HyFlow: A High Performance Distributed Software Transactional Memory Framework

Mohamed M. Saad

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Mohamed A. Ismail
Paul E. Plassmann
Robert P. Broadwater

April 20, 2011
Blacksburg, Virginia

Keywords: Distributed Systems, Software Transactional Memory, Dataflow, Control-Flow, Contention Management, Cache Coherence, Directory Protocols

HyFlow: A High Performance Distributed Software Transactional Memory Framework

Mohamed M. Saad

(ABSTRACT)

We present *HyFlow* — a distributed software transactional memory (D-STM) framework for distributed concurrency control. Lock-based concurrency control suffers from drawbacks including deadlocks, livelocks, and scalability and composability challenges. These problems are exacerbated in distributed systems due to their distributed versions which are more complex to cope with (e.g., distributed deadlocks). STM and D-STM are promising alternatives to lock-based and distributed lock-based concurrency control for centralized and distributed systems, respectively, that overcome these difficulties. HyFlow is a Java framework for D-STM, with pluggable support for directory lookup protocols, transactional synchronization and recovery mechanisms, contention management policies, cache coherence protocols, and network communication protocols. HyFlow exports a simple distributed programming model that excludes locks: using (Java 5) annotations, atomic sections are defined as transactions, in which reads and writes to shared, local and remote objects appear to take effect instantaneously. No changes are needed to the underlying virtual machine or compiler. We describe HyFlow's architecture and implementation, and report on experimental studies comparing HyFlow against competing models including Java remote method invocation (RMI) with mutual exclusion and read/write locks, distributed shared memory (DSM), and directory-based D-STM.

This work was supported in part by the US National Science Foundation CCF (Software and Hardware Foundations) under Grant 0915895.

Dedication

To my wife, Shaimaa.

Without your patience and support this would not come true...

Acknowledgments

I am heartily thankful to my advisor, Prof. Binoy Ravindran, for his continues encouragement, help and guidance on both technical and personal topics. It has been an honor to work under him. I would also like to thank Prof. Mohamed M. Ismail, Prof. Paul Plassmann, and Prof. Robert Broadwater for serving on my committee and providing their valuable feedback and direction. And special thanks for Prof. Sedki Riad and VT-MENA program administration for their support.

Lastly, I offer my regards and blessings to my family and friends who supported me in any respect during the completion of this work.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Lock-based synchronization is inherently error-prone. Coarse-grained locking, in which a large data structure is protected using a single lock is simple and easy to use, but permits little concurrency. In contrast, with fine-grained locking [74, 60], in which each component of a data structure (e.g., a bucket of a hash table) is protected by a lock, programmers must acquire necessary and sufficient locks to obtain maximum concurrency without compromising safety. Both these situations are highly prone to programmer errors. In addition, lock-based code is non-composable. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is difficult: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the two tables' methods; if the methods were to export their locks, that will compromise safety. Furthermore, lock-inherent problems such as deadlocks, livelocks, lock convoying, and priority inversion haven't gone away. For these reasons, lock-based concurrent code is difficult to reason about, program, and maintain [52].

The difficulties of lock-based synchronization are exacerbated in distributed systems with nodes, possibly multicore, interconnected using message passing links, due to additional, distributed versions of their centralized problem counterparts. For example, RPC calls, while holding locks, can become remotely blocked on other calls for locks, causing distributed deadlocks. Distributed versions of livelocks, lock convoying, priority inversion, and scalability and composability challenges similarly occur. Besides, distributed race conditions are complex to reason about. In addition, locks provide naive serialization of concurrent code with partial dependency, which is a severe problem for long critical sections (in both centralized and distributed systems).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) [47] is a promising alternative to lock-based concurrency control. It was proposed as an alternative model for accessing shared memory data objects, without exposing locks in the programming interface, to avoid the drawbacks of locks. With

A = 10, B = 20;

THREAD A

```
1   atomic{
2         B = B + 1;
3       A = B * 2;
4     }
5     ....
```

THREAD B

```
1   atomic{
2     B = A;
3     }
4     ....
5     ....
```

Figure 1.1: An example of transactional code using atomic language/library constructs

TM, programmers write concurrent code using threads, but organize code that read/write shared memory objects as atomic sections. Atomic sections are defined as *transactions* in which reads and writes to shared objects appear to take effect instantaneously. A transaction maintains its read set and write set, and at commit time, checks for conflicts on shared objects. Two transactions conflict if they access the same object and one access is a write. When that happens, a contention manager [95] resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, often immediately. 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 [30, 6] and is composable [48].

TM originated as a hardware solution (HTM). Examples include TCC [45], UTM [5], OneTM [20], and LogSPoTM [43]. They often extend multiprocessor cache coherence protocols, or modify underlying hardware to support transactions. Later, it was extended in software, called STM. Example STM implementations include DSTM2 [53], RSTM [73], and Deuce [63]. They often use basic atomic hardware primitives (e.g., compare-and-swap) to provide atomicity, and thread-local memory locations are used to provide consistent view of memory. TM has also been provided in hardware/software combination, called Hybrid TM (HyTM). Example HybridTM implementations include LogTM [77] and HyTM [65].

Figure 1.1 shows an example transactional code. Atomic sections are executed as transactions. Thus, the possible values of A and B are either 42 and 42, or 22 and 11, respectively. An inconsistent view of a member (e.g., A=20 and B=10), due to atomicity violation or interleaved execution, causes one of the transactions to abort, rollback, and then re-execute.

Similar to multiprocessor TM, distributed software transactional memory (or D-STM) is an alternative to lock-based distributed concurrency control. Though TM has been extensively studied for multiprocessors [47], relatively little effort has focused on supporting it in distributed systems with nodes interconnected using message-passing links. 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. In addition, distributed systems are

inherently asynchronous, and usually do not have a global clock (providing a global clock often incurs significant message overhead [82]). Communication overhead, balancing network traffic, and network failures are additional concerns for D-STM.

Distributed STM 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 execution model [10, 69, 99], 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 [100, 80], transactions are immobile, and objects move through the network to requesting transactions, while guaranteeing object consistency using cache coherence protocols. The dataflow model has been primarily used in past D-STM efforts—e.g., [56, 105, 106, 88]. A hybrid model [21], where transactions or objects are migrated, heuristically, based on properties such as access profiles, object size, or locality, is also possible. The different models have their concomitant tradeoffs.

D-STM implementations also exist. Examples include Cluster-STM [21], D^2STM [27], DiSTM [64], and Cloud-TM [88]. These implementations are mostly specific to a particular programming model (e.g., the partitioned global address space (PGAS) [2]), and often needs compiler or virtual machine modifications (e.g., JVSTM [23]), or assume specific architectures (e.g., commodity clusters).

The least common denominators for supporting D-STM in any distributed execution model include mechanisms/protocols for directory lookup [57, 29, 56, 105, 106], transactional synchronization and recovery [21, 71, 64, 27, 87], contention management [95, 94], cache coherence, and network communication.

This thesis presents HyFlow [89, 90, 92] — a Java D-STM framework that provides pluggable support for these mechanisms/protocols as modules. HyFlow exports a simple distributed programming model that excludes locks: atomic sections are defined as transactions using (Java 5) annotations. Inside an atomic section, reads and writes to shared, local and remote objects appear to take effect instantaneously. No changes are needed to the underlying virtual machine or compiler.

HyFlow supports both dataflow and control flow models, and ensures distributed transactional properties including atomicity, consistency, and isolation. HyFlow's architecture is module-based, with well-defined APIs for further plugins. Default implementations exist for all needed modules. The framework currently includes two algorithms to support distributed memory transactions: the transactional forwarding algorithm (TFA) [93], and a distributed variant of the UndoLog algorithm [85]. A wide range of transaction contention management policies (e.g., Karma, Aggressive, Polite, Kindergarten, Eruption [95, 94]) are included in HyFlow. Four directory protocols [57, 29, 105] are implemented in HyFlow to lookup objects which are distributed over the network. HyFlow uses a voting algorithm, the dynamic two phase commitment protocol (D2PC) [83], to support control flow transactions. Network communication is supported using protocols including TCP [25], UDP [81], and SCTP [97]. We also implement a suite of distributed benchmark applications in HyFlow,

which are largely inspired by the multiprocessor STM STAMP benchmark suite [24], to evaluate D-STM. The benchmarks include Bank, Loan, and Vacation. In addition, we implement micro-benchmarks — distributed versions of data structures including Linked-List and Binary Search Tree.

Figure 1.2 shows example transactional code in HyFlow, in which two bank accounts are accessed and an amount is atomically transferred between them. At the programming level, no locks are used, the code is self-maintained, and atomicity, consistency, and isolation are guaranteed (for the `transfer` transaction). Composability is also achieved: the atomic `withdraw` and `deposit` methods have been composed into the higher-level atomic `transfer` operation. A conflicting transaction is transparently retried. Note that the location of the bank account is hidden from the program. It may be cached locally, migrate to the current node, or accessed remotely using remote calls, which is transparently accomplished by HyFlow.

We experimentally evaluated HyFlow against competing models including Java remote method invocation (RMI) with mutual exclusion and read/write locks, distributed shared memory (DSM) [103], and directory-based D-STM. Our studies show that HyFlow with its underlying algorithms outperform competitors by as much as 40-190% on a broad range of transactional workloads on a 120-node system, with more than 1000 active concurrent transactions.

The thesis makes the following contributions:

- We present the HyFlow D-STM framework, which exports a simple distributed programming model, based on annotations as a meta-data mechanism. In contrast to using a particular programming model (e.g., PGAS [2]), HyFlow is based on Java's general-purpose programming model. Additionally, HyFlow provides pluggable support for different D-STM algorithms, cache coherence protocols, contention policies, and network protocols. This is the first such D-STM framework. We show that HyFlow delivers high performance, without any changes to the underlying virtual machine or compiler.

- We present the TFA algorithm — the first distributed transactional locking algorithm for dataflow D-STM.

- We present Snake-DSTM [91], a control-flow D-STM implementation, based on the Java RMI mechanism for supporting remote procedure calls. This is the first D-STM design and implementation for the control flow distributed execution model.

- We identify and formally characterize the factors of overhead and trade-off between control-flow and dataflow D-STM execution models.

- We implement a set of distributed benchmark applications and micro-benchmarks for evaluating D-STM designs, algorithms, and implementations.

```
1  public class BankAccount implements IDistinguishable {
2   ....
3   @Override
4   public Object getId() {
5       return id;
6     }
7
8     @Remote  @Atomic{retries=100}
9     public void deposit(int dollars) {
10      amount = amount + dollars;
11    }
12
13    @Remote  @Atomic
14    public boolean withdraw(int dollars) {
15      if(amount>=dollars) return false;
16          amount = amount - dollars;
17        return true;
18    }
19 }
```

```
1  public class TransferTransaction {
2   @Atomic{retries=50}
3   public boolean transfer(String accNum1, String accNum2, int amount) {
4     DirectoryManager locator = HyFlow.getLocator();
5     BankAccount account1 = locator.open(accNum1);
6     BankAccount account2 = locator.open(accNum2);
7
8     if(!account1.withdraw(amount))
9         return false;
10    account2.deposit(amount);
11
12    return true;
13    }
14 }
```

Figure 1.2: A bank transaction using an atomic TM method.

The rest of the work is organized as follows. Section 1 describes the problem domain, challenges, and motivations for this research. We overview past and related efforts in Chapter 2. In Chapter 3, we detail our system model, and introduce HyFlow's programming model and architecture. Chapter 4 describes HyFlow's implementation for the two execution models, dataflow, and control-flow, and Chapter 5 formally describes the factors of overhead in both models. We experimentally evaluate HyFlow against competing efforts using benchmarks and micro-benchmarks in Chapter 6. We conclude in Chapter 7.

Chapter 2

Related Work

2.1 Transactional Memory.

The classical solution for handling shared memory during concurrent access is lock-based techniques [7, 61], 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. TM, proposed by Herlihy and Moss [55], is an alternative approach for shared memory access, with a simpler programming model. Memory transactions are similar to database transactions: a transaction is a self-maintained entity that guarantees atomicity (all or none), isolation (local changes are hidden till commit), and consistency (linearizable execution). TM has gained significant research interest including that on STM [96, 75, 49, 46, 53, 54, 73], HTM [55, 45, 5, 20, 76], and HyTM [14, 28, 77, 65]. STM has relatively larger overhead due to transaction management and architecture-independence. HTM has the lowest overhead, but assumes architecture specializations. HyTM seeks to combine the best of HTM and STM.

STM can be broadly classified into static or dynamic. In static STM [96], all accessed objects are defined in advance, while dynamic STM [53, 54, 73] relaxes that restriction. The dominant trend among STM designs is to implement the single-writer/multiple-reader pattern, either using locks [31, 30] or obstruction-free techniques [85, 53], while few implementations allow multiple writers to proceed under certain conditions [86]. In fact, it is shown in [32] that obstruction-freedom is not an important property and results in less efficient STM implementations than lock-based ones. Another orthogonal TM property is object acquisition time: pessimistic approaches acquire objects at encounter time [32, 6], while optimistic approaches do so at commit time [31, 30]. Optimistic object acquisitions generally provide better concurrency with acceptable number of conflicts [30]. STM implementations also rely on write-buffer [72, 73, 53] or undo-log [77] for ensuring a consistent view of memory. In write-buffer, object modifications are written to a local buffer and take effect at commit

time. In the undo-log method, writes directly change the memory, and the old values are kept in a separate log to be retrieved at abort.

2.2 Distributed Shared Memory

Supporting shared memory access in distributed systems has been extensively studied through the DSM model. Earlier DSM proposals were page-based [68, 36, 4, 9] that provide sequential consistency using single-writer/multiple-reader protocol at the level of memory pages. Though they still have a large user base, they suffer from several drawbacks including *false sharing*. This problem occurs when two different locations, located in the same page, are used concurrently by different nodes, causing the page to "bounce" between nodes, even though there is no shared data [35]. In addition, DSM protocols that provide sequential consistency have poor performance due to the high message overhead incurred [4]. Furthermore, single-writer/multiple-reader protocols often have "hotspots," degrading their performance. Also, most DSM implementations are platform-dependent and does not allow node heterogeneity.

Variable-based DSM [17, 16] provides language support for DSM based on shared variables, which overcomes the false-sharing problem and allows the use of multiple-writer/multiple-reader protocols. With the emergence of object-oriented programming, object-based DSM implementations were introduced [10, 69, 99, 100, 80] to facilitate object-oriented parallel applications.

2.3 Distributed STM

Similar to multiprocessor STM, D-STM was proposed as an alternative to lock-based distributed concurrency control. Figure 2.1 shows a taxonomy of different distributed memory concurrent access models. Broadly, it can be classified based on the mobility of transactions and objects. Mobile transactions [10, 69, 99] use an underlying mechanism (e.g., RMI) for invoking operations on remote objects.

The mobile object model [100, 80, 56, 105, 88] allows objects to move through the network to requesting transactions, and guarantees object consistency using cache coherence protocols. Usually, these protocols employ a directory that can be tightly coupled with its registered objects [57], or permits objects to change their directory [29, 56, 105, 11].

