On Improving Distributed Transactional Memory through Nesting, Partitioning and Ordering

Alexandru Turcu

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

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Chao Wang
Roberto Palmieri
Robert P. Broadwater
Eli Tilevich

January 22, 2015
Blacksburg, Virginia

Keywords: Distributed Transactional Memory, Distributed Systems, Nested Transactions, Automated Partitioning, Consensus.

On Improving Distributed Transactional Memory through Nesting, Partitioning and Ordering

Alexandru Turcu

(ABSTRACT)

Distributed Transactional Memory (DTM) is an emerging, alternative concurrency control model that aims to overcome the challenges of distributed-lock based synchronization. DTM employs transactions in order to guarantee consistency in a concurrent execution. When two or more transactions conflict, all but one need to be delayed or rolled back.

Transactional Memory supports code composability by nesting transactions. Nesting however can be used as a strategy to improve performance. The closed nesting model enables partial rollback by allowing a sub-transaction to abort without aborting its parent, thus reducing the amount of work that needs to be retried. In the open nesting model, sub-transactions can commit to the shared state independently of their parents. This reduces isolation and increases concurrency.

Our first main contribution in this dissertation are two extensions to the existing Transactional Forwarding Algorithm (TFA). Our extensions are N-TFA and TFA-ON, and support closed nesting and open nesting, respectively. We additionally extend the existing SCORe algorithm with support for open nesting (we call the result SCORe-ON). We implement these algorithms in a Java DTM framework and evaluate them. This represents the first study of transaction nesting in the context of DTM, and contributes the first DTM implementation which supports closed nesting or open nesting.

Closed nesting through our N-TFA implementation proved insufficient for any significant throughput improvements. It ran on average 2% faster than flat nesting, while performance for individual tests varied between 42% slowdown and 84% speedup. The workloads that benefit most from closed nesting are characterized by short transactions, with between two and five sub-transactions.

Open nesting, as exemplified by our TFA-ON and SCORe-ON implementations, showed promising results. We determined performance improvement to be a trade-off between the overhead of additional commits and the fundamental conflict rate. For write-intensive, high-conflict workloads, open nesting may not be appropriate, and we observed a maximum speedup of 30%. On the other hand, for lower fundamental-conflict workloads, open nesting enabled speedups of up to 167% in our tests.

In addition to the two nesting algorithms, we also develop Hyflow2, a high-performance DTM framework for the Java Virtual Machine, written in Scala. It has a clean Scala API and a compatibility Java API. Hyflow2 was on average two times faster than Hyflow on high-contention workloads, and up to 16 times faster in low-contention workloads.

Our second main contribution for improving DTM performance is automated data partition-

ing. Modern transactional processing systems need to be fast and scalable, but this means many such systems settled for weak consistency models. It is however possible to achieve all of strong consistency, high scalability and high performance, by using fine-grained partitions and light-weight concurrency control that avoids superfluous synchronization and other overheads such as lock management. Independent transactions are one such mechanism, that rely on good partitions and appropriately defined transactions. On the downside, it is not usually straightforward to determine optimal partitioning schemes, especially when dealing with non-trivial amounts of data. Our work attempts to solve this problem by automating the partitioning process, choosing the correct transactional primitive, and routing transactions appropriately.

Our third main contribution is Alvin, a system for managing concurrently running transactions on a geographically replicated data-store. Alvin supports general-purpose transactions, and guarantees strong consistency criteria. Through a novel partial order broadcast protocol, Alvin maximizes the parallelism of ordering and local transaction processing, resulting in low client-perceived latency. Alvin can process read-only transactions either locally or globally, according to the desired consistency criterion. Conflicting transactions are ordered across all sites. We built Alvin in the Go programming language. We conducted our evaluation study on Amazon EC2 infrastructure and compared against Paxos- and EPaxos-based state machine replication protocols. Our results reveal that Alvin provides significant speed-up for read-dominated TPC-C workloads: as much as 4.8x when compared to EPaxos on 7 datacenters, and up to 26% in write-intensive workloads.

Our fourth and final contribution is M^2Paxos, a multi-leader implementation of Generalized Consensus. Single leader-based consensus protocols are known to stop scaling once the leader reaches its saturation point. Ordering commands based on conflicts is appealing due to the potentially higher parallelism, but is imperfect due to the higher quorum sizes required for fast decisions and the need to compare commands and track their dependencies. M^2Paxos on the other hand exploits fast decisions (i.e., delivery of a command in two communication delays) by leveraging a classic quorum size, matching a majority of nodes deployed. M^2Paxos does not establish command dependencies based on conflicts, but it binds accessed objects to nodes, making sure commands operating on the same object will be ordered by the same node. Our evaluation study of M^2Paxos (also built in Go) confirms the effectiveness of this approach, getting up to 7× improvements in performance over state-of-the-art consensus and generalized consensus algorithms.

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

Dedication

To my family and Hoai

Acknowledgments

I would like to thank my advisor, Dr. Binoy Ravindran, for his effective guidance, inspiration and for keeping me motivated during my progress through the PhD program. I could not have done this without his help.

I would also like to thank my committee members: Dr. Robert P. Broadwater, Dr. Roberto Palmieri, Dr. Eli Tilevich and Dr. Chao Wang, for their guidance and advice during my preliminary and defense examinations. It is a great honor for me to have them serving in my committee.

I am especially grateful to Dr. Roberto Palmieri and Dr. Sebastiano Peluso for the amazing input they provided throughout my research. Their contributions helped shape the direction of my work and have helped significantly strengthen the quality of my contributions.

Moreover, many thanks to all my peers in the Systems Software Research Group for their support and the many technical and non-technical discussions. They include, in no particular order, Dr. Antonio Barbalace, Sachin Hirve, Ahmed Hassan, Robert Lyerly, Mohamed Mohamedin, Dr. Junwhan Kim, Mohamed Saad, Duane Niles, Dr. Alastair Murray, and many others.

Last but not least, I am grateful to all my family for all their love and support. My parents supported, encouraged and motivated me throughout my education. My little sister, Diana, although she didn't help a lot during my study, occasionally provided a well deserved break. My girlfriend, Hoai, has devoted her love and endured all my difficulties over the past years. I couldn't have completed my dissertation without her love and support.

This dissertation is dedicated to all the people that helped me and are helping me all the way.

Contents

1 Introduction **1**

1.1 Transaction Nesting 3

1.2 Automatic Data Partitioning 3

1.3 Alvin: Geo-Replicated Transactional System 4

1.4 M^2Paxos: Faster General Consensus 5

1.5 Summary of Dissertation Contributions 6

1.6 Dissertation Organization 8

2 Previous and Related Work **9**

2.1 Distributed Transactional Memory 9

2.2 Nesting in Transactional Memory 11

2.3 Other Unconventional Database Systems 12

2.4 Automatic Partitioning 14

2.5 Geo-Replicated Transactional Systems 14

2.6 Consensus 16

3 Preliminaries: TM, Nesting, TFA **18**

3.1 Primer on Transactional Memory and Nesting 18

3.2 System Model 21

3.2.1 Nesting Model 22

3.2.2 Multi-Level Transactions 23

3.2.3 Open Nesting Safety 24

3.3 Transactional Forwarding Algorithm 24
3.4 SCORe 26
3.5 DTM Frameworks 26

4 Closed Nesting **28**
4.1 N-TFA Algorithm Description 28
4.2 Properties 30
4.3 Evaluation 32
4.3.1 Implementation Details 32
4.3.2 Experimental Settings 32
4.3.3 Experimental Results 33
4.4 Conclusion 35

5 Open Nesting **38**
5.1 TFA with Open Nesting (TFA-ON) 38
5.1.1 SCORe with Open Nesting (SCORe-ON) 40
5.2 Mechanisms and implementation 40
5.2.1 Abstract locks 41
5.2.2 Defining Transactions and Compensating Actions 41
5.2.3 Transaction Context Stack 42
5.3 Evaluation 42
5.3.1 Experimental Settings 43
5.3.2 Experimental Results 45

6 Hyflow2: A High-Performance DTM Framework in Scala **52**
6.1 The Hyflow DTM framework. Motivation 52
6.2 Hyflow2 API 54
6.2.1 ScalaSTM 54
6.2.2 Hyflow2 Objects 55
6.2.3 Hyflow2 Directory Manager 56

6.3 Transaction Nesting . . . 57
6.3.1 Nesting API . . . 57
6.4 Java Compatibility API . . . 58
6.4.1 Defining Transactions . . . 58
6.4.2 Defining Hyflow2 Objects . . . 60
6.5 Mechanisms and Implementation . . . 60
6.5.1 Actors and Futures . . . 61
6.5.2 Network Layer . . . 61
6.5.3 Serialization . . . 61
6.5.4 Hyflow2 Architecture . . . 62
6.5.5 Conditional Synchronization . . . 63
6.5.6 Parallel Object Open . . . 63
6.5.7 Transaction Checkpoints . . . 63
6.5.8 Performance . . . 64
6.6 Experimental Evaluation . . . 64

7 Automated Data Partitioning in DTM **67**
7.1 Background . . . 67
7.1.1 Granola: Independent Transactions . . . 67
7.1.2 Schism: Graph-Based Partitioning . . . 69
7.2 Overview . . . 71
7.2.1 Generating Partitions . . . 71
7.2.2 Using the Partitions at Run-Time . . . 72
7.3 Static Analysis . . . 73
7.3.1 First Pass . . . 73
7.3.2 Second Pass . . . 75
7.3.3 Third Pass . . . 75
7.3.4 Byte-code Rewriting and Trace Collection . . . 76
7.4 Graph Representation and Partitioning . . . 77

7.4.1 Edge Weights . . . 78
7.4.2 Partitioning and Explanation . . . 79
7.5 Transaction Routing . . . 80

8 Evaluating Automated Data Partitioning in DTM 83
8.1 Experimental Setup . . . 83
8.1.1 Benchmarks . . . 84
8.2 Evaluation . . . 86

9 Alvin: A General, Consistent, Geo-Replicated Transactional System 92
9.1 Assumptions and System Model . . . 94
9.2 Alvin: Geo-Replicated Transactional System . . . 94
9.2.1 Partial Order Broadcast Layer . . . 95
9.2.2 Parallel Concurrency Control Layer . . . 100
9.2.3 Garbage Collection . . . 101
9.3 Evaluation . . . 102

10 M^2Paxos: Faster General Consensus 106
10.1 System Model and Consensus . . . 107
10.2 Protocol Overview . . . 108
10.2.1 The Fastest Delivery . . . 108
10.2.2 Forwarding Requests . . . 110
10.2.3 Requesting Ownership . . . 111
10.3 Evaluation . . . 112
10.3.1 Discussion . . . 117

11 M^2Paxos: Protocol Details 119
11.1 Data Structures . . . 119
11.2 The Protocol . . . 120
11.2.1 Coordination Phase . . . 120

