Transactional Forwarding: Supporting Highly Concurrent STM in Asynchronous Distributed Systems

Mohamed M. Saad and Binoy Ravindran
Electrical & Computer Engineering Department
Virginia Tech
Blacksburg, VA 24061, USA,
Email: {msaad, binoy}@vt.edu

***Abstract*—Distributed software transactional memory (or DTM) is an emerging promising model for distributed concurrency control, as it avoids the problems with locks (e.g., distributed deadlocks), while retaining the programming simplicity of coarse-grained locking. We consider DTM in Herlihy and Sun's data flow distributed execution model, where transactions are immobile and objects dynamically migrate to invoking transactions. To support DTM in this model and ensure transactional properties including atomicity, consistency, and isolation, we develop an algorithm called Transactional Forwarding Algorithm (or TFA). TFA guarantees a consistent view of shared objects between distributed transactions, provides atomicity for object operations, and transparently handles object relocation and versioning using an asynchronous version clock-based validation algorithm. We show that TFA is opaque (its correctness property) and permits strong progressiveness (its progress property). We implement TFA in a Java DTM framework and conduct experimental studies on a 120-node system, executing over 4 million transactions, with more than 1000 active concurrent transactions. Our implementation reveals that TFA outperforms competing distributed concurrency control models including Java RMI with spinlocks, distributed shared memory, and directory-based DTM, by as much as $13\times$ (for read-dominant transactions), and competitor DTM implementations by as much as $4\times$.**

Keywords: Distributed Systems, Software Transactional Memory, Asynchronous Network, Concurrency

I. Introduction

Concurrency control is difficult in distributed systems—e.g., distributed race conditions are complex to reason about. The problems of lock-based concurrency control only exacerbate in distributed systems. Distributed deadlocks, livelocks, lock convoying, and composability are significant challenges. In addition, locks provide naive serialization of concurrent code with partial dependency, which is a severe problem for long critical sections. There is a trade off between decreasing lock overhead (lock maintenance requires extra resources) and decreasing lock contention when choosing the number of locks for synchronization.

Transactional memory (TM) is an alternative model for accessing shared memory objects, without exposing locks in the programming interface, to avoid the drawbacks of locks. TM originated as a hardware solution, called HTM [21], was later extended in software, called STM [38], and in hardware/software combination, called Hybrid TM [28]. With TM, programmers organize code that read/write shared objects as *transactions*, which appear to execute atomically. Two transactions conflict if they access the same object and one access is a write. When that happens, a contention manager resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started. Thus, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). In addition to a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking implementations [8].

Though TM has been extensively studied for multiprocessors [19], relatively little effort has focused on supporting it in distributed systems. Distributed applications introduce additional challenges over multiprocessor ones. For example, scalability necessitates decentralization of the application and the underlying infrastructure, which precludes a single point for monitoring or management.

Distributed STM (or DTM) can be supported in any of the classical distributed execution models, which can be broadly classified based on the mobility of transactions or objects. In the control flow model (e.g., [3]), objects are immobile and transactions access objects through remote procedure calls (RPCs), and often use locks over objects for ensuring object consistency. In contrast, in the data flow model (e.g., [22]), transactions are immobile, and objects move through the network to requesting transactions, while guaranteeing object consistency using cache coherence protocols. The dataflow model[1] has been primarily used in past DTM efforts—e.g., [22], [34]. Hybrid models have also been studied (e.g., [7]), where transactions or objects are migrated, based on access profiles, object size, or locality.

Two strategies currently exist for handling concurrent updates on distributed objects in DTM: a) broadcasting read/write sets to all nodes (e.g., [24]), and b) stamping objects with a version number to distinguish between each update (e.g., [27]) (see Section II for past and related work). Broadcasting (transactional read/write sets, or memory differences) in distributed systems is inherently non-scalable, as messages transmitted grow quadratically with the number of nodes. On the other hand, object versioning is a significant challenge in DTM, because distributed systems are inherently asynchronous, and using a global clock often incurs significant message overhead [32].

We develop an algorithm called Transactional Forwarding Algorithm (or TFA) that ensures (distributed) transactional properties including atomicity, consistency, and isolation in the dataflow DTM model (Section IV). In TFA, each node maintains its own local clock, which is asynchronously advanced (e.g., whenever any local transaction commits), and the "happens-before" relationship [25] is efficiently established (message-wise) between relevant events (e.g., write-after-write, read-after-write) for distributed transactional conflict detection. Additionally, the algorithm employs an early valida-

[1]We use "dataflow" to refer to the Herlihy and Sun immobile transaction/-mobile object TM model [22]. In other contexts (e.g., [13]), dataflow may refer to data-driven computations.

tion mechanism to accommodate asynchronous clocks. TFA permits multiple non-conflicting updates to proceed concurrently, and allows multiple concurrent threads to execute transactions at each node. By Transaction Forwarding, we don't mean forwarding transactions from one node to another; instead, the idea of "transaction forwarding" is about advancing the timestamps on some transactions to produce fewer conflicts in a timestamp-based concurrency control mechanism.

We show that TFA is opaque i.e., its correctness property, and permits strong progressiveness i.e., its progress property (Section V). Informally, opacity ensures transactional linearizability and consistent memory view for committed and aborted transactions. Strong progressiveness ensures that non-conflicting transactions are guaranteed to commit, and at least one transaction among conflicting transactions is guaranteed to commit. We also establish a message upper bound for TFA. In TFA, objects are acquired at commit time, while other pessimistic approaches (e.g., [1]) acquire objects at encounter time.

We implement TFA in the HyFlow DTM framework [36], and conduct experimental evaluations (Section VI). Our results reveal that TFA outperforms competitor DTM implementations [5], [29] by as much as $2-4\times$, and Java RMI with spinlocks and distributed shared memory by as much as $13\times$ (for read-dominant transactions) and as low as $1\times$ (for write-dominant transactions). These results were obtained for workloads including a distributed STAMP [9] benchmark, other distributed applications, and distributed data structures on a 120-node system, executing over 4 million transactions, with more than 1000 active concurrent transactions.

