Asynchronous Lease-Based Replication of Software Transactional Memory*

Nuno Carvalho, Paolo Romano, and Luís Rodrigues

INESC-ID/IST
{nonius,romanop}@gsd.inesc-id.pt, ler@ist.utl.pt

Abstract. Software Transactional Memory (STM) systems have emerged as a powerful middleware paradigm for parallel programming. At current date, however, the problem of how to leverage replication to enhance dependability and scalability of STMs is still largely unexplored. In this paper we present Asynchronous Lease Certification (ALC), an innovative STM replication scheme that exploits the notion of *asynchronous lease* to reduce the replica coordination overhead and shelter transactions from repeated abortions due to conflicts originated on remote nodes. These features allow ALC to achieve up to a tenfold reduction of the commit latency phase in scenarios of low contention when compared with state of the art fault-tolerant replication schemes, and to boost the throughput of long-running transactions by a 4x factor in high conflict scenarios.

Keywords: Dependability, Software Transactional Memory, Replication, Leases.

1 Introduction

The advent of multi-core architectures has decreed the end of the free performance gains' era for single-threaded applications. Hereafter, unleashing the full potential of multi-core processors demands a radical shift in the way software is developed, moving parallel programming from the niche of scientific and high performance computing to mainstream application domains. Building on the abstraction of atomic transactions, and freeing the programmer from the complexity of conventional lock-based synchronization schemes, Transactional Memory (TM) allows simplifying the development and verification of concurrent programs, enhancing code reliability and boosting productivity [7]. Over the last years, a wide body of literature has been developed in the area of STMs and, recently, the first real-world, enterprise-class STM-based applications have started to be deployed in production systems [7,33]. One of the key lessons learnt from the development and deployment of these applications [32] is that existing

* This work has been partially supported by the project "Cloud-TM" (co-financed by the European Commission through the contract no. 257784), the FCT project ARISTOS (PTDC/EIA- EIA/102496/2008) and by FCT (INESC-ID multiannual funding) through the PIDDAC Program Funds.

I. Gupta and C. Mascolo (Eds.): Middleware 2010, LNCS 6452, pp. 376–396, 2010.

STM platforms suffer of a significant limitation: the lack of efficient replication schemes capable of fulfilling the scalability and reliability requirements of real-world, service-oriented applications.

Replication of Transactional Memory systems represents a promising new approach for building fault-tolerant distributed systems, providing powerful building blocks that shelter programmers from the complexity of dealing with machine failures and developing lock-based (distributed) synchronization schemes. At current date, only a handful of solutions have been proposed and evaluated [24,5,27,11]. On the other hand, since transactional memory and databases share the common notion of atomic transaction, the large body of literature developed in the area of replicated databases represents a natural source of inspiration for the design of replication schemes for transactional memory. Among the plethora of database replication schemes, recent approaches based on Atomic Broadcast (AB) [18] and distributed certification procedures [31,23,30], appear to be particularly attractive for employment in a TM context. In fact, unlike classic eager replication schemes (based on fine-grained distributed locking and atomic commit), that suffer of large communication overheads and fall prey of distributed deadlocks [15], certification based schemes avoid the costs of replica coordination during the execution phase, running transactions locally in an optimistic fashion. The consistency of replicas (typically, 1-Copy serializability [4]) is ensured at commit-time, via a distributed certification phase that uses AB to enforce agreement on a common transaction serialization order, avoiding distributed deadlocks, and providing non-blocking guarantees in the presence of replicas' failures.

Unfortunately, as previously observed in [11,32] (and confirmed by the experimental results presented later in this paper), the overhead of previously published certification schemes based on AB can be particularly detrimental in STM environments. In fact, unlike in classical database systems, STMs incur neither in disk access latencies nor in the overheads of SQL statement parsing and plan optimization. This makes the execution time of typical STM transactions normally much shorter than in database settings [32], making the cost of inter-replica coordination a major source of overhead. Further, distributed certification schemes are based on an inherently optimistic approach: transactions are only validated at commit time and no bound is provided on the number of times that a transaction will have to be re-executed due to the occurrence of conflicts. This can lead to undesirably high abort rates in high conflict scenarios or with heterogeneous workloads that contain mixes of short and long-running transactions (as it is actually the case for several well-known TM benchmarks [17,2]). In this case, the latter ones may be repeatedly aborted due to the occurrence of (remote) conflicts with a stream of short-lived transactions, leading to fairness violation that might be regarded as unacceptable by the users of interactive applications.

In this paper we tackle the above issues by presenting the Asynchronous Lease Certification (ALC) protocol. In the core of the ALC scheme is the notion of *asynchronous lease*. Analogously to classic lease schemes [13,14], asynchronous leases are used by a replica to establish temporary privileges in the management of a

subset of the replicated data-set. Specifically, in ALC, the ownership of an asynchronous lease on a set of data items provides a replica with two key benefits: *i*) reducing the commit phase latency of the transactions that access those data items and; *ii*) sheltering transactions by repeated abortions due to remote conflicts.

While the ALC protocol may rely on any STM for locally regulating the concurrent execution of transactions, we chose to integrate the ALC scheme with a multi-versioned STM, namely JVSTM [8]. This allows sheltering read-only transactions from the possibility of aborts (due both to local or remote conflicts), as well as to prevent them from incurring in stalls due to concurrent conflicting accesses. Through an extensive experimental evaluation, based on both synthetic micro-benchmarks, as well as complex STM benchmarks we show that ALC permits to achieve up to a tenfold reduction of the commit latency, and a 4x factor increase of throughput when compared with competing replicated STMs [11].

The rest of this paper is organized as follows. Section 2 discusses related work. A description of the considered system model and of the consistency criteria ensured by ALC is provided in Section 3, which also describes the architecture of an ALC-based system and discusses the issues related to the integration with JVSTM. The ALC scheme is presented in Section 4 and Section 5 presents the results of our experimental evaluation study. Finally, Section 6 concludes the paper.

2 Related Work

The only distributed STM solutions we are aware of are those in [24,5,27,11]. Except for the work in [11], none of these solutions leverages on replication in order to ensure cluster-wide consistency and availability in scenarios of failures. In ALC, on the other hand, dependability is seen as a first class design goal, and the STM performance is optimized through a holistic approach that tightly integrates low level fault-tolerance mechanisms (such as AB) with a novel, highly efficient lease based distributed transaction certification scheme.

