On Transactional Memory Concurrency Control in Distributed Real-Time Programs

Sachin Hirve*, Aaron Lindsay†, Binoy Ravindran* and Roberto Palmieri*
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, Virginia 24060
Email: *{hsachin, binoy, robertop}@vt.edu, †aaron@aclindsay.com

***Abstract*—We consider distributed transactional memory (DTM) for concurrency control in distributed real-time programs, and present an algorithm called RT-TFA. RT-TFA transparently handles object relocation and versioning using an asynchronous clock-based validation technique, and resolves transactional contention using task time constraints. We implement the RT-TFA on top of JChronOS, a layer extending the scheduling capabilities of ChronOS for Java programs. We conduct an extensive evaluation study comparing RT-TFA with well known competitors for real-time distributed applications. Our results reveal that RT-TFA outperforms competitors in mostly scenarios up to 43% with added advantage of better programmability and composability.**

I. Introduction

Distributed embedded software is inherently concurrent, as they monitor and control concurrent physical processes. Often, their computations need to concurrently access (i.e., read, write) shared data objects, which must be properly coordinated so that consistency properties (e.g., linearizability [1], serializability [2]) can be ensured. Furthermore, they must satisfy application time constraints. The usual way for managing concurrency of different processes in a system is using locks, which inherently suffers from programmability, scalability, and composability challenges [3]. Additionally, the implementation of complex algorithms based on mutual exclusion becomes hard to debug, resulting in higher developing time.

Software transactional memory (STM) [4] is an novel synchronization abstraction that promises to alleviate these difficulties. In fact, leveraging the proven concept of atomic and isolated transactions, STM spares programmers from the pitfalls of conventional lock-based synchronization, significantly simplifying the development of parallel and concurrent applications. With STM, in-memory operations are organized as *memory transactions* that read/write shared objects. These transactions optimistically execute, logging object changes in a private space. 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 committing the other, yielding (the illusion of) atomicity. Aborted transactions are rolled-back and re-started.

The challenges of lock-based concurrency control are exacerbated in distributed systems, due to the additional complexity of multi-computer concurrency (e.g., distributed deadlocks, priority inversion, and composability challenges). Distributed TM (or DTM) [5]–[7] has been similarly motivated as an alternative to distributed locks. DTM can be supported in any distributed execution model, with concomitant tradeoffs, including a) control flow [8], where objects are immobile and transactions invoke object operations through RMIs/RPCs and b) dataflow [9], where transactions are immobile, and objects are migrated to invoking transactions.

Given DTM's programmability, scalability, and composability advantages, it is also a compelling abstraction for distributed embedded real-time programs. In fact, developing distributed concurrent applications with time constraints is several order of magnitude harder than conventional, non real-time, distributed applications. DTM is the potential solution, however, in order to cope with time constraints, it requires bounding (distributed) transactional retries, as (distributed) real-time tasks, which subsume transactions must satisfy deadlines. Bounding transactional retries requires resolving transactional contention using time constraints.

Motivated by these needs, in this paper we present RT-TFA, *Real-Time Transaction Forwarding Algorithm*. RT-TFA extends the distributed concurrency control scheme called TFA [10], used in recent works such as [11]–[13], with real-time contention management. RT-TFA has been implemented on top of *ChronOS* [14]. It enables a thread to execute part of it under real-time constraints in comparison to other existing real-time OS (e.g. Integrity, μItron etc.) where whole thread is executed under real-time constraint. In order to allow JAVA applications to define time constraints and inject into ChronOS, we designed and implemented *JChronOS*, a user-space library which provides the hooks to interface with the ChronOS kernel from Java application.

We assess the performance of RT-TFA by an extensive evaluation study, comparing our approach with the widely used protocols for implementing real-time distributed applications. Our results show that RT-TFA outperforms competitors in mostly scenarios up to 43%. To the best of our knowledge, RT-TFA is the first algorithm that enables DTM concurrency control in real-time programs.