II. Related Work

The classical solution for handling shared memory during concurrent access is lock-based techniques [2], [23], where locks are used to protect shared objects. Locks have many drawbacks including deadlocks, livelocks, lock-convoying, priority inversion, non-composability, and the overhead of lock management.

Distributed Transactional Memory. DTM models can be classified based on the mobility of transactions and objects. Mobile transactions (e.g., [3]) use an underlying mechanism (e.g., RPC) for invoking operations on remote objects. The mobile object model (e.g., [22], [34]) allows objects to move to requesting transactions, and guarantees object consistency using cache coherence protocols (e.g.,[22]). DTM models can also be classified based on the number of objects. Some proposals allow multiple copies or replicas of objects. Object changes can then be a) applied locally, invalidating other replicas [31], b) applied to one object (e.g., latest version of the object [14]), which is discovered using directory protocols (e.g., [12]), or c) applied to all replicated objects [26]. DTM can also be classified based on system architecture: cache-coherent DTM (cc DTM) [22], where a small number of nodes (e.g., 10) are interconnected using message-passing links [22], [12], and a cluster model (cluster DTM), where a group of linked computers work closely together to form a single computer [7], [27], [24], [10], [33]. The most important difference between the two is communication cost. cc DTM assumes a metric-space network between nodes, while cluster DTM differentiates between memory access to local cluster and other remote clusters.

While these efforts focused on DTM's theoretical properties, several other efforts developed implementations. In [7], Bocchino *et. al.* proposed a word-level cluster DTM. They decompose a set of existing cache-coherent STM designs into a set of design choices, and select a combination of such choices to support their design. They show how remote communication can be aggregated with data communication to improve scalability. However, in this work, each processor is limited to one active transaction at a time, which limits concurrency. Also, in their implementation, no progress guarantees are provided, except for deadlock-freedom. In [27], Manassiev *et. al.* present a page-level distributed concurrency control algorithm for cluster DTM, which detects and resolves conflicts caused by data races for distributed transactions. Their implementation yields near-linear scaling for common e-commerce workloads. In their algorithm, page differences are broadcast to all other replicas, and a transaction commits successfully upon receiving acknowledgments from all nodes. A central timestamp is employed, which allows only a single update transaction to commit at a time. The use of broadcasting object changes and a central timestamp technique yield acceptable performance for small number of nodes (8 nodes are used in [27]). Yet, both techniques suffer from scalability with increasing number of nodes.

Kotselidis *et. al.* present the DiSTM [24] object-level, cluster DTM framework, as an extension of DSTM2 [20], for easy prototyping of TM cache-coherence protocols. They compare three cache-coherence protocols on benchmarks for clusters. They show that, under the TCC protocol [18], DiSTM induces large traffic overhead at commit time, as a transaction *broadcasts* its read/write sets to all other transactions, which compare their read/write sets with those of the committing transaction. Using lease protocols [15], this overheard is eliminated. However, an extra validation step is added to the sole master node. In addition, bottlenecks are created upon acquiring and releasing the leases, besides serializing all update transactions at the single master node. These implementations assume that every memory location is assigned to a *home* processor that maintains its access requests. Also, a central, system-wide ticket is needed at each commit event for any update transaction (except [7]).

Inspired by the recent database replication approaches [30], Carvalho *et. al.* present $GenRSTM$ [29] as a generic STM framework based on D^2STM [10]. Here, STM is replicated on distributed system nodes, and strong transactional consistency is enforced at commit time by a non-blocking distributed certification scheme. GenRSTM shows good performance for upto eight replicas in [10]. However, the Atomic Broadcast (ABcast) primitive [11] that they use limits extending the replication technique for larger number of nodes, thereby limiting scalability.

Bieniusa *et. al.* present $DecentSTM$ [5], a fully decentralized object-based STM algorithm, which implements snapshot isolation semantics. It relies on immutable data structures, and uses a randomized consensus protocol to guarantee consistency of the mutable parts. They host distributed global shared memory on a concept called *runtimes*, which are small memory servers. Application threads operate normally on local data, and access global shared data under the control of the DecentSTM library.

TFA is an object-level, lock-based algorithm with lazy acquisition. TFA is fully distributed without the need for central components, or central clocking mechanisms. Unlike other DTM implementations [24], [27], [29], TFA doesn't rely on message broadcasting or a central clock [27]. This approach reduces network traffic, and enables the algorithm to perform well for write-dominated workloads, while yielding comparable or superior performance for read-dominated workloads, with respect to other distributed concurrency control models (e.g., Java RMI, distributed shared memory [31]), as we show in Section VI.

III. System Model and Preliminaries

We consider an asynchronous distributed system, similar to Herlihy and Sun [22], consisting of a set of N nodes, $N_1, N_2, ..., N_n$, which are fully connected using message-passing FIFO links or through an overlay network. Each shared object has an unique identifier, and is initially assigned to a "home" node. However, an object may be replicated or may migrate to any node. TFA is responsible for caching local copies of remote objects and changing object ownership. It is also responsible for ensuring that only one writable version of an object exists at any given time in the network. Without loss of generality, objects export only *read* and *write* methods (or operations). Thus, we consider them as shared registers.

Transactions are immobile, and each transaction is associated with a certain node. Thus, a node N_x executes a transaction T, which is a sequence of operations on objects $o_1, o_2, ..., o_s$, where $s \geq 1$. We assume that the majority of transactions are concurrent. A

transaction can have one of three status: live, committed, or aborted. An aborted transaction is restarted as a new transaction. When a transaction attempts to access an object, a cache-coherence protocol (e.g., Arrow [12], Ballistic [22]) locates the current cached copy of the object in the network, and moves it to the requesting node's cache. Changes to the ownership of an object occurs at the successful commit of the object-modifying transaction.

Each node has a local clock, lc, which is advanced whenever any local transaction commits successfully. Since a transaction runs on a single node, it uses lc to generate a timestamp, *write version* (wv), during its commit step. The current clock value is piggybacked on all messages, recipient node updates its clock to higher value than the received clock if its local clock value is lower.