In our previous work [11], we introduced D^{2}STM, which is, to the best of our knowledge, the first and only fault-tolerant distributed STM platform presented up to date. D^{2}STM adopts an optimistic certification scheme, which avoids any remote synchronization during transaction's execution and relies on a commit-time AB based distributed validation to ensure global consistency. In order to reduce the AB latency, D^{2}STM uses a Bloom-filter based encoding that minimizes the amount of information to be sent through the AB primitive, at the cost of a small, tunable additional abort rate. ALC provides two significant advantages with respect to D^{2}STM, as described in the following. Firstly, rather than atomically broadcasting the writeset and the (Bloom filter encoded) read-set of a committing transaction, ALC relies on the cheaper Uniform Reliable Broadcast (URB) [18] primitive to disseminate exclusively the writesets. This results in a significant reduction of the inter-replica synchronization overhead. Secondly, the optimistic certification approach used in D^{2}STM may force transactions to undergo a high (and theoretically unbounded) number of aborts. The lease based certification scheme of ALC, conversely, shelters transactions from

repeated aborts due to remote conflicts, which leads to remarkable throughput increases in high conflict scenarios.

The problem of replicating a STM is naturally closely related to the problem of database replication, given that both STMs and DBs share the same key abstraction of atomic transactions. Modern database replication schemes [31,30,9,23] rely on AB to enforce, in a non-blocking manner, a global transaction serialization order without incurring in the scalability problems affecting classical eager replication mechanisms based on distributed locking and atomic commit protocols [15]. Existing database replication schemes based on AB can be coarsely classified in two main categories, depending on whether transactions are executed optimistically [31,23] or conservatively [22]. In the conservative case, which can be seen as an instance of the classical state machine/active replication approach [18], transactions are serialized through AB *prior* to their actual execution and are then deterministically scheduled on each replica in compliance with the resulting total order. This prevents aborts due to concurrent execution of conflicting transactions in different replicas. On the other hand, the need for enforcing deterministic thread scheduling at each replica requires a careful identification of the conflict classes to be accessed by each transaction, prior to its actual execution. Further, as update transactions need to be fully executed by all replicas, these approaches do not scale in presence of write intensive workloads [30]. Optimistic approaches avoid the above problems by relying on a commit-time certification phase but may generate unacceptable abort rates in high conflict scenarios. ALC shares the key benefits of these schemes (no need to determine the data-sets to be accessed by transactions prior to their execution; no need to fully execute update transactions on every replica), but leverages on the notion of asynchronous lease to lower the commit phase latency and shelter transactions from repeated aborts in presence of high conflict workloads.

The large body of literature on distributed shared memory (DSM) is clearly related to our work. Early DSM implementations [26] enforced strong consistency guarantees at the granularity of a single memory access. Those systems have proved hard to implement with good performance. Due to this reason, a significant body of research was devoted to build DSM systems that aim at achieving better performance at the cost of relaxing memory consistency guarantees [21]. Unfortunately, developing software for relaxed DSM's consistency models can be challenging as programmers are required to fully understand sometimes unintuitive consistency models. Conversely, the simplicity of the atomic transaction abstraction, at the core of (distributed) STM platforms, allows to increase programmers' productivity [7] with respect to both locking disciplines and relaxed memory consistency models. Further, the strong consistency guarantees provided by atomic transactions can be supported through efficient algorithms that, like in ALC, incur only in a single synchronization phase per transaction, amortizing the communication overhead across a (possibly large) set of memory accesses.

Atomic transactions play a key role also in the recent Sinfonia [1] platform, where these are referred to as "mini-transactions". However, unlike in conventional STM settings or in D^{2}STM, Sinfonia assumes transactions to be static,

i.e. that their data-sets and operations are known in advance, which limits the generality of the programming paradigm provided by this platform.

The notion of leases has been widely used in the context of replicated systems to simplify and/or optimize the replica consistency mechanisms, e.g. [13,14]. However, traditional leases are time-based contracts, being therefore tightly coupled to the notion of real-time. As a consequence, lease schemes have been traditionally designed and implemented assuming strong, and hence restrictive, synchrony levels (such as bounded communication delay and clock skew across processes). Conversely, the ALC replication scheme is implementable in a partially/eventually synchronous system [18], namely in any system where AB is implementable. The only other lease based solution we are aware of that is designed for employment in an asynchronous system is the one in [6]. There are a number of significant differences between the lease notion defined in [6] with respect to the one leveraged on by the ALC protocol. First, in [6], users are required to pre-declare the number of operations to be executed while holding the lease. In ALC, instead, leases are held by a replica as long as possible, i.e. as long as no conflicting transaction is executed by a different replica. Second, upon crash of a process p that has successfully established a lease, in [6] the remaining processes are forced to block until p recovers and "uses" all the intervals over which it has acquired a lease. Also, the success in the acquisition of an Asynchronous Lease is conditioned to the fact that there are no contending requesters. In our protocol, instead, conflicting leases requests are globally ordered, in a non-blocking fashion, using AB.

3 System Model and Architecture

We consider a classical asynchronous distributed system model [18] consisting of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate via message passing and can fail according to the fail-stop (crash) model. We assume that a majority of processes is correct and that the system ensures a sufficient synchrony level to permit implementing a View Synchronous Group Communication Service (GCS) [10]. GCS integrates two complementary services: *membership* and *multicast communication*. Informally, the role of the membership service is to provide, each participant in a distributed computation with information about which process is active (or reachable) and which one is failed (or unreachable). Such information is called a *view* of the group of participants. The multicast service allows a member to send a message to the group of participants with different reliability and ordering properties.

We assume that the GCS provides a primary-component group membership service [3], which maintains a single agreed view of the group at any given time and provides processes with information on whether they belong to the primary component. Specifically, the GCS delivers to the application a *viewChange* event to notify the alteration of the (primary component) view, and an *ejected* event to notify the exclusion of the process from the primary component (typically because of a false failure suspicion). We say that a process is v_i-correct in a

given view v_i if it does not fail in v_i and if v_{i+1} exists, it transits to it. We assume a GCS ensuring the following properties on the delivered views:

Self-inclusion if process p delivers view v_i, then p belongs to v_i.
Strong virtual synchrony messages are delivered in the same view in which they were sent.
Primary component view the sequences of views delivered are totally ordered and for any two consecutive views v_i, v_{i+1} there always exists a v_i correct process
Non-Triviality when a process fails or it is partitioned from the primary view, it will be eventually excluded from the primary component view
Accuracy a correct process is eventually included in every view delivered by the GCS

The GCS offers two communication services, namely: Optimistic Atomic Broadcast (OAB) [12] and Uniform Reliable Broadcast (URB) [18]. URB is defined by the primitives *UR-broadcast*(m) and *UR-deliver*(m). Three primitives define OAB: *OA-broadcast*(m), which is used to broadcast message m; *Opt-deliver*(m), which delivers message m without providing ordering guarantees; *TO-deliver*(m), which delivers message m in the final total oder. *Opt-deliver*(m) provides an early estimate of the final order of the corresponding *TO-deliver*(m); this estimate may be inaccurate without violating the safety of ALC.