II. Preliminaries and Assumptions

We consider a distributed system of N nodes, where each node has one or more processing cores. We consider a programming model similar to Real-Time CORBA [15]: a distributed task (or simply called task, hereafter) is programmed as a thread that may read/write local as well as remote objects. Unlike CORBA, which is based on control flow execution, we consider Herlihy and Sun's dataflow execution model [16], where threads are immobile and objects are (transparently) migrated to invoking threads using a directory-based lookup protocol (e.g., [16]). A time constraint on a task is expressed using a *scheduling segment* (like in CORBA), which is a task code segment subject to the time constraint.

We assume $n \neq N$ sporadic tasks τ_1, τ_2, $\cdots$, τ_n, with a minimum period of $t(\tau_i)$. A task τ_i's m^{th} instance is denoted τ_i^m, and has a relative deadline $D(\tau_i^m) = t(\tau_i)$. Since we focus on real-time (distributed) tasks, for the rest of the paper, we restrict our attention to code inside task scheduling segments. Each task's scheduling segment may contain non-atomic as well as atomic code sections. The worst case execution time (WCET) c_i of a task τ_i is the sum of its worst case execution time of non-atomic sections (C_i) and atomic sections (s_i).

Each object θ, can be accessed by multiple atomic sections (and therefore multiple tasks). An atomic section may involve reading/writing (one or more) local/remote shared objects. An atomic section's length is the sum of the execution costs of: 1) acquisition of objects from remote node, 2) local modifications on object, 3) acquisition of locks from remote node, and 4) communication cost for remote requests ρ. If one or more atomic sections conflict, it is resolved by committing one section and aborting/retrying the others, which increases the time to commit the aborted ones. The retry cost $RC(\tau_i)$ is the total time that a task τ_i executes its aborted atomic sections. Response time of τ_i i.e. $R(\tau_i)$, is sum of τ_i's WCET (c_i), retry cost $RC(\tau_i)$, and the interference caused by other tasks to τ_i.

We assume a network with bounded communication delay (e.g., [17]), denoted ρ_{max}. Node clocks are synchronized (e.g., [18]), and clock drift is bounded by δ_{max}. Each task is assumed to have a programmer-supplied exception handler. We consider a termination model [19] for task failures including those due to time-constraint violations. If a task has missed its deadline, the handler is immediately executed on all nodes where the task may be accessing objects. The handler is assumed to perform the necessary recovery actions.

III. Overview of TFA

For completeness, we overview TFA. TFA [20] builds on the TL2 algorithm proposed for multiprocessor TM [21]. It is a data flow based, distributed transaction management algorithm, which provides atomicity, consistency, and isolation properties. Under TFA, operations on distributed objects are buffered and locks on objects are acquired at commit time. On successful acquisition of locks, objects are updated. Otherwise, the transaction is aborted by releasing all previously acquired locks and retried.

In contrast to TL2's central clock, TFA uses independent, per-node transactional clocks and provides a mechanism to establish the "happens before" relationship between significant events (e.g., write-after-write, read-after-write). Upon a transaction's successful commit, a node increments its local clock. An object's version is defined by the local clock at the time of the object's last modification. When a local object is accessed by a transaction, as part of validation, the object version is compared with the transaction's starting time. If the object's version is newer, the transaction is aborted and retried.

For validating remote objects, TFA employs a technique called "transaction forwarding:" when a transaction requests access to a remote object, the local clock is piggybacked with the request to the remote node. The remote node advances its clock to the sender's clock if its clock is older; otherwise, no update is made to the remote clock. The remote node then sends the object copy with its clock value. Upon receipt, the local node (i.e., the sender) compares the remote clock value with the transaction starting time. If the remote clock is newer, the transaction's read-set is validated by checking whether any other object in the read-set has been updated to a version newer than the transaction starting time. If the read-set validation succeeds, then the transaction starting time is advanced to the remote clock value (i.e., "forwarded"). Otherwise, the transaction is aborted and re-issued.

When a transaction reaches the commit stage, it first acquires locks on all the objects in its write-set. On successful lock acquisition, the objects are updated and the transaction committing node is published as the new host of the updated objects. If lock acquisition fails for any object, all acquired locks are released, and the transaction is aborted and re-issued.