11.2.2 Accept phase 122
11.2.3 Decision phase 123
11.2.4 Acquisition phase 124
11.3 Correctness Arguments 126

12 Conclusions **129**
12.1 Contributions 131
12.2 Future Work 131

Bibliography **133**

A Alvin: Pseudocode and Proofs **143**
A.1 Partial Order Broadcast Layer 143
A.1.1 Correctness of POB 147
A.2 Parallel Concurrency Control Layer 149
A.2.1 Correctness of P-CC 152

List of Figures

1.1 Example atomic blocks. In *a.* objects are assumed to not need opening before being accessed, as is common for Software Transactional Memory (STM). *b.* shows the same atomic block written to Hyflow2's API, also including object opening. 2

3.1 Simple example showing the execution time-line for two transactions under flat, closed and open nesting. 20

3.2 Transactional Forwarding Algorithm Example, from [115] 25

4.1 Nested Transactional Forwarding Algorithm Example 30

4.2 Performance change by benchmark. 33

4.3 Bank monetary application. First group varies transaction length while keeping read-ratio constant. Second group varies the read-ratio for short transactions. . . 33

4.4 Loan monetary application. First group varies transaction length while keeping read-ratio constant. Second group varies the read-ratio for short transactions. . . 34

4.5 Linked-list micro-benchmark. First group varies read-ratio for short transactions. In the second group, transaction length is varied. 34

4.6 Hash-table micro-benchmark. First group shows increasing number of calls on hash-tables with 7 buckets. Second group shows the effect of increasing transaction length. Third group shows increasing number of calls on hash-tables with 11 buckets. 35

4.7 Hash-table read-ratio plot. First group shows the effect of increasing read-ratio on short transactions with 3 calls. Second group shows the effect of the same parameter on longer transactions. Third group targets transactions with 5 calls. 35

4.8 Skip-list micro-benchmark. Shows the effect of increasing read-ratio on tests with one, two and three operations performed in a transaction, respectively. 36

4.9 Skip-list micro-benchmark. Shows the effect of increasing transaction length. First group contains transactions with 2 calls. For the second group, the number of calls is 3. The last group has 80% reads. 36

4.10 Effect of number of nodes participating in the experiment. First group shows the Bank benchmark with 16 threads/node. Second group shows the Skip-list micro-benchmark, with 4 threads/node. 37

5.1 Simplified source code for supporting Open Nesting in TFA's main procedures. 39

5.2 Simplified transaction example for a BST insert operation. Code performing the actual insertion is not shown. 42

5.3 Performance relative to flat transactions, with $c = 3$ calls per transaction and varying read-only ratio. Both closed nesting and open nesting are included. (TFA-ON/Hyflow) 43

5.4 Performance relative to flat transactions at a fixed read-ratio with varying number of calls. Closed-nesting is depicted, but the individual curves are not identified to reduce clutter. (TFA-ON/Hyflow) 44

5.5 Time spent in committed vs. aborted transactions, on hash-table with $r = 20$ and $c = 4$. Lower lines (circle markers) represent time spent in committed transactions, while the upper lines (square markers) represent the total execution time. The difference between these lines is time spent in aborted transactions. (TFA-ON/Hyflow) 46

5.6 Overhead of successful open-nested transactions. Plotted is the relative ratio of the average time taken by successful open-nested transactions to the average time taken by successful flat transactions. Closed-nested transactions are also shown, with dotted markers and without identification. (TFA-ON/Hyflow) . 46

5.7 Breakdown of the duration of various components of a transaction under open nesting, on hash-table with $r = 20$ and $c = 4$. (TFA-ON/Hyflow) 47

5.8 Number of aborted transactions under open nesting, with various parameters. The figure shows the effect of read-only ratio, number of calls, and key domain size. Note that all aborts depicted in this plot are full aborts due to abstract lock acquisition failure. The number of committed transactions is fixed for each experiment. (TFA-ON/Hyflow) 47

5.9 Throughput relative to flat nesting with increased key space $k = 1000$ and write-dominated workloads $r = 20$. (TFA-ON/Hyflow) 48

5.10 Relative throughput for TFA-ON implementation in Infinispan. 50

5.11 Relative throughput for SCORe-ON implementation in Infinispan. 50

5.12 Relative throughput for SCORe-ON implementation in Hyflow. 51

6.1 Example of the original Hyflow API. Transactions are marked using the @Atomic annotation. 52

6.2 An example transaction in ScalaSTM (common usage). 54
6.3 A more verbose version of the code in Figure 6.2, with several Scala syntactic shortcuts written explicitly. 54
6.4 Conditional synchronization using retry. Transaction can only proceed once there is at least one item in the list. 55
6.5 Hyflow2 Object example for a bank account. 56
6.6 Hyflow2 transaction example. Transaction must open an object before operating on it. 56
6.7 Open nesting in Hyflow2 57
6.8 ScalaSTM Java compatibility API. 58
6.9 Hyflow2 Java compatibility API using the Atomic class. 59
6.10 Scala-style Hyflow2 Object definition in Java. Notice how accessing Refs in this style is more verbose. 59
6.11 Java-style Hyflow2 Object definition in Java. Compact Ref access. 60
6.12 Hyflow2 system diagram 62
6.13 Summary of relative performance across benchmarks. 65
6.14 Throughput on Bank, for the high-contention workload, for different ratios of read-only transactions. 65

7.1 Example graph representation in Schism. The shaded areas are the transactions, which are represented in the graph by edges connecting all accessed objects. . . . 69
7.2 Example graph representation in Schism, with replication. 70
7.3 Forward data-flow analysis example for extracting the intra-method dependency graph. Rectangles represent nodes (units of interest), rounded rectangles are values. 77
7.4 Example partitioning graph for a txn with data deps. 78
7.5 Partitioning graph example in the presence of aborts. The correspondence between the static invocation sites and objects accessed at runtime is: 1-A, 2-B, 3-C, 4-D. 80

8.1 Partition quality on the various benchmarks (lower is better). 83
8.2 Results on TPC-C for a workload configured with 3 warehouses, with different classifiers. The trace used contains approx. 1200 transactions. Boxes show average values. Black markers show best classifier. Horizontal line in 8.2(a) shows theoretical best. 87

8.3 Best partition quality on TPC-C with increasing number of warehouses (lower is better). Horizontal line is theoretical minimum. Differences are minimal (less than 0.4%) 88

8.4 Quality of partitioning and routing with respect to increasing trace size, on TPC-C (15 warehouses). In 8.4(a), horizontal line is optimal manual partitioning. 88

8.5 Total transactional throughput by benchmark. 90

8.6 Total throughput on TPC-C. The bar is the average. The lower error bar is the standard deviation. The upper error bar is the maximum. 90

8.7 Total transactional throughput (TPC-C, 3 warehouses), with a varying fraction of distributed txns. 91

9.1 Throughput of TPC-C benchmark. 103

9.2 Throughput Vs Latency using TPC-C benchmark varying application threads. 104

9.3 Throughput under write-intensive workload for {3,5,7} sites and {1,3} nodes per site using Bank benchmark. 105

10.1 Scalability in a practical deployment. 64 client threads per node, and 5 ms think time. Command locality is 100%. 113

10.2 Scalability plot. (a) Reflects the maximum attainable throughput. (b) Shows the median latency when the system is underloaded. Command locality is 100%. 113

10.3 Maximum throughput for 11-nodes deployments with different machine types. The number of cores are 4, 8, 16 and 32 respectively. 114

10.4 Throughput varying the fraction of complex commands. The value in parenthesis is the number of possible objects per node. The deployment consists of 49 replicas. 115

10.5 Latency vs. throughput plots, with 0% and 100% command locality for M^2Paxos and EPaxos. 116

List of Tables

2.1 Summary of DTM related work 13

7.1 Method invocations and reference types that aid in identifying transactional methods and features (static analysis, first pass). 74

8.1 Per phase running time, on TPC-C with 15 warehouses and a 89MB input trace containing 42k txns. 85

Chapter 1

Introduction

Until recently, CPU manufacturers were able to increase the performance of their devices by running them at ever higher frequencies. However around 2004, this trend became unsustainable, and adding multiple processing cores on the same chip became the new standard. Software developers were forced to embrace *concurrency* as the means for their programs to run faster, or perform more advanced processing in a short time.

Handling concurrency correctly is a difficult task. The simple strategy of using a single global lock may be easy to implement, but it hardly brings any performance benefits, effectively executing all *critical sections* sequentially. Using fine-grained locks to protect individual pieces of data enables the much desired *scalability*, but is inherently error prone. Any mistakes can lead to hard to trace problems such as *deadlocks* and *race-conditions*. Moreover, due to the randomness of concurrency, these problems may not manifest during testing, misleading the programmer to ship a defective product.

Using locks also makes code composition difficult. Suppose a library uses locks to control access to a hash-table and the programmer needs to apply two hash-table operations in an atomic manner in order to hide the intermediary state from other threads. In this situation, he or she may introduce extra locks protecting both data structures at the same time, but this can lead to race conditions or loss of performance if not implemented carefully. Alternatively, the programmer may try to expose the implementation of the library in order to understand and then extend its locking mechanism, but again, this is error prone and moreover it contradicts the encapsulation concept of object-oriented programming.

Transactional Memory (TM) and subsequently, *Software Transactional Memory* (STM) were proposed to bring a successful abstraction from the database community, the *transaction*, into regular multi-processor programming [48, 106]. Transactions were originally developed to provide four important properties: atomicity, consistency, isolation and durability (the ACID properties). In this context, atomicity (or failure atomicity) means that the operations making up a transaction either all execute to completion, or they appear as if they

```
atomic {

  acc1.amt += value
  acc2.amt -= value
}
```

a

```
atomic { implicit txn =>
  val acc1 = Hyflow.dir.open[BankAccount](("acc",1))
  val acc2 = Hyflow.dir.open[BankAccount](("acc",2))
  acc1.amt() += value
  acc2.amt() -= value
}
```

b

Figure 1.1: Example atomic blocks. In *a.* objects are assumed to not need opening before being accessed, as is common for Software Transactional Memory (STM). *b.* shows the same atomic block written to Hyflow2's API, also including object opening.

never started executing. This effectively prevents a transaction from executing partially and leaving the system in an inconsistent state. The isolation property prevents a transaction from observing the intermediary states that another parallel transaction may produce while running. Thus, ACID transactions are *serializable*: although they may execute concurrently, the overall effect is the same as if they executed serially, one after another, without any overlap. The A and I properties give the illusion that a transaction either executes at a single instant in time (i.e. **atomic execution**), or not at all. In TM, transactions are expressed in the same programming language as the rest of the application through the use of atomic blocks. An example STM atomic block is shown in Figure 1.1(a).

Lock-based concurrency is even more challenging in the distributed setting. The same problems are further amplified by the difficulty of debugging distributed systems and their inherent uncertainty. One solution for this problem, *Distributed Transactional Memory* (DTM) is an extension of STM to distributed systems. It provides the same easy-to-use abstraction of transactions expressed in a regular programming language, as illustrated in Figure 1.1(b).