We use a grammar similar to the one in [17], but extend it for distributed systems. Let $O = \{o_1, o_2, ...\}$ denote the set of objects shared by transactions. Let $T = \{T_1, T_2, ...\}$ denote the set of transactions. Each transaction has an unique identifier, and is invoked by a node (or process) in a distributed system of N nodes. We denote the sets of shared objects accessed by transaction T_k for read and write as *read-set(T_k)* and *write-set(T_k)*, respectively.

A history *H* is defined as a sequence of operations, read, write, commit, and abort, on a given set of transactions. Transactions generate events when they perform these operations. Let the relation $\prec$ represent a partial order between two transactions. Transactions T_i and T_j are said to be conflicting in H on an object O_x, if 1) T_i and T_j are live (i.e., non-committed or non-aborted yet) in H, and 2) O_x is accessed by both T_i and T_j, and is present in at least one of the *write-sets* of T_i or T_j.

We denote the set of conflicting objects between T_k and any other transaction in history H as $conf(T_k)$. Let Q be any subset of the set of transactions in a history H. We denote the union of sets $conf(T_k) \forall T_k \in Q$ as $conf(Q)$. Any operation on $conf(Q)$ represents a relevant transactional event to our algorithm. Using a clock synchronization mechanism, we build a partial order between relevant transactions; otherwise any arbitrary order of transactions can be used to construct H.

IV. The TFA Algorithm

Rationale. The problem of locating objects is outside the scope of our work — we can use any directory or cache coherence protocol for this (e.g., Arrow [12], Ballistic [22]). We assume a *Directory Manager* module that will locate objects. The *Directory Manager*'s interface includes two methods: 1) *publish(x, N_c)* that registers the current node, N_c, as the owner of a newly created object O_x with identifier x, or modifies O_x's old owner to the called node; and 2) *locate(x)*, which finds the owner node of object O_x.

Each transactional memory location (e.g., word, page, or object, according to the desired granularity) is associated with a versioned-write-lock. A versioned-write-lock uses a single bit to indicate that the lock is taken, while the rest of the bits hold a version number. This number is changed by every successful transactional commit. Each node maintains its own local clock. When a transaction starts, it reads the current node clock, and can subsequently commit only when all its read objects have a lower version than the one it obtained at the start time (i.e., the objects weren't updated by other concurrent transactions). Upon successful commit, a transaction stamps its modified objects with the current clock value, and advances the node clock.

Clocks are asynchronously advanced, which invalidates the commit procedure that compares transaction starting times (relative to node local clocks) and object versions (relative to different node clocks). To solve this problem, we develop a transaction "forwarding" mechanism which, at certain situations, shifts a transaction to appear as if it started at a later time (unless it is already doomed due to conflicts in the future). This technique helps in detecting doomed transactions and terminates them earlier, and handles the problem of asynchronous clocks.

Algorithm Overview. Figures 1–4 describe TFA's main procedures. When a transaction begins, it reads the current clock value of the node on which it is executing (Figure 1). Let us call this clock value "write version" (*wv*). During execution, a transaction will maintain the read-set and the write-set as mentioned before. However, read and write operations may involve access to remote objects. Whenever a remote object is accessed, a local object copy is created and cached at the current node till the transaction terminates. A transaction makes object modifications to a local copy of the object. At a read operation, the Bloom Filter [6] is used to check if the read-object appears in the write-set. If so, the last value written by the current transaction is retrieved.

An object may be accessed locally or remotely. Access of local objects is preceded by a post-read validation step to check if the object version $<$ *wv*; otherwise the transaction is aborted. In contrast, as remote objects use different clocks (clocks of their owner nodes), such a straightforward validation cannot be done. Providing clock versioning for validation of remote objects, without affecting performance through additional synchronization messages, is the main challenge in the design of TFA.

Recall that each node has a local clock that works asynchronously according to its local events and can be advanced only when needed. We present a novel mechanism, called *Transaction Forwarding*, which efficiently provides early validation of remote objects and guarantees a consistent view of memory, in the presence of asynchronous clocks.

Transaction forwarding. By this, we imply that a transaction, which started at time *wv* needs to advance its starting time to *wv'*, where *wv'* $>$ *wv*. To apply such a step to a transaction, none of the objects of the transaction's *read-set* must have changed their version to a higher value than *wv*; otherwise, the transaction is aborted as one of its read-set objects has been changed by another transaction, producing a higher version number than the original *wv*. To ensure this, an early commit-validation procedure is performed. If the validation succeeds, then we are sure that no intermediate changes have happened to read-set objects, and the transaction can safely change its starting time to *wv'*.

Figures 2–3 illustrate Transaction Forwarding, which works as follows:

- The *sender node (transaction node)* sends a remote read request to the object owner node. The current node clock value, called *lc*, is piggybacked on this message.
- Upon receiving the message at *receiver node (object owner node)*, a copy of the object is sent back, and the current clock value *rc* is included in the reply. In addition, the incoming clock value *lc* is extracted and compared against the current clock value *rc*. If *rc* $<$ *lc*, then *rc* is advanced to the value of *lc*; otherwise nothing is changed.
- When the sender node receives the reply, validation is done as follows: if *rc* $\leq$ *wv*, then the object can be read safely; otherwise, the current clock value, *lc*, is advanced to the value *rc*, and the transaction is forwarded to *rc*.