The mobile object model can also be classified based on the number of active object copies. Most implementations assume a single active copy, called *single version*. Object changes can then be a) applied locally, invalidating other replicas [103], b) applied to one object (e.g., latest version of the object [33, 51]), which is discovered using directory protocols [29, 57], or c) applied to all replicated objects [3, 13, 70]. In contrast, multiversion concurrency control (MVCC) proposals allow multiple copies or replicas of an object in the network [59, 85].

- Distributed Memory Models
  - Control Flow [10, 69, 99]
  - Data Flow
    - Single version [33, 51]
      - Directory Based
        - Tightly coupled [57]
        - Variable [29, 56, 105, 11]
      - Invalidate [103]
      - Replicate [3, 13, 70]
    - Multiversion
      - Linearizable [59]
      - Non-Linearizable [85]
  - Hybrid Flow [21]

Figure 2.1: A Distributed STM taxonomy

The MVCC models often favor performance over linearizable execution [58]. For example, in [85], reads and writes are decoupled to increase transaction throughput, but allows reading of older versions instead of the up-to-date version to prevent aborts.

System architecture and the scale of the targeted problem can affect design choices. With a small number of nodes (e.g., 10) interconnected using message-passing links, cache-coherent D-STM (cc D-STM) [56, 29, 105] is appropriate. However, for a cluster computer, in which a group of linked computers work closely together to form a single computer, researchers have proposed cluster D-STM [21, 71, 64, 27, 87]. The most important difference between the two is communication cost. cc D-STM assumes a metric-space network between nodes, while cluster D-STM differentiates between access to local cluster memory and remote memory at other clusters.

Object conflicts and object consistency are managed and ensured, respectively, by contention management and cache coherence protocols. In [56], they present a cache-coherence protocol, called Ballistic. Ballistic models the cache-coherence problem as a distributed queuing problem, due to the fundamental similarities between the two, and uses the Arrow queuing protocol [29] for managing transactional contention. Ballistic's hierarchical structure degrades its scalability—e.g., whenever a node joins or departs the network, the whole structure has to be rebuilt. This drawback is overcome in Zhang and Ravindran's Relay protocol [105, 106], which improves scalability by using a peer-to-peer structure. Relay assumes encounter time object access, which is applicable only for pessimistic STM implementations, which, relative to optimistic approaches, suffer from large number of conflicts [30].

While these efforts focused on D-STM's theoretical properties, several other efforts developed implementations. In [21], Bocchino *et. al.* proposed a word-level cluster D-STM. 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 com-

munication can be aggregated with data communication to improve scalability. However, each processor is limited to one active transaction at a time. Also, in their implementation, no progress guarantees are provided, except for deadlock-freedom. In [71], Manassiev *et. al.* present a page-level distributed concurrency control algorithm for cluster D-STM, which automatically detects and resolves conflicts caused by data races for distributed transactions accessing shared memory data. 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.

Kotselidis *et. al.* present the DiSTM [64] object-level, cluster D-STM framework, as an extension of DSTM2 [53], 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 [45], 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 [39], this overheard is eliminated. However, an extra validation step is added to the master node, as well as bottlenecks are created upon acquiring and releasing the leases. 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 [21]).

Couceiro *et. al.* present D^2STM [27]. Here, STM is replicated on distributed system nodes, and strong transactional consistency is enforced at commit time by a non-blocking distributed certification scheme. In [62], Kim and Ravindran develop a D-STM transactional scheduler, called Bi-interval, that optimizes the execution order of transactional operations to minimize conflicts, yielding throughput improvement of up to 200%. Romano *et. al.* extend cluster D-STM for Web services [87] and Cloud platforms [88]. In [87], they present a D-STM architecture for Web services, where application's state is replicated across distributed system nodes. D-STM ensures atomicity and isolation of application state updates, and consistency of the replicated state. In [88], they show how D-STM can increase the programming ease and scalability of large-scale parallel applications on Cloud platforms.

The high popularity of the Java language for developing large, complex systems has motivated significant research on distributed and concurrent programming models. DISK [98] is a distributed Java Virtual Machine (DJVM) for network of heterogenous workstations, and uses a distributed memory model using multiple-writer memory consistency protocol. Java/DSM [104] is a DJVM built on top of the TreadMarks [4] DSM system. JESSICA2 [107] provides transparent memory access for Java applications through a single system image (SSI), with support for thread migration for dynamic load balancing. These implementations facilitate concurrent access for shared memory. However, they rely on locks for distributed concurrency control, and thereby suffer from (distributed) deadlocks, livelocks, lock-convoying, priority inversion, non-composability, and the overhead of lock management.

HyFlow is an object-level D-STM framework, with pluggable support for the least common D-STM denominators, including directory lookup, transactional synchronization and recovery, contention management, cache coherence, and network communication. It supports both control and data flow, and implements a variety of algorithms as defaults. In addition, it doesn't require any changes to the underlying virtual machine or compiler, unlike [2, 23].

Chapter 3

System Architecture

3.1 System Model

We consider an asynchronous distributed system model, similar to Herlihy and Sun [56], consisting of a set of N nodes $N_1, N_2, ..., N_n$, communicating through weighted message-passing links E. Let $G = (N, E, c)$ be an undirected graph representing the network, where c is a function that defines the link communication cost. Let M denote the set of messages transferred in the network, and $Size(M_i)$ the size of a message M_i. A message could be a remote call request, vote request, resource publish message, or any type of message defined in HyFlow's protocols. A fixed minimum spanning tree S or G is used for broadcasting. Thus, the cost of message broadcasting is $O(|N|)$, which we define as the constant Ω.

We assume that each shared object has an unique identifier. We use a grammar similar to the one in [41], but extend it for distributed systems. Let $O = \{o_1, o_2, ...\}$ denote the set of objects shared by transactions. An object may be replicated or may migrate to any other node. Without loss of generality, objects export only *read* and *write* methods (or operations). Thus, we consider them as shared registers. Let $T = \{T_1, T_2, \ldots\}$ 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 transaction can be in one of three states: *active*, *aborted*, or *committed*. When a transaction is aborted, it is retried by the node again using a different identifier.

Every object has, at least, one "owner" node that is responsible for handling requests from other nodes for the owned object. Let $Own(O_i)$ and $Size(O_i)$ be functions that represent the owner and size of object O_i, respectively. In the data-flow model, a cache-coherence protocol locates the current cached copy of the object in the network, and moves it to the requesting node's cache. Under some circumstances, the protocol may change the object's owner to a new owner. Changes to the ownership of an object occurs at the successful commit of the

object-modifying transaction. At that time, the new owner broadcasts a *publish* message with the owned object identifier.

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.

In the control flow model, any node that wants to read from, or write to an object, contacts the object's owner using a remote call. The remote call may in turn produce other remote calls, which construct, at the end of the transaction, a global graph of remote calls. We call this graph, a *call graph*.

3.2 Programming Model

We introduce our programing model by example. We will walk through the construction of a distributed linked-list implementation, in which nodes are scattered and distributed over the network. Under the multiple-reader single-writer pattern, a linked-list node can be replicated over the network as long as there is no active writer transactions. In Section 6.3, we will show that this implementation outperforms a fine-grained, handcrafted lock-coupling implementation [15], which acquires locks in a "hand-over-hand" manner.

Since objects are distributed over the network, normal references cannot be used to access the objects. Instead, any distributed class must implement the `IDistinguishable` interface with a single method `getId()` that retrieves an unique object identifier. A `Locator` object, which encapsulates a directory lookup protocol is used to retrieve objects using their identifiers.

Objects are opened in three modes: *shared*, *read* and *write*. "Shared" accessed objects are retrieved by the directory manager (using the underlying cache coherence protocol), but not added to the transaction read set. While objects opened in "read" or "write" modes are added to the read set or write set, respectively.

```
DirectoryManager locator = HyFlow.getLocator();
// for shared access
Node head = locator.open(headId, "s");
// for reading
Node head = locator.open(headId, "r");
// for writing (default)
Node head = locator.open(nodeId); // or
Node head = locator.open(nodeId, "w");
```

A distributed class exports one (or more) remote methods. Remote methods can be invoked regardless of their objects' physical location. A remote method is defined by annotating it as `@Remote`. Remote methods are analogous to RMI stub methods.

```
@Remote
public void setNext(String nextId){
  ....
}
```

Critical sections of the code that must execute atomically are defined as atomic methods (annotated as `@Atomic`). This approach has two advantages. First, it retains the familiar familiar Java programming model where `@Atomic` replaces `synchronized` methods. Secondly, it simplifies transactional memory maintenance, which has a direct impact on performance, as the Transaction Manager, which is responsible for transaction synchronization and recovery (see Section 3.3.3), need not handle local method variables as part of a transaction.

```
1  public class Node implements IDistinguishable{
2      private String id;
3      private Integer value;
4      private String nextId;
5      public Node(String id, Integer value) {...}
6      @Remote
7      public void setNext(String nextId){...}
8      @Remote
9      public String getNext(){...}
10     @Remote
11     public Integer getValue(){...}
12     @Override
13     public Object getId() {...}
14 }
```

Figure 3.1: The code of a Node in the Distributed Linked List implementation.

```
@Atomic{retries=50, timout=1000}
public static void add(Integer value){
  ....
}
```

With atomic methods, a maximum number of retries can be specified or the maximum allowed time for the transaction to terminate, both parameters are optional and the default is infinity. These option is analogous to `tryLock()` and `InterruptedException` in the context of classical lock, which gives programmer more temporal control over transaction execution.

The Transaction Manager will attempt to execute the method atomically, retrying until commit, up to the maximum number of times specified (or the timout period expires), after which, it will throw a `TransactionalException` to the caller method.

Figures 3.1 and 3.2 show a code snippet of our distributed linked-list implementation. List manipulation operations—e.g., *add*, *remove*, and *contains*, are defined as `@Atomic` methods, while node fields are accessed through setter/getter `@Remote` methods.

Shared object access permits a further optimization for data structure algorithms. For example, in the shown implementation, if we want to delete a node at the end of the list, there is no need to add all nodes starting from the head to current node to the transaction read set. Instead, these objects can be opened using "shared" mode and relying on premoting the `prevNode` and `deletedNode` to "write" mode at the removal step. The same can be applied for `contains`, with promoting last accessed node to "read" mode right before return to force adding it to the read-set.

In contrast to the Java RMI programming model, in HyFlow's model, no additional interfaces are defined, and method prototypes remain unchanged. Furthermore, atomicity is implicit

```
1  public class List{
2    // head sentinel node identifier
3    String final HEAD = ...;
4      ....
5        @Atomic
6        public static void add(Integer value){
7         Locator locator = HyFlow.getLocator();
8         Node head = (Node)locator.open(HEAD);
9         String oldNext = head.getNext();
10        String newNodeId = ... //generate random id
11        Node newNode = new Node(newNodeId, value);
12        newNode.setNext(oldNext);
13        head.setNext(newNodeId);
14       }
15       ....
16       @Atomic
17       public boolean delete(Integer value){
18        Locator locator = HyFlow.getLocator();
19        String next = HEAD, prev = null;
20        do{    // find the node
21          Node node = locator.open(next, "r");
22            if(value.equals(node.getValue())){
23              //reopen for write to be deleted
24                Node deletedNode = locator.open(next);
25                //open previous node for write
26                Node prevNode = locator.open(prev);
27                prevNode.setNext(deletedNode.getNext());
28                locator.delete(deletedNode);
29                return true;
30            }
31            prev = next;
32            next = node.getNext();
33        }while(next!=null);
34        return false;
35       }
36 }
37 ....
38 @Atomic
39   public boolean contains(Integer value){
40       Locator locator = HyFlow.getLocator();
41       String next = HEAD;
42       do{ // find the node
43           Node node = locator.open(next, "r");
44           if(value.equals(node.getValue()))
45                   return true;    // Found
46           next = node.getNext();
47       }while(next!=null);
48       return false;   // Not Found
49   }
```

Figure 3.2: Distributed Linked List Example.

in the model without the need to handle locks or suffer from its drawbacks and pitfalls. In addition, unlike other TM implementations [53], there is no need to define factories to create objects; a normal usage of the `new` keyword registers that object with the object registry. Note that newly created (or deleted) objects within an atomic method are populated only at the end of that method and upon a successful termination of the transactional code, which preserves transactional isolation.

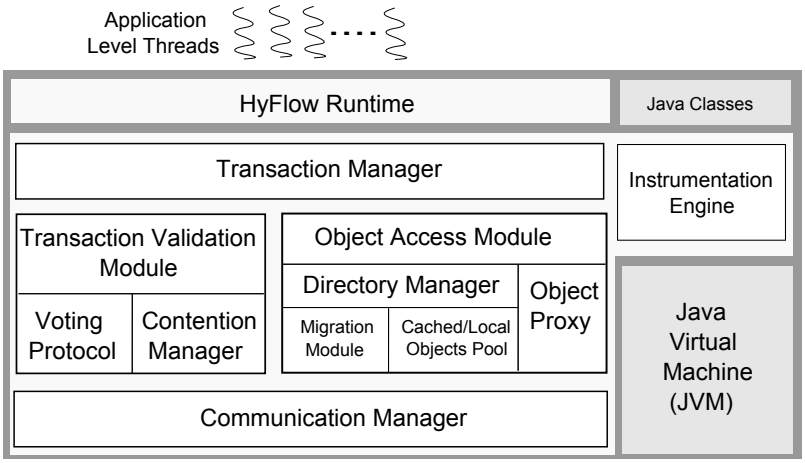


Figure 3.3: HyFlow Node Architecture

3.3 Architecture

Figure 3.3 shows the nodal architecture of HyFlow. Five modules and a *runtime handler* form the basis of the architecture. The modules include the *Transaction Manager*, *Instrumentation Engine*, *Object Access Module*, *Transaction Validation Module*, and *Communication Manager*.

The HyFlow runtime handler represents a standalone entity that delegates application-level requests to the framework. HyFlow uses run-time instrumentation to generate transactional code, like other (multiprocessor) STM such as Deuce [63], yielding almost two orders of magnitude superior performance than reflection-based STM (e.g., [53]).

The Transaction Manager contains mechanisms for ensuring a consistent view of memory for transactions, validating memory locations, and retrying transactional code when needed. Based on the access profile and object size, object migration is permitted.

The 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.

Every node employs a Transaction Manager, which runs locally and handles local transactional code. The Transaction Manager treats remote transactions and local transactions equally. Thus, the distributed nature of the system is seamless at the level of transaction management.

The Object Access Module has three main tasks: 1) providing access to the object owned by the current node, 2) locating and sending access requests to remote objects, and 3) retrieving any required object meta-data (e.g., latest version number). Objects are located with their

IDs using the Directory Manager, which encapsulates a directory lookup protocol [57, 29, 105]. Upon object creation, the Directory Manager is notified and publishes the object to other nodes. The Migration Module decides when to move an object to another owner or keep it locally. The purpose of doing so is to exploit object locality and reduce the overall communication traffic between nodes.

The Transaction Validation Module ensures data consistency by validating transactions upon their completion. It uses two sub-modules:

- *Contention Manager.* This sub-module is consulted when conflicts occur—i.e., when two transactions access a shared object, and one access is a write. When local transactions conflict, a contention management policy (e.g., Karma, Aggressive, Polite, Kindergarten, Eruption [95, 94]) is used to abort or postpone one of the conflicting transactions. However, when one of the conflicting transactions is remote, the contention policy decision is made globally based on heuristics (we explain this later in Section 3.3.3).
- *Global Voting handler.* In order to validate a transaction based on control flow, a global decision must be made across all participating nodes. This sub-module is responsible for collecting votes from other nodes and make a global commit decision such as by a voting protocol (e.g., D2PC [83]).

