Supporting Multiple Data Replication Models in Distributed Transactional Memory

João A. Silva, Tiago M. Vale, Ricardo J. Dias, Hervé Paulino, and João M. Lourenço
CITI—Departamento de Informática, Universidade NOVA de Lisboa, Portugal

ABSTRACT

Distributed transactional memory (DTM) presents itself as a highly expressive and programmer friendly model for concurrency control in distributed programming. Current DTM systems make use of both data distribution and replication techniques as a way of providing scalability and fault tolerance. But both techniques have their advantages and disadvantages, and therefore each one is suitable for different target applications/services and different deployment environments. In this paper we address the support of different data replication models in DTM. To that end we propose ReDstm, a modular and non-intrusive framework for DTM, that supports multiple data replication models in a general purpose programming language (Java). We show its application in the implementation of distributed software transactional memories with different replication models, and evaluate the framework via a set of well-known benchmarks, analysing the impact of different data replication models on the memory usage and transaction throughput.

Keywords

Data Replication; Distributed Transactional Memory; Concurrency Control; Distributed Systems

1. INTRODUCTION

Cloud computing has democratized the access to distributed computing infrastructures, popularizing the use of distributed systems to meet the ever-growing scalability, availability and low latency requirements of modern Internet services. Many of these distributed software systems require the sharing of data among their (distributed) processes, a fact that increases the systems' overall design complexity and hinders their performance. In this context, distributed transactional memory (DTM) presents itself as a highly expressive and programmer friendly alternative for concurrency control in such software systems. It extends the scope of the software transactional memory (STM) model [8, 19] to distributed environments, combining it with data replication and distribution. The result is a high-level concurrency control mechanism that offers transactional semantics over a distributed shared memory addressing space.

Despite being initially studied in the context of chip-level multiprocessing, the benefits of STM over traditional concurrency control methods may also be harvested in distributed environments. To that end, recent research has led to the development of multiple DTM frameworks [4, 12, 16, 21]. The majority of these have grown from existing STM frameworks, by adding communication layers, schedulers, mechanisms for replication and contention management, and object lookup. On top of that, these frameworks offer *intrusive* programming models, demanding the rewriting of the application code in order to comply to their specificities. This makes the decision of using a specific framework a rather serious commitment as new applications become tied to the provided APIs, and later porting of existing applications to another framework is a tedious and error prone job.

Another issue is that the bulk of the DTM frameworks are tailored for a concrete data replication strategy, not allowing the unbiased evaluation of different strategies under the same circumstances. Also, each data replication strategy has its advantages and disadvantages, being therefore suitable for different target applications/services and deployment environments. In the control-flow model [16, 21] data is immobile and transactions access data through remote procedure calls. On the other hand, in the data-flow model [16, 21] transactions are immobile and data moves through the network to requesting transactions. In what regards data replication, the full replication model [4] replicates all data items in all the system's nodes. This strategy provides the best possible tolerance to data loss but limits the system's total storage capacity and requires coordination between all the nodes, which raises scalability issues [18]. Conversely, the partial replication model replicates each data item in only a subset of the system's nodes, i.e., data items are partially replicated. This model provides a reasonable tolerance to data loss but requires nodes to perform remote read operations in search for the data items that are not locally replicated.

In this paper we present ReDstm, a *modular*, *flexible* and *non-intrusive* framework for DTM. ReDstm's modularity allows for different implementations of the local STM algorithm, of the data distribution logic, of the distributed transaction validation protocol, of the communication system, among others. It also intrinsically provides support for multiple data replication models, with different levels of replication (per data structure in the program), ranging from no replication to partial and to full replication. Another dis-

tinctive feature of ReDstm is its non-intrusive, annotation-based, programming model. The choice of a particular replication level for a data structure may have a considerable impact of the system's performance, and hence such should derive from a careful analysis. Ergo, despite the high-level nature of the programming model, the task of choosing the appropriate replication level may not be trivial. To aid in this enterprise, we present a generic methodology for the design of distributed transactional data structures when faced with a choice of different replication levels.

The contributions of this paper can be thus summarized as follows: (i) we propose and define a modular Java DTM framework that supports different strategies for data placement and replication, and offers a highly expressive and non-intrusive programming model; (ii) we propose a methodology for the design of distributed transactional data structures in ReDstm, and; (iii) we evaluate the impact of applying different replication levels to well know transactional benchmarks running on top of ReDstm.

The remainder of this paper is organized as follows: §2 presents ReDstm; §3 elaborates on how to implement distributed STMs, with different data replication models, on top of ReDstm; §4 proposes a methodology for the design of distributed transactional data structures in ReDstm; §5 presents and analyses experimental results; §6 discusses related work; and finally §7 presents our concluding remarks.

2. A MODULAR FRAMEWORK FOR DTM

ReDstm builds upon the TribuSTM framework [5] to offer a generic framework for DTM. It extends TribuSTM with support for (i) distributed objects, including data placement and data replication strategies; and (ii) the distributed commit of transactions accessing these objects. This section begins by giving a general overview of TribuSTM, and follows by the presentation of the ReDstm framework and of its programming model.

2.1 TribuSTM

All STM algorithms associate some kind of information (the *metadata*) to each managed memory location, whose nature varies with the algorithm itself, e.g., locks, timestamps, version lists. Such metadata can be stored either in an external table (*out-place* strategy) or adjacent to each memory location (*in-place* strategy).

The out-place strategy is implemented using a table-like data structure that efficiently maps memory locations to metadata. Storing metadata in such a pre-allocated table avoids the overhead of dynamic memory allocation, but incurs in the overhead of evaluating the mapping function. Additionally, the bounded size of the external table induces a false sharing situation where multiple memory locations share the same table entry and hence the same metadata, resulting in a *many-to-one* relation between memory locations and metadata. This approach suits STM algorithms with weak ties between the metadata and the associated memory locations, e.g., the TL2 algorithm [6], whose metadata are locks. However, it falls short when these ties grow stronger, e.g., the version list associated with each memory location in JVSTM [3]. A direct dependency relationship between the metadata and the associated memory locations requires them to be mapped by a *one-to-one* relation.

The in-place strategy provides such *one-to-one* relation, preventing the occurrence of false sharing situations. This strategy is usually implemented by using the *decorator* design pattern [7], where the functionality of an original is wrapped in a decorator class that contains the required metadata. This allows direct access to the metadata but is very intrusive to the application's code, which must be rewritten to use the decorator classes.

