve significantly higher scalability levels [11]. The protocol in [7] provides a genuine solution also supporting strict consistency, namely 1CS. However, differently from the present proposal, this protocol imposes that read-only transactions undergo a distributed validation phase. Also, these transactions are potentially subject to rollback/retry. Instead, the SCORe protocol we propose never aborts read-only

transactions, since it guarantees that they always observe a consistent snapshot of data, and consequently spares them from expensive remote validations.

Analogously to SCORe, the solutions proposed in [11, 12] are genuine and do not require read-only transactions to be remotely validated. However, differently from SCORe, they do not guarantee that read operations behave as if they were performed within transactions executed on a given serial schedule. For the protocol in [11], this anomalous behavior can occur only for read-only transactions, while for the protocol in [12] it may arise also for update transactions. Overall, both these protocols target weaker consistency semantics than SCORe. Similar arguments can be used when comparing SCORe with the recent proposal in [24], which does not guarantee strong consistency in the case of read operations performed on nodes maintaining distinct partitions of the replicated data.

As for the reliance on multiversions, our proposal is also related to the one in [25], where a multiversion concurrency control mechanism is provided in order to cope with distributed transaction processing in the context of distributed software transactional memories. However this protocol does not cope with (partially) replicated data and it guarantees Snapshot Isolation (SI).

3 Model of the Target System

We consider a classic asynchronous distributed system model composed of $\Pi = \{N_1, \ldots, N_n\}$ nodes, each one representing a transactional process within the replicated system. We consider the classic crash-stop failure model. Hence, nodes may fail by crashing, but never behave maliciously. A node that never crashes is said to be correct, otherwise it is said to be faulty. We assume that nodes only communicate through message passing, thus not having access to a shared memory nor to a global clock. Messages are delivered via reliable asynchronous channels, i.e., messages are guaranteed to be eventually delivered unless either the sender or the receiver crashes. However, messages may experience arbitrarily long (but finite) delays, and we assume no bound on relative process speeds or clock skews.

We assume a simple key-value model for the data maintained by the nodes in Π. Also, data are assumed to be multiversioned, hence each data item d, maintained by whichever node, is represented as a sequence of versions $\langle k, val, ver \rangle$, where k is a key representing d's identifier, val is its value and ver is a scalar, monotonically increasing logical timestamp that identifies (and totally orders) the versions of data item d. Each node N_i is assumed to store a partial copy of the whole data set. We abstract over the data placement policy by assuming that data are subdivided across m partitions, and that each partition is replicated across r nodes. We denote with $\Gamma = \{g_1, \ldots, g_m\}$ the set of groups of nodes belonging to Π, where g_j represents the group of those nodes that replicate the j-th data partition. Each group is composed of exactly r nodes (the value of r being selected in order to ensure the target replication degree), of which at least one is assumed to be correct. Given a data item d, we denote as $replicas(d)$ the set of nodes that maintain a replica of d, namely the nodes of group g_j that repli-

cate the data partition containing d. The same notation is used to indicate sets of nodes maintaining replicas of sets of data items. As an example, $replicas(S)$, with $S = \{d, d'\}$, is used to indicate the set of nodes maintaining a copy of d or a copy of d'.

In order to maximize flexibility of the data placement strategy, we do not require groups to be disjoint (they can have nodes in common), and assume that a node may belong to multiple groups, as long as $\bigcup_{j=1...m} g_j = \Pi$. We highlight that the assumed partitioning model allows capturing a wide range of data distribution algorithms, and, in particular, algorithms based on consistent hashing, which are very popular in NoSQL transactional data stores thank to their ability to: (i) minimize data transfers upon joining/leaving of nodes (which, for ease of presentation, we do not model explicitly in this work, although we will briefly discuss how to cope with dynamic groups in Sect. 4.4); (ii) ensure the achievement of target replication degrees; and (iii) avoid distributed lookups to retrieve the identities of the group of processes storing the replicas of the requested data items.

We model transactions as a sequence of read and write operations on data items, which are preceded by a begin operation, and are followed by a commit or an abort operation. A transaction can be originated on whichever node $N_i \in \Pi$, and can read/write data belonging to any partition. Also, we do not assume any a-priori knowledge on the set of data items that will be read or written by transactions. In addition a history over a set of transactions consists of a partial order of events that reflects the operations (begin, read, write, abort, commit) of those transactions.

4 The SCORe Protocol

4.1 Overview

SCORe is a genuine (hence highly scalable) partial replication protocol that implements a one-copy serializable distributed multiversion scheme. As in typical non-distributed multiversion algorithms [26], SCORe replicas store multiple versions of the data items that they maintain, each tagged with a scalar timestamp. However, SCORe introduces a novel distributed timestamp management scheme that addresses two main issues: (i) establishing the snapshot visible by transactions, i.e. selecting which one, among the multiple versions of a datum (replicated across multiple nodes) should be observed by a transaction upon a read operation; (ii) determining the final global serialization order for update transactions via a distributed agreement protocol that takes place during the transactions' commit phase.

To this end SCORe maintains two scalar variables per node, namely $commitId$ and $nextId$. The former one maintains the timestamp that was attributed to the last update transaction when committed on that node. $nextId$, on the other hand, keeps track of the next timestamp that the node will propose when it will receive a commit request for a transaction that accessed some of the data that it maintains.

Snapshot visibility for transactions is determined by associating with each transaction T a scalar timestamp, which we call *snapshot identifier* or, more succinctly, *sid*. The *sid* of a transaction is established upon its first read operation. In this case the most recent version of the requested datum is returned, and the transaction's *sid* is set to the value of $commitId$ at the transaction's originating node, if the read can be served locally. Otherwise, if the requested datum is not maintained locally, $T.sid$ is set equal to the maximum between $commitId$ at the originating node and $commitId$ at the remote node from which T reads. From that moment on, any subsequent read operation is allowed to observe the most recent committed version of the requested datum having timestamp less than or equal to $T.sid$, as in classical multiversion concurrency control algorithms.