3.3.1 Instrumentation Engine

Instrumentation is a Java feature that allows the addition of byte-codes to classes at run-time. In contrast with reflection, instrumentation works just once at class load time, which incurs much less overhead. HyFlow's Instrumentation Engine (HyIE) is a generic Java source code processor, which inserts transactional code at specified locations in a given source code. HyIE employs *Annotations* — a Java 5 feature that provides runtime access to syntactic form of metadata defined in source code, to recognize portions of code that need to be transformed. Using run-time instrumentation to generate transactional code yielding almost two orders of magnitude superior performance than reflection-based STM (e.g., [53]). HyIE is built as an extension of the Deuce (multiprocessor) STM [63], which is based on ASM [18], a Java bytecode manipulation and analysis framework.

Like Deuce, we consider a Java method as the basic annotated block. This approach has two advantages. First, it retains the familiar programming model, where `@Atomic` replaces `synchronized` methods and `@Remote` substitutes for RMI calls. Secondly, it simplifies transactional memory maintenance, which has a direct impact on performance. The Transaction Manager need not handle local method variables as part of a transaction.

Our Instrumentation Engine works in two phases; the first phase processes *remote objects*. For any class with one (or more) methods annotated as `@Remote`, a `Remote` interface is

```
1  public class BankAccount implements IDistinguishable {
2    // Remote access flag
3    boolean remote_obj$ = false;
4    ....
5    // Modified constructor
6    BankAccount(String id){
7      ....
8      DirectoryManager.register(id, this);
9    }
10   ....
11   // Synthetic duplicate method
12     public void deposit(int dollars, Context c) {
13         amount__Setter$(c, amount__Getter$(c) + dollars);
14     }
15     // Original method instrumented
16     public void deposit(int dollars) {
17       if(remote_obj$){
18         //Invoke remote call
19         Proxy.open(id).deposit(dollars);
20         return;
21       }
22       //Transaction active thread
23       Context context = ContextDelegator.getInstance();
24     boolean commit = true;
25       for (int i=100; i>0; --i) {
26         //Initialize transaction
27       context.init();
28       try{
29           //Try execute
30         result=deposit(dollars,context);
31       } catch(TransactionException ex) {
32         commit = false;  //Aborted
33       } catch(Throwable ex) {
34         //Application Exception
35         throwable = ex;
36       }
37       if(commit){
38         if (context.commit()) {
39           if (throwable == null)
40             return result; //Committed
41           //Rethrow Application exception
42           throw (IOException)throwable;
43         }
44      }else{
45         context.rollback(); //Rollback
46         commit = true;
47      }
48     }
49     //Maximum retries reached
50     throw new TransactionException();
51     }
52 }
```

Figure 3.4: Instrumented version of BankAccount class.

generated with the remote method's signature. Further, a delegator class that implements the `Remote` interface is generated to work as the RMI-client stub. The original class constructors are modified to register objects at the object registry and populate object IDs to other nodes. That has two purposes: i) objects are accessed with a reference of the same type, so objects and object proxies are treated equally and transparently; and ii) no changes to remote method signatures are required, as the modified signature versions are defined by delegator generated code. This phase simplifies the way remote objects are accessed, and reduces the burden of writing complex code.

Any distributed class must implement the `IDistinguishable` interface with a single method `getId()`. The purpose of this restriction is to decouple object references from their memory locations. HyIE detects any loaded class of type `IDistinguishable` and transforms it to a transactional version. Further, it instruments every class that may be used within transactional code. This transformation occurs as follows:

- ***Classes.*** A synthetic field is added to represent the state of the object as local or remote. The class constructor(s) code is modified to register the object with the Directory Manager at creation time.
- ***Fields.*** For each instance field, setter and getter methods are generated to delegate any direct access for these fields to the Transaction manager. Class code is modified accordingly to use these methods.
- ***Methods.*** Two versions of each method are generated. The first version is identical to the original method, while the second one represents the transactional version of the method. During the execution of transactional code, the second version of the method is used, while the first version is used elsewhere.
- ***@Atomic methods.*** Atomic methods are duplicated as described before, however, the first version is not similar to the original implementation. Instead, it encapsulates the code required for maintaining transactional behavior, and it delegates execution to the transactional version of the method.
- ***@Remote methods.*** RMI-like code is generated to handle remote method calls at remote objects. In the control flow model, the Directory Manager can open the object, but cannot move it to the local node. An object appears to application code as a local object, while transformed methods call their corresponding original methods at the remote object.

Figure 3.4 shows part of the instrumented version of a `BankAccount` class defined in Figure 1.2. It is worth noting that the *closed nesting* model [84], which extends the isolation of an inner transaction until the top-level transaction commits, is implicitly implemented. HyIE "flattens" nested transactions into the top-level one, resulting in a complete abort on conflict, or allow partial abort of inner transactions. Whenever an atomic method is called

within the scope of another atomic method, the duplicate method is called with the parent's `Context` object, instead of the instrumented version.

3.3.2 Object Access Module

During transaction execution, a transaction accesses one or more shared objects. The Directory Manager delegates access to all shared objects. An object may reside at the current node. If so, it is accessed directly from the *local object pool*. Or, it may reside at another node, and if so, it is considered as a *remote object*. Remote objects may be accessed differently according to the transaction execution model—i.e., control or dataflow. In the dataflow model, a *Migration Module* guarantees local access to the object. It can move the object, or copy it to the current node, and update the directory accordingly. In the control flow model, a *Proxy Module* provides access to the object through an empty instance of the object "facade" that acts as a proxy to the remote object. At the application level, these details are hidden, resulting in an uniform access interface for all objects.

It is interesting to see how the example in Figure 1.2 works using the dataflow and control flow models. Assume that the two bank accounts accessed in this example reside at different nodes. In the dataflow model, the transaction will call the Object Access Manager, which in turn, will use the Directory Manager to retrieve the required objects. The Directory Manager will do so according to the underlying implementation and contention management policy. Eventually objects (or copies of them) will be transferred to the current node. Upon completion, the transaction will be validated and the new versions of the objects will be committed.

Now, let us repeat the scenario using the control flow model. In this case, the Object Access Manager will employ an *Object Proxy* to obtain proxies to the remote object. Remote calls will be sent to the original objects. As we explain in the next section, once the transaction completes, a voting protocol will decide whether to commit the transaction's changes or to retry again.

3.3.3 Transaction Validation Module

The main task of this module is to guarantee transaction consistency, and to achieve system-wide progress. Recall that, in HyFlow, a transaction may be mobile or immobile. Thus, this module employs two sub-modules: 1) a Voting Manager, which is used for mobile transactions to collect votes from other participating nodes, and 2) a Global Contention Manager, which is consulted to resolve conflicting transactions (this is needed for both mobile and immobile transactions).

Voting Manager In the control flow model, a remote call on an object may trigger another remote call to a different object. The propagated access of objects forms a *call graph*, which is

composed of nodes (sub-transactions) and undirected edges (calls). This graph is essential for making a commit decision. Each participating node may have a different decision (on which transaction to abort/commit) based on conflicts with other concurrent transactions. Thus, a voting protocol is required to collect votes from nodes, and the originating transaction can commit only if it receives an "yes" message from all nodes. By default, we implement the D2PC protocol, however any other protocol may substitute it. We choose D2PC, as it yields the minimum possible time for collecting votes [83], which reduces the possibility of conflicts and results in the early release of acquired objects. Furthermore, it balances the overhead of collecting votes by having a variable coordinator for each vote. For completeness, in Section4.2.1.5 we overview the key idea of D2PC in the context of transactional memory (more information is available in [83]).

Global Contention Manager In contention manager-based STM implementations, the progress of the system is guaranteed by the contention policy. Having a special module for global contention management enables us to achieve effective decisions for resolving distributed transactional conflicts. Using classical non-distributed contention policies for this may be misleading and expensive. This module employs a set of heuristics for making such decisions, including the following:

- A local transaction that accesses local objects is aborted only when it conflicts with any distributed transaction.
- A distributed transaction that follows the dataflow model is favored over one that uses control flow, because the former is usually more communication-expensive.
- If two distributed transactions in the dataflow model conflict, we abort the one that a) accesses objects having a smaller total size, or b) communicates with less number of remote nodes.
- In all other cases, a contention manager employs any local methodology such as Karma, Aggressive, Polite, Kindergarten, or Eruption [95] [94].

Chapter 4

Distributed Software Transactional Memory

4.1 Dataflow D-STM

In this section, we consider Herlihy and Sun's dataflow D-STM model [56]. In this model, transactions are immobile and objects are dynamically migrated to invoking transactional nodes. To ensure (distributed) transactional properties including atomicity, consistency, and isolation in such a model, we develop a distributed transactional management algorithm which we call TFA. TFA is fully distributed and does not rely on a central clock. In TFA, each node has its own clock and the "happens-before" relationship [67] is established between relevant events (e.g., write-after-write, read-after-write). This approach enables the algorithm to perform well for read-dominated transactional workloads, while yielding comparable or superior performance for read-dominated workloads, with respect to other distributed concurrency control models (e.g., Java remote method invocation (RMI), distributed shared memory (DSM) [103], directory-based D-STM [57]).

TFA is an object-level lock-based algorithm with lazy acquisition. Network traffic is reduced by limiting broadcasting to just the object identifiers.

We show that TFA is opaque [41] (i.e., its correctness property) and permits strong progressiveness [41] (i.e., its progress property). 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. The happens-before relationship alone is not sufficient to ensure opacity and distributed atomicity. We also develop a validation mechanism to guarantee these correctness properties.

We also establish a message upper bound for TFA. In TFA, objects are acquired at com-

mit time, while other pessimistic approaches [57, 32, 77, 5] acquire objects at encounter time. TFA's optimistic approach provides better concurrency with ten times less number of conflicts.

4.1.1 The TFA Algorithm

We now describe the TFA algorithm that orchestrates distributed transactions running over a set of nodes. The problem of locating objects is outside the scope of our work — we can use any directory or cache coherence protocol that solves this problem such as the Arrow protocol [29], or the Ballistic protocol [56]. 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 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, wv, during its commit step. The current clock value is piggybacked on all messages, and a variant of Lamport' synchronization mechanism [67] is used to keep the clocks synchronized.

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 incremented by every successful transactional change.

Figures 4.1.1– 4.1.5 describe the algorithm's main procedures. When a transaction begins, it reads the current clock value of the node on which it is executing (Figure 4.1.1). Let us call this clock value 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 [19] 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.

Algorithm 4.1.1 TFA Transaction::Init

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

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

An object may be accessed locally or remotely. Accessing of local objects is preceded by a post-read validation step to check if the object version $<$ wv; otherwise the transaction is

Algorithm 4.1.2 TFA Locator::OpenTransactional

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:    if trans.wv < obj.owner.clock then
7:       for all obj in transaction.readSet do
8:          if obj.version > obj.owner.clock then
9:             return rollback()
10:      trans.wv = obj.owner.clock
11: else
12:    if obj.version > trans.wv then
13:       return rollback()
14: return obj
```

Algorithm 4.1.3 TFA Node::RetriveObject

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: return LocalObjects.get(id)
```

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 propose a novel mechanism, called *Transaction Forwarding*, which efficiently provides early validation of remote objects and guarantees a consistent view of memory, even when using asynchronous clocks. Figures 4.1.2 and 4.1.3 illustrate Transaction Forwarding. The mechanism 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, a "transaction forwarding" step is needed (explained below), and the current clock value *lc* is advanced to the value *rc*.

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 change its starting time to *wv'* safely.

When a transaction completes, we need to ensure that it reads a consistent view of data (Figure 4.1.4). 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 is needed to ensure that a transaction sees a consistent view of objects. Upon successful completion of this step, a transaction can commit safely as discussed in the next step.
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 to the main copy, while for remote objects, we simply publish the current node as the new owner of the object using the *Locator* publish service.

Algorithm 4.1.4 TFA Transaction::Commit

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

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

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.

Algorithm 4.1.5 TFA Transaction::Abort

Require: Transaction *trans*
Ensure: Abroting transaction.

```
1: for all obj in transaction.writeSet do
2:    obj.releaseLock()
3: trans.readSet.clear()
4: trans.writeSet.clear()
```

An aborted transaction releases all acquired locks (if any), clears its read and write sets, and restarts again by reading new *wv* (Figure 4.1.5).

Figure 4.1 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, respectively. 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

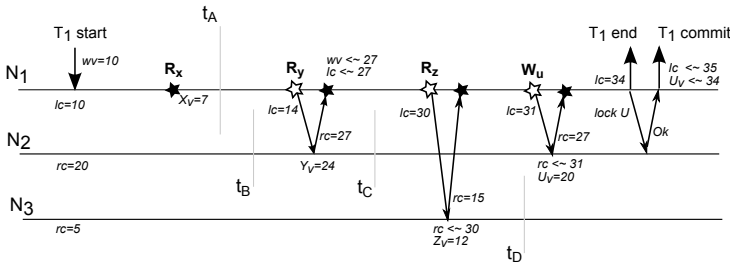


Figure 4.1: An execution of a distributed transaction under TFA.

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. $T_{1_{wv}}$ 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. 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 4.1, 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 at 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 our algorithm, 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.

4.1.2 Properties

We now prove the correctness and the progress guarantees of TFA. We also establish the message bound of the algorithm, and analyze the cost of the early-validation step.

Correctness. A correctness criterion for TM, called Opacity [40] [42], has been proposed as a safety property for TM implementations that suits the special nature of memory transactions. Similar to strict serializability [79], 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. In [40], it is shown that other correctness properties such as Linearizability [58], Serializability [79], Global Atomicity, Rigorous Scheduling [22], and Recoverability [44] are not sufficient to describe TM correctness, while Opacity is.

Theorem 4.1.1. *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 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 should see either the old values of *all* objects or the new values of *all* objects. If T_k was aborted, then any other transaction should see the old values of *all* objects O. As the abortion case already covered by the second opacity condition, we will prove the successful commit case in the next paragraph.

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.

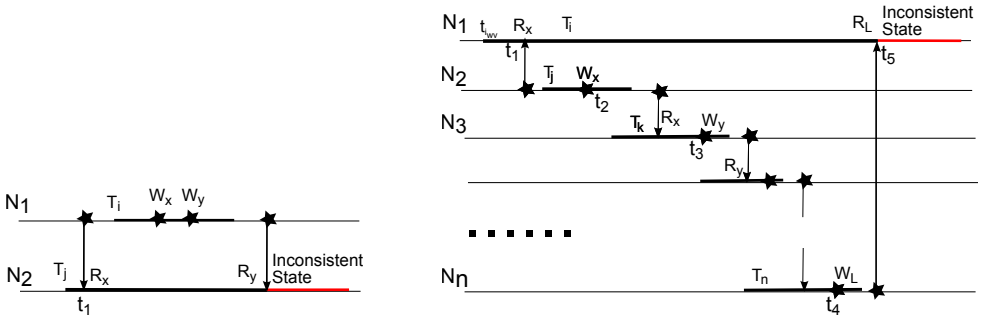
Consider the simplest case of two conflicting transactions, Figure 4.2(a). Here, T_j reads the old value of O_x, before T_i modifies both O_x and O_y, and commits successfully. Thus, $T_j \Leftarrow_{old} T_i$. Later, if T_j retrieved the new value of O_y, then it violates consistency, as $T_j \Leftarrow_{new} T_i$. 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 must be aborted before entering the inconsistent state.