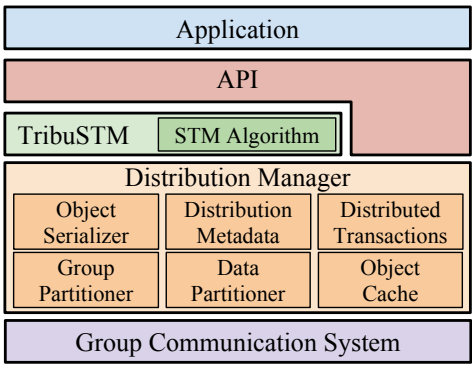


Figure 1: ReDstm's architecture overview.

Deuce [11] is a Java STM framework that allows for the implementation of various STM algorithms, standing out for its non-intrusion to the application programmer. However, its applicability is bound to STM algorithms that can use the out-place strategy. TribuSTM extends the Deuce STM framework with support for the in-place metadata placement strategy, *without* making use of the decorator pattern.

2.2 Supporting Distribution and Replication

Supporting DTM on top of TribuSTM raises some challenges that drive the organization of this subsection, namely: (i) how to design a modular DTM framework; (ii) where to store the distributed data and metadata and how to access them, and; (iii) how to validate and apply transactions in a distributed context.

2.2.1 Architecture Overview

The architecture of ReDstm comprises several modules, organized in three layers (see Fig. 1). From a bottom-up perspective, the group communication system (GCS) provides inter-node communication, the distribution manager (DM) is responsible for all the distribution-related issues, and finally the TribuSTM layer is in charge of the local concurrency control at each node. ReDstm modularity allows different implementations of its modules, such as the STM algorithm, the distributed transactions validation protocol, and the group communication system, among others.

Directing our attention to the first two layers, the purpose of the DM is twofold: to implement a distributed (shared) memory space that installs the desired data placement strategy and data replication model (distribution, full or partial replication), and to implement protocols to support the execution of distributed transactions. To that end, it is comprised by the following modules:

Object Serializer (ObS) Implements the serialization logic for communication with remote objects;

Distribution Metadata (DMd) Each data placement model requires its own distribution metadata, where it stores information on the data's whereabouts. Together with the ObS, they establish a logic for the objects' distribution.

Distributed Transactions (DT) Regulates the implemen-

tation of distributed transaction validation protocols, embedding all the interaction with TribuSTM.

Group Partitioner (GP) Divides the system's nodes into well defined groups. The operation is particularly relevant in the context of partial data replication. A group defines the system's *units of replication*, i.e., given a group of nodes $\mathcal{G}$ and a distributed object $\mathcal{O}$, if $\exists\ \mathcal{N} \in \mathcal{G} : \mathcal{O}$ is replicated in $\mathcal{N} \Rightarrow \forall \mathcal{N}' \in \mathcal{G} \setminus \{\mathcal{N}\}, \mathcal{O}$ is replicated in $\mathcal{N}'$

Data Partitioner (DP) This is a module tailored for partial data replication environments. It establishes a mapping function between the objects that comprise the distributed shared memory space and the groups in which they are replicated on.

Object Cache (ObC) Provides a generic caching service for ReDstm distributed objects. It is a mapping from distributed metadata onto generic objects, which must be specialized to cope with the singularities of each DM implementation.

The GCS layer encapsulates the communication primitives required to implement the DM (e.g., atomic multicast, reliable unicast/multicast) providing an uniform interface regardless of the concrete communication system being used.

2.2.2 Metadata & Distributed and Shared Memory

ReDstm supports objects distribution by combining metadata with the Java serialization infrastructure. One of the ideas highlighted in §2.1 is that STM algorithms associate metadata to the managed memory locations. Similarly, the implementation of different data replication models may also grow from the metadata associated to each managed memory location. For instance, in a full data replication environment one may associate an unique identifier to each memory location, allowing the system to recognize that location in the local replica. In turn, in a purely distributed context, the metadata may contain the information necessary for the execution of a remote procedure call targeted to the *owner* of the corresponding memory location.

Note that we are referring to two different kinds of metadata. STM algorithms associate metadata to each field of an object, in order to manage concurrency. Plus, the support for distributed objects builds from object-level metadata to implement the desired data placement and data replication strategies. These are clearly two distinct kinds of metadata, thus we shall refer to both as *transactional* and *distribution* metadata, respectively, to disambiguate if necessary.

Metadata. Regardless of the data placement and replication strategies in place, the framework requires the association of distribution metadata to objects. As such, in order to be compliant with ReDstm distribution support, an object must implement the `DistributedObject` interface—with methods `getMetadata` and `setMetadata(DistMetadata)`. This interface provides the means for the framework to access the object's metadata, which must be a subclass of `DistMetadata`.

Serialization. A ReDstm distributed object is serialized in function of its distribution metadata, so that different implementations of the DM may serialize objects differently, according to their replication model. For this purpose, the object needs to implement methods `writeReplace` and `readResolve` (from the Java `java.io.Serializable` interface). The first must encode the original object $\mathcal{O}$ into its replacement $\mathcal{R}$, and the second simply de-serialize and return $\mathcal{R}$.

Table 1: Operations provided by the reflexive API.

Operation	Description
onStart($\mathcal{T}$)	Start of transaction $\mathcal{T}$.
onRead($\mathcal{T}$, m)	Read on transactional metadata m by transaction $\mathcal{T}$.
onWrite($\mathcal{T}$, m, v)	Write of value v on transactional metadata m by transaction $\mathcal{T}$.
onCommit($\mathcal{T}$)	Commit request issued from transaction $\mathcal{T}$.
onAbort($\mathcal{T}$)	Abort of transaction $\mathcal{T}$.

In order for ReDstm to be both a flexible and a non-intrusive framework, instead of requiring the application programmer to explicitly implement the `DistributedObject` interface, the framework features an instrumentation agent that automatically injects such code into the application.

The instrumentation is automatically applied to every loaded class, transforming the latter's implementation (via Java bytecode rewriting/injection) into one that is ReDstm distributed object compliant. Let $\mathcal{C}^{\mathcal{I}} = \{f_1, \ldots, f_n, m_1, \ldots, m_k\}$ denote a class $\mathcal{C}$ that implements the set of interfaces $\mathcal{I}$ and where f_i and m_j, with $i \leq n$ and $j \leq k$, represent the fields and methods of the class, respectively. The transformations applied upon $\mathcal{C}$ aim at turning it into a subtype of the `DistributedObject` interface, to insert a new field f^{dm} of type `DistMetadata`, and to add methods `writeReplace` and `readResolve`. Namely:

$$\mathcal{C}^{I \cup \{\texttt{DistributedObject}\}} = \left\{ \begin{array}{c} f^{dm}, \\ f_1, \ldots, f_n,\ m_1, \ldots, m_k, \\ \texttt{getMetadata}, \texttt{setMetadata}, \\ \texttt{writeReplace}, \texttt{readResolve} \end{array} \right\}$$

2.2.3 Distributed Transactions

To allow the flexible integration of various realizations of the DM layer with TribuSTM, the DM and TribuSTM interact through two distinct and well defined interfaces. In order to support distributed transactions, the DM must be aware of some events triggered by the application on TribuSTM, such as transactional accesses or commit requests. It might also need to query or modify the state of the local STM, e.g., to apply updates made by a remote transaction.