SCORe relies on a genuine atomic commit protocol that can be seen as the fusion of the Two-Phase Commit algorithm (2PC) and the Skeen's total order multicast [6]. 2PC is used to validate update transactions and to guarantee the atomicity of the application of their post-images. Overlapped with 2PC, SCORe runs a distributed agreement protocol, similar in spirit to Skeen's total order multicast algorithm, which allows to achieve a twofold goal: (i) totally ordering the commit events of transactions that update any data item in a partition j among all the nodes that replicate j (namely, g_j); (ii) tracking the serialization order between *update* transactions that exhibit (potentially transitive) data dependencies by totally ordering them via a scalar *commit timestamp* that is also used as version identifier of the post-images of committed transactions.

A key mechanism used in SCORe to correctly serialize transactions, and in particular to track write-after-read dependencies [26], is to update the $nextId$ of a node upon the processing of a read operation. Specifically, if a node receives a read operation from a transaction T having a *sid* larger than its local $nextId$, this is advanced to $T.sid$. This mechanism allows to guarantee that any update transaction T^{up} that requests to commit on node N_i at time t is attributed a commit timestamp larger than the timestamp of any transaction T that read a value from N_i before time t, hence ensuring that T^{up} is serialized after T.

Finally, since a transaction is attributed a snapshot identifier upon its first read, which is used throughout its execution, SCORe guarantees that the snapshot read by a transaction is always consistent with respect to a prefix of the (equivalent serial) history of committed transactions. As a consequence, in SCORe read-only transactions never abort and do not need to undergo any distributed validation.

The pseudocode of the SCORe protocol is reported in Algorithms 1, 2, 3, 4, and discussed and analyzed in the following. For the sake of presentation, we will first assume that the transaction's coordinator does not crash, and then discuss how to relax this assumption in Sect. 4.4.

4.2 Handling of Read and Write Operations

SCORe buffers write operations of transactions in a private writeset (denoted as *ws* in Algorithm 1), which is only made visible upon transaction's commit.

Read operations on a datum d first check whether d has already been updated by the transaction, returning in this case the value present in the transaction's writeset. Otherwise, it is necessary to establish which of the versions of d is visible to the transaction. As already mentioned, transactions establish the *sid* that they use to determine version's visibility upon their first read. If this read operation is local, the transaction's *sid* is simply set equal to the originating node's $commitId$. Otherwise, it is set equal to the maximum between the $commitId$ of the remote node from which the data is read and the $commitId$ of the transaction's originating node. Further, if the transaction's *sid* is higher than the node's $nextId$, the latter is set equal to $T.sid$. This ensures that update transactions that subsequently issue a commit request on that node are serialized after T.

Next, the version visible by transaction T is determined, as in conventional MVCC algorithms [26], by selecting the most recent version having commit timestamp less than T's snapshot identifier. Before doing so, however, T first waits for the completion of the commit phase of any transaction T' that i) is updating d, and ii) is currently in its commit phase. In fact, in case T' is committed successfully, as it will be clearer in the following, it might be attributed a timestamp smaller than $T.sid$. Hence, T' would be totally ordered before T and the version of d created by T' would be visible to T. If T' aborted, on the other hand, T should not see its updates. In order to enforce the correct tracking of this read-after-write dependence, SCORe forces any transaction T reading a data item d to wait until there are no longer transaction commit events pending on d and with a (either final or temporary) commit timestamp smaller than $T.sid$

The logic for handling remote read operations is defined by Algorithm 2. It is worthy to highlight that, even though transactions update their own *sid* only upon their first read operation, a node attempts to advance its local timestamps $commitId$ and $nextId$ whenever it receives a message (associated with the request or the response of a read operation) from another node in the system informing it that snapshots with higher timestamps have been already committed. This mechanism, which aims to maximize the freshness of visible snapshots, is encapsulated by the *updateNodeTimestamps* function. This function advances immediately the $nextId$ timestamp, which is used to determine the timestamp proposed for future commit requests. However, additional care needs to be taken before advancing the node's $commitId$ timestamp. As this timestamp determines the (minimum) snapshot visible by locally generated transactions, in fact, it can be increased to a new value, say $commitId'$, only if it is found that there are no committing transactions that may be given a timestamp less than or equal to $commitId'$.

Finally, SCORe includes a simple, yet effective, optimization that consists in immediately aborting update transactions which, based on their snapshot identifier, are forced to observe, upon a read operation, data item versions that have been already overwritten by more recently committed transactions.

Algorithm 1 Begin, read and write events (node N_i).

```
upon Write(Transaction T, Key k, Value val) do
    T.ws ← T.ws \ {< k, − >} ∪ {< k, val >};

upon Value Read(Transaction T, Key k) do
    if (∃ < k, val >∈ T.ws) then
        return val;
    if (is first read of T) then
        T.sid ← N_i.commitId;
    if N_i ∈ replicas(k) then
        < val, maxCommitted, mostRecent >← doRead(T.sid, k);
    else
        if (is first read of T) then
            send ReadRequest[T, k, T.sid, ⊤]) to all N_j ∈ replicas(k);
        else
            send ReadRequest[T, k, T.sid, ⊥]) to all N_j ∈ replicas(k);
        wait receive ReadReturn[T, val, maxCommitted, mostRecent] from N_j ∈ replicas(k);
    if (is first read of T) then
        T.sid ← maxCommitted;
    if T.isUpdate ∧ ¬mostRecent then
        T.abort();
    T.rs ← T.rs ∪ {< k, val >};
    return val;

function < Value, SnapshotId, boolean > doRead(SnapshotId sid, Key k)
    // Track write-after-read dependence
    N_i.nextId ← max(N_i.nextId, readSid);
    // Enforce read-after-write dependence
    wait until (N_i.commitId ≥ readSid ∨ k.exclusiveUnlocked());
    Version ver ← k.getLastVersion();
    while ver.vn > sid do
        ver ← ver.prev;
    return < ver.value, N_i.commitId, k.isLastVersion(ver) >;
```