Now, we will generalize this for any number of transactions (Figure 4.2(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 happen-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 $T_{iwv} \prec t_1$. 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 $N_n > T_{iwv}$. 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 [41] 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 variable, then at least one of them is guaranteed to commit.

Theorem 4.1.2. *TFA is strongly progressive.*

Proof. Assume, by way of contradiction, that TFA is not strongly progressive. Then, there exists a maximal set Q, and all transactions in Q were aborted. (By *maximal*, we mean that no transaction in Q has a conflict with a transaction outside of Q.)



(a) Simple inconsistent state.

(b) Inconsistent state with more than 2 transactions.

Figure 4.2: Possible opacity violation scenarios

Assume that *conf(Q)* $= \Phi$, which means that no transaction conflicts with another on a shared object. Recall that none of the transactions had successfully committed, which implies a failure in the validation step. This validation failure can imply either 1) a failure in acquiring the locks because some other transaction already acquired those locks, or 2) a read-set validation failure. In both cases, there must exist a conflicting object that either caused a lock-failure or a validation-failure. Therefore, *conf(Q)* $\neq \Phi$, which yields a contradiction.

Now, let us assume that $|conf(Q)| \geq 1$. Suppose that no transaction manages to acquire the lock on $O_x \in conf(Q)$, or that all transactions failed in their read-set validation due to a change in O_x's version. This implies that a foreign transaction not in Q, managed to acquire the lock on O_x, or managed to change its value, which contradicts our first assumption that Q is a maximal set. If the value of O_x has been changed by some transaction in Q, then, that indicates that the modifying transaction has committed, which contradicts the assumption that none of the transactions were able to commit. The other possibility is that a transaction, T_k, failed to acquire O_x's lock, which only occurs when another transaction $T_j in Q$ already has the lock and deadlocks with T_k. Clearly, that cannot happen due to the incremental way of acquiring the object locks, as described earlier, which yields a contradiction. □

Strong progressiveness is not the strongest possible progress property. However, it is the *de facto* progress property for most lock-based STM implementations such as TL2 [30], TinySTM [34], RSTM [73], and McRT-STM [6]. The strongest progress property mandates that no transaction is ever forcefully aborted, which is impractical to implement due to its significant complexity and overhead.

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 committing transactions safely outweigh the number of early validation operations. We now establish an upper bound for these costs.

Theorem 4.1.3. *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. □

Lemma 4.1.4. *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. It is clear that the maximum number of early validations is the number of objects accessed by transaction $|O_s|$. □

Lemma 4.1.5. *The worst case number of messages in an early validation step = 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 4.1.6. *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 4.1.5 and Lemma 4.1.4, 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. From Lemma 4.1.6, we can calculate the total number of messages for all early validations during transaction lifetime. □

4.2 Control D-STM

Most of the previous research on D-STM has largely focused on the dataflow model [100, 80], in which objects are replicated (or migrated) at multiple nodes, and transactions access local object copies. Using cache coherence protocols [56, 29, 105], consistency of the object copies is ensured. However, this model is not suitable in applications (e.g., P2P), where objects cannot be migrated or replicated due to object state dependencies, object sizes, or security restrictions. A control flow model, where objects are immobile and transactions invoke object operations via remote procedure calls (RPCs), is appropriate in such instances.

This section focuses on the design and implementation of D-STM based on Java's Remote Method Invocation (RMI) mechanism. We are motivated by the popularity of the Java language, and the need for building distributed systems with concurrency control, using the control flow model. Support for distributed computing in Java is provided using RMI since release 1.1. However, distributed concurrency control is (implicitly) provided using locks. Besides, the RMI architecture lacks the transparency required for distributed programming. We present *Snake D-STM*, an RMI/D-STM implementation that uses D-STM for distributed concurrency control in (RMI's) control flow model, and exports a simpler programming model with transparent object access. Distributed atomicity, object registration, and remote method declarations are handled transparently using instrumentation engine HyIE 3.3.1 without any changes to the underlying virtual machine or compiler.

4.2.1 Snake D-STM

4.2.1.1 Programming Model

The Java RMI specifications require defining a `Remote` interface for each remotely accessible class, and modifying class signatures to throw *remote* exceptions. Server side should register the implementation class, while client uses a delegator object that implements the desired `Remote` interface.

In our model, a programmer annotates remotely accessible methods with the `@Remote` annotation, and critical sections are defined as methods annotated with `@Atomic`. An object that contains at least one `@Remote` method is named *remote object*, and it must implement the `IDistinguishable` interface to provide our registry with a unique object identifier. Remote objects register themselves automatically at construction time, and are populated to other node registries. A transactional object is one that defines one (or more) `@Atomic` methods. Atomic annotation can be, optionally, parametrized by the maximum number of transactional retries.

Transactional or remote objects are accessed using *locators*. Traditional object references cannot be used in a distributed environment. Further, locators monitor object accesses and

```
1  public class SearchAgent implements IDistinguishable {
2   ....
3   @Override
4   public Object getId() {
5       return id;
6     }
7
8     @Remote  @Atomic{retries=10}
9     public Object transfer(URI name) {
10      ....
11    }
12
13    @Remote
14    public URI find(String keyword) {
15      ....
16    }
17
18    @Atomic
19    public static Object search(String keyword) {
20      for(String tracker: trackers){
21        SearchAgent agent = Locator.open(tracker);
22        URI url = agent.find(keyword);
23        if(url!=null)
24            return agent.transfer(url);
25        return null;
26      }
27    }
28  }
```

Figure 4.3: A P2P agent using an atomic remote TM method.

act as early detectors for possible transactional conflicts. Objects can be located (or opened) in read-only or read-write modes. This classification permits concurrent access for concurrent read transactions.

Figure 4.3 shows a distributed transactional code example. A peer-to-peer (P2P) file sharing agent atomically searches for resources and transfers them to the caller node. The agent may act recursively and propagate the call to a set of neighbor agents.

4.2.1.2 Layered Overview

Figure 4.4 shows a layered architecture of our implementation. Similar to the official RMI design, we have the three layers of: 1) *Transport Layer*, where actual networking and communication handling is performed, 2) *Remote Reference Layer*, which is responsible for managing the "liveliness" of the remote objects, and 3) *Stub/Skeleton Layer*, which is responsible for managing the remote object interface between hosts. Additionally, we define an *Object*

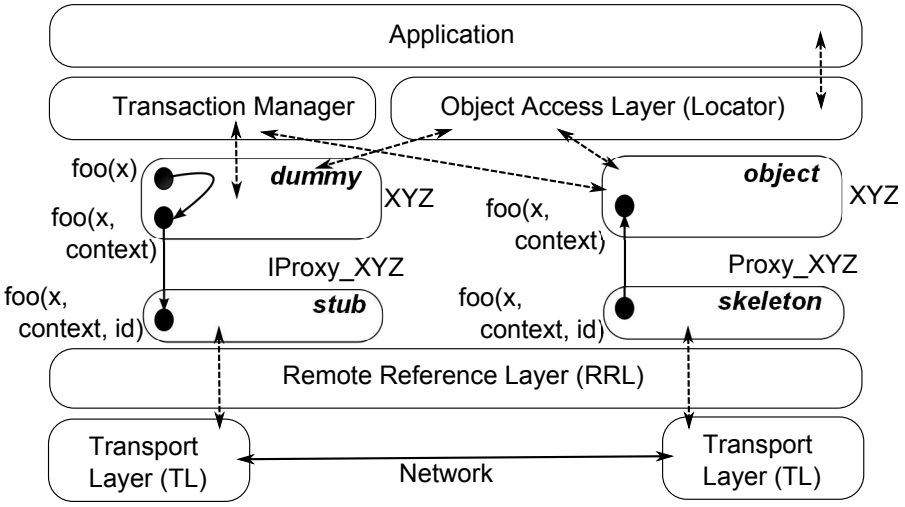


Figure 4.4: Snake D-STM layered architecture overview.

Access Layer, which provides the required transparency to the application layer. Local and remote objects are accessed in an uniform manner, and a dummy object is created to delegate calls to the RMI stub. Transactional code is maintained by a *Transaction Manager* module, which provides distributed atomicity and memory consistency for applications. As described in Section 3.3.1, an *Instrumentation Engine* is responsible for load-time code modifications, which is required for the Transaction Manager and Object Access Layer.

4.2.1.3 Mobile Transactions

Supporting shared memory-like access in distributed systems requires an additional level of indirection. Each transaction must preserve memory consistency, and must expose its local changes instantaneously. In order to do that, old or new values of modified objects must be stored at local-transaction buffers till commit time. Two strategies can be used to achieve this: i) *undo-log* [77, 85], where changes are made to the main object, while old values are stored in a separate log; and ii) *write-buffer* [72, 73, 53], where changes are made to transaction-local memory and written to the main object at commit time. Both strategies are applicable in the distributed context. However, (distributed) transactions cannot move between nodes during their execution with all these metadata (undo-logs or write-buffers) due to high communication costs. Instead, transaction metadata must be detached from the transaction context, while keeping the minimal information mobile with the transaction. In Snake D-STM, we implemented both approaches. Using a distributed mechanism for storing transaction read-set and write-set, distributed transactions are managed with minimum amount of mobile data (e.g. transaction id, priority).

Before (and after) accessing any transactional object field, the transaction is consulted for read (or written) value. A transaction builds up its write and read sets, and handles any

private buffers accordingly. At commit time, a distributed validation step is required to guarantee consistent memory view. In this phase, transaction originator nodes trigger a voting request to the participating nodes. Each node uses its portion of write and read sets to make its local decision. If validation succeeds on all nodes, the transaction is committed; otherwise, an abort handler rolls-back the changes. During the validation phase, the transaction state is set to *busy*, which ensures that a transaction cannot be aborted. This helps in ensuring the correctness of the validation (i.e., all nodes unanimously agree on the transaction to be committed and the transactions to be aborted), and also, it prevents transactions at later stages from being aborted by newly started ones.

4.2.1.4 Distributed Control-Flow Contention Management

Two transactions conflict if they concurrently access the same object, and one of them is a write transaction. Upon detecting a conflict, a *contention management policy* (CP) is used to resolve this situation (arbitrarily or priority-based) e.g., one of the transactions is stalled or aborted and retried. A wide range of transaction contention management policies has been studied for non-distributed STM [95, 94]. We classify CPs into three categories: 1. *Incremental CP* (e.g., Karma, Eruption, Polka), where the CP builds up the priorities of the transactions during transaction execution; 2. *Progressive CP* (e.g., Kindergarten, Priority, Timestamp, Polite), which ensures a system-wide progress guarantee (i.e., at least one transaction will proceed to commit); and 3. *Non-Progressive CP* (e.g., Backoff, Aggressive), which assumes that conflicting transactions will eventually complete, however, livelock situations can occur.

As mentioned earlier, in the control flow model, a distributed transaction T_x is executed over multiple nodes. Under Incremental CP, T_x can have different priorities at each node. This is because, a transaction builds its priority during its execution over multiple nodes. Under this behavior, a live-lock situation can occur. Consider transactions T_x and T_y with priorities P_x, P'_y and P'_x, and P_y at nodes N_1 and N_2, respectively. It is clear that, if $P'_x > P_y$ and $P'_y > P_x$, then both transactions will abort each other, and this will continue forever. The lack of a central store for transactional priorities causes this problem. However, having such a central store will significantly increase the communication overhead during transaction execution, causing a system bottleneck. Non-Progressive CP shows comparable performance for non distributed STM [8]. Nevertheless, our experiments show that it cannot be extended for D-STM due to the expensive cost of retries (see Section 6).

4.2.1.5 Voting Manager

A mobile transaction executes over multiple nodes through the access propagation of objects. Call graph, which is composed of nodes (sub-transactions) and undirected edges (calls), is essential for making a commit decision. Each participating node may have a different

decision (on which transaction to abort/commit) based on conflicts with other concurrent transactions. In this section we overview the key idea of D2PC, our default implementation for voting manager, in the context of transactional memory (more information is available in [83]).

In D2PC, the originating node, which started the transaction, sends a *PREPARE* message to its neighbors, containing the transaction identifier. Each node forwards this message to its neighbors in the call graph except its parent. If a node receives the *PREPARE* message again, it discards it. The number of messages sent is the number of edges in the call graph. Each node consults its *Contention Manager* for committing the requested transaction. The message propagation results in the construction of a spanning tree of the *call graph*: each node remembers its parent and the children nodes to which it propagates its message to. D2PC doesn't distinguish between parent and children nodes; it treats them equally as "neighbors." It worth noting that this simple construction for a spanning tree is better than trying to obtain the minimum spanning tree (MST). The problem of finding MST has well known algorithms such as Prim and Kruskal's algorithms [26], also using parallel distributed algorithms [37, 12, 38, 66]. Although the best of this algorithms has $O(\sqrt{|N|}log^*|N| + D)$, where D is the network diameter, it involves sending many number of messages to construct MST. In our case, the voting protocol involve broadcasting three messages at maximum so it is impractical to construct MST. In addition to that the spanning tree is changing according to the call-graph which is constructed per each transaction, so we cannot rely on pre-generated MST.

Now, assume that one or more nodes decide to send an *ABORT* message. The nodes will forward the message to their neighbors, which in turn will recursively forward to theirs, except the sender. Sub-transactions will be terminated and the originating transaction will be retried. However, in the success case, all nodes will send a *COMMIT* message. Upon receiving a "COMMIT" message from all its neighbors, including its parent, except the last, a node forwards the commit decision to the last neighbor. It can be shown that there will be just one node that will receive all the votes, and this node is selected dynamically based on message speed and nodal delays (Lemma 4.2.1 proves this argument). This node will be elected as a *coordinator* for the current vote. Similar to the failure case, the coordinator populates the *COMMIT* decision to others, commits the distributed changes, and the transaction completes.

Lemma 4.2.1. *Under D2PC there will be one and only one node that will receive all the sent votes, assuming no partitions exist in the network.*

Proof. The proof is by contradiction. We will assume that the protocol ends with two nodes that work as coordinators. Thus, each of these nodes must receive replies from *all* their neighbors, which implies that one node sent its votes to two of its neighbors. However, according to the described protocol, any node will send its collected votes to just *one* node (the last neighbor that did not reply). This contradicts the initial assumption, implying that there is exactly one coordinator at the end of the election protocol. □

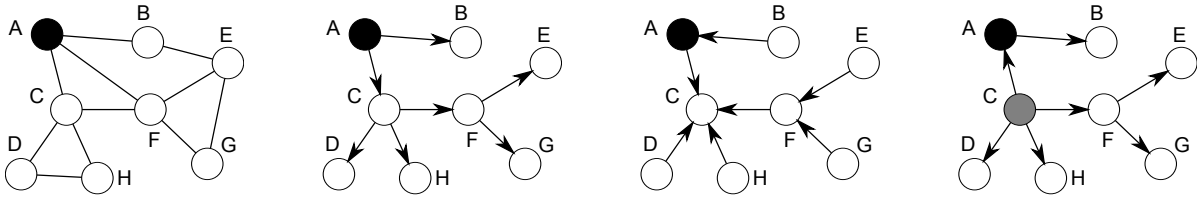


Figure 4.5: (a) Call graph: each node represents a sub-transaction, and edges denote remote calls between them. (b) Originating node A sends a PREPARE message that is forwarded to other nodes. (c) The vote is replied, and the coordinator is selected through the process of last-neighbor forwarding. (d) Coordinator node C publishes the vote result

Figure 4.5 shows a possible execution of D2PC for collecting votes for a transaction distributed over seven nodes.