Separate recent efforts brought main-memory storage to traditional Relational DataBase Management Systems (RDBMS, [109, 52]), which are interfaced using the SQL language. Yet another separate effort brought transactions to main-memory data-stores [75]. While this dissertation explicitly targets DTM systems, all three communities mentioned above relate to in-memory distributed transactions and thus stand to benefit in some degree from our work.

Performance in DTM is paramount. As a new abstraction trying to replace its predecessor, DTM must at least match the performance characteristic of distributed locking. As such, a majority of the research focused on DTM is aiming for increasing its performance. This quest can be observed in research on directory protocols [119, 8], Multi-Version Concurrency Control [73], scheduling transactions [9, 55], transaction protocols [102, 11, 101] and DTM implementations [100, 16, 22].

Our work continues this quest to improve DTM performance and, more generally, the performance of distributed transactions. Towards this purpose we look, in turn, at transaction nesting, automatic data partitioning, geographically replicated transactions and lease-based

consensus.

1.1 Transaction Nesting

To solve the code composability problem, TM uses the concept of nesting. A transaction is nested when it is enclosed within another transaction. Since in TM concurrency control is handled by an underlying framework or library, application code does not need to know the details of its callers or callees, thus maintaining encapsulation.

The outer transaction is called parent and the inner transaction is the child. Child transactions can also have their own children, resulting in a tree-like structure. Transactions may have multiple children, leading to inner transactions that can execute concurrently [116]. However, in this dissertation we will only consider linear nesting [81], where each transaction can only have at most one child, and the bottom-most transaction in the chain is the only active transaction.

While transaction nesting was studied extensively in non-distributed TM ([83, 4, 78, 80, 81, 116]), this topic was never touched upon in the DTM literature. Nesting can potentially help improve DTM performance (as it does for non-distributed TM), and we consider important to evaluate any such improvements.

Closed nesting, as a generic partial rollback mechanism, reduces the amount of work that needs to be retried in case of transaction aborts. In the distributed context, such work usually involves opening remote objects — an inherently slow operation due to network latency. For these reasons, we expect to see closed nesting improving DTM performance. We also seek to identify what factors and workload characteristics have an influence on performance.

Open nesting reduces isolation by releasing memory locks early, and thus allows more transactions to execute concurrently without aborting. We expect this will directly translate into greater system throughput (as measured in transactions per second). We again seek to identify influencing factors.

Finally, irrespective of which algorithms are being used, DTM performance also depends on the efficiency of the system's implementation. Being dissatisfied with our current framework, Hyflow, we develop a new framework from scratch, which we named Hyflow2. In implementing Hyflow2, we focus on performance, ease of use, and rapid prototyping.

1.2 Automatic Data Partitioning

Modern distributed systems require increasing isolation levels, scalable performance, fault-tolerance and a simple programming model for being easily integrated with transactional

applications. The recent growth of large scale infrastructures with dozens or hundreds of nodes needs transactional support ready to scale with them.

Many of the modern transactional storage systems have abandoned strong consistency (e.g., serializability) in order to achieve good scalability and high performance [17, 28, 61]. Weak consistency models (e.g., eventual consistency) incur the expense of allowing some non-serializable executions, which, if at all tolerated by the application requirements, are more difficult to deal with for the developers [107]. In fact, it was observed that developers prefer strong consistency when possible [21].

For this reason, transactional storage systems that offer serializability without forsaking high speed and high scalability, represent a very promising sweet spot in the overall design space. One system that approaches this sweet spot for On-Line Transaction Processing (OLTP) workloads is Granola, as proposed by Cowling and Liskov in [24]. Granola employs a novel transactional model, *independent transactions*, to keep overheads and synchronization to a minimum while guaranteeing serializable distributed transactions. To help reach its high transaction throughput, Granola relies on storing the data in main memory and operating upon it using transactions expressed in the application's native programming language, and is essentially a DTM system.

One key enabler for good performance in the Granola model is having the data organized in fine-grained, high-quality partitions that promote the use of single-partition and independent distributed transactions. This can be considered a drawback for Granola, as developers need to manually organize the data, choose the transaction primitives, and route transactions appropriately. Our work focuses on eliminating this drawback by automating the three tasks.

To reach our goal, we adapt and extend an existing graph-based data partitioning algorithm, Schism [25], originally proposed for traditional, SQL-based databases. By targeting a DTM environment, we can reach much higher transaction throughputs than traditional OTLP workloads [109], and we are presented with some interesting problems that allow us to innovate.

1.3 Alvin: Geo-Replicated Transactional System

In recent years, there has been an increase in demand for data storage systems capable of executing distributed transactions spanning across geographic regions. This was reflected by a significant research interest in geographically distributed computer systems (or GDS, [107, 60, 120, 21, 77]).

One motivating factor for this trend is the growing number of Internet-scale applications. Geographically distributed systems are generally employed for two reasons. Firstly, if properly engineered, a GDS can tolerate major disasters that would bring a whole data-center (site)

off-line, without service disruptions or data loss. Secondly, GDS can optimize application performance by exploiting the locality of accesses from geographically-dispersed users.

It is typical for geographically distributed applications not to exhibit high conflicts among transactions spawned at different sites [77]. This property has encouraged the development of geo-replicated transactional concurrency control protocols that lower communication costs when transactions do not conflict, thus boosting performance and scalability. Furthermore, the ease of programmability is an important goal for any computer system, but it is challenging to achieve both high performance and good programmability. With distributed systems, strongly consistent transactions generally lead to great programmability, because they shield the developers from inconsistent data and partial transactional states, and enable them to easily reason about application behavior. On the downside, strong consistency is also associated with poor performance and scalability due to the higher synchronization requirements and insufficient exploitation of locality.

Thus the concurrency control protocols typically used with geo-replicated transactional systems can be classified in two categories. The first approach targets high consistency, but restricts the type of transactions that are allowed [120, 77] and exploiting specific protocol optimizations to achieve high performance. The second approach allows general-purpose transactions, but compromises with a weaker consistency criterion in order to better the performance [6, 107]. This has the negative effect of reduced programmability, as programmers must cope with the aforementioned issues.

Motivated by this gap between current solutions in this space, we propose a geo-replicated transactional system called Alvin, which finds an effective trade-off between performance and strong consistency. Alvin was built around a novel Partial Order Broadcast protocol (*POB*) that globally orders only conflicting transactions while minimizing the number of communication steps for non-conflicting transactions and avoids relying on a single designated leader. Alvin was implemented as a DTM framework in the Go programming language.

1.4 M^2Paxos: Faster General Consensus

Paxos [64, 65] is the most commonly deployed solution for the consensus problem [19]. It is able to reach agreement between participants interconnected by asynchronous networks, even in the presence of faults, and it can be employed to easily build strongly consistent transactional systems [21, 50, 60, 72, 36]. Despite its widespread use, Paxos (as well as its most commonly deployed variant, Multi-Paxos [65]) is still limited due to its usage of a single designated leader. When the leader's resources are exhausted (e.g., CPU, network bandwidth), the protocol stops scaling, despite the other nodes being underutilized.

Recent research tried to eliminate the bottleneck associated with having a single leader by allowing all nodes to take leadership for a particular subset of the commands [77, 74, 113].

These proposals however introduce other costs that prevent scalability, stemming from the absence of a single point of decision and the competition between leaders in the presence of conflicting commands.

To provide any benefits at all, multi-leader protocols relax the guarantees they can provide. Instead of totally ordering all commands, they instead agree on a partial order where conflicting commands are totally ordered, whereas non-conflicting commands may be delivered in different orders at the different nodes participating to the consensus. They solve the problem of Generalized Consensus [63], which is a generalization of the consensus problem that requires agreement on an increasing sequence of commands for less than a permutation of non-conflicting commands. In practice, generalized consensus is enough to provide strong consistency on top a a transactional replicated data storage system.

The scalability and performance of past solutions to the Generalized Consensus problem are the result of competing benefits and drawbacks. Firstly, these protocols reduce the number of communication steps required to reach consensus from three to two, matching the lower bound for the problem of Consensus in asynchronous systems [66]. Secondly, these protocols require replies from a super-majority of nodes before a decision is reached. For instance, assuming N is the number of nodes participating to the consensus, some protocols wait for $\lceil \frac{3}{4}N \rceil$ or $\lceil \frac{3}{4}N - 1 \rceil$ replies from other nodes [63, 77, 113], whereas (Multi-)Paxos only waits for a simple majority of $\lfloor \frac{N}{2} \rfloor + 1$ replies. Finally, previous Generalized Consensus solutions need to track, communicate and process the conflict relations between commands, which in the general case is a rather expensive process.

We thus propose M^2PAXOS, a solution to the Generalized Consensus problem that employs a multiple leader strategy, while avoiding the costs incurred by past solutions (i.e., larger quorum sizes, processing of dependencies). M^2PAXOS achieves its goals by exploiting the locality of data accesses, and granting leases to nodes for exclusive ordering privileges on individual sub-sets of data (partitions). We implement M^2PAXOS in the Go programming language.

1.5 Summary of Dissertation Contributions

We design *N-TFA* and *TFA-ON*, extensions to the existing Transactional Forwarding Algorithm (TFA) with support for closed nesting, and respectively, open nesting. We implement N-TFA and TFA-ON in Hyflow, a Java DTM framework and evaluate them on a set of micro-benchmarks. We find closed nesting has an average performance improvement of only 2% compared to flat nesting, but with a maximum speedup of 84%. Open nesting enabled up to 30% improvement for write-dominated workloads and increased fundamental conflicts, and as high as 167% under reduced fundamental conflict workloads. We observe that closed nesting applies best when transactions do not access many objects, and when the number of sub-transactions is between 2 and 5. The best speedups for open nesting occur when trans-

actions are composed of a high number of sub-transactions, and when fundamental conflict rates are low. To the best of our knowledge, this dissertation contributes the first closed nesting and open nesting implementations for DTM.

We further verify our results for open nesting by extending another existing STM algorithm (SCORe, [89]) with support for open nesting. The resulting algorithm (SCORe-ON) was implemented in Infinispan [75].

We design and implement Hyflow2, a novel high-performance DTM framework for the JVM. Hyflow2 is the first DTM implementation in Scala, with a Java compatibility API, and with support for features such as transaction nesting, checkpointing and conditional synchronization. At high contention, Hyflow2 proved on average 2x faster than Hyflow, with a peak of up to 7x at low node counts. At low contention however, Hyflow2 is consistently 8-15x faster.

We extend the Schism automatic data partitioning algorithm with support for independent distributed transactions. Thus the newly developed partitioning algorithm suggests partitions that favor both fast single-partition and independent transactions against the slower, Two-Phase Commit (2PC) coordinated transactions. We develop a mechanism based on static program analysis for determining edge weights in the graph that Schism uses for proposing partitions. This essentially enables applying an algorithm like Schism (which originally only works on SQL workloads) to DTM transactions expressed in a native programming language. We contribute a machine-learning based mechanism for routing transactions, which is essential for enabling any kind of automatic partitioning in a DTM environment, where a transaction's access set is not known a priori. Additional minor contributions include automatic program refactoring for run-time trace collection, and automatic choice of an appropriate transaction primitive based on static program analysis. To the best of our knowledge, this is the first work that provides an end-to-end automated framework for exploiting independent transactions.