Algorithm 2 Handling of remote reads (node N_i).

```
upon receive ReadRequest[T, k, readSid, firstRead]) from N_j do
    SnapshotId newReadSid ← readSid;
    if firstRead ∧ N_i.commitId > newReadSid then
        newReadSid ← N_i.commitId;
    < Value, mostRecent > val ← doRead(newReadSid, k);
    send ReadReturn[T, val, mostRecent, N_i.commitId];
    updateNodeTimestamps(readSid);

upon receive ReadReturn[T, val, lastCommitted, mostRecent] from N_j do
    updateNodeTimestamps(lastCommitted);

function updateNodeTimestamps(SnapshotId lastCommitted);
    // Update global snapshot knowledge
    N_i.nextId ← max(N_i.nextId, lastCommitted);
    N_i.maxSeenId ← max(N_i.maxSeenId, lastCommitted);

upon N_i.maxSeenId > N_i.commitId ∧ pendQ.isEmpty() ∧ stableQ.isEmpty() atomically do
    N_i.commitId ← max(N_i.maxSeenId, N_i.commitId);
```

4.3 Commit Phase

As already mentioned, with SCORe read-only transactions can be committed without undergoing distributed validation phases (unlike, for instance, in [7]).

Update transactions, on the other hand, execute a Two-Phase Commit protocol, which is detailed in the following. To guarantee genuineness, SCORe involves in the commit phase of a transaction T only the nodes that maintain replicas of the data items that T accessed. More in detail, when a node N_i requests to commit transaction T, it broadcasts a PREPARE message to all nodes N_j belonging to $Replicas(T.rs \cup T.ws)$. Upon the receipt of this message, node N_j verifies whether the transaction can be serialized after every transaction that has locally committed so far. To this end, it attempts to acquire exclusive, resp. shared, locks for the data in T's writeset, resp. readset, that it locally maintains. This lock acquisition is non-blocking since the node waits for a busy lock only for a certain amount of time, which is determined by means of a configurable timeout parameter. Next, if the acquisition of the locks succeeds, the node validates T's readset, verifying that none of the items read by T has been overwritten by a more recently committed transaction. If any of these operations fails, T is simply rolled back, which will yield to the abort of the whole distributed transaction, as in classic 2PC.

If the transaction passes the validation phase, however, the VOTE message of 2PC is exploited to overlap a distributed agreement scheme similar in spirit to Skeen's total order multicast algorithm that aims to establish the final serialization order for the transaction. More in detail, N_j increments the $nextId$ timestamp, inserts the pair $< T, N_j.nextId >$, defined on the domain $TransactionId \times SnapshotId$ in a queue of pending committing transactions (denoted as $pendQ$) ordered by $SnapshotId$, and sends back to the transaction coordinator the value of $N_j.nextId$ in piggyback to the VOTE message. The coordinator gathers the VOTE messages (aborting the transaction in case one of the contacted node does not respond within a predefined timeout), determines the final commit timestamp for T as the maximum among the timestamps proposed by the transaction's participants, and broadcasts back a DECIDE message with the transaction's final commit timestamp.

Upon the receipt of the DECIDE message with a positive outcome, unlike classical 2PC, the transaction is not necessarily immediately committed. In fact, as each data item is replicated over more than one node, and since we want to ensure 1CS without requiring the validation of read-only transactions, SCORe guarantees that the commit events of all update transactions (even non-conflicting ones) are totally ordered across all the replicas of a same partition. To ensure this result, when a DECIDE message is received on N_j for transaction T with final commit timestamp fsn, T is removed from the pending queue and is immediately committed (atomically increasing $N_j.nextId$) only if there are no other transactions in both the pending queue and a second queue, denoted as $stableQ$, with snapshot id less than fsn. If this is not the case, T is buffered in $stableQ$, which is ordered by $SnapshotId$ as well, till it can be ensured that no other pending transaction will ever receive a final commit snapshot id less than fsn (see Algorithm 4).

We conclude by remarking that the idea of intertwining an atomic commit algorithm and the Skeen's total order multicast algorithm was, to the best of

our knowledge, first employed in our recent proposal GMU [11]. Differently from SCORe, however, GMU relies on a vector-clock-based timestamping mechanism that guarantees a weaker consistency criterion (Extended Update-Serializability [10]).

Algorithm 3 Commit phase (node N_i).

```
upon boolean Commit(Transaction T) do
    if T.ws = ∅ then
        return ⊤;
    boolean outcome ← ⊤;
    Set proposedSn ← ∅;
    send PREPARE[T, T.sid, T.rs, T.ws] to all N_j ∈ replicas(T.rs ∪ T.ws)
    for all N_j ∈ T.involvedNodes do
        wait receive VOTE[T, sn, res] from N_j or timeout;
        if (res = ⊥ ∨ timeout) then
            outcome ← ⊥;
            break;
        else
            proposedSn ← proposedSn ∪ sn;
    T.sid ← max(proposedSn);
    send DECIDE[T, T.sid, outcome] to all N_j ∈ T.involvedNodes
    wait until T.completed = ⊤;
    return T.outcome;

upon receive PREPARE[T, sid, rs, ws]) from N_j
    boolean outcome ← (getExclLocksWithTimeout(ws) ∧ getSharedLocksWithTimeout(rs)
∧ validate(rs, sid)); SnapshotIdsn ← NULL_SID
    if outcome then
        sn ← N_i.nextId ← N_i.nextId + 1;
        pendQ ← pendQ ∪ {< T, sn >};
    send VOTE [T, sn, outcome] to N_j;

upon receive DECIDE[T, fsn, outcome]) from N_j atomically do
    if outcome then
        N_i.nextId ← max(N_i.nextId, fsn);
        stableQ ← stableQ ∪ {< T, fsn >};
    pendQ ← pendQ \ {< T, − >};
    if ¬outcome then
        releaseSharedLocks(T.rs);
        releaseExclusiveLocks(T.ws);
        if T.origin = N_i then
            T.outcome ← ⊥;
            T.completed ← ⊤;

boolean validate(Set readSet, SnapshotId sid) do
    for all k ∈ readSet do
        if k.getLastVersion().vn > sid; then
            return ⊥;
    return ⊤;
```