Chapter 5

Analysis

We now illustrate the factors of communication and processing overhead through a comparison between control flow and dataflow D-STM models. A compromise between the two models can be used to design a hybrid D-STM model. This chapter is complementry of Chapter 6, in which we present a comprehensive experiments on each of the two models: dataflow and control-flow, while our analysis forms the basis for creating a hybrid model.

5.1 Dataflow Model

In the dataflow model, transactions are immobile, and objects move through the network. To estimate the transaction cost under this model, we state the following theorem. For simplicity, we consider a single transactional execution.

Theorem 5.1.1. *The communication cost for a transaction T_i running on node N_S and accessing k remote objects O_j, $1 \leq j \leq k$, using the dataflow model is given by:* $DF_{cost}(T_i) = \sum_{1<j<k}[[Size(O_j)+\Pi(T_i,O_j)]*c(N_S,Own(O_j))+\lambda+\beta*(1-\Pi(T_i,O_j))]$. *Here, λ is the lookup cost, β is the directory update cost, and Π is a function that returns 1 if the transaction accesses the object for read-only and 0 for read-write operations.*

Proof. To execute a transaction using the dataflow model, three steps must be done for each remote object: locate the object, move the object, and validate or own the object.

There has been significant research efforts on object lookup (or discovery) protocols in distributed systems. Object (or service) lookup protocols can be either directory-based [10, 101, 102] or directory-less [1, 78, 50], according to the existence of a centralized directory that maintains the locations of services/resources. There could be one or more directories in the network or a single directory that is fully distributed.

Directory-based architectures suffer from scalability and single points-of-failure problems. Directory-less protocols can be classified as *push protocols* or *pull protocols*. In a push protocol, a node advertises its object to other nodes, and is responsible for updating other nodes with any changes to the provided object. In a pull protocol, when a node needs to access an object, it broadcasts a discover request to find the object provider. Usually, caching of object locations is exploited to avoid storms of discover requests. We denote the cost for object location by λ, which may differ according to the underlying protocol.

The cost for moving an object to the current node is proportional to the object size and the total communication cost between the current node and the object owner.

At commit time, a transaction needs to validate a read object, or obtain the ownership of the object, and thus will need to update the directory. β is an implementation-specific constant that represents the cost to update the directory. It may vary from a single message cost as in Arrow and Relay directory protocols [29, 105], logarithmic in the size of nodes as in the Ballistic protocol [29], or may require message broadcasting over the entire network as in the Home directory protocol [57]. □

5.2 Control Flow Model

In contrast to the dataflow model, in the control flow model, a transaction can be viewed as being composed of a set of sub-transactions. Remote objects remain at their nodes, and are requested to do some operations on behalf of a transaction. Such operations may request other remote objects. For simplicity, we assume that the voting protocol will use a static minimum spanning tree S for sending messages, and the nodes which are not interested in voting will not respond.

Theorem 5.2.1. *The communication cost for a transaction T_i running on a set of nodes $N_{t_i} = \{N_{1t_i}, N_{2t_i}, \ldots, N_{nt_i}\}$, and accessing k remote objects O_j, $1 \leq j \leq k$, using the control flow model is given by:*
$CF_{cost}(T_i) = Voting(N_{t_i}) + \sum_{\substack{1<j<k \\ 1<s<n}} [c(N_{st_i}, Own(O_j)) * Calls(T_i, O_j) * \Theta(N_{st_i}, O_j)]$. *Here, $Calls$ is the number of method calls per object in transaction T_i, $Voting$ is the cost of collecting votes of a given set of nodes, and Θ is a function that returns 1 if a node needs to access a remote object during its execution, and 0 otherwise.*

Proof. Let us divide the distributed transaction T_i into a set of sub-transactions. Each sub-transaction is executed on a node, and during the sub-transaction, the node can access one (or more) remote object(s). The communication cost per each sub-transaction is the cost for accessing remote objects using remote calls. Each remote call requires a round-trip message for the request and its response. The total communication cost per node, is the sum of the costs of all sub-transactions which run on the node.

The second term of the equation of the theorem shows the aggregate cost for all nodes involved in executing the distributed transaction. The D2PC voting protocol [83] needs to broadcast at most three messages, one for requesting the votes, one for collecting the votes, and one for publishing the result of the global decision—i.e., $Voting(N_{t_i}) \leq 3 * \Omega$. □

5.3 Tradeoff

The previous two sections show that there is a trade-off between using the dataflow or control flow model for D-STM, which hinges on the number of messages and the size of objects transferred. The definition of functions $Calls$, Θ, and Π could be obtained either by code-analysis that identifies *object-object* relationships for objects defined as shared ones, or by transaction profiling based on past retries. The $Own(O)$ function is implemented as one of the functions of the Directory Manager, while λ, β, and $Voting$ functions are implementation-specific according to underlying protocols. Under a network with stable topology conditions, we can define a fixed communication cost metric.

Another factor that affects the communication cost is the locality of objects. Exploiting locality can reduce network traffic and thereby enhance D-STM performance. Consider an object, which is accessed by a group of geographically-close nodes, but far from the object's current location. Sending several remote requests to the object can introduce high network traffic, causing a communication bottleneck. The following two lemmas show the cost of moving an object versus sending remote requests to it.

Lemma 5.3.1. *The communication cost introduced by relocating an object O_j to a new node N_i is given by:*
$Reloc_{cost}(O_j, N_i) = \beta + Size(O_j) * c(N_i, Own(O_j))$.

Proof. Object relocation or duplication recquires; i) updating the objects directory (β), and ii) moving or copying the object over the network, which is proportional to the object size and depends on the link costs between the source and distination nodes. □

Lemma 5.3.2. *For a distributed transaction in the control flow model, the communication cost for sending a remote request to an object O_j is given by:*
$Msg_{cost}(O_j, Own(O_j)) = \sum_{1<s<n}[c(N_{st_i}, Own(O_j)) * \Theta(N_{st_i}, O_j)]$.

Proof. As distributed trabsaction can run over multiple nodes, a remote object may be accessed by some of the participating nodes. We formulate this using the Θ funtion, and aggregate the cost over all the participating nodes. □

We conclude that even in the control flow model, object relocation may be beneficial. For any object O_j accessed by some transaction running on a set of nodes N_{t_i} under the control flow model, if $Msg_{cost}(O_j, Own(O_j)) > Msg_{cost}(O_j, N_i) + Reloc_{cost}(O_j, N_i)$, then the object should be moved to node N_i.

Chapter 6

Experimental Results

6.1 Distributed Benchmark

We developed a set of distributed benchmarks to evaluate the usefulness and performance of our implementations against competing models including: 1. classical RMI, which uses mutual exclusion locks (read/write) with random timeout mechanism to handle deadlocks and livelocks; 2. distributed shared memory (DSM), which uses the Home directory protocol such as Jackal [103]; and

Our benchmarks include the following.

Loan Benchmark. Loan is a simple money transfer application, in which a set of asset (i.e., with monetary value) holders is distributed over nodes. A loan transaction is configured by two parameters: 1) branching, and 2) nesting. In a loan transaction, an account is issued a loan request to one or more other accounts (determined by the branching parameter). The request recipient forwards this request to other accounts, and this propagation continues till the level (determined by the nesting parameter) is reached. An entire loan request takes effect atomically, by virtue of the request being implemented as an atomic transaction. The number of inter-node remote object calls grow exponentially with the number of participating objects (i.e., the nesting level). For example, for a single transaction, 20 inter-node calls occur for a nesting level of 6, and 376 inter-node calls occur for a nesting level of 12. This benchmark is interesting for evaluating control-flow D-STM, as it involves significant number of remote calls that grows exponentially with nesting.

Bank Benchmark. This is a distributed banking application, which maintains a set of accounts distributed over bank branches. The application contains two atomic transactions: a) *transfer transaction*, which transfers a given amount between two accounts, and b) *total balance transaction*, which computes the total balance for given accounts. A combination of these two transactions can be used to measure the performance of distributed concurrency

control models. The object size and the number of operations per object during a transaction is configurable.

Vacation Benchmark. This is a distributed version of the STAMP [24] STM vacation benchmark application. This application implements an online transaction processing system that tracks customer reservations for air travel. The distributed object here is the customer and trip records. The system supports three types of transactions: reservation, cancelation, and update. These transactions are read/write, as they change the object. However, another administrative transaction, which simply prints-out all current reservations, is provided to assess read-only transactions.

File Sharing Benchmark. This is a P2P file sharing agent (see Figure 4.3), which searches shared file contents by keywords, atomically transfers data from other remote agents, and updates some application-specific data. DSM and dataflow D-STM cannot be used with this benchmark, as a remote agent must search files at its node. This is an example of immobile objects with system-state property.

Micro Benchmarks. A novel implementation of a distributed linked list and a distributed binary search tree. Nodes are located at different nodes, while links between nodes are replaced by keys of neighbor nodes. Element manipulation operations are defined as transactions—e.g., *add*, *remove*, and *contains*. The RMI version of these micro benchmarks uses fine-grained lock-coupling implementation [15] that acquires locks in a kind of "hand-over-hand" order.

6.2 Testbed

We conducted our experiments on a multiprocessor/multicomputer network comprising of 120 nodes, each of which is an Intel Xeon 1.9GHz processor, running Ubuntu Linux, and interconnected by a network with $1ms$ end-to-end delay. We ran the experiments using one processor per node, because we found that this configuration was necessary to obtain stable runtimes. This configuration also generated faster runtimes than using multiple processors per node, because it eliminated resource contention between two processors on one node.

6.3 Evaluation

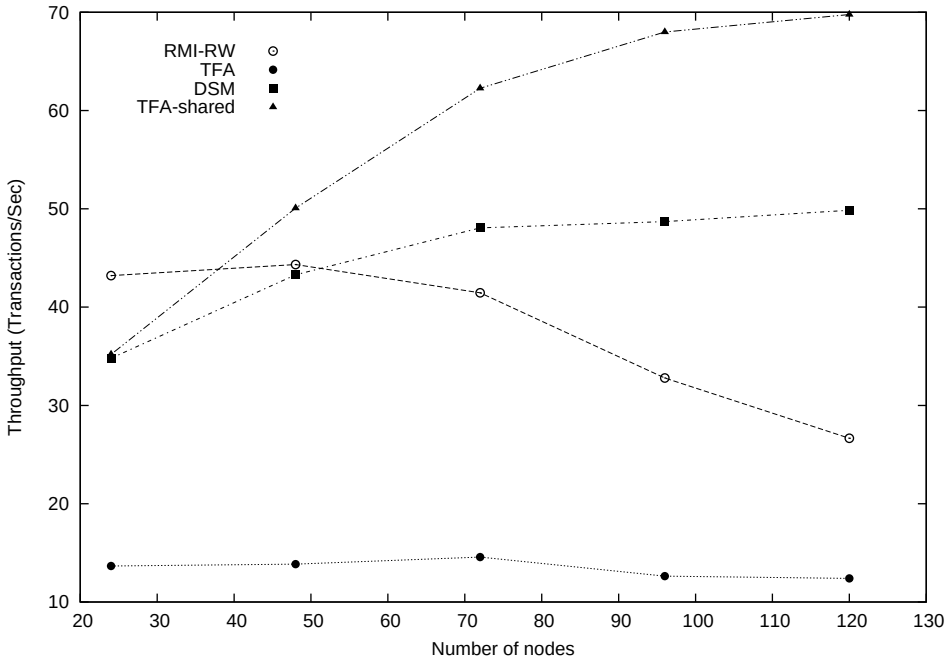
6.3.1 Microbenchmark

We evaluated the throughput of concurrent nodes which access the same data structure, using distributed linked list and binary search tree implementations. The number of concurrent nodes varies from 24 nodes to 120 nodes, every node running a single process invokes a set of 200 sequential transactions. Four linked list implementations were conducted. The first two uses transactional memory for defining data structure operations as transactions. One of them uses normal "read"/"write" object access, while the other uses "shared" object access with promoting objects to "read" or "write" access whenever needed. The other two implementations are a fine-grained, handcrafted lock-coupling implementations [15], which acquires locks in a "hand-over-hand" manner. That was implemented using RMI with read/write locks, and using DSM with Home directory manager.

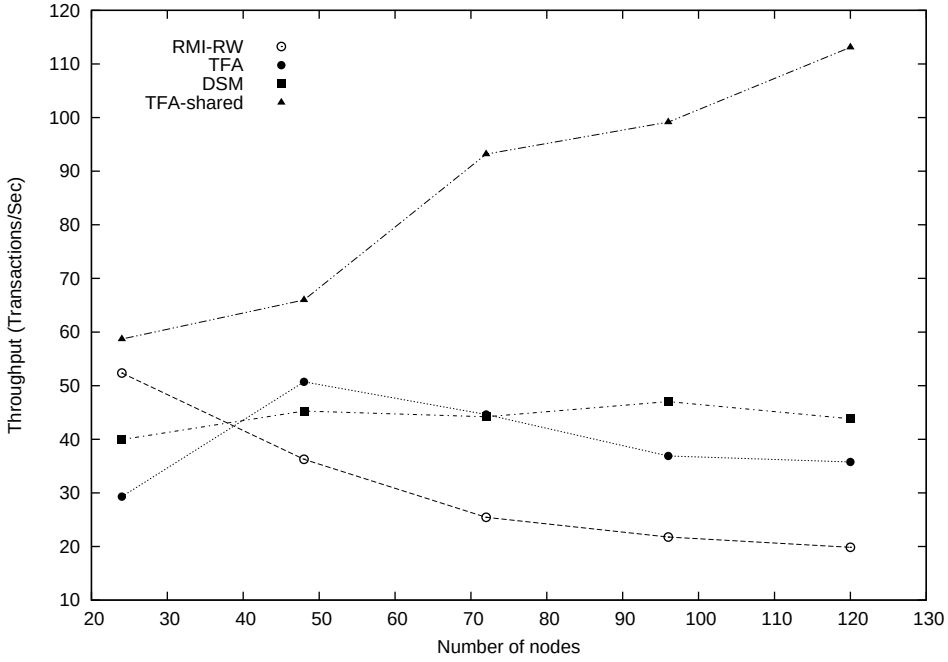
Figures 6.1 and 6.2 the nodal throughput with increasing contention by adding more concurrent nodes and using different schemes at 50% and 90% read-only transactions (`contains()` opeartion).

From Figure 6.1 we observe that the shared access TM implementation of linked list outperforms all other approaches (except at 24 nodes, 50% reads), while the normal TM implementation has the lowest throughput. That is due to the large number of elements added to the read set which increase the probability of conflicts, especially with `add()` that changes in the head node, and make the validation step at commit time very expensive. Traversing objects as "shared" allows TFA algorithm to validate object at access time and doesn't add them to the read-set (except promoted elements to read/write access). It worthnoting that this optimization works only for algorithms that guarantees opacity. Notice that the performance gain, due to using TM implementation, in Figure 6.1(b) is much higher than Figure 6.1(a) because the write-transactions increase the probability of conflict and hence the number of retries increases.