The properties of the OAB are as follows:

Validity If a v_i-correct process p *OA-broadcasts* message m in v_i, then p *Opt-delivers* and *TO-delivers* m.
Integrity Any message m is *Opt-delivered* and/or *TO-delivered* by a process p at most once, and only if it had been previously *OA-broadcast*.
Optimistic Order If a node p *TO-delivers* m, then node p has previously *Opt-delivered* m.
Uniform Agreement If process p *TO-delivers* m in view v_i, then any v_i-correct process *TO-delivers* m in view v_i.
Total Order If two processes p and q *TO-deliver* messages m and m', then they do so in the same order.

The properties of the URB are as follows:

Validity If a v_i-correct process p *UR-broadcasts* message m in v_i, then p *UR-delivers* m.
Integrity Any message m is *UR-delivered* by a process p at most once, and only if it had been previously *R-broadcast*.
Uniform Agreement If process p *UR-delivers* m in view v_i, then any v_i-correct process $q \in v_i$ *UR-delivers* m in view v_i.
Causal Order If a process p *UR-delivers* m and m' such that m causally precedes m', according to Lamport's causal order [25] (denoted $m \rightarrow m'$), then p *UR-delivers* m before m'.

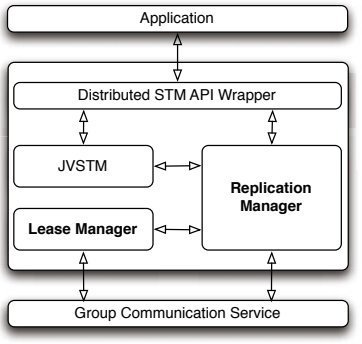


Fig. 1. Middleware architecture of an ALC replica

Note that in addition to the URB properties, OAB ensures total order of the *TO-deliver* events, preceded by a guess of the final order through the *Opt-deliver* event. Providing the total order property is more expensive than causal order in terms of exchanged messages and communication latency.

We now describe the software architecture of the middleware running on each replica, illustrated in Figure 1. The top layer is a wrapper that intercepts the application level calls for transaction demarcation (i.e. to begin, commit or abort transactions), not interfering with the application accesses (read/write) to the transactional data items, which are managed directly by the underlying JVSTM layer. This approach allows for transparently extending the classic STM programming model to a distributed setting.

JVSTM implements a multi-version scheme which is based on the abstraction of a *versioned box* (VBox). A VBox is a container that keeps a tagged sequence of values - the history of the versioned box. Each of the history's values corresponds to a change made to the box by a successfully committed transaction and is tagged with the timestamp of the corresponding transaction. To this end, JVSTM maintains an integer timestamp, *commitTimestamp*, which is incremented whenever a transaction commits. Each transaction stores its timestamp in a local *snapshotID* variable, which is initialized at the time of the transaction activation with the current value of *commitTimestamp*. This information is used both during transaction execution, to identify the appropriate values to be read from the VBoxes, and, at commit time, during the validation phase, to determine the set of concurrent transactions to check against possible conflicts. JVSTM relies on an optimistic approach which buffers transactions' writes and detects conflicts only at commit time, by checking whether any of the VBoxes read by a committing transaction T was updated by some other transaction T' with a larger timestamp value. In this case T is aborted. Otherwise, T's *commitTimestamp* is increased, its *snapshotID* is set to the new value of *commitTimestamp* and the new values of all the VBoxes it wrote are atomically updated. The integration of JVSTM within the ALC replication protocol entailed extending, in a non-intrusive manner, the JVSTM's original API so to

allow the Replication Manager to (*i*) extract information concerning transactions' read-set, write-set, and *snapshotID* timestamp; (*ii*) explicitly trigger the transaction validation procedure, which detects conflict generated by a transaction T_x with any other (local or remote) transaction that committed after T_x started; and (*iii*) atomically apply, the write-set WS of a remotely executed transaction (i.e. atomically updating the VBoxes of the local JVSTM with the new values written by a remote transaction) and simultaneously increasing the JVSTM's *commitTimestamp*.

The bottom layer is a Group Communication Service (GCS) [10] which provides the view synchronous membership, OAB and URB services. In our middleware implementation, we use the Appia GCS [29].

The core components of ALC are the Lease Manager (LM) and the Replication Manager (RM). The role of the LM is to ensure that there are never two replicas simultaneously disseminating updates for conflicting transactions. To this end, the LM exposes an interface consisting of two methods, namely GETLEASE() and FINISHEDXACT(), which are used by the RM to acquire/free leases on a set of DataItems. The RM is responsible of managing the transactions' commit phase, implementing a distributed certification scheme which leverages the local JVSTM replica to commit and certify local and remote transactions, as well as the services provided by the LM and the GCS.

Just like JVSTM, which ALC encapsulates, ALC preserves the strong atomicity [28] and opacity [16] properties. The former property avoids conflicts among transactional and non-transactional memory accesses. Opacity [16], on the other hand, can be informally viewed as an extension of the classical database serializability property with the additional requirement that even non-committed transactions are prevented from accessing inconsistent states. Our target consistency criterion for replication is 1-copy serializability [4], which ensures that transaction execution history across the whole set of replicas is equivalent to a serial transaction execution history on a not replicated (JV)STM.

4 The ALC Protocol

For the sake of clarity, we present the ALC protocol in an incremental fashion. We start by presenting a baseline version that relies on a simple, yet quite inefficient, lease establishment scheme. We will initially assume that the set of data items accessed by transactions do not vary across different re-executions of a same transaction and show how to deal with the case of transactions accessing different sets of data items across different executions in Section 4.4. In the Section 4.5 we introduce two optimizations that permit to drastically reduce the communication latency associated with the lease transfer mechanism by achieving full overlapping with the distributed certification phase. Finally, to simplify presentation, we will assume a single threaded execution model. This will allow us to avoid describing in detail the intra-replica synchronization scheme required to ensure consistency in a multi-threaded environment, and to focus on the description of the inter-replica coordination protocol.