4.4 Garbage Collection and Fault-Tolerance

For space constraints we can only briefly overview which standard mechanisms could be integrated in SCORe to deal with garbage collection of obsolete data versions and fault-tolerance.

Algorithm 4 Finalizing the commit phase of transaction T (node N_i).

```
1: upon ∃ < T, fsn > : {< T, fsn > = stableQ.head ∧
2:           (∄ < T', sn > : < T', sn > = pendQ.head ∧ sn < fsn} atomically do
3:     apply(T.ws, fsn);
4:     releaseSharedLocks(T.rs);
5:     releaseExclusiveLocks(T.ws);
6:     stableQ ← stableQ \ {< T, fsn >};
7:     if T.origin = N_i then
8:         T.outcome ← ⊤;
9:         T.completed ← ⊤;
10:
```

As in non-distributed MVCC algorithms, versions of a data item d having timestamps less than the *sid* of any active transaction can be safely removed, provided that most recent versions of d have already been committed. In a distributed platform, it is required to disseminate the information on the *sid* of the oldest active transaction at each node. This information can be spread by relying, e.g., on lazy approaches based on piggybacking or gossip [27].

For simplicity, we have opted to present SCORe as layered on top of a 2PC protocol, which is well known to be blocking upon failure of the coordinator. However, the issue of how to ensure high availability of the transaction coordinator state is well understood, and a range of orthogonal solutions have been proposed in literature to deal with such failure scenarios. One may use, for instance, protocols such as Paxos Commit [28] or other consensus based abstractions [29], to replicate the state of the coordinator of a transaction T across the replicas of any of the data partitions accessed by T. Note that, as we are assuming that at least one process is correct for each replica group, failures of transactions' participants will not lead to blocking scenarios during the execution of a remote read operation. Failures of transactions' participants can, instead, lead to aborts during the commit phase, as the coordinator unilaterally aborts the transaction if it times out while waiting for some reply during the prepare phase. To ensure the liveness of the commit protocol, SCORe relies on an underlying Group Communication System [5] in order to handle the removal of faulty replicas from the system and manage its reconfiguration, which may imply the re-distribution of data across replicas to guarantee a desirable replication degree.

Aiming at ensuring 1CS, SCORe opts for sacrificing availability (by aborting transactions that span remote nodes) in order to ensure consistency in presence of network partitions. This is not surprising, given the existence of well known results, such as the CAP theorem [30], concerning the impossibility of achieving both availability and consistency in presence of partitions.

Finally, SCORe does not introduce additional issues concerning the management of dynamic process groups with respect to classic 2PC-based transactional replication systems. Conversely, its supports for multiversion simplify significantly the design of state-transfer mechanisms [31] aimed to synchronize the state of newly joining nodes.

5 Correctness Proof

Preliminary definitions. Let us start by briefly recalling some basic notions and nomenclature on multiversioned histories [26]. Let us denote as x_i the version of data item x committed by transaction T_i. A multiversioned history H defines a partial order on the operations executed by transactions on the multiversioned dataset, and each operation can be a *read*, a *write*, a *begin*, a *commit* or an *abort* operation. We use the following notation for the five types of operations: b_i denotes the begin of a transaction T_i, while c_i and a_i represent respectively its commit and its abort; the notation $r_i(x_j)$ is used to indicate the transaction T_i performs a read on the version x_j, while $w_i(x_i)$ denotes a write of a new version x_i issued by transaction T_i. In addition, a multiversioned history H implicitly defines a total order $\ll_x$ for each data item x. A version order $\ll$ on H is the union of the $\ll_x$ for each x in H.

Given a multiversioned history H and a version order $\ll$ on the written data item versions, a *Direct Serialization Graph* $DSG(H, \ll)$ (as in [10, 26]) is a direct graph having a vertex V_{T_i} for each committed transaction T_i in H (i.e. c_i is in H) and a direct edge $V_{T_i} \xrightarrow{E} V_{T_j}$ from a vertex V_{T_i} to a vertex V_{T_j} if one of the following statements holds:

- T_j *directly read-depends on* T_i *(*$V_{T_i} \xrightarrow{wr} V_{T_j}$*).* There exists a data item x such that both $w_i(x_i)$ and $r_j(x_i)$ are in H.
- T_j *directly write-depends on* T_i *(*$V_{T_i} \xrightarrow{ww} V_{T_j}$*).* There exists a data item x such that both $w_i(x_i)$ and $w_j(x_j)$ are in H and version x_i immediately precedes version x_j according to the total order defined by $\ll$.
- T_j *directly anti-depends on* T_i *(*$V_{T_i} \overset{rw}{\dashrightarrow} V_{T_j}$*).* There exists a data item x and a committed transaction T_k in H such that $k \neq i$, $k \neq j$, both $r_i(x_k)$ and $w_j(x_j)$ are in H and version x_k immediately precedes version x_j according to the total order defined by $\ll$.