When a transaction completes, we need to ensure that it reads a consistent view of objects (Figure 4). This is done as follows:

1) Acquire the lock for each object in write-set in any appropriate order to avoid deadlocks. As some (or all) of these objects may be remote, a lock request is sent to the owner node. The owner node will try to acquire the lock. If the lock cannot be acquired, the owner will spin till it is released or the owner will lose object ownership. If the lock cannot be acquired for any of the objects, the transaction is aborted and restarted.
2) Revalidate the read-set. This ensures that a transaction sees a consistent view of objects. Upon successful completion of this step, a transaction can proceed to commit safely.
3) Increment and get local clock value *lc*, and write the retrieved clock value in the version field of the acquired locks. For local objects, changes to the object can be safely committed

Require: Transaction $trans$, Node $node$
Ensure: Initialize transaction.

```
1: trans.node = node
2: trans.wv = node.clock
```

Fig. 1. Transaction::Init

Require: Transaction $trans$, ObjectID id
Ensure: Open shared object for current transaction.

```
1: Node owner = findObjectOwner(id)
2: Object obj = node.RetrieveObject(trans.node, id)
3: if obj.remote then
4:    if trans.node.clock < obj.owner.clock then
5:       trans.node.clock = obj.owner.clock
6:    end if
7:    if trans.wv < obj.owner.clock then
8:       for all obj in transaction.readSet do
9:          if obj.version > trans.wv then
10:            return rollback()
11:         end if
12:      end for
13:      trans.wv = obj.owner.clock
14:   end if
15: else
16:   if obj.version > trans.wv then
17:      return rollback()
18:   end if
19: end if
20: return obj
```

Fig. 2. DirectoryMgr::OpenTransactional

Require: Node $requester$, ObjectId id
Ensure: Send a copy of object identified by given id and owned by current node to the requester node.

```
1: if this.clock < requester.clock then
2:    this.clock = requester.clock
3: end if
4: return LocalObjects.get(id)
```

Fig. 3. Node::RetrieveObject

Require: Transaction $trans$
Ensure: Commit transaction if valid and rollback otherwise.

```
1: for all obj in transaction.writeSet do
2:    obj.acquireLock()
3: end for
4: for all obj in transaction.readSet do
5:    if obj.version > trans.wv then
6:       return rollback()
7:    end if
8: end for
9: trans.node.clock ++
10: for all obj in transaction.writeSet do
11:    obj.commitValue()
12:    obj.setVersion(trans.node.clock)
13:    obj.releaseLock()
14:    if obj.remote then
15:       DirectoryManager.setOwner(obj, trans)
16:    end if
17: end for
```

Fig. 4. Transaction::Commit

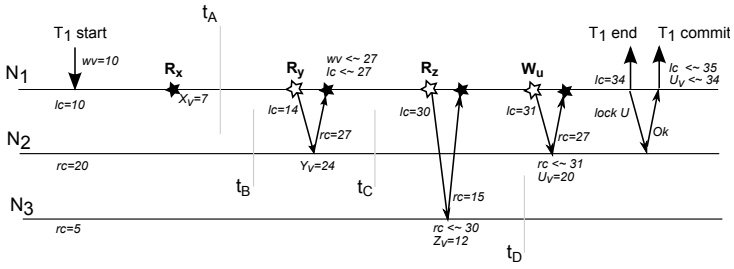


Fig. 5. An execution of a distributed transaction under TFA.

to the main copy, while for remote objects, we simply publish the current node as the new owner of the object using the *DirectoryManager* publish service.

4) For local objects in the write-set, release the acquired locks by clearing the write-version lock bit. The remote locks need not be released, as changing the ownership handles this implicitly.

An aborted transaction releases all acquired locks (if any), clears its read and write sets, and restarts again by reading new *wv*.

Example. Figure 5 illustrates an example of how TFA operates in a network of three asynchronous nodes, N_1, N_2, and N_3. Initial values of the respective node clocks are 10, 20, and 5. Lines between the nodes represent requests and replies, and stars represent object access. Any changes in the clock values are due to successfully committed transactions. Such clock changes are omitted from the figure for simplicity.

Transaction T_1 is invoked by node N_1 with a local clock value, *lc*, of 10. Thus, T_{1wv} equals 10. Afterwards, T_1 reads the value of local object *X*, finds its version number $7 < T_{1wv}$, and adds it to its read-set. The remote object *Y* is then accessed for read. N_1 sends an access request to N_2 (*Y*'s owner node) with its current clock value *lc*. Upon receiving the request at N_2 at time 27 (according to N_2's clock), N_2 replies with the object value and its local clock. N_1 processes the reply and finds that it has to advance its local clock to time 27. In addition, transaction forwarding needs to be done. T_{1wv} is therefore set to 27. Furthermore, early commit-validation is done on the read-set to ensure that this change will not hide changes happened to any object in the read-set since the transaction started (at any time t_A).

Subsequently, T_1 accesses object *Z* located at node N_3, and includes its local clock value to the request. After N_3 replies with a copy of the object and its local time, N_3 detects that its time lags behind N_1's time. Thus, N_3 will advance its time to 30 (the last detected clock value from N_1). Note that in this case, N_1 will not advance its clock, nor will do transaction forwarding, as it has a leading clock value.

Now, T_1 requests object *U* at node N_2. Assume that N_2's clock value is still 27 since the last request, while N_1 advances its clock due to other transactions' commit. Now, N_2 will advance its clock to 31 upon receiving object *U*'s access request.

Eventually, T_1 completes its execution and does the commit-validation step by acquiring locks on objects in its write-set (i.e., *U*), and validating versions of objects in its read-set (i.e., *X*, *Y*, and *Z*). Upon successful validation, N_1's local clock is incremented atomically and its old value is written to *U*'s versioned-lock. N_1 is published as the new owner of the write-set objects.

Several points are worth noting in this example:

- Clocks need not be changed, unless there is a significant event like transaction commit or remote object access. By using those events to stamp object versions, we can determine the last object modification time relative to its owner node.
- The process of advancing clock at nodes N_1, N_3, and finally at N_2 builds a chronological relationship between significant events at participating nodes, and those that occur *only* when needed and just for the nodes of concern. For example, if any

of T_1's read-set objects has been changed at any arbitrary time t_B, as shown in Figure 5, it does not cause a conflict to T_1. However, if this change occurs after R_y's request, it will be easily detected by T_1 as a conflict. As T_1 advances its time at N_2, any further object changes at N_2, say, at time t_C, will write a version number higher than the recorded communication event time. Similarly, advancing the clock at N_3 upon R_z's request enables T_1 to detect further changes to *Z* at any later time t_D.