Illustrative Example. Consider three nodes, N_1, N_2, and N_3, each running transactions T_1, T_2, and T_3, respectively (see Figure 1). N_2 hosts object θ and is considered θ's "owner." Nodes maintain a local transactional clock, which is incremented when transactions running on them commit. T_3 starts at local clock $lc_3 = 12$ and requests θ from N_2 at $lc_3 = 16$. N_2 compares the received clock value with its local clock $lc_2 = 12$ and advances its clock to $lc_2 = 16$. After some time, T_1 starts at local clock $lc_1 = 14$ and requests θ from N_2 at $lc_1 = 19$. At N_2, no change is made to its local clock lc_2, since $rc = 19 < lc_2 = 21$. When N_1 receives the response from N_2, it observes that $lc_1 = 19 < rc = 21$. Therefore, it forwards T_1 to start at $lc_1 = 21$ and validates all other objects accessed earlier by T_1. Later, T_3 acquires the lock on θ at N_2 and updates θ at local clock $lc_3 = 25$. Now, N_3 holds the ownership of θ, but leaves θ locked at N_2. When T_1 tries to acquire the lock on θ, it may find N_2 or N_3 as the object owner depending on when N_3 successfully publishes its ownership of θ. If T_1 requests the lock on θ from N_2, it fails due to the existing lock on θ and aborts. Otherwise, if T_1 finds N_3 as the owner, it acquires the lock, but fails during read-set validation. Therefore, T_1 releases the lock acquired on θ at N_3 and aborts. T_1 retries and requests θ again from the new owner N_3. Concurrently, another transaction T_2 also receives θ, but T_1 acquires the lock on θ earlier than T_2 and commits. As a result, T_2 aborts and retries. T_2 finally commits at $lc_2 = 44$ and becomes the new object owner.

IV. RT-TFA

Overview. Since TFA is agnostic to task time constraints, it can cause priority inversions, causing higher priority tasks to miss their time constraints. For example, in Figure 1, if the deadlines d_1, d_2, and d_3, of the parent tasks of transactions T_1, T_2, and T_3, respectively, are such that $d_1 > d_2 > d_3$, then it is possible that transactions will commit in a different order than their respective deadlines. From Figure 1, even though T_2's urgency is greater than T_1's, T_1 commits earlier than T_2. Therefore, T_2 may not be able to meet its deadline, though there might be enough slack time for T_1 to commit later.

RT-TFA extends TFA to support transactions that execute under time constraints. Under RT-TFA, task scheduling segments are scheduled at each node using EDF. Transactions inherit the deadlines of their parent tasks: when atomic sections attempt to access and modify local or remote shared objects, they may be subject to object access conflicts. Those conflicts are resolved using the deadlines of the sections' parent tasks.

RT-TFA inherits all properties of TFA: objects are acquired at encounter time, transaction clock values are exchanged, and

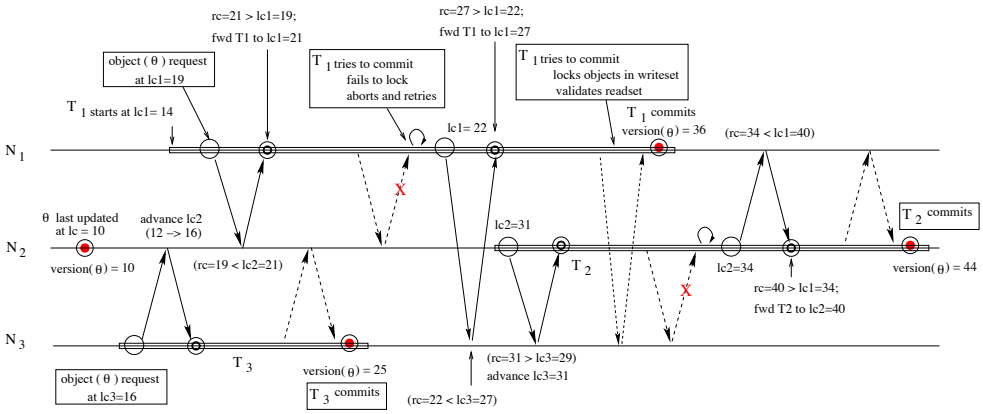


Fig. 1. Example transaction execution under TFA.

transaction forwarding is conducted. However, in addition to the local clock, the object requester also sends the deadline of the transaction (when the object is remote). The object holder maintains a list of all the requesters for the shared object in a deadline-ordered queue, and grants the object to the earliest deadline transaction. If an earlier deadline transaction T_a requests an object θ which was previously granted to a later deadline transaction T_b, then T_b is aborted and θ is granted to T_a. T_b joins the deadline-ordered wait queue for θ. When T_a commits, it signals T_b to retry.