With the first interface, the *reflexive* API (see Table 1), the DM is able to react to certain events from TribuSTM. It registers callbacks for transactional-related events such as transaction start (onStart), transactional accesses (onRead, onWrite), and commit and abort (onCommit, onAbort).

The second interface, the *actuator* API (see Table 2), enables the DM to inspect the state of the local STM and act upon it. It can acquire an opaque representation of a transaction's state (createState) which can then be used to recreate the transaction (recreateTx). It can also explicitly trigger the validation (validate) and apply the updates (applyWs) of a transaction. The reaction of the DM to an event may trigger synchronous request-reply communication with remote nodes. Accordingly, the execution of the DM blocks until one of the callback methods (e.g., startProcessed, readProcessed, writeProcessed) notifies the reception of the response.

2.2.4 Communication System

Different implementations of the DM can have specific requirements for the GCS. For instance, a certification scheme running on a fully replicated environment requires a GCS with support for an atomic broadcast primitive. However, in a purely distributed context we can devise scenarios where

Table 2: Operations provided by the actuator API.

Operation	Description
createState($\mathcal{T}$) : $\mathcal{S}$	Returns a representation of transaction's $\mathcal{T}$ state.
recreateTx($\mathcal{S}$) : $\mathcal{T}$	Returns a transaction $\mathcal{T}$ recreated from state $\mathcal{S}$.
validate($\mathcal{T}$) : *bool*	Validates transaction $\mathcal{T}$.
applyWS($\mathcal{T}$)	Applies all updates by transaction $\mathcal{T}$ on the local STM.

Table 3: Operations provided by the communication API.

Operation	Description
ABcast(m)	Atomically broadcasts message m.
RBcast(m)	Reliably broadcasts message m.
AMcast(m, g)	Atomically multicasts message m to group g.
RUcast(m, dst)	Reliably unicasts message m to node dst.
RMcast(m, g)	Reliably multicasts message m to group g.
self(s) : *bool*	Returns true if s is the local node, false otherwise.

only point-to-point communication is necessary.

Table 3 shows the communication primitives offered by the GCS to the DM. To be notified of incoming messages, the DM subscribes to the deliveries using the *observer* pattern [7] (the on*Deliver(m, src) operations notify of the delivery of message m from sender src).

2.2.5 Cache Support

With the exception of full data replication, read/write operations over objects in the distributed shared memory space may imply remote communication. This naturally raises performance problems that may even cancel out other benefits. To reduce time variance in the access to distributed objects, ReDstm provides support for a generic caching service, leveraging locality. The service offers a set of generic operations over a container that is also generic. The caching service maps keys, of type `DistMetadata`, to generic objects, and includes the basic map operations (e.g., contains, get, insert, remove). This approach promotes flexibility, allowing for the development of cache mechanisms tailored for specific transaction validation protocols, given that only these specific implementations may handle issues such as: what to cache, and the validity of cached objects.

2.3 Programming Model

Pursuing our non-intrusiveness goals, the programming model exported by ReDstm is annotation-based, so no code re-writing is required.

Transactions. As legacy from TribuSTM, the programmer only has to add a `@Atomic` annotation to the methods that must execute as transactions. Then, in a transparent way to the programmer, ReDstm rewrites the application's bytecode, intercepting and conferring control to the framework at the beginning and end of transactions, and at memory accesses performed in a transactional context.

Partial Data Replication. In a partial data replication environment, it is useful to be able to express what is to be fully and partially replicated. So, our key insight is to grant the programmer the power to express what data should be partially replicated.

In the Java virtual machine (JVM), the heap can be seen as a directed graph, where each node represents an object that, in turn, may have references to other objects. Figure 2 depicts an example of such graph, on which edges represent references, leaf nodes represent values of primitive data types, and the remainder nodes represent (composite) objects. Nodes A and H can be seen has two references pointing to the same memory location, i.e., aliasing.

In our approach the heap graph is replicated in all nodes, i.e., fully replicated, by default. Exceptions must be *explicitly* conveyed by the means of an annotation—`@Partial`—to be applied to class' fields. The annotation expresses that everything *downstream* of such field, in the graph, is to be partially replicated. Listings 1-2 illustrate the application of the annotation to the simple case of a linked list.

3. ON THE IMPLEMENTATION OF DISTRIBUTED STMS IN REDSTM

In this section we reason on how to use the mechanisms provided by ReDstm to implement distributed STMs with different data replication models.

3.1 A Distributed STM

Consider that the distribution metadata is comprised by an unique identifier, *id*; and the address of the node which owns the object associated with the metadata, *owner*. Let $id(\mathcal{O})$ and $owner(\mathcal{O})$ denote the *id* and *owner* of a distributed object $\mathcal{O}$, respectively. If a distributed object $\mathcal{O}$ is created at node $\mathcal{N}$, $owner(\mathcal{O})$ is set to $\mathcal{N}$.

Let $host(\mathcal{T})$ denote the node where transaction $\mathcal{T}$ executes, and $id(\mathcal{T})$ the unique identifier of $\mathcal{T}$. Let $proxy_n(id)$ denote the proxy transaction of $\mathcal{T}$ on node n such that $id(\mathcal{T}) = id$, and consider, for simplicity, that every node contains a proxy transaction for every transaction which is currently executing in the system.

When a transaction $\mathcal{T}$ issues a read access on transactional metadata m (onRead at Table 1), if $owner(m) = host(\mathcal{T})$ the read access is a regular local access. Otherwise, $host(\mathcal{T})$ sends a message $[id(\mathcal{T}), id(m)]$, henceforth rd, to $owner(m)$, and waits response. When $owner(m)$ receives rd, it resolves $id(m)$ to m, issues a local read on m on behalf of $proxy_{owner(m)}(id(\mathcal{T}))$, and responds to $host(\mathcal{T})$ with message $[id(\mathcal{T}), v, a]$ where v is the value read and a is true if there has been a conflict. When $host(\mathcal{T})$ delivers the response message, the execution of $\mathcal{T}$ is resumed. A write access behaves similarly.