Similarly, at Figure 6.2 that the shared access TM implementation of binary search tree is the best. However the other TM implementation also gives comparable performance to RMI and DSM, the reason is the lower probability of conflicts due to the branching property of tree.

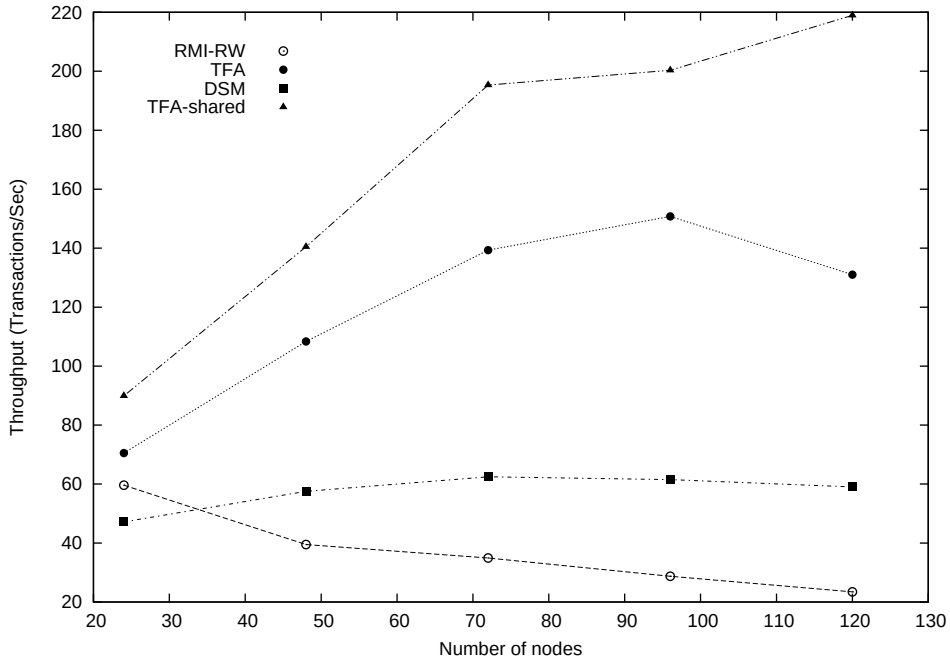


(a) 50% reads.

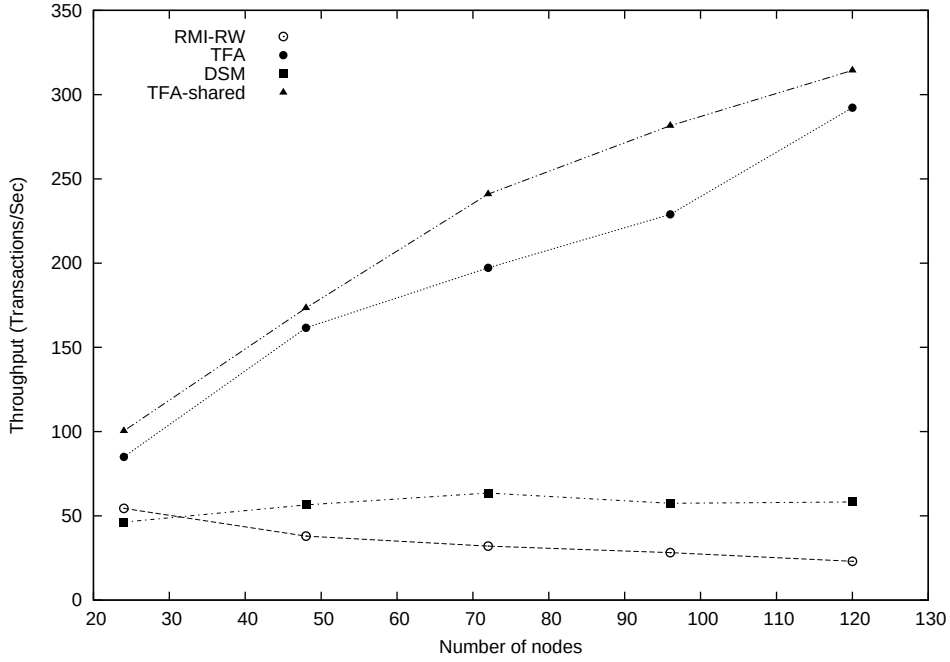


(b) 90% reads.

Figure 6.1: Distributed Linked List Micro-benchmark



(a) 50% reads.



(b) 90% reads.

Figure 6.2: Distributed Binary Search Tree Micro-benchmark

6.3.2 Benchmarks

6.3.2.1 Vacation

Our first experiment uses a distributed version of Vacation benchmark [24] at different schemes (low and high contention), where the number of objects at *high* scheme were reduced to half the ones used at *low* contention scheme. Every transaction makes ten operations that differs from; price comparisons to get the best offer, or deleting customer or updating offers.

Figures 6.3 shows the throughput under increasing number of nodes, which increases contention (with all else being equal). The confidence intervals of the data-points of the figure are in the 5% range.

From Figures 6.3(a) and 6.3(b) we observe that TFA outperforms the best of other distributed concurrency control models by 60-120% at 120 nodes. In contrast to RMI and DSM, TFA is scalable and provides linear throughput at large number of nodes.

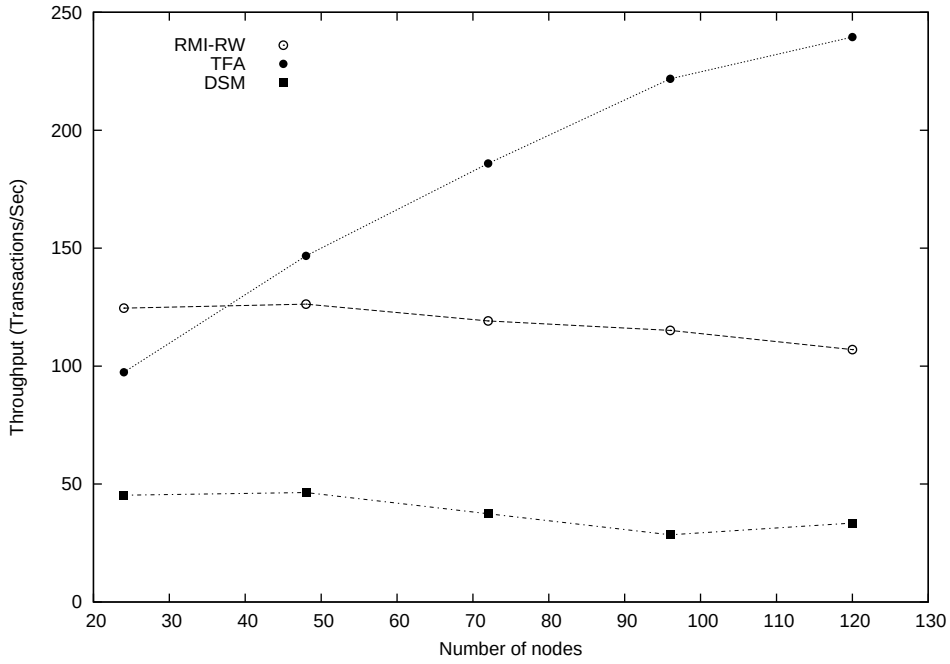
6.3.2.2 Bank

At the Bank benchmark, accounts were distributed over nodes (which represents bank branches), and every node invokes eight concurrent transactions (either checking balance or transfer operations). The number of concurrent nodes (or branches) were increased to increase contention over the number objects which is the same during the whole experiment. The number of concurrent transactions at the end of our experiment reaches a thousand concurrent at a time.

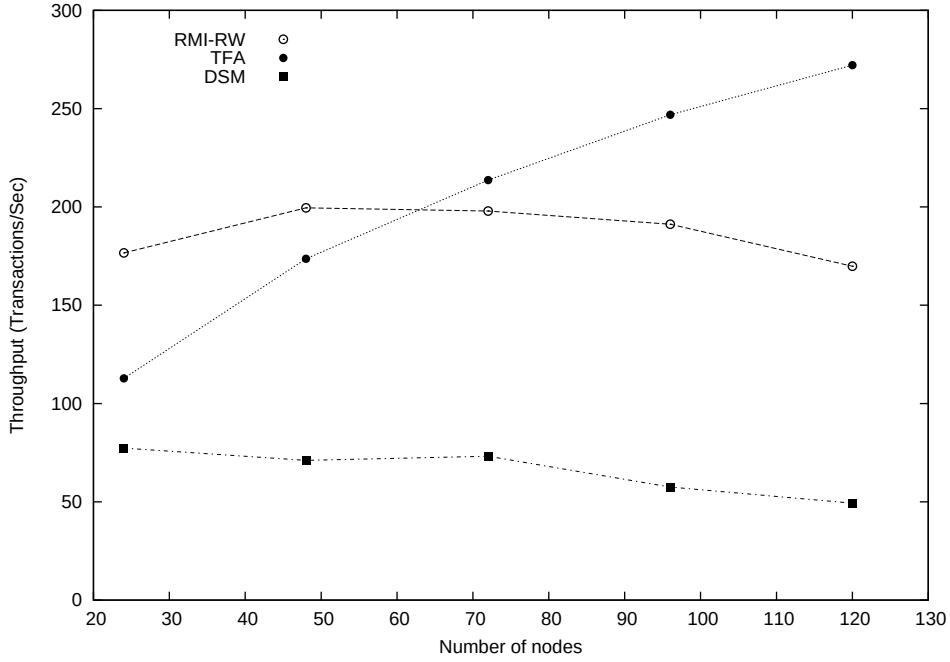
What Figures 6.4 shows is how the throughput will be affected at different schemes (at 50% and 90% read-only transactions, respectively) under increasing number of nodes. Again, we notice from From Figures 6.4(a) and 6.4(b) that TFA outperforms other distributed concurrency control models at large number of nodes. Further, from Figure 6.5, we observe the excellent scalability of HyFlow/TFA under high contention (72 nodes, more than 500 concurrent transactions).

It important to mention that the high throughput of RMI comes because the number of calls per object at this experiment was just one per object, which is not the case in most real life applications. Figure 6.6 shows the performance degradation of RMI based implementations with increasing the number of calls per object. In dataflow TM implementations, the number of calls is insignificant as calls occur at the local copy of the object.

Figure 6.6 demonstrates the effect of not employing locality of reference: in the control flow model, each remote call incurs a round-trip network delay. As shown in the figure, it reduces throughput by 25% for four to eight calls. This should be considered in environments with high link latency.

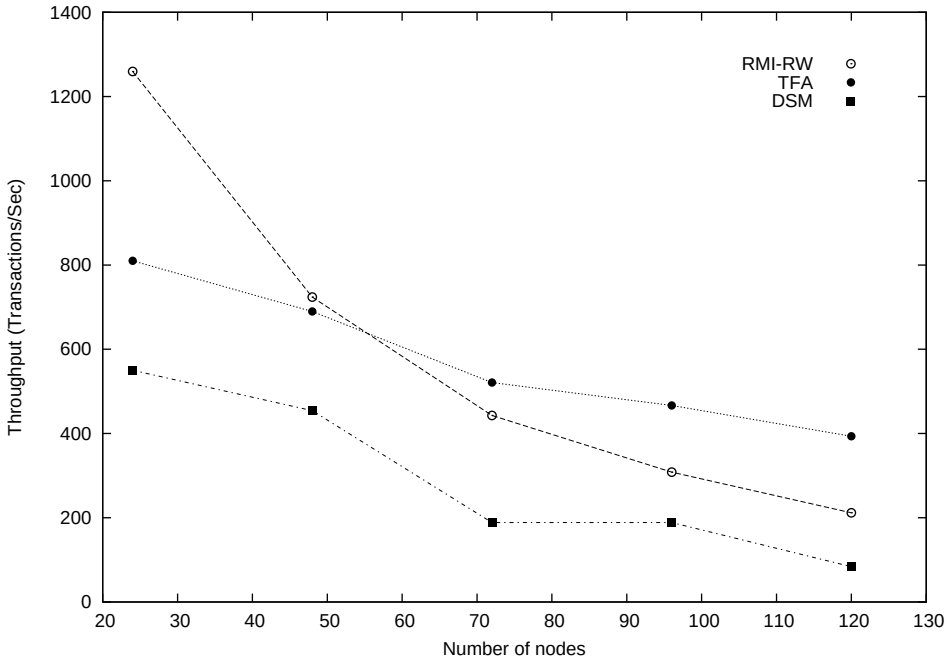


(a) High Contention.

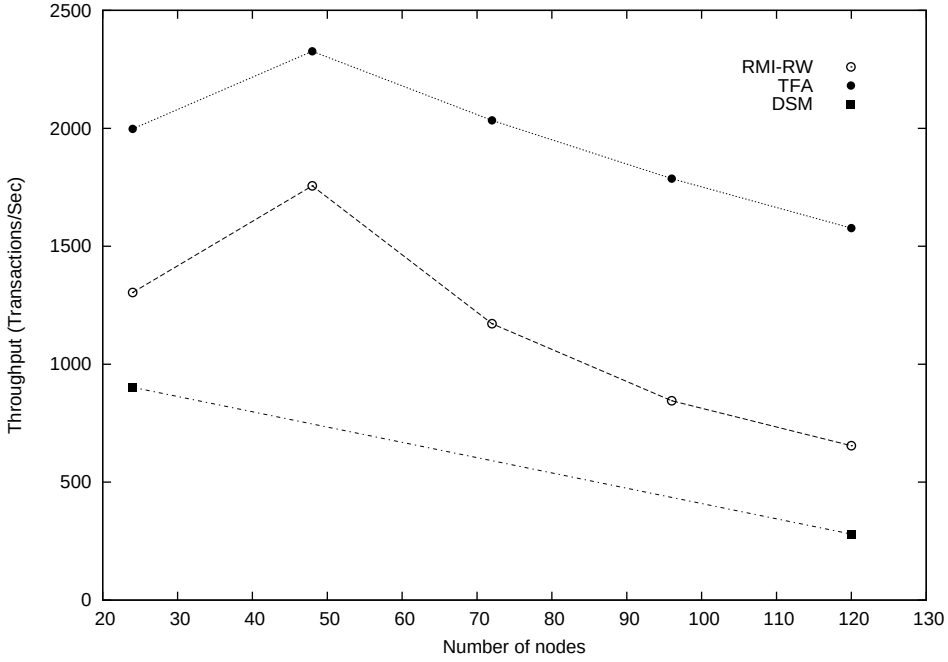


(b) Low Contention.