A multiversioned history H is *One-Copy Serializable* iff there exists a version order $\ll$ such that the $DSG(H, \ll)$ graph does not contain any oriented cycle.

One-Copy Serializability. Our proof is based on establishing a mapping between each vertex V_{T_i} in $DSG(H, \ll)$ and the value of the commit timestamp of T_i, denoted as $commitSId(T_i)$. We prove the acyclicity of the $DSG(H, \ll)$ by showing that for each edge $V_{T_i} \xrightarrow{E} V_{T_j} \in DSG(H, \ll)$ SCORe guarantees that $commitSId(T_i) \leq commitSId(T_j)$.

Note that, if T_i is a read-only transaction, $commitSId(T_i)$ is equal to the *sid* assigned to T_i upon its first read operation. On the other hand, in case T_i is an update transaction, $commitSId(T_i)$ is computed during T_i's commit phase and is equal to the maximum identifier among the ones proposed by the nodes involved in the commit of T_i .

Let us start by assuming that E is a direct write-dependence edge, and show that SCORe ensures that $commitSId(T_i) < commitSId(T_j)$. This is because T_i and T_j are both update transactions and they commit on a common subset S of

the nodes in the system (at least the nodes storing the data item on which the write-dependence is materialized). In fact, in accordance with the design of the commit phase, it is ensured that: (i) T_j cannot enter the commit phase of the protocol before T_i has committed, since T_j has to wait for the release of some exclusive lock owned by T_i at least on the nodes in S; (ii) T_i updates the $nextId$ on the nodes in S to a value at least equal to $commitSId(T_i)$ before finalizing its commit; (iii) the $commitSId(T_j)$ is chosen as the maximum among the $nextId$ values, incremented by one, of the nodes involved in the commit of T_j.

Now assume that E is a direct read-dependence edge. This means that T_j has read a version committed by T_i. Therefore the snapshot identifier used by T_j to perform read operations, i.e. $T_j.sid$, is greater than or equal to the T_i's commit snapshot identifier due to the reading rule defined by the protocol. So, if T_j is a read-only transaction, this entails that $commitSId(T_i) \leq T_j.sid = commitSId(T_j)$; otherwise, if T_j is an update transaction its commit snapshot identifier will be always greater than its reading snapshot identifier, since the value proposed by each node involved in the commit of T_j (i.e. the incremented $nextId$) is greater than every snapshot seen by T_j. As a consequence, $commitSId(T_i) < commitSId(T_j)$ holds.

Finally, if E is a direct anti-dependence edge, we have to distinguish two scenarios. In the former, if T_i is a read-only transaction, then the $commitSId(T_j)$ is greater than $commitSId(T_i)$ since (i) the T_j's commit snapshot identifier is at least equals to all the values proposed for its commit and (ii) there exists a value among the one proposed that is guaranteed to be greater than T_i's reading snapshot identifier (i.e. $commitSId(T_i)$ in this scenario) due to the visibility rule adopted on each read operation of T_i. In particular, T_i performs a read operation on a data item x of a node N only after it has ensured that (i) the $nextId$ value on N will be greater than its reading snapshot identifier and (ii) no transaction will commit an update on x using a snapshot id not greater than $commitSId(T_i)$. Otherwise, if T_i is an update transaction, it is guaranteed that at the time T_j commits, T_i has been already successfully committed otherwise T_i's read-set would have been invalidated by T_j. This case is analogous to the one in which E is a write-dependence edge since we have two update transactions, T_i and T_j, that commit on a common subset of nodes S, and T_i commits before T_j; therefore $commitSId(T_i) < commitSId(T_j)$ holds.

Executing 1CS. Indeed, the SCORe protocol provides a consistency criterion stronger than 1CS. In fact, the protocol ensures that the read operations issued by every transaction $T \in H$, even those that eventually abort, observe the state generated by a sequential history equivalent to H. This is verifiable by considering that: (i) since a write operation is externalized only upon a successful commit, a live or an aborted transaction at time t can be considered as a read-only committed transaction that contains its read prefix performed until t, except the operation which has triggered an abort (if any); (ii) the $DSG(H, \ll)$ graph has a node for each committed transaction or an aborted/live transaction reduced to its read prefix.

This property, which was also called Executing 1CS by Adya [10], is also implied by the more recent Opacity [34] property. However, it is easy to show that, since SCORe fixes the timestamp of a transaction upon its first read operation, it does not guarantee real-time ordering, as required by Opacity.

6 Experimental Data

In this section we report the results of an experimental study aimed at evaluating the performance and scalability of SCORe. This study is based on a prototype implementation of SCORe[3] that has been integrated within the Infinispan data grid system, a JAVA based open source NoSQL data platform developed by Red Hat [13]. Similarly to what done by other distributed, in-memory data platforms, Infinispan externalizes a simple key-value store interface. Also, it targets scalability by natively relying on weak data consistency models, and on a lightweight consistent hashing scheme [32], which allows partitioning data efficiently across the nodes, while ensuring good load balancing and minimum reshuffling of keys in presence of joins/departures of nodes from the platform. Further, Infinispan natively supports partial replication, allowing to store each key across a fixed, user-tunable number of replicas.

The strongest consistency level ensured by Infinispan is Repeatable Read [2] (RR), which guarantees that no intermediate or aborted values are ever observed, and that no two reads on the same key within the same transaction can return different values. RR is definitely weaker than Serializability, as it allows the commit of (both read-only and update) transactions that observe non-serializable schedules [10]. To provide some more architectural details, Infinispan relies on an encounter based two-phase-locking scheme, which is applied only to write operations and that does not synchronize reads. Repeatability of read operations is instead guaranteed by storing (locally caching) the read data items, and returning the stored copies upon subsequent reads. For what concerns the native replication protocol supported by Infinispan, it relies on a classical 2PC-based distributed locking algorithm [33].