After acquiring the object, a transaction moves forward to acquire the remaining objects or to make changes on locally buffered objects. Shared objects are updated after the transaction acquires locks on the objects. A committing transaction signals the next transaction in the (deadline-ordered) wait queue to resume, which then retries the object acquisition from the new object owner. In this way, transactions commit in their deadline order.

Illustrative Example. Nodes N_1, N_2, and N_3, which are clock-synchronized, run tasks τ_1, τ_2, and τ_3, respectively. The absolute deadlines of these tasks are in time order of $d_1 > d_2 > d_3$. N_3 hosts an object θ, and is its owner.

τ_1 starts executing on N_1 and contains a transaction T_1. T_1 starts a read or write operation on θ, as shown in Figure 2. T_1 sends the request to access θ to N_3, and piggybacks its transaction clock and absolute deadline (lc_1, d_1) with the request. When N_3 receives this request, it replies back with its remote clock (rc_3) and adds this entry to its object ID/deadline list map $< ID(\theta), list(d(\tau_i)) >$. Little later τ_2 at N_2 initiates transaction T_2 on θ and sends request for θ to N_3. On receiving this request, N_3 finds a later deadline transaction T_1 accessing θ. N_3 replies to N_2 with rc_3 and sends a request to N_1 to abort T_1. On receiving this request at N_1, T_1 is aborted and τ_1 is suspended. Later when T_2 commits on θ, it sends a request to resume τ_1, thereby T_1 retries and requests access to θ from the new host N_2. Meanwhile τ_3 at N_3 starts T_3 on θ and sends request for θ to N_2. N_2 replies with its clock (rc_2) and adds this entry to its object ID/deadline list map. Later when N_2 receives a request from T_1, it declines the request since T_1 has a later deadline than T_3. On receiving the response, T_1 is aborted and τ_1 is again suspended. When T_3 commits, τ_1 resumes. Finally T_1 receives θ from N_2 and commits.

Transactions are committed in deadline order $T_1 \succ T_2$ and $T_1 \succ T_3$, only if there is contention for a shared object. On other hand, a later deadline task may start early depending on its arrival pattern and commit on an object, before another earlier deadline task may attempt to access this object (e.g., T_2 commits earlier than T_3).

V. Architecture

The nodal architecture of RT-TFA's implementation consists of a stack of the ChronOS Linux kernel, JChronOS, the JVM, RT-TFA and the application. In the following we describe the core modules of this architecture.

ChronOS. ChronOS [14] is derived from the 2.6.33.9 version of the Linux kernel. It uses the $CONFIG\_PREEMPT\_RT$ real-time patch [22] which enables complete preemption in the Linux kernel and improves interrupt latencies. ChronOS provides a set of APIs and a scheduler plugin infrastructure that can be used to implement and evaluate a variety of single- and multi-processor scheduling algorithms. The ChronOS real-time scheduler is implemented as an extension to the Linux O(1) scheduler. ChronOS includes implementations of various single-processor (e.g., EDF, DASA, RMA etc.) and multiprocessor scheduling algorithms (e.g., G-EDF, NG-GUA, G-GUA etc.). Single-processor schedulers in ChronOS support priority inheritance to address priority inversion.

In ChronOS, a time constraint on a thread is expressed using the notion of a *scheduling segment* (introduced in Section II). A real-time application in ChronOS specifies the start and end of a scheduling segment. Scheduling segments of a task are identified using special APIs, and are scheduling events. A scheduling event is defined as a trigger that forces the system into a scheduling cycle, resulting in a call to the scheduler where new task(s) are selected based on the scheduling algorithm. In ChronOS, we define four scheduling events (e.g., beginning of a scheduling segment, end of a scheduling segment, resource request, and resource release).