We develop Alvin, the first geo-replicated transactional system that guarantees a strong consistency level and supports the execution of general-purpose transactions in classic asynchronous environments. We contribute Alvin POB, a novel multi-leader protocol for partially ordering transactions, enabling high scalability in geo-replicated environments. In addition, the protocol does not need complex local processing for determining the final delivery order, yielding reduced client-perceived latency. In an extensive evaluation study, Alvin is shown to outperform state-of-the-art competitors on well-known transactional benchmarks.

Finally, we propose M^2Paxos, a novel solution for the Generalized Consensus problem. M^2Paxos does not have the bottleneck of a single leader, while also employing minimal quorum sizes and not having a slow fall-back path for when contention is encountered. Additionally, M^2Paxos has cheap local processing as it doesn't need to track relations between commands. We show M^2Paxos scales well and outperforms competitors up to 7× in a deployment of 49 nodes.

All of our work is publicly available as at https://bitbucket.org/talex/ .

1.6 Dissertation Organization

This dissertation proposal is organized as follows. In Chapter 2 we summarize relevant and related previous work. In Chapter 3 we describe our basic system model, our nesting model, and TFA, the base algorithm for all our contributions. Chapter 4 introduces and evaluates N-TFA, our closed-nested extension to TFA. Open nesting and TFA-ON are discussed in Chapter 5. In Chapter 6 we introduce Hyflow2, our new-generation DTM framework. Chapters 7 and 8 introduce and respectively evaluate our automatic data partitioning methodology. Chapter 9 focuses on Alvin. Chapter 10 discusses M^2Paxos, while Chapter 11 provides its detailed algorithm description. Finally, Chapter 12 concludes the dissertation and discusses potential future work.

Chapter 2

Previous and Related Work

2.1 Distributed Transactional Memory

DTM was first proposed by Herlihy and Sun [49] as an alternative to standard distributed transactions using *Two-Phase Locking* and *Two-Phase Commit Protocols* (2PC) as is standard in database environments. They use a *dataflow*-based approach where transactions execute on a fixed node while the data migrates to the transactions that requires it. One claimed advantage of this approach is that it does not require a distributed commit protocol, making successful commits fast. In order to manage the location of data, the authors propose a distributed cache-coherence protocol called Ballistic. This protocol, alongside a contention manager, manage data conflicts and ensure its consistency. On the downside, it relies on an existing distributed queuing protocol, Arrow [30], that does not take contention into account, and due to its hierarchical structure, scalability is limited — the entire structure needs rebuilding every time a node joins or leaves the network.

Zhang and Ravindran [119] developed the Relay protocol which takes transactional conflicts into account and scales better due its use of peer-to-peer data structures. The authors also introduce Location Aware Cache-coherence protocols (LAC, [118]), where nodes closer to the data (in terms of communication cost) are guaranteed to locate the object earlier. They show that LOC protocols, in conjunction with the optimal Greedy contention manager, improve the *makespan* competitive ratio, a measure of the efficiency of a transaction execution.

Unlike previous proposals, which do not tolerate unreliable links, Attiya et al. present Combine [8], a directory protocol that works even in the presence of partial link failures and non-FIFO message delivery. Combine is however still not network partition tolerant.

Bocchino et al. took an implementation based approach and developed Cluster-STM [12]. They observe that remote communication overheads are the main impediment for scalability, and thus try to make an appropriate set of design choices, sometimes different than other

cache-coherent STMs. Examples of such choice are aggregating the remote communication with data communication, and using a single access-set rather than separate read and write-sets.

Kotselidis et al. developed DiSTM [59], an DTM system optimized for clusters. DiSTM can be configured with three cache coherence protocols. TCC [42], an existing decentralized protocol, suffers from large traffic overheads at commit time, as transactions broadcast their read and write-sets. These overheads are avoided using two newly proposed lease-based protocols, at the expense of introducing lease bottlenecks and an additional validation step. In benchmarks, no one protocol achieved greater performance, but rather the best protocol choice depended was dependent on the amounts of contention and network congestion.

In contrast to cache-coherent DTM, replicated DTM stores multiple writable copies of the data, and this is a promising approach for achieving fault-tolerance. D2STM is the first replicated DTM system. Introduced by Couceiro et al [22], it provides strong consistency even in the presence of failures by using a non-blocking distributed certification scheme. This scheme is inspired by recent database replication research [85, 87], but employs Bloom filters in order to reduce the overheads of replica coordination, at the expense of an increased probability of false aborts.

Asynchronous Lease-based Certification protocol (ALC, [15]) is a protocol for distributed transaction processing based on asynchronous leases, that enables up to ten times lower commit latencies. ALC relies on an underlying Optimistic Atomic Broadcast primitive [29] to establish lease ownership. Once a node has all the leases required for a transaction, it can certify the transaction locally and finally disseminate its updates using a more efficient broadcast primitive.

Lilac-TM [46] improves upon the ALC algorithm by dynamically deciding whether and when to migrate lease ownership or alternatively forward transactions to the node currently owning the required leases.

Romano et al. report in [96] on implementing a web application using Distributed Transactional Memory, and the experience of its first two years in production. The authors make several important observations, such as the workload being comprised of only 2% write transactions, and the average write-set being orders of magnitude smaller than the average read-set. In [97], they show how DTM would be an appropriate programming model for applications running in cloud environments (i.e., clusters of hundreds of nodes or more), and point to several research directions that would help reach this goal.

A number of researchers focused on consistency criteria weaker than serializability in in order to improve DTM performance. In particular, Multi-Version Concurrency Control (MVCC) and its associated consistency criterion, Snapshot Isolation (SI) have the advantage of not generally having to abort read-only transactions. MVCC has been extensively studied in the database environments and multi-processor STMs [14, 92, 91].

Several DTM systems also use MVCC. ALC [15] relies on MVCC to enable only acquiring

leases for writes. Peluso et al. introduce the GMU protocol [90], which is the first protocol to provide Snapshot Isolation and Genuine Partial Replication (i.e., only nodes replicating used data are involved in the transaction protocol). GMU relies upon several mechanisms. It employs a new scheme based on Vector Clocks (VC) to determine which version of an object to be returned by a read operation, and to achieve agreement upon next object version and the VC value attached to committed transactions. Additionally, prepared transactions wait in a commit queue (sorted by a particular VC entry) before they are allowed to commit. The commit operation is effectuated in a standard Two Phase Commit (2PC) fashion. By disseminating the VC of the oldest transaction still running, old object versions can be safely garbage collected.

2.2 Nesting in Transactional Memory

Nested transactions (using closed nesting) originated in the database community and were thoroughly described by Moss in [79]. His work focused on the popular two-phase locking protocol and extended it to support nesting. In addition to that, he also proposed algorithms for distributed transaction management, object state restoration, and distributed deadlock detection.

Open nesting also originates in the database community [34], and was extensively analyzed in the context of undo-log transactions and the two-phase locking protocol [117]. In these works, open nesting is used to decompose transactions into multiple levels of abstraction, and maintain serializability on a level-by-level basis.

One of the early works introducing nesting to Transactional Memory was done by Moss and Hosking in [81]. They describe the semantics of transactional operations in terms of *system states*, which are tuples that group together a transaction ID, a memory location, a read/write flag, and the value read or written. They also provide sketches for several possible HTM implementations, which work by extending existing cache coherence protocols. Moss further focuses on open-nested transactions in [80], explaining how using multiple levels of abstractions can help differentiate between fundamental and false conflicts and thus improve concurrency. Ni et al. also discuss the implications of open nesting in [83], and additionally provide the first open nesting implementation for STM.

Moravan et al. [78] implement closed and open nesting in their previously proposed LogTM HTM. They implement the nesting models by maintaining a stack of log frames, similar to the run-time activation stack, with one frame for each nesting level. Hardware support is limited to four nesting levels, with any excess nested transactions flattened into the inner-most sub-transaction. In this dissertation, open nesting was only applicable to a few benchmarks, but it enabled speedups of up to 100%.

Agrawal et al. combine closed and open nesting by introducing the concept of transaction ownership [4]. They propose the separation of TM systems into transactional modules (or

Xmodules), which *own* data. Thus, a sub-transaction would commit data owned by its own Xmodule directly to memory using an open-nested model. However, for data owned by foreign Xmodules, it would employ the closed-nesting model and would not directly write to the memory.

From a different perspective, Herlihy and Koskinen propose transactional boosting [47] as a methodology for implementing highly concurrent transactional data structures. Boosted transactions act as an abstraction above the physical memory layer, internally employing open nesting (or a suspension mechanism) and abstract locks. Boosting works with an existing concurrent data structure (which it treats as a black box), captures a different (possibly better) performance-complexity balance than pure open nesting, and is easier to use and reason about.

2.3 Other Unconventional Database Systems

While not considered Distributed Transactional Memory, a recent line of research is proposing a complete rewrite of conventional database systems. Stonebraker et al. argue [109] that conventional DBMS systems, while trying to be provide a solution applicable to a wide range of problems such as on-line transactional processing (OLTP), data-warehousing, and stream processing, in reality they do a bad job at all such problems. In [43] the authors analyze how is the CPU time spent in a conventional DBMS and find out that only a fraction of time is used to do useful work, while the vast majority of time is spend in tasks such as resource management, locking and synchronization. Thus, they propose a new architecture that specifically targets OLTP workloads, and they implement H-Store [53] to demonstrate its superiority. H-Store stores all the data in the main-memory. Durability is achieved using network-based replication instead of disk-backed logs. Data is horizontally partitioned across multiple sites, and each such repository is backed up by a number of replicas. At each repository, transactions are processed in a single thread. A majority of transactions in common OLTP workloads only require data from a single site. Such single-repository transactions are run by H-Store without any concurrency control. H-Store is able to outperform a leading commercial DBMS by almost two orders of magnitude, while still guaranteeing strong consistency (serializability).

Cowling and Liskov improve upon H-Store by introducing independent transactions in Granola [24]. Under this model, single-shot distributed transactions can execute without coordination between nodes on the critical path as long as the same commit/abort decision can be individually taken by each repository without an agreement protocol. Read-only and non-aborting single-shot transactions are good candidates for this model. Repositories vote on a proposed time-stamp for each transaction and then execute all transactions in time-stamp order. Voting can be handled asynchronously in a thread separate to the thread executing transactions.

Aguilera et al. tackle a similar problem and propose minitransactions [5] as a way to achieve good performance and scalability. Minitransactions are the result of applying a number of optimizations to the standard two-phase commit protocol. They significantly reduce the number of round-trips required to commit a transaction, at the expense of having a severely constrained transaction primitive (essentially a multi-object compare-and-set) that requires all its accessed data to be specified in advance. Using this primitive, the authors quickly implemented a cluster file-system and a group communication service.

We summarize relevant related DTM work in Table 2.1, along with some representative STM works.