Being Infinispan designed to achieve high scalability in the context of weak data consistency models, we argue that it represents an ideal baseline to evaluate the costs incurred in by the SCORe protocol in order to provide 1CS (i.e. strong consistency) guarantees.

Benchmarks. We have evaluated SCORe using two different benchmarks. The first one is a porting of TPC-C [14] adapted to execute on a NoSQL platform such as Infinispan. TPC-C is a benchmark representative of OLTP environments, and is characterized by complex and heterogeneous transactions, with very skewed access patterns, and hence non-minimal conflict probability. In our study we configured the benchmark to generate two workloads: one including 50% of update transactions, and a second one including 90% of read-only transactions.

[3] The SCORe prototype is publicly available at the URL `http://www.cloudtm.eu`.

The second benchmark is YCSB (Yahoo! Cloud Serving Benchmark) [15], which is specifically targeted at the assessment of key-value data grids and cloud stores. This benchmark is somehow complementary to TPC-C since its transactional profile is characterized by simpler transactions that rarely conflict.

Test-bed Platforms. We performed our study on two different experimental testbeds. The first one, denoted as Cloud-TM platform, is a dedicated cluster of 20 homogeneous nodes, where each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 16 GB of RAM, running *Linux 2.6.32-33-server* and interconnected via a private Gigabit Ethernet. This platform is representative of small/medium private clouds or data-center environments, with dedicated servers and a fairly large amount of available (computational and memory) resources per node. In all the experiments performed on the Cloud-TM platform we used four threads per node to inject transactions (in closed loop), which guaranteed a high utilization of the machine's resources without overloading them, which would otherwise lead to unreliable results in terms of assessment of the distributed protocol used to handle partial replication.

The second used platform is FutureGrid, which is a public distributed testbed for parallel and cloud computing. This platform allowed us to evaluate SCORe in environments representative of public cloud infrastructures, which are typically characterized by more competitive resource sharing, ample usage of virtualization technology, and relatively less powerful virtualized nodes. On top of the FutureGrid platform we performed experiments using up to 100 virtual machines, equipped with 4GB RAM, two 2.93GHz cores Intel Xeon CPU X5570, running CentOS 5.7 x86_64. All the VMs were deployed in the same physical data-center and interconnected via Gigabit Ethernet. Also, again in order not to saturate machine's resources, in all the experiments performed on FutureGrid we used two threads per node to inject transactions (in closed loop).

Finally, for both deploys on the above described platforms, we have set the replication degree of each data item to the value 2.

Results. In Fig. 1 we show the achieved throughput values for TPC-C on top of the Cloud-TM platform while varying the number of involved nodes between 2 and 20. The plots in the top row refer to the workload composed at the 90% by read-only transactions, denoted as Workload A. The left plot reports the throughput for write transactions, whereas the right plot reports the throughput for read-only transactions. We contrast the performance of SCORe, with that of the native RR scheme supported by Infinispan, and with that of the GMU protocol presented in [11], which has also been integrated within Infinispan. As already discussed, GMU ensures a consistency criterion (namely EUS - Extended Update-Serializability [10]) weaker than 1CS, but stronger than RR. In other words, GMU exhibits intermediate consistency semantics with respect to the other two analyzed protocols.

The plots highlight that SCORe attains throughput values that are even slightly better than those achieved by GMU. This phenomenon is explainable

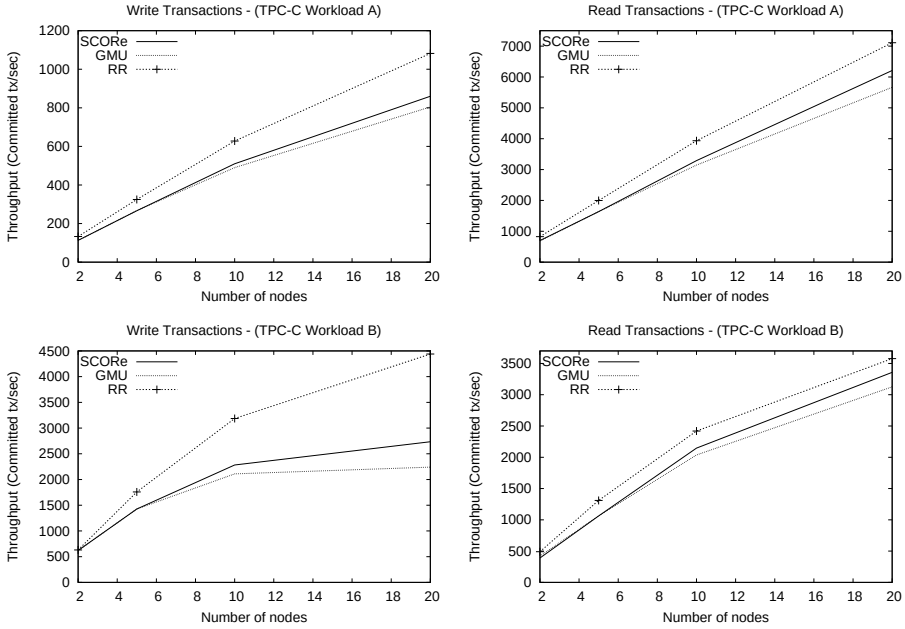

Fig. 1: TPC-C Benchmark (Cloud-TM).

by considering that, while SCORe relies on a timestamping mechanism based on scalar clock values, GMU uses vector clocks, which introduce higher overheads with respect to scalar clocks as the number of nodes in system grows.