When $\mathcal{T}$ requests to commit (onCommit at Table 1), $host(\mathcal{T})$ sends a message to all nodes to trigger the validation of $proxy_n(id(\mathcal{T}))$ on each node n (validate at Table 2). All nodes respond with the result of the validation. Once $host(\mathcal{T})$ has collected all validation results, if none of the validations failed, a message is sent to all nodes instructing the commit of each $proxy_n(id(\mathcal{T}))$. If at least one validation was unsuccessful, all nodes are instructed to discard their proxies and $\mathcal{T}$ aborts.

This is a simple approach, thus there are various optimizations to consider, such as creating transaction proxies on demand only on strictly necessary nodes, caching to reduce network communication, and so on and so forth.

3.2 A Fully Replicated STM

In a fully replicated environment, in which all replicas maintain a local copy of all the objects in the system, each of such objects must be identifiable in the global context of the system. In a centralized system, an object can be uniquely identified by its memory address, but this does not apply in a distributed context, where an object's metadata must contain a globally unique identifier.

```
class Node<T> {
  Node next;
  int id;
  @Partial
  T value;

  ...
}
```

Listing 1: Class `Node`.

```
class List<T> {
  Node<T> head;

  public List() {
    this.head = ...
  }
  ...
}
```

Listing 2: Class `List`.

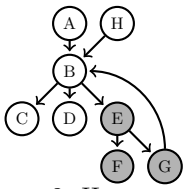


Figure 2: Heap graph.

Algorithm 1 Transactional memory (TM) component.

```
committed ← false                                   ▷ Commit result

procedure BEGIN(T)
    ...
    ONSTART(T)

function COMMIT(T)
    ONCOMMIT(T)
    wait until T is processed
    return committed

procedure COMMITPROCESSED(T, r)
    committed ← r
    T was processed
```

Algorithm 2 Non-voting certification protocol.

```
localTxs ← ∅                          ▷ Map of transaction ids to transactions

procedure ONCOMMIT(T)
    localTxs ← localTxs[T.id ↦ T]
    S ← CREATESTATE(T)
    ABCAST(S)

procedure ONABDELIVER(S, src)
    if SELF(src) then
        T ← localTxs(S.id)
        localTxs ← localTxs \ T
    else
        T ← RECREATETX(S)
    valid ← VALIDATE(T)
    if valid then
        APPLYWS(T)
    COMMITPROCESSED(T, valid)
```

Consider that a fully replicated object $\mathcal{O}$ can be in two possible states: published or private. In the first case it has already been assigned an identifier *oid* (metadata). The serialization of a private object $\mathcal{O}$ consists in generating its respective *oid* and effectively serializing $\mathcal{O}$. If an object is already published, we just need to assign *oid* as its representative, serializing *oid* instead of $\mathcal{O}$. In this case, the de-serialization of *oid* returns the object $\mathcal{O}'$ such that $id(\mathcal{O}') = oid$, i.e., returns the object corresponding to the local replica with identifier *oid*. This means that an already published object is never sent through the network, only its identifier is, resulting in a potentially large decrease in the size of the exchanged messages.

Certification-based protocols [1, 9] are very interesting in the context of a fully replicated DTM because they do not require synchronization between replicas during transaction execution (only at commit time). In Algorithms 1 and 2 we can see the pseudo-code of the non-voting certification protocol implemented using the interfaces provided by ReDstm.

An application's thread executes a transaction locally, and only when it finishes it asks for the confirmation of the transaction to the TM component (COMMIT at Alg. 1), which triggers the certification protocol (ONCOMMIT at Alg. 2). At this point, the thread waits for the decision of the certification protocol, which will validate and confirm the transaction, or abort it. The atomic broadcast performed by the certification protocol is received by all replicas. The replica that broadcasted the message processes the corresponding local transaction, while the other replicas recreate the transaction using the state $\mathcal{S}$ received in the message. All replicas proceed to the validation of the transaction and subsequent application in case of confirmation. The process concludes with the invocation of COMMITPROCESSED (Alg. 1), which causes the application's thread to proceed its execution.

3.3 A Partially Replicated STM

In the context of full replication, an unique system-wide identifier is sufficient to unequivocally represent a distributed object. However, when addressing partial replication, not all objects are replicated by all the system's nodes, and thus read operations may require inter-node communication. Consequently, the system must be able to classify objects as local or remote, and to obtain all the necessary information to request the given object from a node that replicates it. To that end, an object's metadata must complement the aforementioned unique identifier with the identifiers of the *groups* of nodes that replicate the associated object.

Algorithm 3 presents the pseudo-code for object serialization in partial replication environments. The process is delegated to the ObS module, by methods WRITEREPLACE and READRESOLVE to methods OBJECTREPLACE and OBJECTRESOLVE, respectively. As its fully replicated counterpart, a partially replicated object $\mathcal{O}$ can be in two possible states: published or private. The serialization process is also similar to the one described in §3.2: public objects nominate their identifier to be serialized in their place, whilst the serialization of private objects requires the generation of the respective metadata and the subsequent effective serialization of $\mathcal{O}$. Additionally, partial replication compels the serialization algorithm to be aware if it was triggered in the context of a remote read operation (ISREADCONTEXT). This distinction is necessary because, when in such context, the requesting node does not hold a local copy of $\mathcal{O}$, and thus the original object must always be transferred.

The de-serialization process also grows from the fully replicated scenario. The difference lies in the need to check if the local node should replicate the incoming object $\mathcal{O}$, i.e., if it belongs to one of the groups identified in $\mathcal{O}$'s metadata. If not, $\mathcal{O}$ is a replica of the original object and is hence returned, otherwise there are still two scenarios to cover: the local node *holds* or *not* a local replica of $\mathcal{O}$. In the first case, $\mathcal{O}$ comprises only the original object's metadata, which must be used to retrieve the local replica. In the second case, $\mathcal{O}$ is a replica of the original object and thus must be stored in the local memory, before being returned.

Unlike with full replication, it is not easy to support transactions commit in partially replicated environments using certification-based protocols. Nodes do not hold enough in-

Algorithm 3 Object Serializer component.

```
memory ← ∅                                              ▷ Local memory

function objectReplace(O)
    if isReadContext(O) then
        return O
    oid ← getMetadata(O)
    if ∃oid then
        return oid
    else                                        ▷ Object O is not published
        oid ← generate fresh metadata
        setMetadata(O, oid)
        group ← getGroup(oid)
        if isLocal(group) then
            memory ← memory[oid ↦ O]
        return O

function objectResolve(O)
    oid ← getMetadata(O)
    group ← getGroup(oid)
    if isLocal(group) then
        obj ← memory(oid)
        if ∃obj then
            return obj
        else
            memory ← memory[oid ↦ O]
            return O
    else
        return O
```

formation to validate all transactions by themselves. The solution is to opt for a voting algorithm, so that all nodes may reach the same decision about which transactions to commit and abort. The multiple protocols proposed for partial replication are all based in the two phase commit protocol (2PC) [14, 15, 17, 18]. In case of a remote read operation, these protocols need to obtain the remote data while still ensuring that the data is consistent.