	STM						DTM						Other			
	McRT-STM [103]	Deuce STM [57]	Scala STM [13]	SwissTM [32]	RingSTM [108]	NOrec [26]	Cluster-STM [12]	DiSTM [59]	D2STM [22]	ALC / D2STM [15]	GMU / Infinispan [90]	TFA / Hyflow [98]	Sinfonia	H-Store	Granola	Current Work
Language	C++	Java	Scala	C++	C	C	C	Java	Java	Java	Java	Java	C++	C++	Java	Java/Scala
Serializable	Y	Y	Y	Y	Y	Y	Y	Y				Y	Y	Y	Y	Y
MVCC									Y	Y	Y					
Replicated								Y	Y	Y	Y		Y	Y	Y	
Fault-Tolerant									Y	Y	Y		Y	Y	Y	
Closed Nesting	Y			-		Y										Y
Open Nesting																Y
Checkpoints																Y
Strong Atomicity	†		Y		†	†		†		Y	†		Y	Y	Y	Y
Conditional Sync	†					Y										Y

Table 2.1: Summary of DTM related work

† The presence of this feature is not clear from the article.

2.4 Automatic Partitioning

Partitioning techniques have been widely studied in context of DBMS where the typical approach is to enumerate possible partition schemes and evaluate them using different methodologies. Horticulture [86] is a system similar in spirit to our work. It targets OLTP in workloads that may sustain temporal skew, and was implemented in H-Store [53], a main-memory NewSQL system. Horticulture employs a partitioning algorithm derived from a *large neighborhood search*. It however lacks support for independent distributed transactions, and chooses a partitioning strategy at table granularity as opposed to object granularity.

In [82], Nehme and Bruno implement and evaluate several automatic partitioning algorithms into the query optimizer for a data warehouse system with long, complex queries. Due to the nature of such a system, the optimization goals are different: data movement costs must be reduced for each query due to the sheer amount of data involved. The work in [69] also targets data warehouse systems, but optimizes for data skipping instead. In contrast to these two approaches, our system aims to reduce distributed transactions when the workload is composed of short, low-latency OLTP transactions.

In [112], the authors propose a stochastic approach for clustering data in object oriented DBMS. In context of distributed storage systems, [17] and [20] propose systems which continuously re-partition data to increase the balancing. Unfortunately these strategies cannot be easily ported to transaction processing due to the presence of incoming transactional requests. AutoPart [84] is an automated scheme designed for multi-terabyte datasets, without any OLTP requirements. A dynamic vertical partitioning approach based on query patterns was published in [95]. However it is better suited for applications where such information does not tend to change over time.

2.5 Geo-Replicated Transactional Systems

Many modern transactional systems employ geo-replication as a means to reduce data access latency and to provide fault-tolerance and disaster recovery. However, most of such geo-replicated systems chose to relax strong consistency guarantees and instead provide only eventual consistency, requiring developers to reason about state divergence and manual merging routines.

Spanner [21] is Google's globally-replicated database. It provides externally-consistent transactions, and is the first system with such strong guarantees at a global scale. Its architecture is complex: it relies on the TrueTime API, which exposes the absolute time and the uncertainty of the time measurement. The uncertainty is kept low by employing multiple dedicated time servers within each datacenter, each being equipped with modern clock references such as GPS and atomic clocks. Internally, each data partition is managed by a Paxos [64] instance with long-lived leaders in order to agree on a final timestamp for the

commit of write transactions. Using the TrueTime API, the system can guarantee that only one Paxos leader exists at any given time. Spanner provides several types of operations: read-write transactions, read-only transactions, and snapshot reads. Spanner concurrency control uses two-phase locking for write transactions. Upon a transaction's commit, a Paxos write takes place and is assigned a timestamp. This time-stamp is also representative for the original transactions. In order to ensure that writes are not visible to clients until after the commit time-stamp is certainly past, Spanner delays the commit, effectively waiting out the time uncertainty and this introduces additional delays experienced by clients due to the use of TrueTime API.

Walter [107] is a key-value store with support for transactions. It ensures Parallel Snapshot Isolation, which offers strong consistency within a site (data-center), but relaxed consistency across sites. To keep latency low, transactions are only replicated asynchronously across sites. Walter ensures Parallel Snapshot Isolation, which allows non-conflicting write transactions that span multiple sites to commit even if they observed incompatible histories. Walter employs two techniques in order to provide PSI: preferred sites and counting sets. With preferred sites, each object is owned by a single data-center (its preferred site), and writes to the object at its preferred site do not need to check for remote conflicts. This is different from the database concept of primary site, as non-preferred sites are not barred from modifying an object. Instead, they simply need to check for remote conflicts before proceeding with a write. Thus, when a transaction accesses objects only at their preferred sites, it can commit without remote communication. Counting sets (csets) are conflict-free, eventually consistent data types.

MDCC [60] is an optimistic commit protocol that is able to commit transactions with a single round-trip across data-centers. MDCC emphasizes reducing transaction latency, and enables recovery from failure without stalling other transactions. It defaults to Read-Committed consistency, but the authors claim that other consistency properties could be easily plugged in. MDCC commits transactions by using one instance of Multi-Paxos [64] per replication group containing the accessed data items and, if a transaction touches multiple replication groups, one more communication step is required to reach a consensus among leaders of the various Paxos instances. On the other hand, MDCC commits transactions by using one instance of Multi-Paxos [64] (or Generalized Paxos [63] to exploit commutative operations) per replication group containing the accessed data items and, if a transaction touches multiple replication groups, an additional phase is required to reach a consensus among the leaders of the various groups.

Gemini [68] differentiates the desired consistency level depending on operations' types. While some operations in a workload may require a high degree of consistency, many others do not. Authors propose RedBlue consistency, which combines strongly consistent but slow red operations, with eventually consistent but fast blue operations. It involves the programmer on deciding which operation is tagged as red or blue.

COPS [70] is a geo-replicated key-value store which implements *causal consistency with con-*

vergent conflict handling (or causal+) consistency. COPS supports multi-get transactions, but no general purpose transactions. COPS requires the use of a client-side library which tracks dependencies on keys, necessary for ensuring causal+ consistency. Storage nodes then asynchronously replicate the stream of write operations.

Eiger [71] is a geo-replicated system that offers a richer data model, while maintaining low transaction latency and supporting both read-only and write-only transactions. Eiger provides causal consistency by explicitly tracking dependencies on operations, as opposed to COPS's tracking of dependencies on data. Read-only transactions are guaranteed to complete within two rounds of local reads. Write-only transactions use a variant of two-phase commit (2PC) that is always successful, but may wait for other transactions to complete.

Lynx [120] is a geo-distributed transactional storage that works by chopping transactions into sequences of pieces. Each piece executes at a different datacenter, and the system usually replies to clients after the first hop, thus resulting in a very low latency. On the downside, subsequent hops may not abort a transaction. Conflicts are avoided through the use of home geo-replicas, which receive all updates for their respective data. Serializability is ensured by doing an a priori static analysis of the whole workload. Transactions that would not be serializable using the piece-wise execution model are executed as classic distributed transactions using two-phase locking and 2PC.

2.6 Consensus

Paxos [64] is perhaps the most widely used solution for the Consensus problem [19]. Participants to the Paxos algorithm have three possible roles: proposers, acceptors and learners. Proposers issue the values that need to be agreed upon following a round of consensus. Acceptors vote for such values and act as the distributed memory of the protocol. Learners infer the outcome of every consensus round – they learn the values agreed upon and may take action to handle them. In practice, nodes are called replicas and usually perform all three roles at the same time.

In the classic Paxos algorithm, a value is learned after a minimum of four communication delays. Progress guarantees can not be provided as the Prepare phase may fail in the presence of multiple concurrent proposals. Multi-Paxos alleviates this problem by letting promises (the first Paxos round) cover an entire sequence of values. This effectively establishes a distinguished proposer that acts as a coordinator (also called leader or master). Once a leader is elected, new values can be learned in only three communication delays and progress can be guaranteed in the periods of synchrony of the system.

Fast Paxos [67] can eliminate one communication delay by having proposers bypass the leader and broadcast their request directly to the acceptors. This is called a fast round. If a fast round fails due to concurrent proposals, a classic Paxos round is needed to recover after the collision. Thus, in the worst case, it takes six communication delays to learn a value.

Moreover, acceptors in Fast Paxos have to wait for a number of replies that is greater than a majority of nodes in the fast rounds (i.e., $\lceil \frac{3}{4}N \rceil$, a fast quorum).

Generalized Paxos [63] solves Generalized Consensus [63] that is a variant of Consensus generalized to agree on an increasing sequence of commands for less than a permutation of non-conflicting commands. Assuming state machine replication terminology, non-conflicting (commutative) commands may be delivered in any order, while conflicting commands must be totally ordered. Generalized Paxos relies on a distinguished node (leader) to detect command conflicts and enforce an order, and also uses fast quorums as in Fast Paxos. Some of those limitations are removed by the Fast Genuine Generalized Consensus algorithm [110] that is able to use optimal quorums size, but still relies on designated leaders.

EPaxos [77] is a leader-less solution to the generalized consensus problem. EPaxos employs dependency tracking and fast quorums (one smaller than Fast and Generalized Paxos) to deliver non-conflicting commands using a fast path, in two communication delays. In the presence of conflicts however, the protocol takes a slow path and uses four communication delays before delivery.

Chapter 3

Preliminaries: TM, Nesting, TFA

3.1 Primer on Transactional Memory and Nesting

Transactions are a successful abstraction from the database community that give the impression of an atomic execution of a larger piece of code. When a transaction executes successfully, it is said to **commit**. Otherwise, it **aborts** and leaves no evidence that it ever started executing. If a transaction has to abort, it may retry a fixed number of times.

Transactional Memory can be supported in hardware (Hardware Transactional Memory — HTM) [76, 93], in software (Software Transactional Memory — STM) [106] or a combination of the two (hybrid TM) [27]. STM has the unique advantage of being able to run on commodity hardware. The drawback however is a degradation in performance, as reads and writes aren't simple memory operations anymore, but complex functions that implement the TM protocols.

During a transaction's execution *conflicts* may take place. A conflict is said to occur when two transactions try to access the same memory location, and at least one of those accesses is a write.

Conflict detection can take place at different times. Using *pessimistic concurrency control* the conflicts are detected at the time the memory operations are performed, while under *optimistic concurrency control* the detection is postponed and conflicting transactions are allowed to keep running, but not to commit. The two approaches work best in different situations: the optimistic strategy gives better results when conflicts are rare (*low contention* workload) whereas the pessimistic one performs better under *high contention* workloads.

Once a conflict is detected, it has to be resolved. In order to resolve the conflicts, there are two alternatives: (1) *Delay* one of the transactions in order to allow the other one to complete, then continue with its execution. (This only works for eager conflict detection.) (2) *Abort* one of the transactions and retry it later.

In Transactional Memory, *version management* refers to the methods employed by the system for managing writes to the memory. A TM system uses *eager version management* or *direct update* when it writes directly to memory [76]. The previous memory content is recorded in an undo-log, which is later used to *roll back* the transaction in the event of an abort. In such systems, the conflict detection scheme employed must be pessimistic, because write operations that cause conflicts should not be executed.