Algorithm 1. Replication Manager.

```
boolean commit(Transaction T)
  if (¬JVSTM.validate(T)) then // early validation
     JVSTM.abort(T)
     return false
  LeaseRequestID leaseID=LeaseManager.getLease(JVSTM.getReadAndWriteSet(T))
  if (leaseID=⊥ ∨ ¬JVSTM.validate(T)) then // final validation
     JVSTM.abort(T)
     return false
  else
     trigger UR-broadcast([ApplyWS,T,leaseID,JVSTM.getWriteset(T)])
     wait until (committedXact(T) ∨ ejected)
     if ( ejected ) then
        JVSTM.abort(T)
        return false
     else
        return true

upon event UR-deliver([ApplyWS,T,leaseID,ws]) from p_j do
  if (p_j = p_i) then
     JVSTM.commitLocalXact(T)
     trigger committedXact(T)
     LeaseManager.finishedXact(leaseID)
  else
     JVSTM.commitRemoteXact(ws)
```

The intuition behind the ALC approach is the following. Analogously to classic certification schemes, transactions are run based on local data, avoiding any inter-replica synchronization till they enter commit phase. At this stage, however, ALC ensures to have established a lease for the accessed data items, prior to proceed with transactions' validation. In case a transaction T is found to have accessed stale data, this is re-executed without releasing the lease. This ensures that, during T's re-execution, no other replica can update any of the data items accessed during the first execution of T, guaranteeing the absence of remote conflicts on the subsequent re-execution of T provided that this deterministically accesses the same set of data items accessed during its first execution.

The ownership of the lease, in fact, ensures that no other replica will be allowed to validate any conflicting transaction, making it unnecessary to enforce distributed agreement on the global transactional serialization order. ALC takes advantage of this by limiting the use of OAB exclusively for establishing the lease ownership. Subsequently, as long as the lease is owned by the replica, transactions can be locally validated and their updates can be disseminated using URB, which can be implemented in a much more efficient manner than OAB.

Unlike classic lease based approaches, where the lease duration is defined at the time of the lease establishment, in ALC leases are said to be asynchronous since the concept of lease is detached from the notion of time. Conversely, once that a replica acquires a lease on a set of data items, it holds the lease as long as it does not require an explicit lease request from another replica. In order to avoid distributed deadlocks during the lease acquisition phase, lease requests are disseminated via OAB, and atomically enqueued at each node in the

TO-delivery order. Fairness is ensured by establishing leases in FIFO order and leases are transferred to a requesting replica as soon as the transactions (in execution at the lease-owner) to which those leases had been granted have committed.

4.1 Replication Manager

As already stated, transactions are executed locally, without any inter-replica synchronization, until the commit phase is reached. At this stage, if the committing transaction did not issue any write operation, it can be locally committed given that the JVSTM multi-versioned concurrency control scheme ensures the serializability of the observed snapshot. On the other hand, if we are not in presence of a read-only transaction, the STM API wrapper invokes the commit method of the Replication Manager, triggering the execution of the ALC protocol.

The pseudo-code describing the behavior of the RM is shown in Algorithm 1. Following an early validation phase, aimed at detecting any conflict developed with (local or remote) transactions already committed since the activation of the committing transaction, the RM requires the LM to acquire the leases corresponding to the set of data-items read and written during the transaction execution. The lease acquisition phase (described in the following) eventually terminates returning either a lease identifier, or the special value $\perp$ notifying the RM about the impossibility to acquire the requested leases. As we will see, the only case in which the LM ever fails to acquire leases is in case the process is excluded from the primary component view (due to a wrong failure suspicion). In such a case, for the RM it is only safe to keep on processing read-only transactions, and will therefore abort the current transaction. On the other hand, in absence of failures or failure suspicions, the lease manager will eventually succeed in acquiring the requested set of leases and return a lease request identifier to the RM. In this case, p_i is guaranteed to have already installed the updates of every remotely (and locally) executed transaction, and can therefore proceed with the validation. If this is successful, the transaction's writeset (and the corresponding lease request identifier) is disseminated using URB.

The properties of URB ensure that if p_i self-delivers the transaction's writeset in the current view, any other v_i-correct process will also deliver it in view v_i (even if p_i is subject to a failure right after the writeset delivery). This allows to safely commit the local transaction. Finally, the RM informs the LM of the successful execution of the transaction by invoking the FINISHEDXACT method specifying, as input parameter, the identifier of the lease request previously returned by the GETLEASE method.

The RM is also responsible of applying the writeset of remotely executed transactions, which are triggered by the corresponding *UR-deliver*. Note that the Causal Order property of the primitive ensures that the sequence of local transactions committed by a process p_i is delivered in FIFO order (i.e. in the same order in which p_i committed them) by any replica that deliver them.

Algorithm 2. Lease Manager at process p_i - Basic Algorithm.

```
FIFOQueue<LeaseRequest> CQ[NumConflictClasses]={⊥, ..., ⊥}
View currentView={p_1,...,p_i,...,p_n}
boolean inPrimaryComponent=true

LeaseRequestID getLease(Set DataSet)
 if (¬inPrimaryComponent) then return ⊥
 ConflictClass[] CC = getConflictClasses(DataSet)
 if (∃req∈CQ s.t. req.proc=p_i ∧ ¬req.blocked ∧ (∀cc∈CC : cc∈req.cc) ) then
   req.activeXacts++
 else
   LeaseRequest req = new LeaseRequest(p_i,CC)
   trigger OA-broadcast([LeaseRequest,req])
 wait until isEnabled(req) ∨ ¬inPrimaryComponent
 if (¬inPrimaryComponent) then return ⊥
 else return req.getID()

void finishedTransaction(LeaseRequestID reqID)
 getLeaseReqFromId(reqID).activeXacts−−

upon event TO-deliver([LeaseRequest, req]) from p_k do
 freeLocallyEnabledLeases(req.cc)
 ∀ cc∈req.cc do CQ[cc].enqueue(req)

upon event UR-deliver([LeaseFreed, reqs]) from p_k do
 ∀req∈reqs do
   ∀ cc∈req.cc do CQ[cc].dequeue(req)

void freeLocalLeases(LeaseRequest req)
 Set<LeaseRequest> locallyEnabledLeases
 ∀req† ∈CQ s.t. req†.proc=p_i∧ (req†.cc∩req.cc)≠ ∅ do
   req†.blocked=true
   if (req†.isEnabled()) then locallyEnabledLeases=locallyEnabledLeases ∪ req†
 if (locallyEnabledLeases ≠ ∅) then
   wait until ∀req* ∈locallyEnabledLeases : req*.activeXacts=0
   trigger UR-broadcast([LeaseFreed,locallyEnabledLeases)]

boolean isEnabled(LeaseRequest req)
 return ∀cc∈req.cc : CQ[cc].isFirst(req)
```