The plot in the bottom row of Fig. 1 reports the results for TPC-C, obtained on top of the Cloud-TM platform, for the scenario encompassing 50% of read-only transactions, denoted as Workload B. While the comparative behavior of SCORe vs GMU follows trends similar to those observed for 90% read-only transactions, this time the performance loss of SCORe vs RR for update transactions grows significantly. This is essentially due to the fact that the increased volume of update transactions leads to an increased abort rate caused predominantly by failures during the validation phase of the transaction's read set (interestingly, the aborts due to failures in the lock acquisition phase turned out to be statistically marginal). In other words, as the update rate grows, the probability for an update transaction to access a stale snapshot accordingly grows. In particular, for the case of 20 nodes, the abort probability for update transactions with SCORe is on the order of 43%, while RR only exhibits around 8% abort rate for update transactions, with aborts exclusively caused by deadlocks. However, when considering the total throughput for Workload B (including both read-only plus update transactions), SCORe exhibits similar scalability trend when compared to RR. Overall, the data show that, for increased contention scenarios, strong consistency semantics do pay a performance toll, which, in this specific configuration, corresponds to a throughput reduction up to 22% (at 20

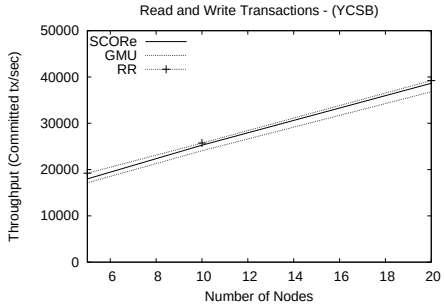

Fig. 2: YCSB Benchmark (Cloud-TM).

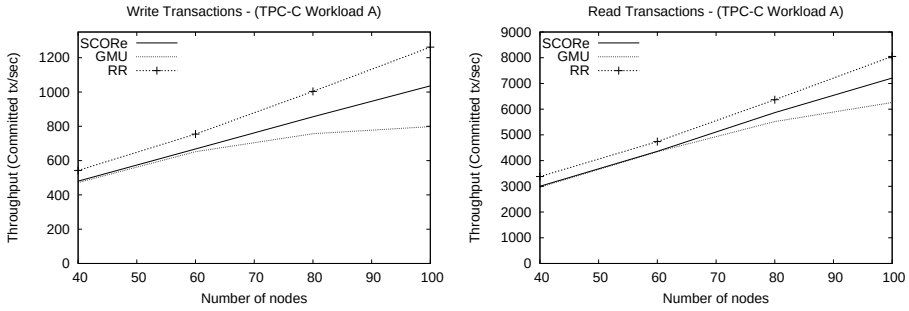

Fig. 3: TPC-C Benchmark (FutureGrid).

nodes). On the other hand, we argue that this is an unavoidable cost to pay in applications whose correctness can be endangered by adopting non-serializable isolation levels.

In Fig. 2 we show the results obtained by running YCSB on the Cloud-TM platform. We used Workload A [15] of the benchmark, which is an update intensive workload (comprising 50% of update transactions) simulating a session store that records recent client actions. We report the maximum throughput (committed transactions per second) achievable by the three considered protocols. The plot shows that the average reduction in throughput for both SCORe and GMU, compared to RR, oscillates around 8%, and that, again, the throughput scales linearly at the same rate as RR, providing an evidence of the efficiency and scalability of the proposed solution when considering transaction profiles featuring applications natively tailored for key-value data stores.

In Fig. 3 we present the results obtained by running Workload A of TPC-C on FutureGrid. The data confirm the general trends already observed on the Cloud-TM platform, highlighting both the high scalability of the proposed solution and its high efficiency when compared to vector-clock-based solutions, such as GMU, whose overheads grow linearly with the scale of the platform.

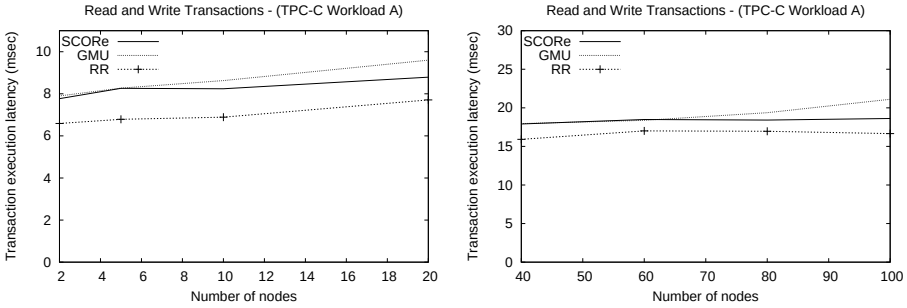

Fig. 4: Average transaction execution latency (TPC-C Workload A) for both Cloud-TM (left) and FutureGrid (right).

Finally, for completeness of the analysis, we report in Fig. 4 the average transaction execution latency for the case of TPC-C (Workload A) run on both Cloud-TM and FutureGrid. By the data we observe that, for all the protocols, latency values stay almost flat while increasing the size on the underlying platform (and consequently of the total workload sustained), which again supports the claim of good scalability of SCORe. Further, the relevance of this result is supported by the fact that all the reported values were related to scenarios where the utilization of infrastructural resources was high (as an example, for the tests with TPC-C on top of FutureGrid the CPU utilization was constantly observed to be over the 80%). Hence, the data refer to scenarios where the throughput was relatively close to the maximum sustainable one.

7 Conclusions

In this article we introduced SCORe, which is, to the best of our knowledge, the first partial replication protocol that jointly guarantees the following properties: (i) genuineness, which maximizes system scalability by demanding that only the replicas that maintain data accessed by a transaction are involved in its processing; (ii) strong consistency of the snapshots observed by read operations, thanks to a one-copy serializable multiversion scheme, which never aborts read-only transactions and that spares them from any (distributed) validation phase.