Some of the DM's modules are specific for partially replicated environments, namely the GP and the DP. In these scenarios, the system's nodes are partitioned into *groups* which in turn replicate a subset of the system's data. This division into groups is carried out by the concrete implementations of the GP. Having knowledge of the nodes that exist in the system and of the desired objects' replication factor, the GP returns a partitioning according to the implemented strategy. Some strategies can be as simple as dividing the nodes by the groups in a round-robin fashion, or as complex as trying to minimize the latency between the nodes in a group. The decision of in which groups to replicate an object is managed by the concrete implementations of the DP. Implementations can range from simple ones, such as distributing the objects randomly by the groups to more complex strategies trying to achieve some kind of load balancing or taking into account previous access patterns.

3.4 Implementation Considerations

Using ReDstm, we implemented a fully and a partially replicated STMs. In order to prove the framework's flexibility and modularity we developed a few different implementations of its components.

For the fully replicated STM we implemented two STM algorithms, namely TL2 [6] and MVSTM [5]. We also implemented two different protocols for the DT module, namely non-voting certification [1] and voting certification [9].

For the partially replicated STM we implemented a STM algorithm as described in [14] and for the DT we implemented the SCORe protocol [14]. Still in the context of partially replicated STM and just for the sake of simplicity, in the evaluation of our framework in §5 we adopted a simple scheme where each data item is replicated by a *single* group and groups are comprised by *disjoint* sets of nodes.

4. COMBINING FULL AND PARTIAL REPLICATION IN REDSTM

In §3.3 we presented the reasoning on how to implement a partially replicated STM in ReDstm, but in fact that is a replicated STM that combines partial with full replication. Supporting partial replication in a DTM framework for a general purpose programming language (GPPL) may build upon the cumulative knowledge of applying such technique to (in-memory) databases. However, the expressiveness of programming languages is much higher than that of database query languages, specially compared to the `put/get` interface of key-value stores. As such, addressing partial replication in this context raises an extra set of challenges, namely: (1) what data should be partially replicated; (2) how to express the replication level in a GPPL; and (3) how to partially replicate objects graphs.

Challenge (2) was addressed in §2.3 with the definition of the `@Partial` annotation, and challenge (3) was addressed in §2.2.2 and §3.3 with the use of specific distribution metadata. This section will thus focus on challenge (1) by reasoning on how to balance full and partial replication in the context of DTM for a GPPL.

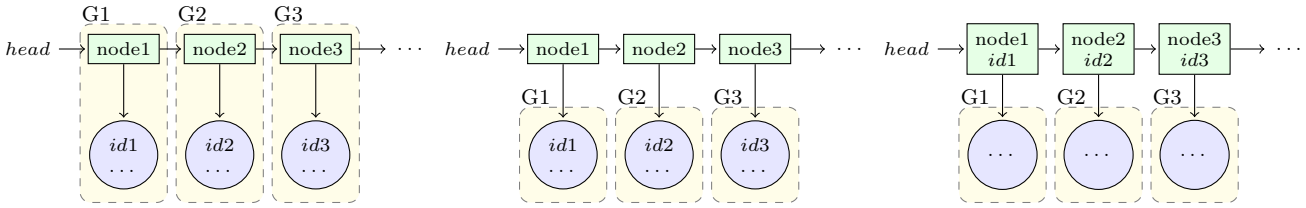
Fully replicated databases keep a copy of every table (and its content) at every node. In turn, *partially* replicated databases confine that strategy to tables, partitioning the enclosed data among the system's nodes. In a way, the information that gives structure to the data (i.e., the data structures—the tables in the case of databases) is always fully replicated. The same approach can be seen in key-value stores, where the data organizing structures (e.g., maps and indexes) exist in every node and just the hard data they contain is partially replicated. In sum, partial replication is, to some extent, always combined with full replication.

We claim that the same reasoning should be applied to DTM in order to mitigate the overhead of remote read operations. A pragmatic example is the linked list. In Fig. 3, we can see three possibilities when partially replicating a list. The dashed rectangles represent different groups, i.e., the data inside a rectangle is partially replicated. Everything outside the dashed rectangles is fully replicated. The circles represent the data stored in each node of the list.

If we imagine that the whole nodes are partially replicated, like in Fig. 3a, the simple task of traversing the list would entail a possibly remote read operation for each traversed node. Even more, since we use Java, a programming language with automatic memory management, we would have to stop relying on its garbage collection mechanism, and we would have to deploy a distributed garbage collection mechanism ourselves. The partial replication of the data structure itself demands the maintenance and management of (strong) references to the objects (at the distributed system level), otherwise the objects could be erased from memory by the Java's garbage collector (GC).

On the other hand, if we just partially replicate the hard data stored in each node (Fig. 3b) the number of remote read operations necessary for traversing the list would be limited by the data we really want to inspect.

An even better option is to just partially replicate the hard data stored in each node and bring the keys associated with



(a) Partially replicated nodes. (b) Partially replicated data (with ids). (c) Partially replicated data (w/o ids).

Figure 3: Different partial replication options.

the data to the nodes themselves, as depicted in Fig. 3c. This approach is particularly good for indexed data structures, such as dictionaries, where this problem gains more relevance. A search in a dictionary traverses the data structure looking for a specific key, and only then inspects the value associated with that key. If we fully replicate the entire structure of the dictionary (including its keys) and partially replicate only the values associated with those keys, we will greatly limit the number of remote read operations.

4.1 Methodology

In line with the previously described possibilities, we can argue that partially replicating the data structures can bring both advantages and disadvantages. On the one hand, if we partially replicate the data structures, the transactions that modify them will only have to be confirmed by a *small* set of the system's nodes (e.g., the nodes that replicate the modified data), but the task of traversing the structures will most likely entail multiple remote read operations. On the other hand, by fully replicating the data structures all the transactions that modify them will have to be confirmed by *all* the system's nodes, but the task of traversing the structures can always be performed locally.

Taking into account all the trade-offs and following the reasoning already described, our key insight to address challenge (1)—what data should be partially replicated—is to fully replicate the (data) structures and to partially replicate only the hard data they store. This way we trade performance when confirming transactions that modify the structures (because they have to be confirmed by all the system's nodes) by performance due to less remote read operations when traversing those structures.