- Validating all read-set objects during transaction-forwarding is required to detect the validity of increasing *wv* to the new clock value. To illustrate this, consider any other transaction that starts and finishes successfully at time t_A. This transaction can modify object *X* by increasing its version to 8 instead. If *wv* is simply changed, such a conflict will not be detected.
- Early validation is the most costly step in TFA, especially when it involves remote read-set objects. However, early validation can detect conflicts at an earlier time and save further processing or communication. Further, as we show in the next section, the worst case analysis of early validation reveals that it is proportional to the number of concurrent committed transactions.

V. Algorithm Properties

Correctness. A correctness criterion for TM, called opacity [16], has been proposed as a safety property for TM implementations that suits the special nature of memory transactions. Opacity requires that: 1) committed transactions appear to execute sequentially, in real-time order, and 2) no transaction observes the modifications to shared state done by aborted or live transactions. In addition, all transactions, including aborted and live ones, must always observe a consistent state of the system.

Theorem 1: TFA ensures opacity.

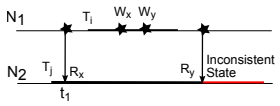
Proof: To prove the theorem, we have to show that TFA satisfies opacity's three conditions. We start with the real-time order condition. We say that a transaction T_j reads from transaction T_i, if T_i writes a value to any arbitrary object O_x, and then commits successfully, and later T_j reads that value. Let us assume that M transactions commit successfully and violate the real-time order by mutually reading from each other in a circular way: $T_1 \prec T_2 \prec T_3 ..... \prec T_M \prec T_1$. For this to happen, T_2 must read from T_1, T_3 must read from T_2, and so on. This means that T_1 must read from T_M, and commit before T_M, which yields a contradiction, as a transaction's local changes are not visible till the commit phase.

The second opacity condition is guaranteed by the write-buffer mechanism of the algorithm: a transaction makes its changes locally through transaction-local copy, and exposes changes only at commit time, after locking all write-set objects.

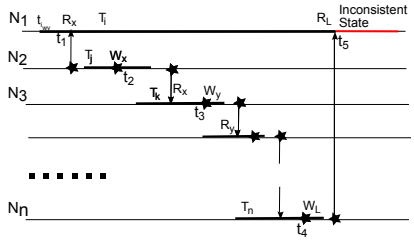
Opacity's last condition ensures consistent state for both live and aborted transactions. By consistent state, we mean that, for a given set of objects *O* modified by some transaction T_k, if T_k was committed successfully, then any other transaction will see either the old values of *all* objects or the new values of *all* objects. If T_k was aborted, then any other transaction will see the old values of *all* objects *O*. As the abort case is already covered by the second opacity condition, we will prove the successful commit case.

Let us define the operator $\Leftarrow_{old}$ (or $\Leftarrow_{new}$) between two transactions to indicate that the first transaction reads old (or new) values of objects changed by the second transaction. We can easily construct such a relation if the event of reading an arbitrary object O_x can be defined relative to the commit event of the other transaction.

Consider the simplest case of two conflicting transactions, shown in Figure 6(a). Here, T_j reads the old value of O_x, before T_i modifies both O_x and O_y, and commits successfully. Thus, $T_j \Leftarrow_{old} T_i$. Later, if T_j retrieves the new value of O_y, then it violates consistency, as $T_j \Leftarrow_{new} T_i$. At this point, the clock value of N_1 is larger than $T_{j_{wv}}$, due to the synchronization point at t_1. This causes an early-validation, and the conflict on O_x will be detected and T_j is aborted before entering the inconsistent state.



(a) Simple inconsistent state.



(b) Inconsistent state with more than two transactions.

Fig. 6. Possible opacity violation scenarios.

Now, we will generalize this for any number of transactions (see Figure 6(b)). Assume that we have n transactions, T_i, T_j, ... , T_n, running on n different nodes N_1, N_2, ... N_n, respectively. At time t_1, T_i accesses O_x located at N_2, and then T_j modifies O_x and commits at time t_2. T_k reads the new value of O_x at time t_3, and then modifies any other object O_y and commits at time t_3. Similarly, the rest of the transactions follow the same access pattern, implicitly constructing the happens-before relationship. At time t_5, T_n reads the new value of O_L. Therefore, we can say that $T_i \Leftarrow_{old} T_j$, $T_n \Leftarrow_{new} T_i$, and $T_{n-1} \Leftarrow_{new} T_n$. Since the last two relations imply that $T_i \Leftarrow_{new} T_j$ indirectly through T_k, ... , T_n, it contradicts the first relation, violating data consistency. This situation is not permitted by TFA, and it is clear that $T_{i_{wv}} \prec t_1$. Since we will have clock synchronization at t_1 between N_1 and N_2, we can say that $t_1 \prec t_2$, and similarly, $t_2 \prec t_3$, etc. The point of interest is t_5, for which the clock value of $N_n > T_{i_{wv}}$. Now, transaction forwarding will occur and early validation will detect the conflict on O_L. Thus, T_i will not proceed to an inconsistent state and will be aborted immediately. The theorem follows. ■

Progress Property. *Strong Progressiveness* was recently proposed [17] as a progress property for TM. A TM system is said to be strongly progressive if 1) a transaction that encounters no conflict is guaranteed to commit, and 2) if a set of transactions conflicts on a single transactional object, then at least one of them is guaranteed to commit.

Note that applying just Lamport clocks in designing DTM is not sufficient to achieve strong progressiveness, and may infact, lead to poor performance. This because, the process of advancing the clock in Lamport is done at every message (i.e., at each interaction between nodes). In contrast, in TFA, it is done only when one of the transactions in the involved nodes commits successfully. Thus, under TFA, many messages are exchanged without affecting the clock on both sides. Moreover, TFA allows a transaction started at an absolute time t_1, to read an object changed at absolute time t_2, where $t_2 > t_1$. Due to transaction forwarding, transactions can detect false positives due to other object changes or other transactions' executions. This has an impact in distributed environments, wherein transactional lifetime is significantly longer than that in non-distributed settings.

Theorem 2: TFA is strongly progressive. (Proof is available in [37])

Cost Measures. As we mentioned before, the most costly operation in TFA is the early validation which occurs whenever transaction forwarding is needed. Although early validation forces validation of the *read-set*, which can be expensive due to the presence of remote objects, we prove that this operation is not frequent. This is essentially because, transaction forwarding will not occur unless

a clock difference has been detected. However, the clock cannot be changed unless some other transactions commit successfully. Thus, the likelihood of safely committing transactions outweigh the number of early validation operations. We now establish an upper bound for these costs.