The alternative is *lazy version management* or *deferred update*. Write operations do not directly affect the main memory, but instead are recorded in a transaction's private redo-log. As a consequence, read operations must also check the redo-log in order to make sure they observe the most recent value of the desired memory location. Upon commit, the changes recorded in the redo-log are saved to the shared memory.

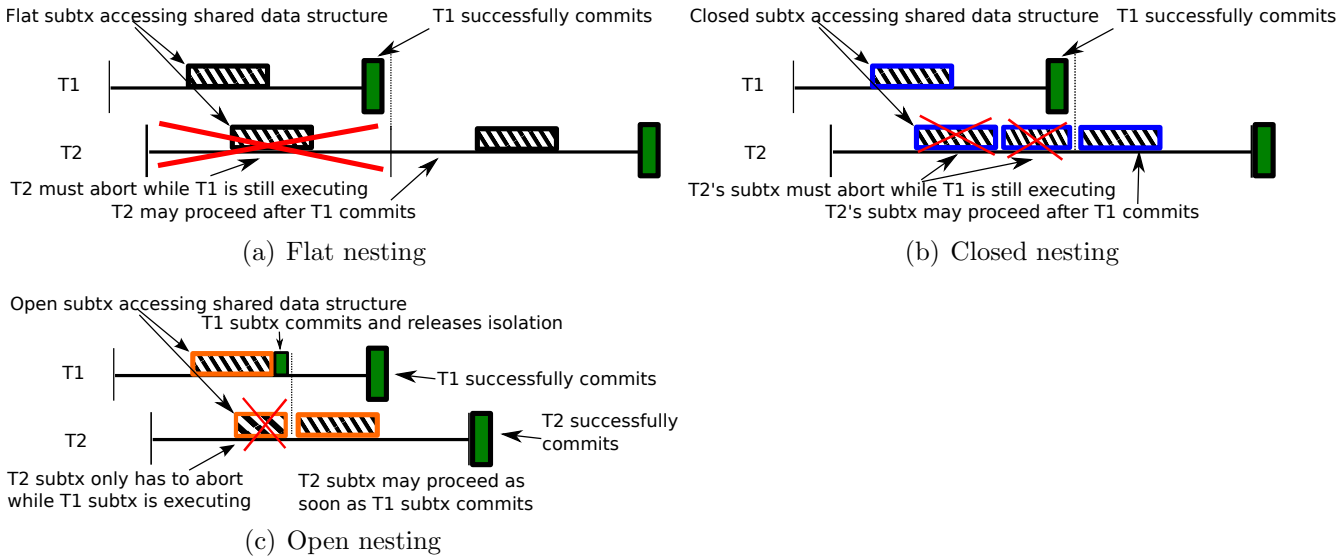
Three types of nesting models have been previously studied [44, 81]: flat, closed and open. They differ based on whether the parent and children transactions can independently abort:

Flat nesting
: is the simplest type of nesting, and simply ignores the existence of transactions in inner code. All operations are executed in the context of the outermost enclosing transaction, leading to large monolithic transactions. Aborting the inner transaction causes the parent to abort as well (i.e., partial rollback is not possible), and in case of an abort, potentially a lot of work needs to be rerun.

Closed nesting
: In closed nesting, each transaction attempts to commit individually, but inner transactions do not publicize their writes to the globally committed memory. Inner transactions can abort independently of their parent (i.e., partial rollback), thus reducing the work that needs to be retried, increasing performance.

Open nesting
: In open nesting, operations are considered at a higher level of abstraction. Open-nested transactions are allowed to commit to the globally committed memory independently of their parent transactions, optimistically assuming that the parent will commit. If however the parent aborts, the open-nested transaction needs to run compensating actions to undo its effect. The compensating action does not simply revert the memory to its original state, but runs at the higher level of abstraction. For example, to compensate for adding a value to a set, the system would remove that value from the set. Open-nested transactions breach the isolation property, thus potentially enabling significant increases in concurrency and performance. However, to be used correctly, logical isolation is still generally required, and the burden for ensuring it now falls on the programmers. Therefore, open nesting must be used with extreme caution, and is generally only recommended for experts.

We illustrate the differences between the three nesting models in Figure 3.1. Here we consider



(a) Flat nesting

(b) Closed nesting

(c) Open nesting

Figure 3.1: Simple example showing the execution time-line for two transactions under flat, closed and open nesting.

two transactions, which access some shared data-structure using a sub-transaction. The data-structure accesses conflict at the memory level, but the conflict is not fundamental (we will explain fundamental conflicts later, in Section 3.2.2), and there are no further conflicts in either T_1 or T_2. With flat nesting, transaction T_2 can not execute until transaction T_1 commits. T_2 incurs full aborts, and thus has to restart from the beginning. Under closed nesting, only T_2's sub-transaction needs to abort and be restarted while T_1 is still executing. The portion of work T_2 executes before the data-structure access does not need to be retried, and T_2 can thus finish earlier. Under open nesting, T_1's sub-transaction commits independently of its parent, releasing memory isolation over the shared data-structure. T_2's sub-transaction can proceed immediately after that, thus enabling T_2 to commit earlier than in both closed and flat nesting. This example assumes the TM implementation aborts the minimum amount of work required to resolve the conflict, thus leading to the maximum performance for each nesting model (in practice, this is accomplished by validating the read operations and determining the minimal set of transactions that should be aborted).

Besides providing support for code composability, nested transactions are attractive when transaction aborts are actively used for implementing specific behaviors. For example, *conditional synchronization* can be supported by aborting the current transaction if a pre-condition is not met, and only scheduling the transaction to be retried when the pre-condition is met (for example, a dequeue operation would wait until there is at least one element in the queue). Aborts can also be used for **fault management**: a program may try to perform an action, and in the case of failure, change to a different strategy (try...orElse). In both these scenarios, performance can be improved with nesting by aborting and retrying only the inner-most sub-transaction.

DTM works can be classified into cache-coherent DTM and cluster-DTM. Cache-coherent DTM [49, 98] maintains copies of the data at the nodes that requires it. A directory protocol is usually employed to locate the primary copy. When a transaction that modifies a data object commits, it invalidates all previous copies of the data and effectively migrates the object to its own node. This approach was proposed by Herlihy and Sun and is called the data-flow execution model [49].

Alternatively, Cluster DTM [22, 12] replicates the data on a set of closely coupled machines. The cluster usually employs a group communication protocol [96], a consensus protocol (i.e., Paxos), or a lease mechanism [59] for ensuring consistency across replicas.

3.2 System Model

As in [49], we consider a distributed system with a set of nodes $\{N_1, N_2, \cdots\}$ that communicate via message-passing links.

Let $O = \{O_1, O_2, ...\}$ be the set of objects accessed using transactions. Each object O_j has an unique identifier, id_j. For simplicity, we treat them as shared registers which are accessed solely through read and write methods, but such treatment does not preclude generality. Each object has an owner node, denoted by $owner(O_j)$. Additionally, they may have cached copies at other nodes and they can change owners. A change in ownership occurs upon the successful commit of a transaction which modified the object.

Let $T = \{T_1, T_2, ...\}$ be the set of all transactions. Each transaction has an unique identifier. A transaction contains a sequence of operations, each of which is a read or write operation on an object. An execution of a transaction ends by either a commit (success) or an abort (failure). Thus, transactions have three possible states: active, committed, and aborted. Any aborted transaction is later retried using a new identifier.

Let $O = \{O_1, O_2, ...\}$ be the set of objects accessed using transactions. Every such object O_j has an unique identifier, id_j. For simplicity, we treat them as shared registers which are accessed solely through read and write methods, but such treatment does not preclude generality. Each object has an owner node, denoted by $owner(O_j)$. Additionally, they may have cached copies at other nodes and they can change owners. A change in ownership occurs upon the successful commit of a transaction which modified the object.

Our implementation executes transactions using the redo-log approach. During the transaction's execution, all object accesses are stored in two temporary buffer called the *read-set* and the *write-set*. At commit-time, if the transaction is still valid, changes are propagated to the shared state.

When an object is read from the globally committed memory (i.e., the shared state), its value is stored in the read-set. Similarly, when an object is written, the value is temporarily buffered in the write-set and does not affect the shared state. Subsequent reads and writes

are serviced by these sets in order to maintain consistency: inside a transaction, two reads of the same object (not separated by a write) must return the same value. On abort, the sets are discarded and the transaction is retried from the beginning. On commit, the changes buffered in the write-set are saved to the globally committed memory.

A detailed description of the basic protocol (TFA) will be given in Section 3.3.

3.2.1 Nesting Model

Our nesting model is based on Moss and Hosking [81]. While their description uses the abstract notion of system states, we describe our model in terms of concrete read and write-sets, as used in our implementation.

With transactional nestings, let $parent(T_k)$ denote the parent (enclosing) transaction of a transaction T_k. A root transaction has $parent(T_k) = \emptyset$. Each transaction may only have one active child, i.e. parallel nested transactions are outside the scope of this dissertation. A parent transaction may execute sub-transactions using any of the three nesting models: flat, closed, or open. We denote this by defining the *nesting model* of any sub-transaction T_k:

$$nestingModel(T_k) \in \{FLAT, CLOSED, OPEN\}$$

Furthermore, root transactions can be considered as a special case of the *OPEN* nesting model.

Let's briefly examine how the four important transactional operations behave in the context of transaction nesting. As mentioned above, each transaction maintains a redo-log of the operations it performs in the form of a read-set and a write-set. Reading an object O_k first looks at the current transaction's (T_k) read and write-sets. If a value is found, it is immediately returned. Otherwise, depending on the transaction's nesting model, two possibilities arise:

- For $nestingModel(T_k) = OPEN$, the object is fetched from the globally committed memory. This case includes the root transaction.
- For $nestingModel(T_k) = CLOSED$, the read is attempted again from the context of $parent(T_k)$.

Read operations are thus recursive, going up T_k's ancestor chain until either a value is found or an open-nested ancestor is encountered. Write operations simply store the newly written value to the current transaction's write-set.

The commit of a closed-nested transaction T_k merges $readset(T_k)$ into $readset(parent(T_k))$ and $writeset(T_k)$ into $writeset(parent(T_k))$. Open-nested transactions commit to the globally committed memory just like root transactions do. They optionally register abort and

commit handlers to be executed when the innermost open ancestor transaction aborts or respectively, commits. These handlers are described in Section 5.2.2.

3.2.2 Multi-Level Transactions

We now introduce the concept of multi-level transactions, which is the theoretical model of open-nesting. Consider a data-structure, such as a set implemented using a skip-list. Each node in the list contains several pointers to other nodes, and is in turn referenced by multiple other nodes. When a (successful) transaction removes a value from the skip-list, a number of nodes will be modified: the node containing the value itself, and all the nodes that hold a reference to the deleted value. As a result, other transactions that access any of these nodes will have to abort. This is correct and acceptable if the transactions exist for the sole purpose, and only for the duration of the data-structure access operations. If however, the transactions only access the skip-list incidentally while performing other operations, aborting one of them just because they accessed neighboring nodes in the skip-list would be in vain. Such conflicts are called *false-conflicts*: transactions do conflict at the memory level, as one of them accesses data that was written by the other. However, looking at the same sequence of events from a higher level of abstraction (the remove operation on a set, etc.), there is no conflict because the transactions accessed different items.