In a generic way, small and/or frequently accessed data should be fully replicated, and big and/or occasionally accessed data should be partially replicated. Extrapolating this into a methodology for distributed data structures:

- the **structure** should be **fully** replicated;
- **search information**, e.g., keys, should also be **fully** replicated; and
- the **hard data** should be **partially** replicated.

The approach taken in §2.3, for the `@Partial` annotation, was devised with the described methodology in mind, so it is tailored for partially replicating just the leafs (or small sub-graphs close to the leafs) of the heap graph. On the one hand, this approach provides a programming model with a certain flexibility that empowers the programmer to adjust the data replication level to the specific characteristics of each application. On the other hand, it requires the programmer to reason about the impact of the replication model in the application's performance. However, given that distributed data structures subsume all the distribution-related concerns, the burden of applying the `@Partial` annotation is confined to the internal implementation of these.

The linked list example in Listings 1-2 applies the replication option depicted in Fig. 3c, where we have the list's structure fully replicated and partially replicate the value object at each node.

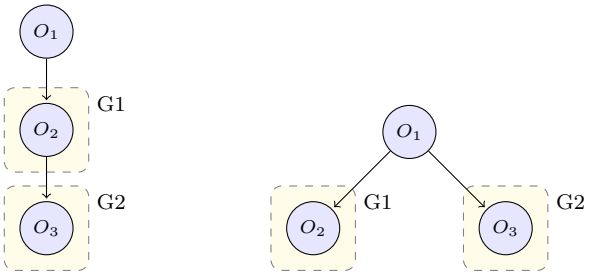
When addressing compound data structures, that build from pre-developed ones, further reasoning may be required. For instance, consider a hash map with collision lists that build up from the list in Listings 1-2. Two possibilities may be considered: (1) to partially replicate the entire collision list of each map entry; or (2) to partially replicate only the nodes of the collision lists. In option (1) our methodology is applied at the hash map level. The structure and keys of the map are fully replicated, while the collision list of each map entry is partially replicated. In option (2) the methodology is applied at the list level. The hash map's implementation is left unaltered from the single-site version, as the `@Partial` annotation only appears in the list's implementation. As a result, we keep the collision lists fully replicated and only partially replicate the nodes of the lists.

4.2 Limitations

As is, our implementation of the programming model has some limitations, namely: (1) cycles including both fully and partially replicated objects; (2) partially replicated objects referring other partially replicated objects in a different replication group. Another aspect that can be seen as a limitation is the fact that the application programmer has no knowledge where the objects are replicated. But that can be fairly controlled by the strategies used by the DP and GP.

Regarding limitation (1) our implementation does not allow cycles in the object graph that include both fully and partially replicated objects. To ensure the semantics of the `@Partial` annotation, edges that flow upstream from partially to fully replicated objects are not permitted. In the example in Fig. 2, if node E was annotated with `@Partial`, the edge between G and B could not exist, since by annotating node E the programmer was declaring that all the objects downstream, represented by the shaded nodes, should be partially replicated. An edge from node G to node B would intuitively mean that node B should be partially replicated, but since it is upstream from node E it also means that it should be fully replicated, creating an ambiguity.

Limitation (2) results from an implementation decision. We take advantage of the Java's GC and use weak references in the map that associates distribution metadata to distributed objects (e.g., *memory* in Algorithm 3). Because we use weak references, when an object is no longer referenced by others in the application, the GC is free to erase that object from memory. Thus, the case in Fig. 4a is not possible in our implementation. Since objects O_2 and O_3 are in different groups, O_3 will not be referenced by O_2 leaving



(a) Cascading partial data replication.

(b) Tree-like partial data replication.

Figure 4: Limitation of our implementation of the programming model.

object O_3 at the mercy of the GC. On the other hand, the case in Fig. 4b is possible since both objects O_2 and O_3 are referenced by a fully replicated object.

During the adaptation of the benchmarks these limitations where overcome with relative ease. In fact, by following the recommended methodology, the management of the `@Partial` annotation is made at the data structure level.

5. EXPERIMENTAL EVALUATION

The evaluation of the ReDstm framework addressed three main questions: (1) What is the impact in memory consumption if we switch from a full replicated to a partially replicated system? (2) How does the replication factor influences the throughput of both fully and partially replicated DTM applications? (3) How does the ratio of structure data vs. hard data influences the throughput in the presence of full and partial replication?

5.1 Experimental Setup

Experimental Test-bed. All the experiments were conducted in a heterogeneous cluster with 8 nodes. The first 5 nodes have 2 × Quad-Core AMD Opteron 2376 2.3 GHz and 16 GB of RAM. The last 3 nodes have 1 × Quad-Core Intel Xeon X3450 2.66 GHz (with Hyper-Threading) and 8 GB of RAM. All machines run *Linux 2.6.26-2* (Debian 5.0.10) and are interconnected via private Gigabit Ethernet. The installed Java platform is OpenJDK 6 (IcedTea 1.8.10).

To allow a comparison between the two implemented replicated STMs, we used two configurations of ReDstm: (i) a full replication configuration that uses MVSTM as the TM layer and the DT implements non-voting certification; and (ii) a partial replication configuration that uses SCORe and its multi-version STM algorithm. Additionally, we have set to *two* the replication factor of each data item [14] (if nothing said on the contrary). With regard to the underlying GCS we used JGroups [2] version 3.4.1.

Benchmarks. The Red-Black Tree (RBT) micro-benchmark is composed of three transactions: insertions, which add an element to the tree (if not present); deletions, which remove an element from the tree (if present); and searches, which search the tree for a specific element. Insertions and deletions are said to be *write* transactions. This benchmark is characterized by very small and fast transactions that perform little work and exhibit low contention.

The Vacation benchmark is part of the STAMP suite [13]. It emulates a travel reservation systemimplemented as a set of binary trees keeping track of customers and their reservations. There are three transactions (all write transactions): reservations, cancellations and updates. We added a new *read-only* transaction that consults reservations.

The TPC-W benchmark [20] models an online book store. Servers handle user requests such as browsing, adding products to a shopping cart, or placing an order. The user requests are much more demanding in terms of processing power when compared to the previous benchmarks. The workload used consists of 95% of read-only transactions.

In all the benchmarks, the use of the `@Partial` annotation was confined to the data structures. So, only the values stored in the nodes of the binary trees and hash maps used by the benchmarks were partially replicated (according to the methodology described in §4.1).

5.2 Results

Memory Consumption. The out-of-the-box RBT benchmark maps integers to integers, i.e., both keys and values are integers. With this benchmark, we found that ReDstm using config. (ii) (partial replication) consumes a larger, but negligible, amount of memory when compared to config. (i) (full replication). In this case, the problem is twofold: partially replicating an integer does not allow to reduce memory usage as the memory for the integer is still reserved (this goes for all primitive data types in Java); and the distribution metadata for config. (ii) store more information than the metadata used by config. (i).