Figure 6.3: Vacation Benchmark

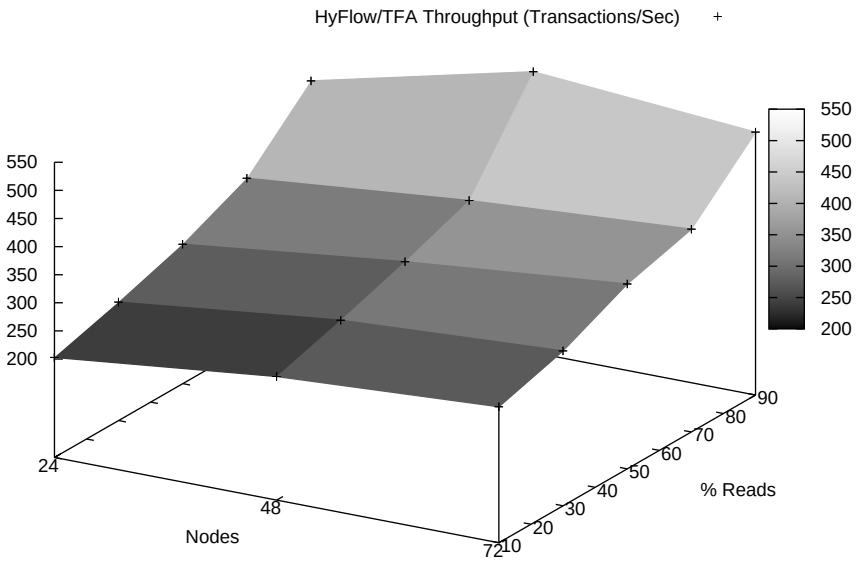


(a) 50% reads.

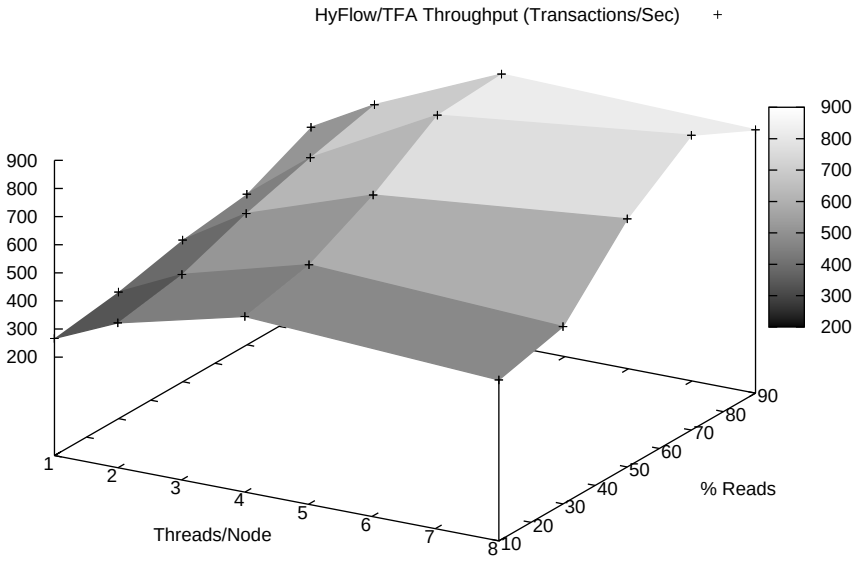


(b) 90% reads.

Figure 6.4: Bank Benchmark



(a) Throughput under increasing number of nodes.



(b) Throughput under increasing number of threads per node.

Figure 6.5: Scalability of HyFlow/TFA.

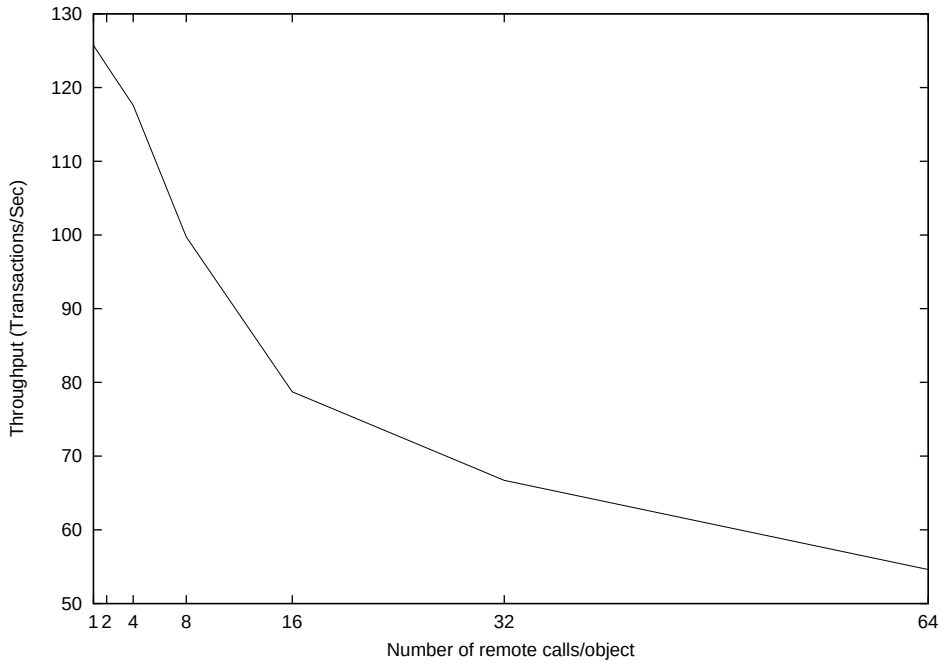


Figure 6.6: Bank benchmark: RMI throughput under increasing number of calls per object.

6.3.2.3 Loan

Using Loan benchmark, with nesting level of six that accesses six objects producing 20 inter-object calls, we evaluated the performance of our implementations dataflow and control-flow. Figure 6.7 shows the throughput of transactions at different number of nodes. Like our previous experiment, eight concurrent transactions were invoked by every node, and the number of nodes changes from 24 to 120 nodes.

From Figure 6.7 we observe that RMI performance degrades dramatically with increasing objects contention due to the need to acquire lock over six objects. Also, TFA was affected by the relatively larger read set or write set that outweigh the performance gain from the algorithm. However, it outperforms other models at high contention situations.

Figure 6.9 shows the throughput under increasing number of participating objects in each transaction (transaction execution time under no contention is $350ms$ in this experiment). Greater the number of accessed objects, higher the algorithm overhead, and higher the number of remote calls per each transaction (e.g., a transaction of twelve objects issues 376 remote object calls during its execution). In Figure 6.10, the effect of progressive contention management policies is shown. Six shared objects for the Loan benchmark (and two for the Bank benchmark) were accessed using twelve nodes issuing concurrent transactions. To increase contention, we forced every transaction to access all shared objects during its execution.

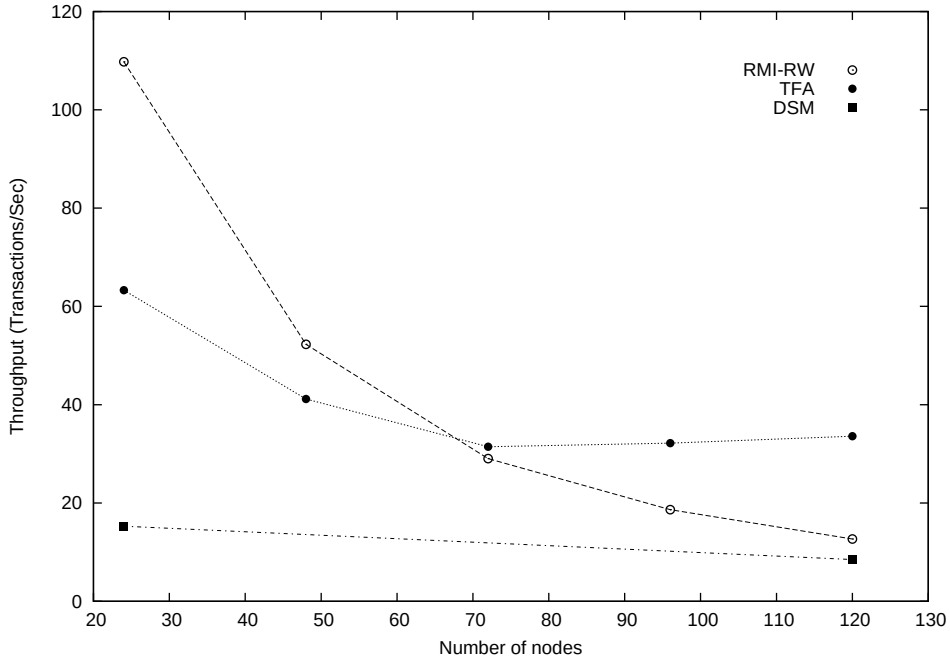


Figure 6.7: Loan Benchmark

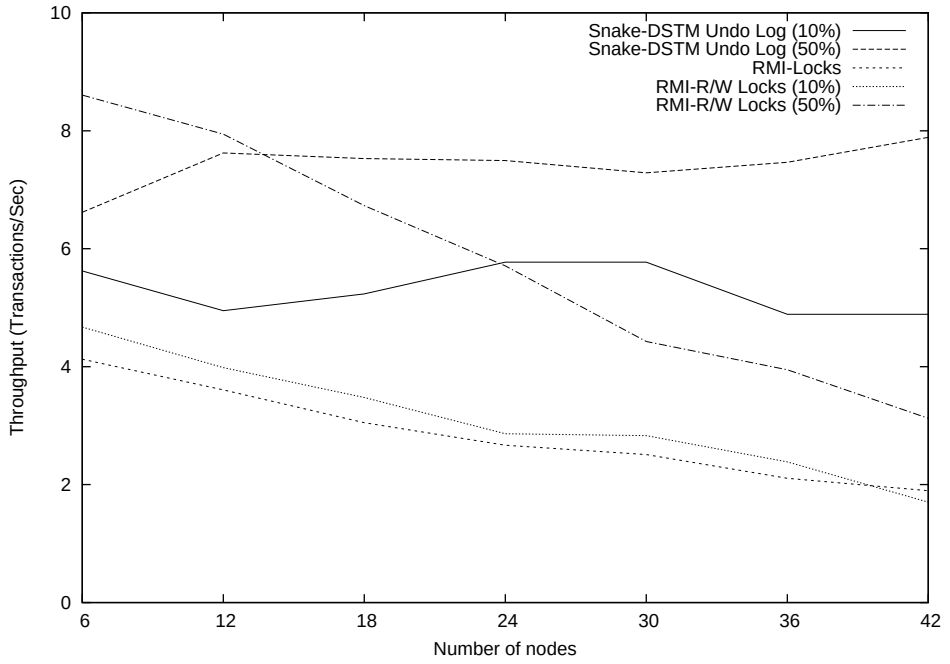


Figure 6.8: Throughput of Loan benchmark under increasing number of nodes.

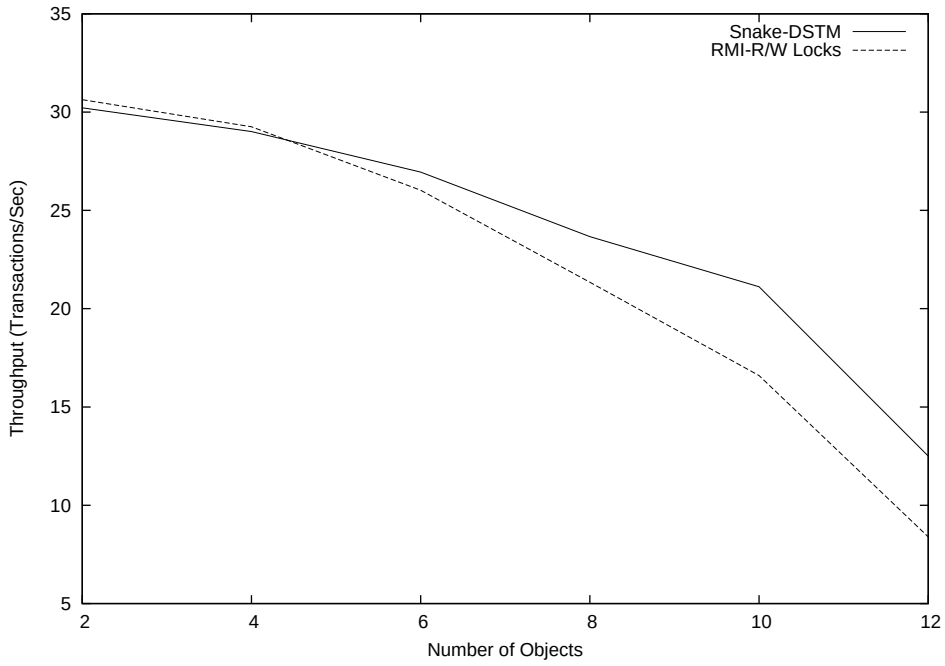


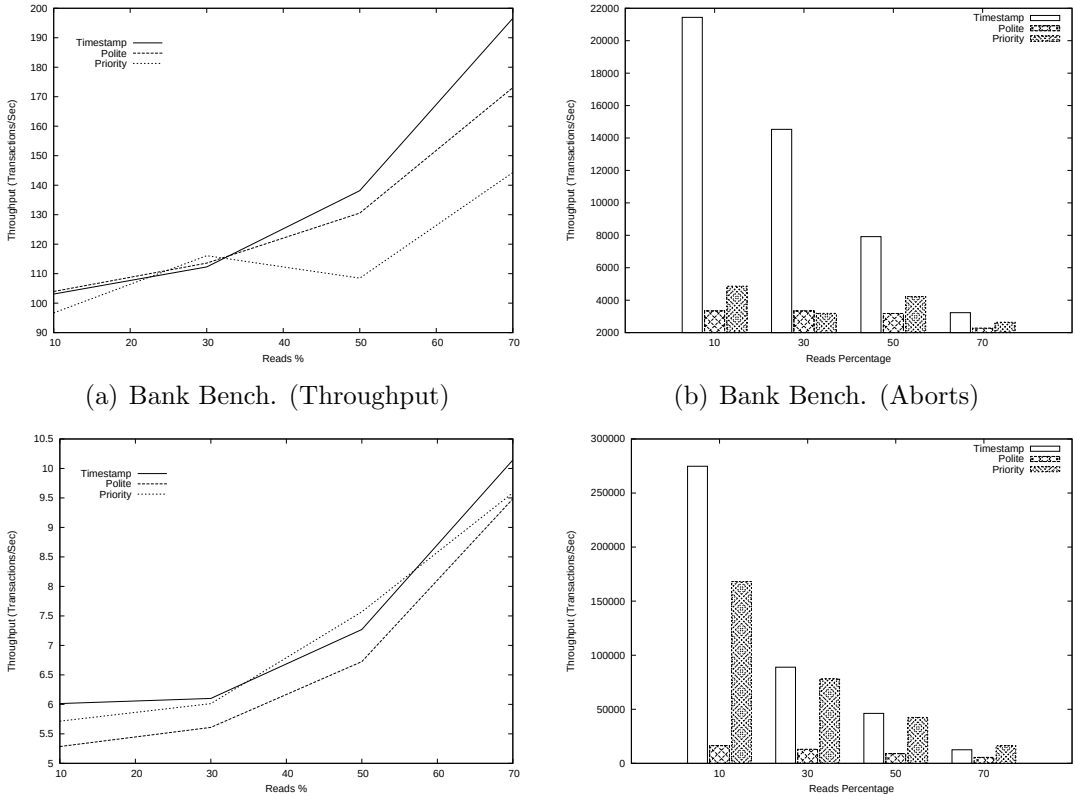
Figure 6.9: Throughput of Loan benchmark using pure read transactions over 12 nodes, and variable object count per transaction.

In Figure 6.11, we show the scalability of Snake-DSTM using File Sharing Benchmark (P2P search agent with two trackers) running over 72 nodes.