4.2 Lease Manager

The LM's pseudo-code for process p_i is reported in Algorithm 2 and Algorithm 3. Let us start by analyzing the pseudo-code in Algorithm 2, which represents the core of the lease establishment protocol. As already hinted, in order to establish/relinquish leases, the LM exposes two interfaces, namely the getLease and finishedXact methods. Leases are associated with data items indirectly, namely through conflict classes. This allows to flexibly control the granularity of the leases abstraction. We abstract over the mapping between a data item and a conflict class (which can in practice be implemented through classic hashing schemes since each transactional object is already uniquely identified) through the `getConflictClasses()` primitive, taking a set of data items as input parameter and returning a set of conflict classes. The trade-off between coarse and fine lease granularity is in that coarse granularity is prone to false sharing, i.e. lease requests associated with disjoint data items' sets may be mapped to common conflict classes, generating unnecessary lease migrations across replicas. On the other hand, fine granularity schemes may generate larger communication and

Algorithm 3. Lease Manager at process p_i - Dealing with View Changes.

```
upon event ViewChange(View newView) do
 if (¬inPrimaryComponent ∨ p_i is joining the group for the first time) then
    perform state transfer
    inPrimaryComponent=true
 else
     ∀p_j s.t. (p_j ∈currentView ∧ p_j ∉ newView) do
       ∀ req∈CQ s.t. req.proc=p_j do CQ.remove(req)
 currentView = newView

upon event ejected do
 inPrimaryComponent=false
```

processing overhead, since they impose the transmission of larger lease request messages among replicas and the management of larger local data structures for detecting conflicts among lease requests.

The data structures maintained by replicas for regulating the establishment/release of leases are the following: CQ, namely an array of FIFO queues, one per conflict class, that serves as a lock table to keep track of the conflict relations among lease requests; $currView$, namely the set of processes belonging to the current view; $inPrimaryComponent$, a boolean flag which indicates whether p_i is in the primary component or not. A `LeaseRequest` type is a structure containing the following fields: cc, namely the set of conflict classes associated with the lease request; $activeXacts$, an integer keeping track of the number of active transactions associated with the lease request, which is initialized to 1 when a lease request is created; $blocked$, a boolean variable indicating whether new transactions can be associated with this lease request or not, which is initialized to $false$ when a lease request is created; a unique identifier, which is transparently generated by p_i and is retrievable through the `getID`() primitive.

When the GETLEASE() method is invoked by the RM to establish a lease on the set of data items accessed by a committing transaction, the LM first checks whether p_i has already been ejected from the primary component. In this case it returns the special value $\perp$, notifying the RM that it is currently impossible to establish new leases. Otherwise, it determines, through the `getConflictClasses`() primitive, the set of conflict classes associated with the data-sets accessed by the transaction. Then it checks whether p_i has already enqueued in CQ a lease request req i) associated with a super-set of the currently requested conflict classes, and ii) which can still be associated with additional transactions (i.e. whose $blocked$ field is set to false). In this case, it is not necessary to issue a new lease request, and the current transaction can simply be associated with req. Otherwise, a new lease request is created and *OA-broadcast*. In both cases, p_i waits either until the corresponding lease request is enabled (this happens when the lease request reaches the first position in all the FIFO queues associated with its conflict classes - see the isEnabled() function), or until p_i is ejected from the primary component. In the latter case, the LM returns the special value $\perp$. If the lease request is eventually enabled, on the other hand, its unique identifier is retrieved via the `getID`() primitive and returned to the RM.

The FINISHEDTRANSACTION() method takes as input parameter a lease request identifier (i.e. the identifier previously returned by the GETLEASE() method when a lease request was associated with the transaction), retrieves the corresponding lease request via the `getLeaseReqFromId`() primitive, and decrements the number of active transactions associated with the lease request.

Upon a *TO-deliver* event of a lease request *req*, p_i first of all checks whether some of his locally issued lease requests need to be freed or blocked. This is done by invoking the freeLocalLeases procedure which, determines whether there is any of p_i's lease requests (denoted as req† in the pseudo-code) already enqueued in CQ which conflicts with *req* (i.e. whether req and req† have at least a conflict class in common). In this case, it sets the blocked field of these lease requests to true. This is the key mechanism employed to ensure the fairness of the lease rotation scheme: in order to prevent a remote process p_j from starving while waiting for process p_i to relinquish a lease, in fact, p_i is prevented from associating new transactions with existing lease requests as soon as a conflicting lease request from p_j is *TO-delivered* at p_i (as explained while describing the GETLEASE method). Next, the LM waits for the successful completion of every transaction associated with any locally issued conflicting lease request that is also currently enabled (note that this implies that such transactions have been already allowed to proceed with the validation phase). When these transactions have successfully committed, the LM triggers a *UR-broadcast* specifying the set of locally owned lease requests that p_i is freeing. The handling of the *TO-deliver* event terminates by enqueueing the corresponding lease request in every associated conflict class.

The logic associated with *UR-deliver* events is very simple: every lease request specified in the uniformly broadcast message is removed from the corresponding conflict class queues.

View Changes. It remains to discuss the replicas' behavior in the presence of view changes and ejections from the primary component view, which is formalized by the pseudo-code in Algorithm 3. Upon delivery of a new view event, if the replica re-joins the primary component or is joining the group of replicas for the first time, it triggers a state transfer procedure that realigns the content of the local replica of the STM, as well as of the state variables of the ALC protocol. Due to space constraints, we do not detail a description of the state transfer procedure, as, indeed, conventional state transfer mechanisms, such as [20] may be used at this purpose. On the other hand, if upon a view change, some processes are eliminated from the current view (because they have crashed or are partitioned away from the primary component), all of their lease requests are purged from the local CQ. Recall also that, if a process gets disconnected from the primary component, it will fail to deliver any pending lease request. This will cause the failure of the lease acquisition procedure at this replica (see the GETLEASE method). Overall, these two mechanisms (the removal of lease requests issued by processes excluded from the primary component, and the failure of the acquisition of lease requests pending at a process that is ejected from the primary component) guarantee the liveness of the lease management protocol. Note also

Algorithm 4. Lease Manager – Optimistic delivery optimization.

```
upon event Opt-deliver(LeaseRequest request) from p_k do
  freeLocalLeases(request)

upon event TO-deliver(LeaseRequest request) from p_k do
  foreach cc ∈ request.cc do
    CQ[cc].enqueue([p_k,request])
```

that replicas outside of the primary component may still continue processing read-only transactions, which will observe a serializable, albeit possibly obsolete, snapshot of the replicated (JV)STM.