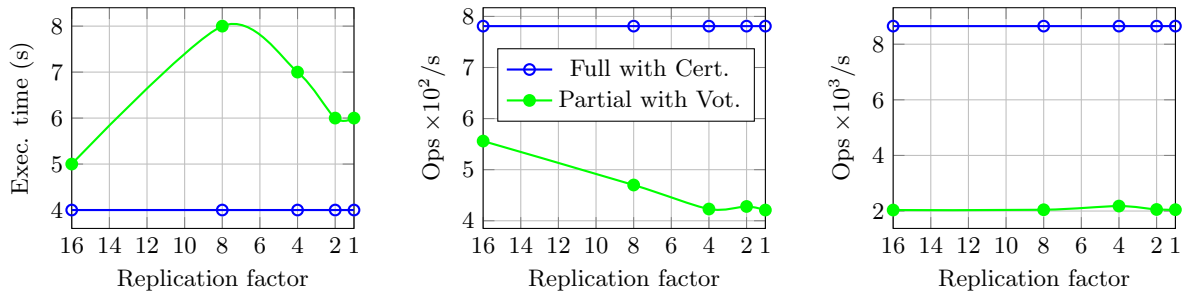
To verify our intuition, we modified the benchmark in order to map integers to jpeg images, each with 3 MB in size. In this case, as expected, by decreasing the data's replication factor, the memory consumed by each node decreases as well, reaching around 55% less memory at replication factor 2.

Partial & Full Data Replication. After running the three benchmarks, namely Adapted Vacation and RBT, both with 10% writes, and TPC-W with the browsing mix, Fig. 5 shows that the performance of config. (ii) is very low regarding its full replication counterpart. This results are explained by the simple reason that, in all these benchmarks, the majority of the write transactions modify the data structures (around 90% in Adapted Vacation, 85% in TPC-W and 100% in RBT), and those structures are *fully replicated*. Thus, those write transactions require a distributed confirmation involving *all* the system's nodes, which becomes very expensive in config. (ii) since it uses 2PC, while config. (i) uses a single atomic broadcast.

These benchmarks reveal a known but rather important aspect of config. (ii): SCORe was not designed for environments mixing both full and partial replication.

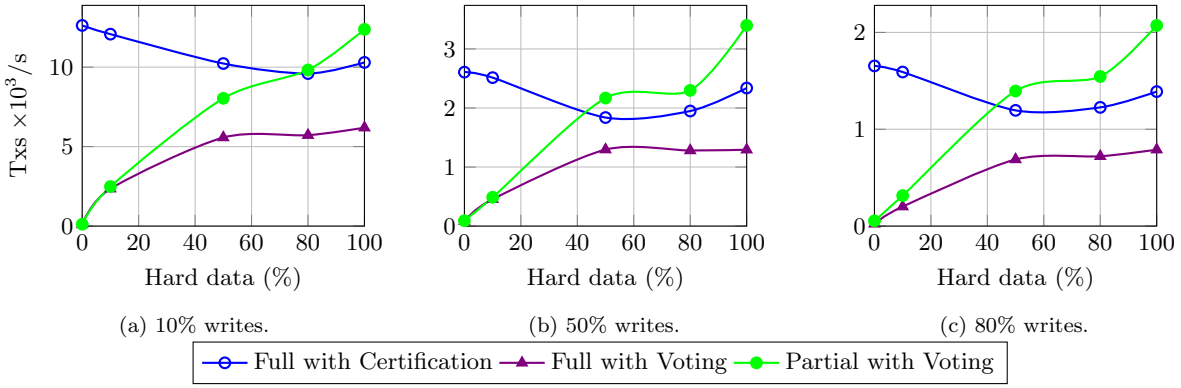
In order to see if the protocol used by config. (ii) can take advantage of its nature in some scenarios, we modified the RBT benchmark in order to have control over the amount of write transactions that modify partially replicated data, i.e., hard data accessed by transactions. Thus, in this version of the benchmark, named *Adapted RBT*, we have transactions that modify the tree's structure while others modify the values stored in the tree's nodes. This way, config. (ii) is able to leverage its protocol and some transactions will only require confirmation of a *subset* of the system's nodes. Fig. 6 depicts the results of running this version of the benchmark with 10, 50 and 80% of write transactions.

With small amounts of hard data accessed, config. (ii) performs poorly, but as we increase that mount it starts to match config. (i) and is even able to surpass it. With 10% of writes, config. (ii) starts to outperform config. (i) at 80%



(a) Adapted Vacation. (b) TPC-W. (c) RBT.

Figure 5: Influence of the replication factor in the throughput of the Adapted Vacation and RBT benchmarks with 10% writes, and of TPC-W with the browsing mix (16 instances).



(a) 10% writes. (b) 50% writes. (c) 80% writes.

Figure 6: Sensitiveness of the replication strategies to the amount of hard data accessed in the Adapted RBT (8 instances).

of hard data accessed, and with 50 and 80% of writes, it outperforms config. (i) starting just at 50% of hard data accessed. We present config. (ii) mimicking full replication (Full with Voting) just as a baseline. Since *all* write transactions have to be confirmed by *all* the system's nodes, it will always perform short of the desired.

Config. (i) is affected by the variation in the amount of hard data accessed due to particularities of the benchmark. The two write transactions of RBT only perform work in certain cases: insertions only add an element if it *is not already present*, and deletions only remove an element if it *is present*. Since elements are randomly chosen, write transactions may become read-only (which do not need distributed confirmation), e.g., if an element already exists when doing an insertion. On the other hand, in our adaptation, *all* write transactions choose elements that *definitely* exist, thus they always perform work and need a distributed confirmation. So, by increasing the amount of hard data accessed, config. (i) performs more work, thus affecting its performance.

Figs. 7 and 8 show the latency when performing remote read operations and when committing transactions, respectively. When executing a remote read, config. (ii) experiences increased latency when the replication factor decreases due to the fact that less nodes replicate each object, which leads to more remote accesses. In Fig. 8, we can see that config. (i) is not influenced by the amount of hard data accessed. But config. (ii) starts to experience lower latencies when the replication factor decreases, since transactions have to be confirmed by less nodes, making commit operations faster.