From Figure 6.8, we observe that Snake-DSTM outperforms classical RMI using mutual exclusion locks (RMI-Locks), and also using read/write locks (RMI-R/W), by 180% at high contention (10% reads), and by 150% at normal contention (50% reads). Though RMI with read/write locks shows better performance at a single point (6 nodes) due to the voting protocol overhead, yet, it suffers from performance degradation at increasing loads.

Figure 6.9 uses the no-contention situation (100% reads) to compare the overhead of Snake-DSTM and RMI-R/W. At small number of shared objects per transaction, the TM overhead outweighs the provided concurrency, and both Snake-DSTM and RMI-R/W incur the same overhead. With increasing number of objects, Snake-DSTM outperforms RMI-R/W by 50%.

As illustrated in Section 4.2.1.4, progressive contention management policies are the most suitable CP for distributed environments. From Figure 6.10, we observe the effect of CPs on throughput under the Loan and Bank benchmarks. The Timestamp CP performs better than the Polite and Priority CPs, but it results in the highest abort rate, and thus incurs more processing overhead. The Polite back-off mechanism with retries manages to significantly reduce aborts (7-14 times less), while yielding moderate throughput. The Static Priority CP gives the worst performance. Besides, it suffers from starvation situations for low priority transactions.



(a) Bank Bench. (Throughput)

(b) Bank Bench. (Aborts)

(c) Loan Bench. (Throughput)

(d) Loan Bench. (Aborts)

Figure 6.10: Throughput and number of aborts of Loan and Bank benchmarks under different progressive contention management policies.

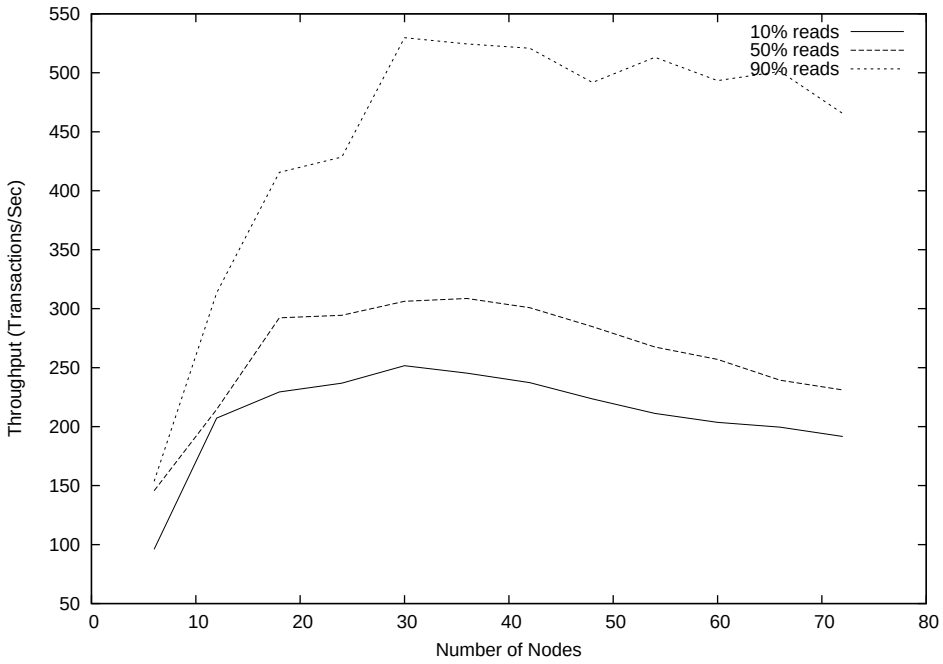


Figure 6.11: Throughput of File Sharing (P2P Agent) benchmark.

Figure 6.11 illustrates the scalability of our model for P2P File Sharing benchmark running over the 72-node system. Other distributed programing models such as DSM or dataflow D-STM cannot be used in such applications, as an agent must search files at its node (immobile object). Besides, code simplicity and maintainability are additional features provided by our approach.

Chapter 7

Conclusions

We presented HyFlow, a high performance, pluggable, distributed STM that supports both dataflow and control flow distributed transactional execution. Our experiments show that HyFlow outperforms other distributed concurrency control models, with acceptable number of messages and low network traffic.

We designed the TFA algorithm for supporting the dataflow D-STM execution model. 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 HyFlow and conduct experimental studies. Our experimental evaluations reveal that TFA outperforms competitors by as much as 70%-600%

We also developed the Snake DSTM, a control-flow D-STM implementation. Snake DSTM is based on RMI as a mechanism for handling remote calls, and defines critical sections as atomic transactions, in which reads and writes to shared, local and remote objects appear to take effect instantaneously. Snake DSTM's default implementation uses the D2PC protocol to coordinate the voting process among participating nodes toward making distributed transactional commit decisions. We show that Snake DSTM outperforms classical RMI with 150% throuput gain.

Our experiments demonstrate that the dataflow model scales well with increasing number of calls per object, as it permits remote objects to move toward geographically-closer nodes that access them frequently, reducing communication costs. Control flow is beneficial under infrequent object calls or calls to objects with large sizes. Our implementation shows that D-STM, in general, provides comparable performance to classical distributed concurrency control models, and exports a simpler programming interface, while avoiding dataraces, deadlocks, and livelocks.

HyFlow provides a testbed for the research community to design, implement, and evaluate algorithms for D-STM. HyFlow is publicly available at hyflow.org.

7.1 Future Work

Despite the advantages of D-STM over other models, it is not a silver bullet. Some of its limitations include handling irrevocable instructions (e.g., I/O operations), and managing the interaction between transactional and non-transactional code (*weak atomicity*). In addition, several directions exist to further optimize D-STM performance. We enumerate these below.

- Retry overheads in distributed systems can cause significant performance degradation. Transaction scheduling, through either a transaction manager or object access modules, may reduce the number of conflicts and hence retries.
- The multi-versioned objects approach, in which multiple versions of objects can co-exist in the network at the same time, can reduce object contention, provided, transactions are able to observe a consistent snapshot of the memory at a certain time point. Similar to the TFA algorithm, a distributed version of the Lazy Snapshot Algorithm [85] is an interesting direction to explore in this context.
- As shown in our analysis, there is a trade off between control-flow and dataflow D-STM execution models. A hybrid model, with a compilation phase, can automatically analyze runtime object access patterns and selectively use either of the execution models or a combination of them.
- Nested transactions provide programmers with greater flexibility, especially in integrating atomic code with third-party libraries. Supporting open and closed nesting is a challenging problem for distributed transactions due to the possibility of node/link failures and the asynchronous nature of communications.

Bibliography

[1] UPnP Forum: Understanding universal plug and play white paper, 2000.

[2] Partitioned Global Address Space (PGAS), 2003.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, (29), 1996.

[5] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[6] B. S. andAli-Reza Adl-Tabatabai andRichard L. Hudson andChi Cao Minh andBen Hertzberg. McRT-STM: a high performance software transactional memorysystem for a multi-core runtime. In *PPOPP*, pages 187–197, 2006.

[7] 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.

[8] M. Ansari, C. Kotselidis, M. Lujn, C. Kirkham, and I. Watson. Investigating contention management for complex transactional memory benchmarks. In *In Proc. Second Workshop on Programmability Issues for Multi-core Computers (MULTIPROG09*, 2009.

[9] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion project: From data integration to data coordination. In *In: SIGMOD RECORD (2003*, 2003.

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

[11] H. Attiya, V. Gramoli, and A. Milani. COMBINE: An Improved Directory-Based Consistency Protocol. Technical report, EPFL, 2010.

[12] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 230–240, New York, NY, USA, 1987. ACM.

[13] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.*, 18(3):190–205, 1992.

[14] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *In Proceedings of the 35th 8 International Symposium on Computer Architecture*, 2008.

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

[16] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *In PPOPP*, pages 168–176. ACM, 1990.

[17] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical report, Carnegie-Mellon University, 1991.

[18] W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. PPPJ '07, pages 135–144, New York, NY, USA, 2007. ACM.

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

[20] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[21] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 247–258, New York, NY, USA, 2008. ACM.

[22] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Trans. Softw. Eng.*, 17:954–960, September 1991.

[23] J. a. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63:172–185, December 2006.

[24] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[25] V. G. Cerf and R. E. Icahn. A protocol for packet network intercommunication. *SIGCOMM Comput. Commun. Rev.*, 35:71–82, April 2005.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Section 23.2: The algorithms of Kruskal and Prim, 2009.

[27] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC '09: Proc. 15th Pacific Rim International Symposium on Dependable Computing*, nov 2009.

[28] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[29] M. J. Demmer and M. Herlihy. The Arrow distributed directory protocol. In *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*, pages 119–133, London, UK, 1998. Springer-Verlag.

[30] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[31] Dice, D. and Shavit, N. What Really Makes Transactions Faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006.

[32] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[33] 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.

[34] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[35] V. W. Freeh. Dynamically controlling false sharing in distributed shared memory. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 403–, Washington, DC, USA, 1996. IEEE Computer Society.

[36] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. MILLIPEDE: Easy parallel programming in easy parallel programming in available distributed environments.

[37] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[38] J. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 659 –668, 3-5 1993.

[39] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

[40] R. Guerraoui and M. Kapalka. Opacity: A Correctness Condition for Transactional Memory. Technical report, EPFL, 2007.

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

[42] R. Guerraoui and M. Kapalka. Transactional Memory: Glimmer of a Theory. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*. ACM, 2009. Invited Paper.

[43] R. Guo, H. An, R. Dou, M. Cong, Y. Wang, and Q. Li. Logspotm: a scalable thread level speculation model based on transactional memory. In *ACSAC 2008. 13th Asia-Pacific*, pages 1 –8, 2008.

[44] V. Hadzilacos. A theory of reliability in database systems. *J. ACM*, 35:121–145, January 1988.

[45] 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.

[46] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, (38), 2003.

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

[48] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[49] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[50] S. Helal, N. Desai, V. Verma, and C. Lee. Konark - A Service Discovery and Delivery Protocol for Ad-Hoc Networks, 2003.

[51] M. Herlihy. The Aleph Toolkit: Support for scalable distributed shared objects. In *CANPC '99: Proceedings of the Third International Workshop on Network-Based Parallel Computing*, pages 137–149, London, UK, 1999. Springer-Verlag.

[52] M. Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 1–2, New York, NY, USA, 2006. ACM.

[53] 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.

[54] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *In Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, 2003.

[55] 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.

[56] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In *In Proc. International Symposium on Distributed Computing (DISC 2005)*, pages 324–338. Springer, 2005.

[57] M. Herlihy and M. P. Warres. A tale of two directories: implementing distributed shared objects in Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 99–108, New York, NY, USA, 1999. ACM.

[58] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990.

[59] R. Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM.

[60] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60:151–157, November 1996.

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

[62] J. Kim and B. Ravindran. On transactional scheduling in distributed transactional memory systems. In S. Dolev, J. Cobb, M. Fischer, and M. Yung, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin / Heidelberg, 2010.

[63] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[64] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 51–58, Washington, DC, USA, 2008. IEEE Computer Society.

[65] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[66] S. Kutten and D. Peleg. Fast distributed construction of small k-dominating sets and applications. *Journal of Algorithms*, 28(1):40 – 66, 1998.

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

[68] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM*, (7), 1989.

[69] B. Liskov, M. Day, M. Herlihy, P. Johnson, and G. Leavens. Argus reference manual. Technical report, Cambridge University, Cambridge, MA, USA, 1987.

[70] J. Maassen, T. Kielmann, and H. E. Bal. Efficient replicated method invocation in Java. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 88–96, New York, NY, USA, 2000. ACM.

[71] 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.

[72] J. Mankin, D. Kaeli, and J. Ardini. Software transactional memory for multicore embedded systems. *SIGPLAN Not.*, 44(7):90–98, 2009.

[73] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[74] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[75] M. Moir. Practical implementations of non-blocking synchronization primitives. In *In Proc. of 16th PODC*, pages 219–228, 1997.

[76] K. E. Moore. Thread-level transactional memory. In *Wisconsin Industrial Affiliates Meeting*. Oct 2004. Wisconsin Industrial Affiliates Meeting.

[77] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *In Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[78] M. Nidd. Timeliness of service discovery in DEAP space. In *ICPP '00: Proceedings of the 2000 International Workshop on Parallel Processing*, page 73, Washington, DC, USA, 2000. IEEE Computer Society.

[79] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[80] M. Philippsen and M. Zenger. Java Party transparent remote objects in Java. concurrency practice and experience, 1997.

[81] J. Postel. Rfc 768: User datagram protocol, internet engineering task force, August 1980.

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

[83] Y. Raz. The Dynamic Two Phase Commitment (D2PC) Protocol. In *ICDT '95: Proceedings of the 5th International Conference on Database Theory*, pages 162–176, London, UK, 1995. Springer-Verlag.

[84] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA, 1978.

[85] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In S. Dolev, editor, *Distributed Computing*, Lecture Notes in Computer Science, pages 284–298. Springer Berlin / Heidelberg, 2006.

[86] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber. From causal to z-linearizable transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 340–341, New York, NY, USA, 2007. ACM.

[87] 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).

[88] 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.

[89] M. M. Saad and B. Ravindran. Distributed Hybrid-Flow STM : Technical Report. Technical report, ECE Dept., Virginia Tech, December 2010.

[90] M. M. Saad and B. Ravindran. Hyflow: A high performance distributed software transactional memory framework. In *In Proceedings of the 20th IEEE International Symposium on High Performance Distributed Computing, HPDC '11*, HPDC '11, 2011.

[91] M. M. Saad and B. Ravindran. RMI-DSTM: Control Flow Distributed Software Transactional Memory: Technical Report. Technical report, ECE Dept., Virginia Tech, February 2011.

[92] M. M. Saad and B. Ravindran. Supporting STM in Distributed Systems: Mechanisms and a Java Framework. In *TRANSACT '11: Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, California, USA, 2011. ACM.

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

[94] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[95] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC '04: Proceedings of Workshop on Concurrency and Synchronization in Java Programs.*, NL, Canada, 2004. ACM.

[96] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[97] R. R. Stewart and Q. Xie. *Stream control transmission protocol (SCTP): a reference guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[98] M. Surdeanu and D. Moldovan. Design and performance analysis of a distributed java virtual machine. *IEEE Trans. Parallel Distrib. Syst.*, 13:611–627, June 2002.

[99] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of legacy Java software. In *In European Conference on Object-Oriented Programming (ECOOP*, 2001.

[100] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP*, 2002.

[101] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol, 1997.

[102] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Salutation consortium: Salutation architecture specification. version 2.0c, 1999.

[103] R. A. F. B. R. Veldema and H. E. Bal. Distributed shared memory management for java. In *In ASCII2000*, page 256, 2000.

[104] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.

[105] B. Zhang and B. Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*, pages 48–53, Berlin, Heidelberg, 2009. Springer-Verlag.

[106] B. Zhang and B. Ravindran. Dynamic analysis of the Relay cache-coherence protocol for distributed transactional memory. In *IPDPS '10: Proceedings of the 2010 24th IEEE International Parallel and Distributed Processing Symposium*, Washington, DC, USA, 2010. IEEE Computer Society.

[107] W. Zhu, C.-L. Wang, and F. Lau. Jessica2: a distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, 2002.