Under different load conditions, real-time segments may violate their time constraints. Until they finish their execution, such segments may block the execution of other real-time segments. ChronOS supports a thread abort mechanism by which the application is notified about the threads with expired time constraints. On receiving this information, the application can run the abort handler for the threads, thereby terminating expired real-time segments. RT-TFA uses this mechanism to abort transactions when they violate their thread deadline.

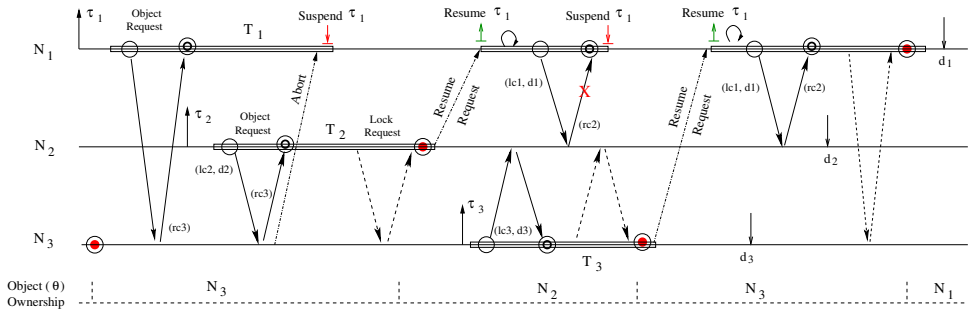


Fig. 2. Example distributed real-time transaction execution under RT-TFA.

JChronOS. In this section we present *JChronOS*, a library that extends the scheduling capabilities of ChronOS for Java programs. This is accomplished by accessing the existing ChronOS library, `libchronos`, through JNI. `libchronos` is written in C and interacts with the ChronOS scheduler through system calls to initialize the real-time scheduler, begin or end a real time segment, acquire a ChronOS mutex, etc.

JChronOS consists of two components. The first component, written in Java, provides the necessary interfaces to applications in Java. It enables Java applications to define scheduling segments and to acquire resources by simply importing the classes provided by its interface. These classes are distributed in a jar file, `libjchronos.jar`. When invoked, these methods call the other component of JChronOS, the native library, through JNI. The second component, maintained in Linux by the shared object `libjchronos.so`, provides a mechanism for converting data structures between their Java forms, and those required by `libchronos`, the existing ChronOS library, and its system calls. Once it has converted any arguments appropriately, `libjchronos.so` calls `libchronos.so`, marshals the data into Java objects, and returns from the JNI call.

VI. Implementation and Evaluation

Implementation. We implemented RT-TFA in the HyFlow Java DTM framework [23], which provides pluggable support for policies for directory lookup, transactional synchronization and recovery, and contention management. We integrated this implementation with the ChronOS real-time Linux kernel. We modified HyFlow to transparently support distributed real-time transactions: atomic sections defined within scheduling segments transparently use thread time constraints for local and remote contention management.

Experimental setting. We used a 14-nodes testbed, where each node is an AMD Opteron 1.9GHz processor. The average network delay is less than $1ms$. Each node ran a set of periodic tasks. Each task is composed of a fixed duration for processing non-atomic section and atomic sections. In the experiments, 5 tasks were fired at each node, resulting in 70 concurrent tasks over 50 Bank objects. Tasks were defined such that, they contained atomic sections with local and remote read/write operations, resulting in time-critical, distributed transactions. Table I shows the task details.

We used a well known benchmark for STM and DTM called Bank, a monetary application. This maintains a set of accounts distributed over bank branches, and contains two transactions: A) a read operation i.e. `transfer`, which transfers a given amount between two accounts, and B) a write operation i.e. `total balance` , which computes the total balance for given accounts. The object size and the number of operations per object during a transaction is configurable.