Theorem 3: For a given set of concurrent transactions Q executing on N nodes, the upper bound on the number of possible early validation steps is $O(committed(Q)^2)$, where $committed(Q)$ is the subset of successfully committed transactions.

Proof: Assume we have a set of Q concurrent transactions. From the definition of early validation, we can't have early validation till one of the transaction commits successfully. At worst case, a transaction $T \in Q$, and assume all other transactions in Q will issue object read/write request to the node of T, so we will at most have $|Q| - 1$ early validation. Repeating that for all other concurrent transactions, then we will have at most $\sum_{i=1..|Q|}(|Q|-i)$ early validation, which can be approximated to $|Q|^2$, given that all transactions trigger early validation to each others, and all of them commit successfully. ∎

Using normal broadcasting (e.g., [29]), $O(Q)$ messages are needed for each committed transaction (to broadcast), which implies a message cost of $O(Q^2)$. Besides, with broadcasting, a transaction needs to broadcast its changes to all nodes, irrespective of whether those changes are relevant to the execution of transactions at remote nodes. In contrast, under TFA, messages are exchanged only between conflicting transactions and on objects of interest.

Note that, Theorem 3 presents the worst-case bound on early validation steps. In Section VI, we show that the number of early validation steps, on average, is significantly less than the worst-case bound.

Lemma 1: For any transaction accessing O_s objects, the number of possible early validation steps is $O(|O_s|)$.

Proof: Early validation by definition occurs whenever some object is accessed for the first time within a transaction. So, the maximum number of early validations is the number of objects accessed by transaction $|O_s|$. ∎

Lemma 2: The worst case number of messages in an early validation step is N.

Proof: During early validation, it is required to validate all objects in transaction read-set. Read-set objects can be distributed over network. Hence, at worst case, these objects can't be distributed on nodes $> N$. As we aggregate the validated objects ids, so we will require at most to N different message for all nodes to validate all read-set objects. ∎

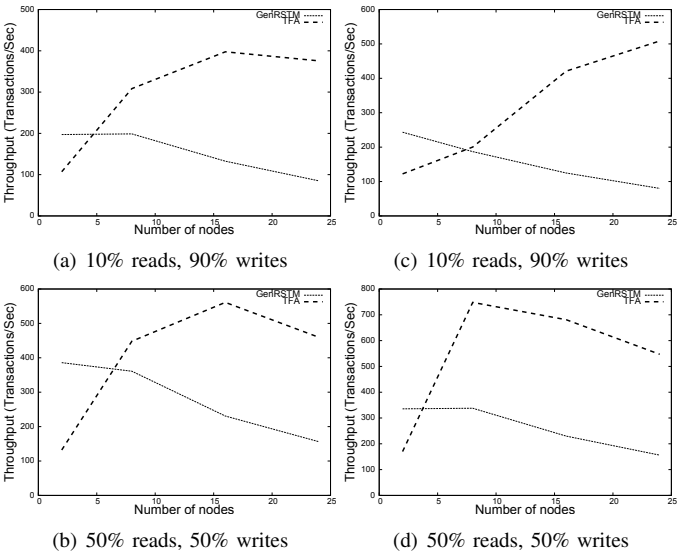
Lemma 3: The upper bound on the number of messages in an early validation step for a single transaction accessing O_s objects is $O(min(committed(Q), |O_s|) * |N|)$.

Proof: From Lemma 2 and Lemma 1, we conclude that early validation can happen due to committed concurrent transaction and based on accessed objects within current transaction, so the minimum of those factors will determine the number of possible early validation events per this transaction. Hence, we can calculate the total number of messages for all early validations during transaction lifetime. ∎

VI. Experimental Evaluation

We implemented TFA in the HyFlow DTM framework [36] for experimentally evaluating its performance. HyFlow is a pluggable framework for DTM. Its instrumentation engine modifies class code at runtime, adds new fields, and modifies annotated methods to support transactional behavior. Further, it generates callback functions that work as "hooks" for Transaction Manager events such as onWrite, beforeWrite, beforeRead, etc.

We developed a set of distributed applications as benchmarks to evaluate TFA against competing models. Our benchmark suite includes a distributed version of the Vacation benchmark from the STAMP benchmark suite [9], two monetary applications (Bank and Loan), and three distributed data structures (Queue, Linked list, and Binary Search Tree) as microbenchmarks. Three versions of the benchmarks were implemented. The first version uses Java RMI, and locks to guard critical sections. We used read-write locks, which permit greater concurrency. A random timeout mechanism was used to handle deadlocks and livelocks. In the microbenchmark implementations, we used a fine-grained, handcrafted lock-coupling implementation [4], which acquires locks in a "hand-over-hand" manner. The second version uses atomic transactions using the TFA implementation, GenRSTM, and Decent STM. The third version is based on a distributed shared memory (DSM) implementation using the Home directory protocol, like Jackal [31], which uses the single-writer/multiple-readers pattern.