It is therefore desirable to separate transactions into multiple levels of abstraction. By making the operations shorter at the lower memory level, isolation at that level is released earlier, thus enabling increased concurrency. This breaches serializability and must be used with care. In practice, it is sufficient in most cases to ensure serializability at each abstraction level with respect to other operations at the same level, while preserving conflicts at higher levels (i.e., level-by-level serializability [117]). Level-by-level serializability can be achieved by reasoning about the commutativity of operations at the higher level of abstraction. Two such operations are conceptually allowed to commute if the final state of the abstract data-structure does not depend on the relative execution order of the two operations [47]. For example, in deleting two different elements from a set, the final state is the same regardless of which of the deletes executes first. In contrast, inserting and deleting the same item from a set can not commute: which of the two operations executes last will determine the state of the set.

In order to achieve level-by-level serialization, non-commutative higher-level operations, when executed by two concurrent transactions, must conflict. Such a conflict is called *fundamental*, as it is essential for a correct execution. One such mechanism for detecting fundamental conflicts is by using *abstract locks* (locks that protect an abstract state as opposed to a concrete memory location). Two non-commutative operations would try to acquire the same abstract lock. The first one to execute succeeds at acquiring the abstract lock. The second operation would be forced to wait (or abort) until the lock is released. Abstract locks are acquired by open-nested sub-transactions at some point during their execution.

When their parent transaction commits, the lock can be released. In case the parent aborts, however, before the lock can be released, the data-structure must be reverted to its original semantic state, by performing compensating actions that undo the effect of the open-nested sub-transaction. Referring back to the set example, to undo the effect of an insertion, the parent would have to execute a deletion in case it has to abort.

Abstract locks can be used to implement read/write locking, mutual exclusion, or even more complex scenarios, such as compatibility matrices (for encapsulating higher-level reasoning about commutativity of abstract operations, e.g., in [47])

3.2.3 Open Nesting Safety

Multi-level transactions become ambiguous when open sub-transactions update data that was also accessed by an ancestor. As described by Moss [80], TM implementations have multiple alternatives for dealing with that situation (such as leaving the parent data-set unchanged, updating it in-place, dropping it altogether, and others), which may be confusing for the programmers using those implementations. We thus decide to disallow this behavior in TFA-ON: open sub-transactions may not update memory which was also accessed by any of their ancestors. We thus impose a clear separation between the memory locations accessed by transactions at the multiple abstraction levels. This separation should make the usage of open nesting less confusing for programmers. Failure to comply to this rule can easily be caught by the run-time system and the programmer notified.

Furthermore, the open nesting model's correctness depends on the correct usage of abstract locking. Should the programmers misuse this mechanism, race conditions and other hard to trace concurrency problems will arise. For these reasons, previous works have suggested that open nesting be used only by library developers [83] – regular programmers can then use those libraries to take advantage of open nesting benefits.

3.3 Transactional Forwarding Algorithm

TFA [99, 102] was proposed as an extension of the Transactional Locking 2 (TL2) algorithm [31] for DTM. It is a data-flow based, distributed transaction management algorithm that provides atomicity, consistency, and isolation properties for distributed transactions. TFA replaces the central clock of TL2 with independent clocks for each node and provides a means to reliably establish the "happens before" relationships between significant events. TFA uses optimistic concurrency control, buffering all operations in per-transaction read and write sets, and acquiring the object-level locks lazily at commit time. Objects are updated once all locks have been successfully acquired. Failure to acquire a lock aborts the transaction, releasing previously acquired locks and thus avoiding deadlocks.

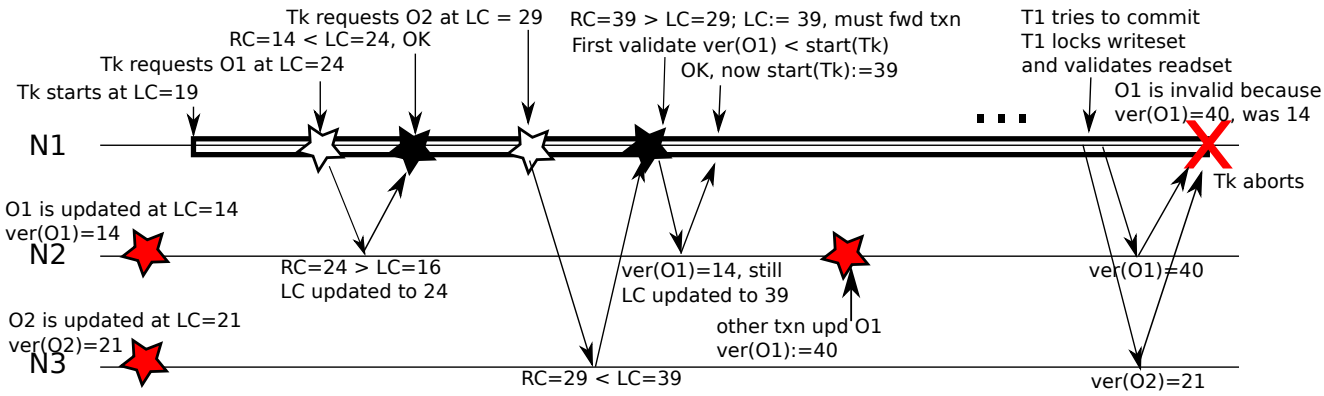


Figure 3.2: Transactional Forwarding Algorithm Example, from [115]

Each node maintains a local clock, which is incremented upon local transactions' successful commits. An object's lock also contains the object's version, which is based on the value of the local clock at the time of the last modification of that object. When a local object is accessed as part of a transaction, the object's version is compared to the starting time of the current transaction. If the object's version is newer, the transaction must be aborted.

Transactional Forwarding is used to validate remote objects and to guarantee that a transaction observes a consistent view of the memory. This is achieved by attaching the local clock value to all messages sent by a node. If a remote node's clock value is less than the received value, the remote node would advance its clock to the received value. Upon receiving the remote node's reply, the transaction's starting time is compared to the remote clock value. If the remote clock is newer, the transaction must undergo a *transactional forwarding* operation: first, we must ensure that none of the objects in the transaction's read-set have been updated to a version newer than the transaction's starting time (early-validation). If this has occurred, the transaction must be aborted. Otherwise, the transactional forwarding operation may proceed and advance the transaction's starting time.

We illustrate TFA with an example. In Figure 3.2, a transaction T_k on node N_1 starts at a local clock value $LC_1 = 19$. It requests object O_1 from node N_2 at $LC_1 = 24$, and updates N_2's clock in the process (from $LC_2 = 16$ to $LC_2 = 24$). Later, at time $LC_1 = 29$, T_k requests object O_2 from node N_3. Upon receiving N_3's reply, since $RC_3 = 39$ is greater than $LC_1 = 29$, N_1's local clock is updated to $LC_1 = 39$ and T_k is forwarded to $start(T_k) = 39$ (but not before validating object O_1 at node N_2). We next assume that object O_1 gets updated on node N_2 at some later time ($ver(O_1) = 40$), while transaction T_k keeps executing. When T_k is ready to commit, it first attempts to lock the objects in its write-set. If that is successful, T_k proceeds to validate its read-set one last time. This validation fails, because $ver(O_1) > start(T_k)$, and the transaction is aborted (but it will retry later).

3.4 SCORe

SCORe [89] is a control-flow based, scalable, one-copy serializable partial replication protocol. It is genuine, as only nodes replicating data touched by a transaction are contacted during the execution and commitment of the transaction. It also allows read-only transactions to commit locally (without any remote communication during the commit phase) by ensuring transactions always read from a consistent snapshot.

SCORe combines a local multi-version concurrency control algorithm with a distributed logical clock synchronization scheme. Each replica holds multiple versions of the objects it maintains, which are tagged with a scalar timestamp. The clock synchronization scheme is used to (a) determine the snapshot visible to transactions, and (b) agree on a final global serialization order for read-write transactions.

All nodes maintain two scalar variables: *commitId* stores the timestamp of the last read-write transaction to commit on that node, and *nextId* holds the timestamp the node will propose at the next commit request. Each transaction is associated with a snapshot identifier (*sid*). The *sid* is recorded at the first read operation within each transaction. It is the greatest of the *commitId* at the current node, and the *commitId* at the node servicing the read (if different). The first read operation in a transaction returns the latest version of the object being read. All further reads may only observe object versions whose tag number is $\leq sid$, in order to maintain a consistent snapshot.

SCORe commits transactions using an algorithm that can be seen as a combination between a Two-Phase Commitment (2PC) and the Skeen total order multicast algorithm [40]. 2PC is used to validate the optimistic execution of update transactions and to ensure the global state is updated atomically. Skeen's algorithm is responsible for agreeing on a final commit ordering across all nodes replicating a certain object. Given that SCORe is a control-flow algorithm, objects are immobile and do not migrate.

Finally, a node's *nextId* is advanced whenever a transaction with a larger *sid* reads from that node. This effectively tracks data dependencies between transactions and ensures that a transaction updating object X is serialized after than all transactions that have observed a previous version of X.

3.5 DTM Frameworks

Two of our proposed algorithms (N-TFA and TFA-ON) were implemented and evaluated in Hyflow [98, 99], a DTM framework for Java. Hyflow's design attempts to be modular by allowing for pluggable support for lookup protocols, transactional synchronization and recovery mechanisms, contention management policies, cache coherence protocols, and network communication protocols. Hyflow extends upon Deuce STM [57] and relies on automatic

byte-code rewriting to provide an API based on annotations, without requiring compiler or JVM support. Hyflow along with its programming interface and its shortcomings will be described in Section 6.1.

A third algorithm (SCORe-ON) is implemented in Infinispan [75]. Infinispan is a popular open-source in-memory data-grid, with support for distributed transactions. Infinispan is highly configurable, extensible but also complex, is supported commercially and is used in production world-wide.

Chapter 4

Closed Nesting

We extend TFA to support Closed Nesting and partial aborts. The resulting algorithm, Nested Transactional Forwarding Algorithm (N-TFA) was implemented in Hyflow and evaluated.

4.1 N-TFA Algorithm Description

In TFA, transactions are immobile. Furthermore, we also consider that all sub-transactions of a transaction T_k are created and executed on the same node as T_k.

Starting from these assumptions, it is straightforward to implement the rules described in Section 3.2.1. Note that there are two types of commit. The original, *top-level commit model* is used when a top-level transaction commits the changes from its replay-log to the globally committed memory. This commit is only performed after the successful validation of all objects in the transaction's read-set, as defined by the TFA algorithm [102]. If the validation fails, i.e. at least one of the objects' version is newer than the current transaction's starting time, the transaction is aborted. The new *merge commit model* is used when a sub-transaction commits the changes from its replay-log to the replay-log of its parent.