4.3 Correctness Arguments

For space constraints we omit a full proof of correctness, but we still present some informal arguments analyzing why ALC ensures 1-copy serializability. First of all, we note that the enqueuing of lease requests at the various replicas takes place in a common order, namely the one determined by the final delivery of OAB, and that the logic for the advancement of the lease requests in the conflict classes' queues is deterministic. Also, the sequence of ApplyWS and LeaseFreed messages is disseminated via URB, which ensures causally ordered delivery. This guarantees that the stream of writesets associated with transactions accessing non-disjoint data items' sets are all applied in the same order at all replicas. The same applies for the delivery LeaseFreed messages, which implies that the order of dequeuing from the the conflict classes' queues for each pair of conflicting lease request is the same at all replicas. This also implies that every pair of conflicting transactions is validated in the same total order by each replica.

4.4 Non-deterministic Re-executions

The above presented lease management scheme guarantees the absence of remote conflicts during the re-execution of a transaction as it avoids releasing the lease on the conflict classes accessed during the previous execution of the transaction until this is successfully commit. This scheme can deterministically guarantee the absence of remote conflicts only if the set of conflict classes accessed when re-executing the transaction do not vary. While this is not always true in general for real applications, on the other hand it is very likely (as also supported by our experimental evaluation) that two re-executions of the same transaction access a large number of conflict classes in common (especially if lease granularity is moderately coarse). In practical settings, therefore, the presented ALC scheme is still very likely to significantly reduce the transactions' abort rate.

A simple, albeit somewhat extreme, workaround to deterministically bound the number of aborts/re-runs undergone by "problematic" transactions dramatically altering their data access patterns upon re-execution would consist in requesting a lease on the whole set of conflict classes. This would clearly suffice to ensure their successful re-execution, at the price of a temporary, though significant, bridling of concurrency.

Finally, it is important to highlight that the scheme presented in Section 4.2 can suffer of deadlocks in case the conflict classes accessed during transactions' re-execution, say cc', are not a subset of those accessed during a previous execution, say cc. This is due to the fact that the LM won't relinquish the lease on cc granted during the first transaction's execution, and will issue a new lease request on cc'. The latter may block if some other replica is simultaneously retaining the lease on cc' while requesting a lease on cc.

Fortunately, such an issue can be resolved by using simple and lightweight deadlock avoidance or detection schemes. A possible deadlock avoidance scheme is to detect whether $cc' \nsubseteq cc$ as a transaction completes its re-execution, and to piggyback a LeaseFreed message to the lease request OA-broadcast for cc'. An alternative deadlock detection scheme could check for the presence of cycles in the wait-for graph of the lease requests locally enqueued in CQ, and use a deterministic rule for breaking the cycle by aborting one of the involved lease requests. Note that as the state of the CQ is consistently replicated by all replicas, the deadlock detection would not require any additional inter-replica coordination.

4.5 Analysis and Optimizations

Provided that a replica owns a lease on the conflict classes accessed by a transaction, ALC allows committing the transaction using a single URB, which can be implemented incurring in a two communication steps latency [18]. This is in contrast with state of the art distributed certification schemes [31], which incur in the latency of (at least) an AB during the commit phase (whose latency is of at least 3 communication steps latency[1]). On the other hand, in the presented ALC scheme, if a transaction has accessed data items for which its process does not hold a lease, it incurs in the latency associated with lease acquisition phase. As depicted in the Figure 2 (a), this entails one AB to deliver the lease request, plus one URB for delivering the lease granted messages, yielding a total latency of 5 communication steps. Including the final URB for the dissemination of the transactions' writeset, we get 7 communication steps latency.

Two optimizations can be employed to reduce to just 3 communication steps latency the cost required for both committing a transaction and acquiring the corresponding lease. The first optimization, reported in Algorithm 4 and depicted in Figure 2 (b), consists in exploiting the *Opt-deliver* of the lease request (which incurs in a single communication step latency [22]) to immediately trigger the relinquishment of the required leases at a remote node (and the corresponding *URB* of a LeaseFreed message). This is safe since, even in the case of mismatches between the optimistic and the final delivery of two conflicting lease requests at some node p_i, the net effect would be anyway to trigger the relinquishment of the leases currently owned by p_i. This allows to totally overlap the execution of

[1] The only exception being AB protocols such as [34] which, relying on additional system assumptions - such as the existence of a bound Δ on the minimum inter-arrival time of messages at the replicas, achieve a latency of to $2+\Delta$ communication steps.

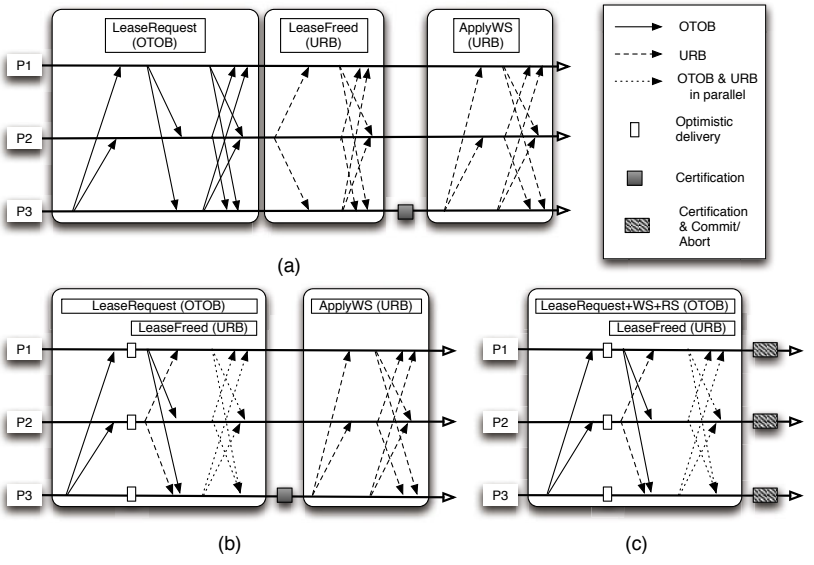


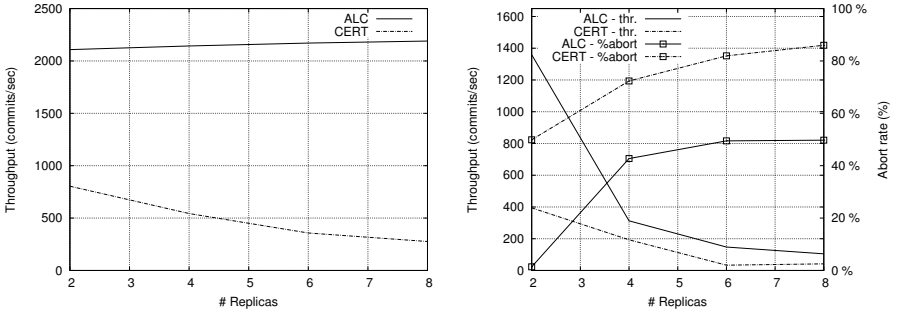
Fig. 2. Message pattern for (a) the baseline ALC protocol, (b) the optimization exploiting optimistic deliveries to free the leases, and (c) the optimization that piggybacks the readset and writeset on the LeaseRequest message ($P3$ requesting a lease owned by $P2$)

the OAB for the lease request and the URB for the lease granted, reducing to three communication steps the latency of the lease acquisition phase.