(a) 10% reads, 90% writes (c) 10% reads, 90% writes

(b) 50% reads, 50% writes (d) 50% reads, 50% writes

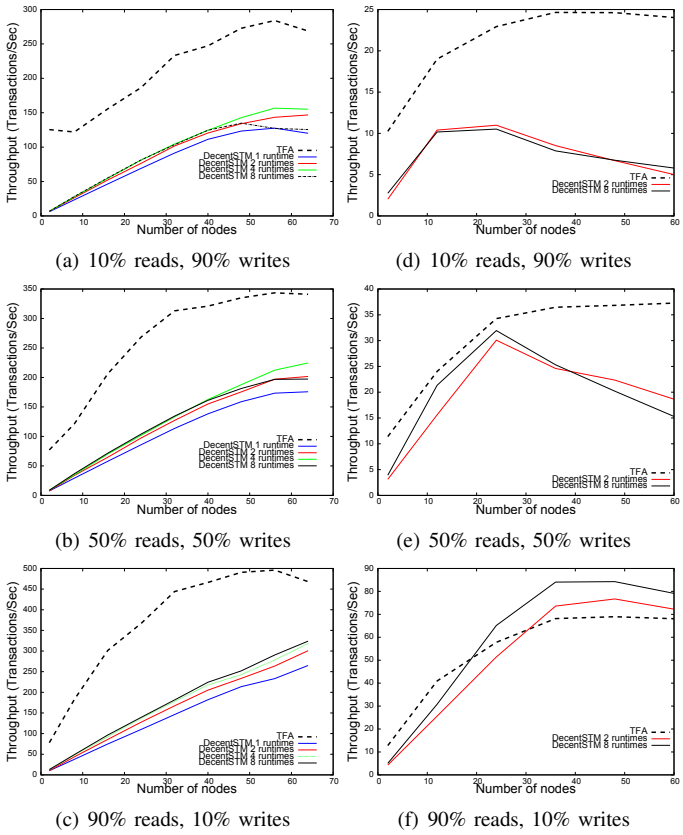
Fig. 7. Throughput of Bank benchmark under TFA and GenRSTM: (a-b) 4 threads per node, (c-d) 8 threads per node.

We conducted the experiments on a network comprising of 120 nodes, each of which is an Intel Xeon 1.9 GHz processor, running Ubuntu Linux, and interconnected by a network with $1ms$ end-to-end link delay. Each node runs eight concurrent threads, with each thread invoking 50-200 sequential transactions (in total, around one thousand concurrent transactions). In a single experiment, we thus executed 200,000 transactions, and measured the throughput for each concurrency model, for each benchmark.

Competitor DTM implementations. We first evaluate the performance of TFA and compare it with two competitor DTM implementations: GenRSTM [29] and DecentSTM [5]. GenRSTM is an example DTM, which relies on broadcasting using group communication. GenRSTM replicates data access nodes, and its replication manager is notified of events reflecting the internal state of the local STMs. On the other hand, DecentSTM implements a sophisticated, fully decentralized snapshot algorithm, which minimizes aborts. Unlike TFA, DecentSTM is a multiversion algorithm. Thus, it keeps a history of object states to allow conflicting transactions to proceed as long as it can see a consistent snapshot of memory.

Figure 7 shows the throughput of TFA and GenRSTM for the Bank benchmark under different number of threads (4 and 8 threads) per node, and read/write transaction percentages. The y-axis shows the number of nodes (or replicas), while x-axis shows the throughput (committed transactions/second). In this experiment, GenRSTM was found to crash after 25 nodes, so we terminate the comparison at this number of nodes.

As Figure 7 shows, GenRSTM outperforms TFA at small number of nodes (2-7), while TFA outperforms it at higher number of nodes.



(a) 10% reads, 90% writes (d) 10% reads, 90% writes

(b) 50% reads, 50% writes (e) 50% reads, 50% writes

(c) 90% reads, 10% writes (f) 90% reads, 10% writes

Fig. 8. Throughput of DecentSTM and TFA: (a-c) Bank benchmark, (d-f) Loan benchmark

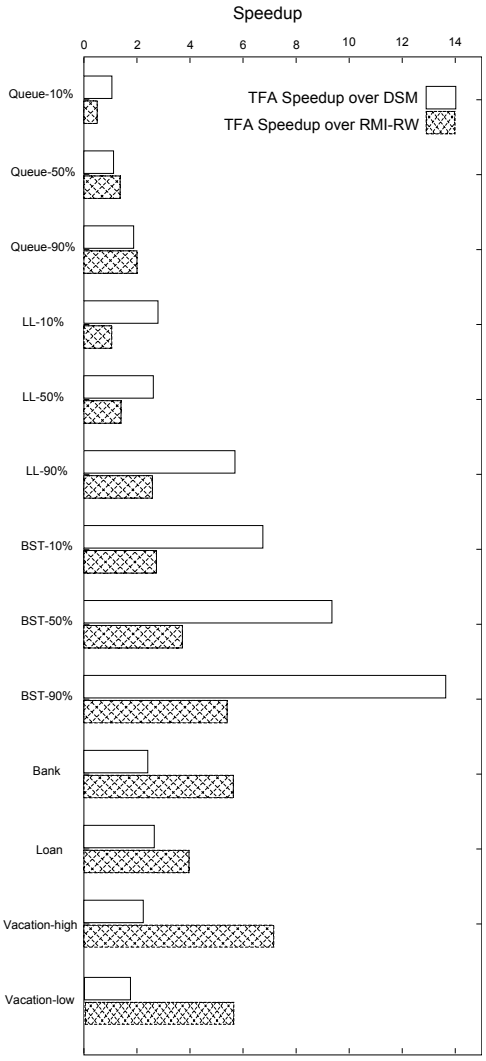


Fig. 9. TFA algorithm speedup for a distributed benchmark suit over 120 nodes.

For write-dominant transactions, TFA performs much better, because of the overhead introduced by GenRSTM for broadcasting changes to all other replicas.

Figure 8 compares the performance of TFA with DecentSTM. In DecentSTM, each application thread works as a distributed entity, so no inter-thread optimization is introduced in its implementation. In contrast, TFA allows multiple threads per node, which reduces the validation and clocking overheads. For the sake of fairness, we limited this experiment to a single thread per node. As mentioned earlier in Section II, DecentSTM introduces the concept of *runtimes* that work as distributed shared memory containers to reduce the network contention. Figure 8 shows the throughput of TFA and DecentSTM using different number of runtimes, and for a range of read/write transaction percentages. We observe from Figure 8 that TFA consistently outperforms DecentSTM. This is precisely due to the higher overhead of DecentSTM's snapshot isolation algorithm (relative to TFA). The snapshot algorithm in DecentSTM incurs relatively larger overhead because it requires searching the history of objects to find a valid snapshot. Besides, in DecentSTM, objects are centrally maintained at runtimes, which incurs high contention and communication cost. It is worth noting that, the variance of the nodal throughput under GenRSTM is around 29 to 836, while that under both TFA and DecentSTM is within the 0.06 to 13.6 range. This means that, TFA and DecentSTM are more fair than GenRSTM, ensuring a uniform execution of transactions over the whole system.