We integrated SCORe in Infinispan, a popular open source distributed data grid, and evaluated its performance by means of an experimental study relying on well-known benchmarks and on the usage of both private and public cloud infrastructures. The experimental results demonstrate that SCORe can scale up to a hundred nodes, delivering throughput and latency comparable to schemes that ensure much weaker consistency criteria.

We argue that the ability of SCORe to ensure strong consistency guarantees without hampering scalability can enlarge significantly the spectrum of applications commonly deployed on large scale NoSQL data grids.

References

1. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A. Sivasubramanian, S.,Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proc. of the 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220 (2007)
2. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proc. of the ACM SIGMOD International Conference on Management of Data, pp. 1–10 (1995)
3. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review. 44, 35–40 (2010)
4. Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: a new paradigm for building scalable distributed systems. ACM SIGOPS Operating Systems Review. 41, 159–174 (2007)
5. Defago, X., Schiper, A., Urban, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. J. ACM Computing Surveys. 36, 372–421 (2004)
6. Guerraoui, R., Schiper, A.: Genuine atomic multicast in asynchronous distributed systems. J. Theoretical Computer Science. 254, 297–316 (2001)
7. Schiper, N., Sutra, P., Pedone, F.: P-Store: Genuine Partial Replication in Wide Area Networks. In: Proc. of the 29th IEEE Symposium on Reliable Distributed Systems, pp. 214–224 (2010)
8. Armendáriz-Iñigo, J. E., Mauch-Goya, A., González de Mendívil, J. R., Muñoz-Escoí, F. D.: SIPRe: a partial database replication protocol with SI replicas. In: Proc. of the 2008 ACM Symposium on Applied Computing, pp. 2181–2185 (2008)
9. Serrano, D., Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B.: Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation. In: Proc. of the 13th Pacific Rim International Symposium on Dependable Computing, pp. 290–297 (2007)
10. Adya, A.: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. PhD Thesis, Massachusetts Institute of Technology (1999)
11. Peluso, S., Ruivo, P., Romano, P., Quaglia, F., Rodrigues, L.: When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In: Proc. of the IEEE 32nd International Conference on Distributed Computing Systems, pp. 455–465 (2012)
12. Sovran, Y., Power, R., Aguilera, M. K., Li, J.: Transactional storage for geo-replicated systems. In: Proc. of the 23th ACM Symposium on Operating Systems Principles, pp. 385–400 (2011)
13. Marchioni, F., Surtani, M.: Infinispan Data Grid Platform. PACKT Publishing (2012)
14. TPC Council: TPC-C Benchmark, Revision 5.11. (2010)
15. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proc. of the 1st ACM Symposium on Cloud Computing, pp. 143–154 (2010)
16. Kemme, B., Alonso, G.: Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In: Proc. of the 26th International Conference on Very Large Data Bases, pp. 134–143 (2000)
17. Lin, Y., Kemme, B., Patiño-Martínez, M. and Jiménez-Peris, R.: Middleware based Data Replication providing Snapshot Isolation. In: Proc. of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 419–430 (2005)

18. Thomson, A., Abadi, D. J.: The case for determinism in database systems. J. VLDB Endowment. 13, 70–80 (2010)
19. Wiesmann, M., Schiper, A.: Comparison of Database Replication Techniques Based on Total Order Broadcast. J. IEEE Transactions on Knowledge and Data Engineering. 17, 551–566 (2005)
20. Pedone, F., Guerraoui, R., Schiper, A.: The Database State Machine Approach. J. Distributed and Parallel Databases. 14, 71–98 (2003)
21. Thomson, A., Diamond, T., Weng, S., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: Proc. of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 1–12 (2012)
22. Carvalho, N, Romano, P., Rodrigues, L.: SCert: Speculative certification in replicated software transactional memories. In: Proc. of the 4th Annual International Conference on Systems and Storage, pp. 10:1–10:13 (2011)
23. Palmieri, R., Quaglia, F., Romano, P.: OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems. In: Proc. of the IEEE 30th International Symposium on Reliable Distributed Systems, pp. 59–64 (2011)
24. Baker, J., Bond, C., Corbett, J., Furman, J.J., Khorlin, A., Larson, J., Leon, J., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: Proc. of the 5th Biennial Conference on Innovative Data Systems Research, pp. 223–234 (2011)
25. Bieniusa, A., Fuhrmann, T.: Consistency in hindsight: A fully decentralized STM algorithm. In: Proc. of the 2010 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–12 (2010)
26. Bernstein, P. A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
27. van Renesse, R., Birman, K. P., Vogels, W.: Astrolabe: A Robust and Scalable Technology For Distributed Systems Monitoring, Management, and Data Mining. J. ACM Transactions on Computer Systems. 21, 164–206 (2003)
28. Gray, J., Lamport, L.: Consensus on transaction commit. J. ACM Transactions on Database Systems. 31, 133–160 (2006)
29. Frølund, S., Guerraoui, R.: Implementing E-Transactions with Asynchronous Replication. J. IEEE Transactions on Parallel and Distributed Systems. 12, 133–146 (2001)
30. Brewer, E. A.: Towards robust distributed systems (abstract). In: Proc. of the 19th annual ACM Symposium on Principles Of Distributed Computing, p. 7 (2010)
31. Jiménez-Peris, R. , Patiño-Martínez, M., Alonso, G.: Non-Intrusive, Parallel Recovery of Replicated Data. In: Proc. of the 21st IEEE Symposium on Reliable Distributed Systems, pp. 150–159 (2002)
32. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: Proc. of the 29th annual ACM Symposium on Theory of Computing, pp. 654–663 (1997)
33. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proc. of the ACM SIGMOD International Conference on Management of Data, pp. 173–182 (1996)
34. Guerraoui, R. , Kapalka, M.: On the Correctness of Transactional Memory. In: Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 175–184 (2008)