The second optimization consists in OA-broadcasting the set of data items accessed by a transaction T while issuing a lease request, rather than the corresponding conflict classes. This would allow each replica to validate T as soon as the corresponding lease request gets locally established, thus avoiding the *URB* of the transaction's writeset and reducing the latency for committing T to three communication steps (see Figure 2 (c)).

5 Performance Evaluation

In this section we report the results of an experimental study aimed at quantifying the performance gains achievable by the proposed ALC protocol with respect to state of the art transactional replication schemes. We use, as baseline, an atomic broadcast based certification scheme, such as the one in [31], which we refer to as CERT. More specifically, we use D^{2}STM [11] since it is the only fully replicated STM that uses a certification based scheme that we are aware of. Analogously to ALC, CERT allows replicas to process transactions locally, avoiding any form of synchronization during transaction execution. This protocol permits to achieve better scalability than pessimistic approaches [22] that



(a) No Conflict. (b) High Conflict.

Fig. 3. Bank Benchmark

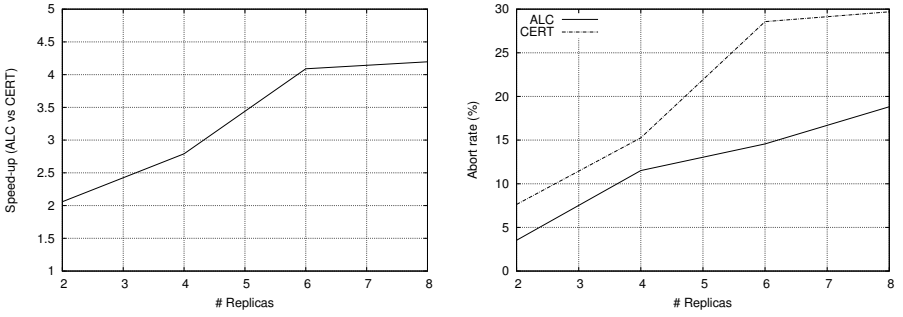
force all replicas to process every update transactions, does not rely on a-priori knowledge on transactions' data access patterns and requires a single atomic broadcast to disseminate the readset and writeset of a certifying transaction.

Concerning ALC, we implemented all the optimizations described in Section 4.5. In order to prevent the possibility of incurring in deadlocks in the presence of transactions altering their data access pattern during transaction execution, we implemented the simplest deadlock avoidance scheme among those previously described in Section 4.4: we piggyback a *LeaseFreed* message to the lease request message OA-broadcast during the commit phase of the re-started transaction if the set of conflict classes accessed is not a subset of those accessed during its former execution. All the results reported in the following were obtained by setting the conflict class granularity to coincide with a single data item. We deployed the prototypes of ALC and CERT[2] on a cluster of 8 nodes, each one equipped with an Intel QuadCore Q6600 at 2.40GHz, 8 GB of RAM, running Linux 2.6.27.7 and interconnected via a private Gigabit Ethernet.

We start by considering a synthetic workload (obtained by adapting the Bank Benchmark originally used in [19]) which serves for the purpose of quantifying the performance of the ALC scheme in two extreme scenarios for what concerns conflicts. In detail, we initialize the STM at each replica with an array of *numMachines*·2 items. In the first scenario, each machine reads and updates a distinct fragment of the array, thus never generating conflicts. In the second scenario, all the machines read and update the same data items, thus always conflicting.

Figure 3 shows the throughput (committed transactions per second) and the abort rate as the number of nodes in the system varies. In the scenario with no conflicts (Figure 3(a)), when using ALC, replicas disseminate transactions exclusively via URB (after establishing the lease upon their first transaction). This allows ALC to achieve a throughput from 3 to 10 times higher than CERT, which, requiring one OAB per committed transaction, puts a significantly higher

[2] Both prototypes are implemented in Java and are publicly available at the url: `http://aristos.gsd.inesc-id.pt`



(a) Speed-up. (b) Abort rate.

Fig. 4. Lee Benchmark

load on the GCS (which represents the bottleneck in this benchmark, being the transaction's logic extremely lightweight) especially as the number of replicas increase. The high conflict scenario (Figure 3(b)) represents a worst case scenario for ALC, since leases are constantly rotated across the replicas, and a lease request must be OA-broadcast for each transaction that commits. Nevertheless, ALC's throughput is on average 3 times higher with respect to CERT. This can be explained by observing that, with CERT, the percentage of transactions that abort is significantly larger than with ALC. In the 8 replicas' scenario, for instance, transactions are re-executed on average around 10 times before committing with CERT. On the other hand, ALC ensures that a transaction can be aborted at most once, as also proved by the fact that the abort rate for ALC never grows larger than 50% independently of the degree of concurrency.

We now consider a complex benchmark, namely Lee-TM [2], which is a parallel, STM-based implementation of the Lee algorithm for routing junctions in a circuit. The Lee-TM generates a very heterogeneous workload encompassing a wide range of transactions' duration and length. More in detail, the benchmark starts by routing the shortest junctions in the circuit - generating transactions whose local processing lasts just a few msecs - and then progressively lays junctions of increasing length - generating transactions whose local processing lasts up to a few seconds. Additionally, in Lee-TM, multiple re-runs of a transaction have a non-negligible probability of accessing different data-sets, permitting to evaluate the performance of the ALC's deadlock avoidance mechanisms proposed in Section 4.4. Figure 4(a) reports the speed-up achieved by ALC with respect to CERT computed considering the time required to route the whole set of junctions of the mainboard circuit [2] when using the two protocols. Also in this case, the performance gains achieved by ALC are clear, ranging from around 2x to more than 4x and growing along with the number of replicas in the system. Being the inter-transaction data locality of this benchmark pretty low (i.e. the likelihood to re-use a previously acquired leases when running two different transactions on a same replica was found to be less than 10%), the reason underlying the performance boost achievable by ALC is mainly imputable to its

ability to reduce the transaction abort rate (see Figure 4(b)), and, in particular, to shelter long-running transactions from repeated aborts. Despite the lack of deterministic guarantees on the immutability of the data accessed during transactions' re-runs, in fact, ALC guaranteed to execute transactions at-most once in the 98% of the cases. On the other hand, with CERT, long running transactions are very likely to be aborted tens of times before being successfully committed, causing a huge waste of computing resources.