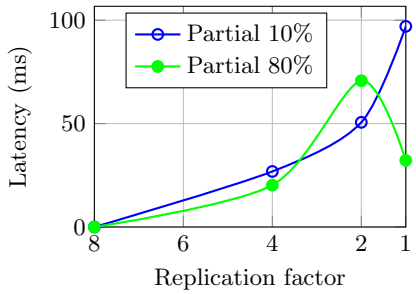


Figure 7: Remote read latency on Adapted RBT with 10% writes and 10% and 80% of hard data accessed (8 instances).

6. RELATED WORK

Transactional Memory. With the recent thrust in DTM research, a handful of different frameworks have been proposed with the goal of facilitating the development, testing and evaluation of the multiple proposed memory consistency protocols. All of the best known DTM frameworks present a great level of intrusion, requiring that a significant part of the application code is rewritten to comply with the specificities of each framework. DiSTM [12] is a master/slave approach to DTM. Objects have to implement special interfaces and are created using transactional factory methods. Transactions have to be executed by a special thread class. D^{2}STM [4] leverages replication to enhance dependability, so data is fully replicated. It is based on JVSTM [3], hence objects have to be wrapped in boxes and have to be accessed by special `get` and `set` methods. Transactions are simple methods annotated with a `@Atomic` annotation. HyFlow [16] is a Java framework for DTM with many pluggable components. Objects have to implement a special interface and are identified by an unique identifier that serves as its reference in the system. These identifiers are exchanged by their corresponding objects using a `Locator` instance. Transactions are

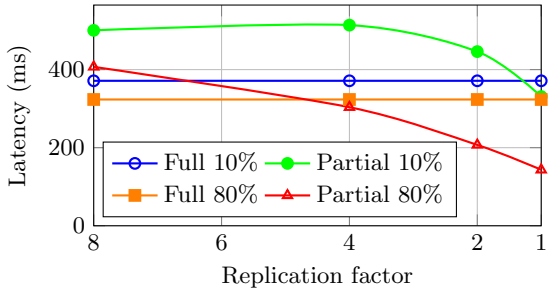


Figure 8: Distributed commit latency on Adapted RBT with 10% writes and 10% and 80% of hard data accessed (8 instances).

defined as `@Atomic`-annotated methods. HyFlow2 [21] is a framework in Scala and its programming model is very similar to one of HyFlow, but instead it uses Scala constructs. Compared with all these frameworks, ReDstm manages to be a modular framework, while presenting a much lower level of intrusion to the application code, requiring only the use of three annotations. This higher-level programming model greatly simplifies the work of the programmer.

Concerning the data replication strategies, DiSTM has a master/slave approach, D^{2}STM supports full data replication, and both HyFlow and HyFlow2 support data-flow and control-flow approaches. HyFlow-related research also dwells into the partial replication field [10], however the focus is on the scheduling of memory transactions, rather than on the support of different levels of replication in DTM As a result, ReDstm is more flexible, allowing the implementation of a wider range of data replication strategies.

Databases. A great part of the work presented on this paper was built upon the cumulative knowledge of the databases research field. Solutions natively oriented to partially replicated transactional systems may be divided depending on whether they can be considered genuine, i.e., when the transaction's confirmation only involves the nodes keeping copies of the data accessed by the transaction, and on the specific consistency guarantees they provide. Serrano et al. [18] provide a non-genuine protocol, where the confirmation of a transaction requires interaction with all the nodes in the system. Compared to this approach, genuine schemes have been shown to achieve significantly higher scalability [15].

Concerning the granted consistency guarantees, in [18], the authors identify 1-copy-serializability (1CS) as a limitation when designing scalable replicated solutions, and propose a protocol offering 1-copy-snapshot-isolation, whereby replicas run under snapshot isolation. In turn, with P-Store [17], Schiper et al. go back to 1CS proposing the first genuine partial replication protocol offering this kind of consistency guarantee. But this protocol still imposes that read-only transactions undergo a distributed confirmation phase.

7. CONCLUSIONS

In this paper we presented ReDstm, a modular DTM framework that provides support for a wide range of data replication models, from full to partial data replication, in a general purpose programming language.

We used ReDstm to study how different replication models can be integrated in the implementation of distributed STMs. From our experiences we devised a methodology for the implementation of distributed data structures in such contexts.

We evaluated the impact of different replication models in the execution of three known benchmarks, being able to conclude that: (i) partial replication contributes to the system's scalability by reducing the amount of data stored at each node; (ii) the impact of the replication model in transaction throughput is sensible to the amount of transactions that manipulate structuring and hard data. The benefits of partial replication are directly proportional to the number of transactions that manipulate only hard data.

Acknowledgments

This work was partially supported by the Euro-TM EU COST Action IC1001 and the Portuguese national research project PTDC/EIA-EIA/113613/2009 (Synergy-VM).

8. REFERENCES

[1] D. Agrawal et al. Exploiting atomic broadcast in replicated databases. In *Euro-Par*, 1997.

[2] Bela Ban. JGroups - A Toolkit for Reliable Multicast Communication. `http://www.jgroups.org`, 2013.

[3] J. Cachopo et al. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 2006.

[4] M. Couceiro et al. D2stm: Dependable distributed software transactional memory. In *PRDC*, 2009.

[5] R. J. Dias et al. Efficient support for in-place metadata in java software transactional memory. *Concurrency and Computation: Practice and Experience*, 2013.

[6] D. Dice et al. Transactional locking II. In *DISC*, 2006.

[7] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. 2004.

[8] M. Herlihy et al. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[9] B. Kemme et al. A suite of database replication protocols based on group communication primitives. In *ICDCS*, 1998.

[10] J. Kim et al. Scheduling transactions in replicated distributed software transactional memory. In *CCGRID*, 2013.

[11] G. Korland et al. Deuce: Noninvasive software transactional memory in java. *Transactions on HiPEAC*, 2010.

[12] C. Kotselidis et al. Distm: A software transactional memory framework for clusters. In *ICPP*, 2008.

[13] C. C. Minh et al. Stamp: Stanford transactional applications for multiprocessing. In *IISWC*, 2008.

[14] S. Peluso et al. Score: A scalable one-copy serializable partial replication protocol. In *Middleware*. 2012.

[15] S. Peluso et al. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2012.

[16] M. M. Saad et al. Hyflow: A high performance distributed software transactional memory framework. In *HPDC*, 2011.

[17] N. Schiper et al. P-store: Genuine partial replication in wide area networks. In *SRDS*, 2010.

[18] D. Serrano et al. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC*, 2007.

[19] N. Shavit et al. Software transactional memory. In *PODC*, 1995.

[20] Transaction Processing Performance Counsil. TPC Benchmark W. http://www.tpc.org/tpcw, 2013.

[21] A. Turcu et al. Hyflow2: A high performance distributed transactional memory framework in scala. In *PPPJ*, 2013.