Due to space limitations, results under other benchmarks are skipped here; they can be found in [37].

Classical distributed programming models. Figure 9 shows the relative *throughput speedup* achieved by TFA over other concurrency models on the benchmarks. The confidence intervals of the data-points of the figure are in the 5% range. We observe that TFA outperforms all other models: the speedup ratio ranges between $1\times$ and $13\times$. For the Linked List and Binary Search Tree microbenchmarks, at different read percentages (10%, 50%, and 90%), DSM shows higher throughput than RMI with the lock-coupling implementation. In contrast to RMI with locks, DSM's multiple-reader pattern permits concurrent operations to proceed, while fine-grained locks serialize node traversals. In Queue, the contention is distributed over both ends. We used *read locks* with RMI to permit concurrent *contains* calls, which improves RMI/locks's throughput. TFA outperforms both approaches by $1\times$ to $13.6\times$ speedup for most workloads. TFA yields a higher speedup for read-dominant transactions (e.g., Queue, Linked List, and Binary Search Tree) due to the low number of conflicts/retries, especially on the Binary Search Tree where transactions operate on different branches. For other benchmarks such as Bank, Loan, and Vacation, RMI outperforms DSM by 40-250%, while TFA achieves $1.6\times$ to $7\times$ speedup over both of them.

In [35], we report extensively on performance under different conditions and under variable number of nodes and threads per node. Our experiments show that TFA performs better at high contention situations and with large number of nodes (e.g., when object concurrent access probability is higher than 12%).

VII. Conclusions

We presented TFA, a fully distributed, scalable cc DTM that ensures both opacity and strong progressiveness. It outperforms other distributed concurrency control models and competitor DTM implementations, with acceptable number of messages and low network traffic. Locality of reference enables TFA to scale well with increasing number of calls per object. In addition, TFA permits remote objects to move toward group of nodes that access them frequently, reducing communication costs. Our implementation shows that DTM, in general, provides comparable performance to classical and modern distributed concurrency control models, and exports a simpler programming interface, while avoiding data-races, deadlocks, and livelocks.

The TFA implementation is publicly available at www.hyflow.org.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[2] T. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6 –16, Jan. 1990.

[3] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977. 10.1007/BF00263762.

[5] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *IPDPS '10*, pages 1 –12, april 2010.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[7] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08*, pages 247–258, New York, NY, USA, 2008. ACM.

[8] R. L. H. C. C. M. Bratin Saha, Ali-Reza Adl-Tabatabai and B. Hertzberg. McRT-STM: a high performance software transactional memorysystem for a multi-core runtime. In *PPOPP '06*, pages 187–197, 2006.

[9] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*, September 2008.

[10] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC '09*, nov 2009.

[11] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36:372–421, December 2004.

[12] M. J. Demmer and M. Herlihy. The Arrow distributed directory protocol. In *DISC '98*, pages 119–133, London, UK, 1998. Springer-Verlag.

[13] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *POPL '95*, pages 37–48, New York, NY, USA, 1995. ACM.

[14] M. Factor, A. Schuster, and K. Shagin. A platform-independent distributed runtime for standard multithreaded Java. *Int. J. Parallel Program.*, 34(2):113–142, 2006.

[15] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89*, pages 202–210, New York, NY, USA, 1989. ACM.

[16] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08*, pages 175–184, New York, NY, USA, 2008. ACM.

[17] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. *SIGPLAN Not.*, 44:404–415, January 2009.

[18] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *in Proc. of ISCA*, page 102, 2004.

[19] T. Harris, J. Larus, and R. Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[20] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. volume 41, pages 253–262, New York, NY, USA, October 2006. ACM.

[21] M. Herlihy, J. E. B. Moss, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[22] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In *DISC '05*, pages 324–338. Springer, 2005.

[23] T. Johnson. Characterizing the performance of algorithms for lock-free objects. *Computers, IEEE Transactions on*, 44(10):1194 –1207, Oct. 1995.

[24] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP '08*, pages 51–58, Washington, DC, USA, 2008. IEEE Computer Society.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[26] J. Maassen, T. Kielmann, and H. E. Bal. Efficient replicated method invocation in Java. In *JAVA '00*, pages 88–96, New York, NY, USA, 2000. ACM.

[27] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06*, pages 198–208. ACM Press, Mar 2006.

[28] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA '06*, 2006.

[29] P. R. N. Carvalho and L. Rodrigues. A generic framework for replicated software transactional memories. In *NCA '11*, August 2011.

[30] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14:71–98, July 2003.

[31] R. V. R A F Bhoedjang and H. E. Bal. Distributed shared memory management for java. In *ASCII '00*, page 256, 2000.

[32] M. Raynal. About logical clocks for distributed systems. *SIGOPS Oper. Syst. Rev.*, 26:41–48, January 1992.

[33] P. Romano, N. Carvalho, M. Couceiro, L. Rodrigues, and J. Cachopo. Towards the integration of distributed transactional memories in application servers clusters. In *Quality of Service in Heterogeneous Networks*, volume 22 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 755–769. Springer Berlin Heidelberg, 2009. (Invited paper).

[34] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo. Cloud-TM: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44:1–6, April 2010.

[35] M. M. Saad. HyFlow: A High Performance Distributed Software Transactional Memory Framework. Master's thesis, Virginia Tech, ECE Dept., Blacksburg, VA, USA, 2011. Available at http://scholar.lib.vt.edu/theses/available/etd-05182011-095228/.

[36] M. M. Saad and B. Ravindran. Supporting STM in Distributed Systems: Mechanisms and a Java Framework. In *TRANSACT '11*, San Jose, California, USA, 2011. ACM.

[37] M. M. Saad and B. Ravindran. Transactional Forwarding Algorithm : Technical Report. Technical report, ECE Dept., Virginia Tech, January 2011.

[38] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*, pages 204–213, New York, NY, USA, 1995. ACM.