6 Conclusions and Future Work

In this paper we have introduced ALC, a novel STM replication scheme that relies on the notion of asynchronous lease to boost the performance of existing AB-based transaction certification schemes. We have integrated ALC within a middleware that allows STM applications to transparently leverage the computational resources available in commodity clusters and shown the significant performance benefits achievable by ALC (up to 10x reduction of the commit phase latency in low conflict scenarios, and up to 4x speed-ups in high conflict scenarios) via a fully fledged, publicly available prototype.

This work opens several interesting research perspectives that we intend to pursue in our future work. In particular, it would be interesting to identify techniques capable of effectively minimizing the frequency of rotation of leases among the replicas, so to maximize the performance gains achievable through the use of ALC. These include locality aware load balancing strategies, as well as mechanisms capable of adaptively adjusting the lease rotation mechanism based on the actual replicas' (spatial/temporal) locality of reference.

References

1. Aguilera, M.K., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: a new paradigm for building scalable distributed systems. In: Proc. of the 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 159–174. ACM, New York (2007)
2. Ansari, M., Kotselidis, C., Watson, I., Kirkham, C.C., Lujin, M., Jarvis, K.: Lee-tm: A non-trivial benchmark suite for transactional memory. In: Bourgeois, A.G., Zheng, S.Q. (eds.) ICA3PP 2008. LNCS, vol. 5022, pp. 196–207. Springer, Heidelberg (2008)
3. Bartoli, A., Babaoglu, O.: Selecting a "primary partition" in partitionable asynchronous distributed systems. In: IEEE Symp. on Reliable Dist. Systems, p. 138 (1997)
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
5. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: Proc. of the Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 247–258. ACM, New York (2008)
6. Boichat, R., Dutta, P., Guerraoui, R.: Asynchronous leasing. In: Proc. of the The International Workshop on Object-Oriented Real-Time Dependable Systems, p. 180. IEEE Computer Society, Washington (2002)

7. Cachopo, J.: Development of Rich Domain Models with Atomic Actions. Ph.D. thesis, Technical University of Lisbon (2007)
8. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63(2), 172–185 (2006)
9. Cecchet, E., Marguerite, J., Zwaenepole, W.: C-JDBC: flexible database clustering middleware. In: Proc. of the USENIX Annual Technical Conference, p. 26. USENIX Association (2004)
10. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. 33(4), 427–469 (2001)
11. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D^{2}STM: Dependable Distributed Software Transactional Memory. In: Proc. of the 15th Pacific Rim Int. Symposium on Dependable Computing, PRDC (2009)
12. Defago, X., Schiper, A., Urban, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys 36(4), 372–421 (2004)
13. Duvvuri, V., Shenoy, P., Tewari, R.: Adaptive leases: A strong consistency mechanism for the world wide web. IEEE Transactions on Knowledge and Data Engineering 15(5), 1266–1276 (2003)
14. Gray, C., Cheriton, D.: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In: Proc. of the Symposium on Operating Systems Principles (SOSP), pp. 202–210. ACM, New York (1989)
15. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proc. of the Conference on the Management of Data (SIGMOD), pp. 173–182. ACM, New York (1996)
16. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 175–184. ACM, New York (2008)
17. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: a benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. 41(3), 315–324 (2007)
18. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer, Heidelberg (2006)
19. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. SIGPLAN Not. 41(10), 253–262 (2006)
20. Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G.: Non-intrusive, parallel recovery of replicated data. In: Proc. of the 21st IEEE Symp. on Reliable Distributed Systems (SRDS), p. 150. IEEE Computer Society, Washington (2002)
21. Keleher, P., Cox, A.L., Zwaenepoel, W.: Lazy release consistency for software distributed shared memory. In: Proceedings of the 19th Int. Symp. on Computer Architecture (ISCA), pp. 13–21. ACM, New York (1992)
22. Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing transactions over optimistic atomic broadcast protocols. In: Proc. of the 19th IEEE International Conference on Distributed Computing Systems, p. 424. IEEE Computer Society, Los Alamitos (1999)
23. Kemme, B., Alonso, G.: A suite of database replication protocols based on group communication primitives. In: Proc. of the International Conference on Distributed Computing Systems (ICDCS), p. 156. IEEE Computer Society, Los Alamitos (1998)
24. Kotselidis, C., Ansari, M., Jarvis, K., Lujan, M., Kirkham, C., Watson, I.: DiSTM: A software transactional memory framework for clusters. In: Proc. of the International Conference on Parallel Processing (ICPP), pp. 51–58 (2008)
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. ACM Commun. 21(7), 558–565 (1978)

26. Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. In: Proc. of the Symp. on Principles of Distributed Computing, pp. 229–239. ACM, New York (1986)
27. Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: Proc. of the Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 198–208. ACM, New York (2006)
28. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Comput. Archit. Lett. 5(2), 17 (2006)
29. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proc. International Conference on Distributed Computing Systems (ICDCS), pp. 707–710. IEEE, Los Alamitos (2001)
30. Patino-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Scalable replication in database clusters. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 315–329. Springer, Heidelberg (2000)
31. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases 14(1), 71–98 (2003)
32. Romano, P., Carvalho, N., Rodrigues, L.: Towards distributed software transactional memory systems. In: Proc. of the Workshop on Large-Scale Distributed Systems and Middleware, LADIS (2008)
33. Romano, P., Carvalho, N.M.R., Couceiro, M., Rodrigues, L., Cachopo, J.: Towards the integration of distributed transactional memories in application servers' clusters. In: The 3rd Int. Workshop on Advanced Architectures and Algorithms for Internet DElivery and Applications, ICST. Springer, Las Palmas (2009)
34. Vicente, P., Rodrigues, L.: An indulgent uniform total order algorithm with optimistic delivery. In: Proc. of the Symposium on Reliable Distributed Systems (SRDS), pp. 92–101 (2002)