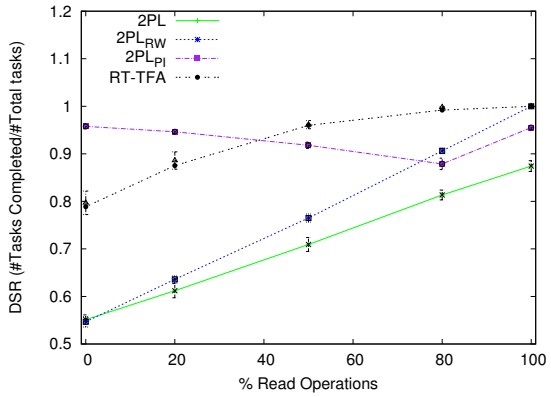
TABLE I. Task Properties

Task	T_1	T_2	T_3	T_4	T_5
Period (ms)	500	1000	1500	3000	5000
Number of transfer	1	1	1	1	1
Bank objects accessed	2	2	2	2	2

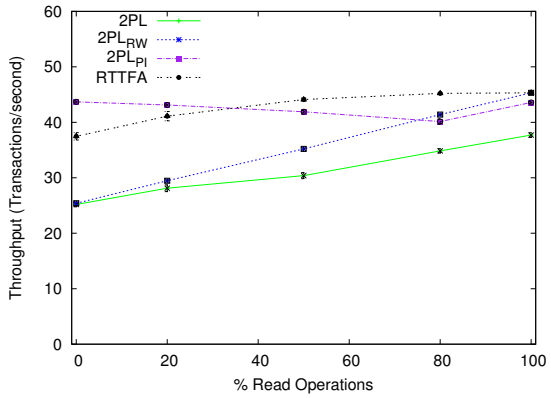
We considered three competitors for RT-FTA, all of them using Java RMI and a version of the classic two-phase-locking protocol (2PL) [24]. Our first competitor ($2PL$) uses mutual exclusion locks to guard critical sections. The second implementation ($2PL_{RW}$) uses readers-writer locks. The last version ($2PL_{PI}$) uses mutual exclusion locks with distributed priority inheritance i.e., the (dynamic) priorities of transactions are propagated [25] with transactions to resolve contention on remote shared objects. The main competitor in our comparison is $2PL_{PI}$, which is widely used in distributed real-time systems for lock-based synchronization. Though it does not provide guarantee against deadlock, it reduces priority inversions by inheriting the priority of highest blocked task. Other competitors further highlight the blocking duration experienced by distributed tasks. We do not consider Distributed Priority Ceiling [26], because it requires prior information about the object access pattern, while in our application we do not impose such limitation.

Results. Figures 3(a) and 3(b) show the deadline satisfaction ratio (DSR) and the throughput, respectively, of RT-TFA and the three $2PL$ versions for different percentages of read operations. All other parameters are fixed: 1 transaction per task, 14 nodes, and 50 Bank objects. DSR is the fraction of the total number of tasks that met their deadlines and throughput is the total number of successfully committed transactions/second.

RT-TFA yields comparable DSR to $2PL$ with priority inheritance ($2PL_{PI}$) and higher DSR than other versions of $2PL$. RT-TFA underachieves against $2PL_{PI}$ for lower percentages (e.g., $<\sim$35%) of read operations due to the higher retry cost for aborted transactions. As percentage of read operations increase, RT-TFA outperforms all three versions of $2PL$, which is directly due to optimistic concurrency control: RT-TFA does lazy locking and acquires locks at the commit stage. In contrast, $2PL$ serializes objects by acquiring the locks before entering the critical section and therefore suffers from higher cost of priority inversion.



(a) DSR



(b) Throughput

Fig. 3. DSR and throughput (0-100% read operations).

Next, we evaluated RT-TFA for different contention levels resulting from varying the number of objects with a fixed number of nodes and transactions. DSR and Throughput plots are omitted due to space constraints, but we describe our observations. RT-TFA's DSR improves gradually; this is due to reduced contention from increased number of objects due to which RT-TFA incurs less cost of transaction aborts and retries. RT-TFA outperforms all competitors except $2PL_{PI}$ for lower percentage (less than 35%) of read operations, where it suffers from higher number of aborted and retried transactions.

VII. Conclusions

We presented the RT-TFA algorithm for supporting DTM in distributed real-time programs. The algorithm inherits TFA's core mechanisms for transactional synchronization, and extends it with real-time contention management. We built RT-TFA on a distributed, real-time OS-based infrastructure, called ChronOS and provided JChronOS, a layer enabling the direct interactions between RT-TFA and ChronOS. We implemented RT-TFA in the HyFlow DTM and compared with different protocols usually adopted for developing distributed real-time applications. Our results revealed that RT-TFA yields comparable deadline satisfactions to $2PL_{PI}$ and outperforms other 2PL-based locking protocols (by as much as 43%). Finally, by the results we can state that DTM can be supported in distributed real-time programs, allowing programmers to reap DTM's better programmability and composability properties.

Acknowledgment

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385

References

[1] J. Aspnes, M. Herlihy, and N. Shavit, "Counting networks," *J. ACM*, vol. 41, no. 5, pp. 1020–1048, 1994.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley '86.

[3] M. Herlihy, V. Luchangco, and M. Moir, "A flexible framework for implementing software transactional memory," in *OOPSLA '06*.

[4] J. Larus and R. Rajwar, *Transactional Memory*. Morgan & Claypool '06.

[5] S. Peluso, P. Romano, and F. Quaglia, "Score: A scalable one-copy serializable partial replication protocol," in *Middleware*, 2012.

[6] R. Palmieri, F. Quaglia, and P. Romano, "Osare: Opportunistic speculation in actively replicated transactional systems," in *SRDS*, 2011.

[7] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D^{2}STM: Dependable Distributed Software Transactional Memory," in *Proc. of PRDC*. IEEE Computer Society Press, 2009.

[8] K. Arnold, R. Scheifler *et al.*, *Jini Specification*. Addison-Wesley '99.

[9] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java application partitioning," in *ECOOP*, 2002.

[10] M. M. Saad and B. Ravindran, "Transactional forwarding: Supporting highly-concurrent stm in asynchronous distributed systems," in *SBAC-PAD*, 2012, pp. 219–226.

[11] A. Turcu and B. Ravindran, "On Open Nesting in Distributed Transactional Memory," in *SYSTOR*, 2012.

[12] J. Kim, R. Palmieri, and B. Ravindran, "Scheduling open-nested transactions in distributed transactional memory," in *COORDINATION*. Springer, 2013.

[13] A. Turcu and B. Ravindran, "On closed nesting in distributed transactional memory," in *TRANSACT*, 2012.

[14] M. Dellinger, P. Garyali, and B. Ravindran, "Chronos Linux: a best-effort real-time multiprocessor Linux kernel." in *DAC*. ACM, 2011.

[15] V. Fay-Wolfe, L. C. DiPippo, G. Copper, R. Johnston, P. Kortmann, and B. Thuraisingham, "Real-time CORBA," *TPDS*, Oct. 2000.

[16] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," *Distributed Computing*, 2007.

[17] Robert Bosch GmbH, "CAN Specification, 2.0 ed." 1991.

[18] D. Mills, "Network time protocol (version 3) specification, implementation," United States, 1992.

[19] E. Curley, J. Anderson, B. Ravindran, and E. D. Jensen, "Recovering from distributable thread failures with assured timeliness in real-time distributed systems," *SRDS*, vol. 0, pp. 267–276, 2006.

[20] M. Saad and B. Ravindran, "Transactional Forwarding Algorithm," Virginia Tech, Tech. Rep., 2010, available at http://hyflow.org/hyflow/chrome/site/pub/tfa_tech.pdf.

[21] O. S. D. Dice and N. Shavit, "Transactional Locking II," in *DISC*, 2006.

[22] Real-time Linux Wiki, "RT-PREEMPT kernel patch for Linux," Available at https://rt.wiki.kernel.org.

[23] M. Saad and B. Ravindran, "Supporting STM in distributed systems: Mechanisms and a Java framework," in *TRANSACT*, 2011.

[24] P. Felber and M. K. Reiter, "Advanced concurrency control in Java," Concurrency and Computation: Practice and Experience, Tech. Rep., 2002.

[25] I. Pyarali, D. C. Schmidt, and R. Cytron, "Techniques for enhancing real-time corba quality of service." *Proceedings of the IEEE*, 2003.

[26] R. Rajkumar, L. Sha and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *RTSS*, 1988, pp. 259–269.