A number of questions about how to apply TFA in the context of nested transactions arise. In TFA, every transaction commit increments the node-local clock and updates the affected objects' lock version. Should these operations also be performed upon the commit of a sub-transaction? Which objects should be processed during the early-validation procedure? What is the meaning of transaction forwarding inside a sub-transaction?

By answering these questions, we design a protocol which we will call Nested Transactional Forwarding Algorithm (N-TFA). We must note that two variations of N-TFA could be obtained based on whether merge commits are conditioned by a read-set validation or occur unconditionally. In what follows we will only refer to unconditional merge-commits, because

any extraneous validations proved in our experiments to have high overheads that decreased performance while bringing no benefits.

Assume that transaction T_k opened and read an object O_1. Let T_{k2} be a sub-transaction of T_k. Assume that T_{k2} also reads object O_1, and moreover, T_{k2} can successfully commit (O_1 was not modified by any other transaction). Intuitively, T_{k2} should not update the object's lock version when it commits, because, the object as seen by other transactions did not change. If the version was updated at this point, other unrelated transactions would be forced to unnecessarily abort due to invalid read-set even if T_k eventually aborts (due to other objects) without changing O_1 in the globally committed memory.

In order to maintain similarity with the original TFA, all objects will be validated against the outer-most transaction's starting time. While we could imagine an algorithm where sub-transaction's start times were used to validate objects, doing so would only add unnecessary complexity and would again provide no real benefit. Therefore, all transaction forwarding operations must be operated upon the starting time of the root transaction.

Summarizing the previous two observations, the starting time of sub-transactions is not used for object validity verification and the object versions are not updated upon a sub-transaction's commit. Consequently, merge-commits and the start of new sub-transactions are not globally important events and should not be recorded by incrementing node-local clocks. If the clocks were incremented on such events, remote nodes would need to perform the transaction forwarding operation unnecessarily, only to find that no objects were changed. This is undesirable as the forwarding operation bears the overhead of validating all objects in the transaction's read-set. Additionally, since no global objects are changed at merge-commits, no locks need to be acquired for such commits.

Early validation is the process that checks for the consistency of all objects in a transaction's read-set before advancing the transaction's starting time. If early validation was performed on only the objects in the current sub-transaction (say, T_{k2}), a situation may arise when an object in a previous sub-transaction (say, T_{k1}) becomes inconsistent. In such a case, the parent transaction's clock would be advanced, thereby erasing any evidence that T_{k1}'s object is inconsistent. Thus, early validation must process all objects encountered to date by the outer-most enclosing transaction and all of its children.

In case one or more objects are detected as invalid, the upper-most transaction that contains an invalid object and all of its children should be aborted. In TFA, it was sufficient to stop the validation procedure when the first invalid object is observed. However, with N-TFA, all objects within the root transaction must be validated (ideally in parallel) in order to determine the best point to roll back to.

Let's now look at an example of N-TFA (Figure 4.1). The top-level transaction T_k is executing on node N_1. A sub-transaction T_{k1} executes and commits successfully. Next, another sub-transaction T_{k2} opens an object O_1, which is located on node N_2. T_{k2} spawns a further sub-transaction, T_{k3} which operates on O_1. Assume that at this point sub-transaction T_{k3}

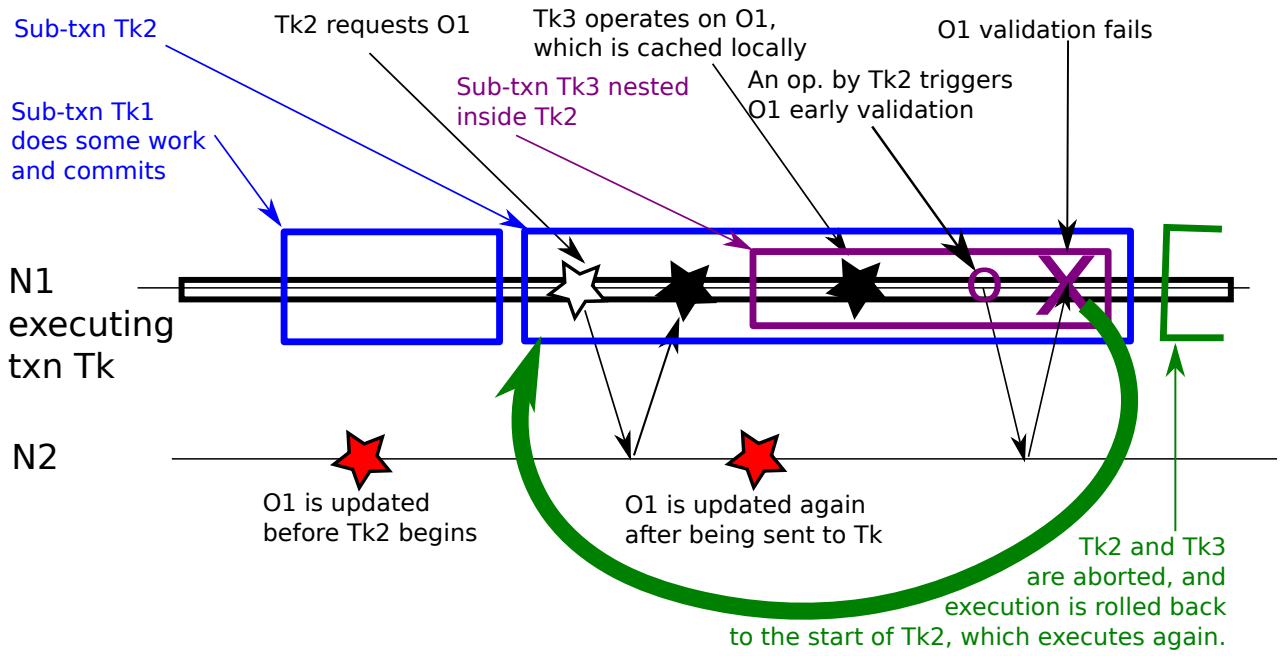


Figure 4.1: Nested Transactional Forwarding Algorithm Example

performs an operation that attempts to validate O_1 (such as an early validation or a merge-commit) and this validation fails. Under TFA, this would abort the root transaction T_k, including the work done by sub-transaction T_{k1}. N-TFA on the other hand only aborts as many sub-transactions are needed to resolve the conflict. In this case, only T_{k2} and T_{k3} need to abort. The transaction will be rolled back to the beginning of T_{k2}, such that the next operation performed is retrieving a new copy of the previously invalid object, O_1.

4.2 Properties

We show that N-TFA maintains the properties of the original TFA, in particular, opacity and strong progressiveness.

Opacity [38] is a **correctness criterion** proposed for memory transactions. A transactional memory system is opaque if the following conditions are met:

- Committed transactions appear to execute sequentially, in their real-time order.
- Any modifications done by aborted or live transactions to the shared state are never observed by any other transaction.
- All transactions observe a consistent view of the system at all times.

Theorem 4.2.1. *N-TFA ensures opacity.*

Proof. The proof for opacity in TFA can be trivially extended to cover N-TFA. The real-time ordering condition is satisfied as shown in [102], because changes made to objects by a transaction are not exposed to other unrelated transactions until the outermost transaction's commit phase, when the ordering is ensured through the usage of locks. Within a transaction, sub-transactions execute serially. There is no need to discuss the ordering of sub-transactions of different top-level transactions: they are effectively invisible to each other.

Uncommitted changes within a transaction are isolated from outside transactions through the use of a write-buffer, just as in TFA. Sub-transactions are executed serially and therefore always observe the correct values. The second condition for opacity is thus satisfied.

Transactions always observe a consistent system state. When N-TFA loads a new object with a version newer than the outermost transaction's starting time, it validates all objects observed by any child sub-transaction. This behavior is identical to the original TFA, satisfying the third condition for opacity. □

Strong Progressiveness is TFA's **progress property**. On a transactional memory system, strong progressiveness implies the following:

- A transaction without any conflicts must commit.
- Among a set of transactions conflicting on a single shared object, at least one of them must commit.

Theorem 4.2.2. *N-TFA ensures strong progressiveness.*

Proof. This follows immediately from the proof that TFA is strongly progressive [102], because the behavior of top-level transactions is identical for both TFA and N-TFA. This is because, sub-transactions as implemented by N-TFA do not introduce any operations that can disturb progress:

- External transactions are not affected because no objects are changed and the node-local clocks are not incremented upon merge model commits.
- Sub-transactions are aborted and retried such that any invalid objects will be re-opened on retry.
- After a validation procedure, no invalid objects will remain in a transaction that does not abort.

□

4.3 Evaluation

We implemented N-TFA in order to quantify the performance impact of closed nesting in the distributed STM environment. We also seek to identify the kinds of workloads that are most appropriate for using closed nesting instead of flat transaction.

4.3.1 Implementation Details

In order to support nesting, we inserted an additional layer of logic between the code of a parent transaction and the code of its sub-transactions. This extra logic handles the partial rollback mechanism and the merge-commits. It was designed to be flexible and to provide support for all three types of nesting: flat, closed and open. While it supports flat nesting and could, in theory, be automatically inserted for every function call within a transaction, doing so would unnecessarily degrade performance.

Instead, we chose to manually insert this logic only in those locations where spawning sub-transactions is desirable. The downside of this approach, at least for now, is that the programmer must acknowledge the difference between regular function calls and closed-nested sub-transactions and write his or her code accordingly. Regular function calls must pass a transactional context variable as an additional parameter (compared to non-transactional code). Methods that spawn sub-transactions do not need any extra parameters, but must include the code implementing the extra logic mentioned above. (Modifying the automatic instrumentation present in both Deuce STM and HyFlow to support this behavior was deemed unnecessary for our research purposes.)

4.3.2 Experimental Settings

The performance of N-TFA was experimentally evaluated using a set of distributed benchmarks consisting of two monetary applications (bank and loan) and three micro-benchmarks (linked list, skip list, and hash table). We record the throughputs obtained when running the benchmarks with the same set of parameters under both closed and flat nesting, and we report on the relative difference between them. Most of our figures relay two values: the average and the maximum. The average value represents multiple runs of the experiment under increasing number of nodes, while the maximum settles on the number of nodes that gives the best results in favor of closed nesting. Unfortunately, we cannot compare our results with any competitor D-STM, as none of the two competitor D-STM frameworks that we are aware of support closed nesting or partial aborts [11, 16].

We targeted the effect of several parameters:

- Ratio of read-only transactions to total transactions (denoted in figure legends with

%).

- Length of transaction in milliseconds (L) is used in some tests to simulate transactions that perform additional expensive processing and therefore take longer time.
- Number of objects (o) is used to control the amount of contention in the system. The meaning of this number is benchmark-dependent.
- Number of calls (c) controls the number of operations performed per test. In closed-nested tests, this directly controls the number of sub-transactions.

Our experiments were conducted using up to 48 nodes. Each node is an AMD Opteron processor clocked at 1.9GHz. We used the Ubuntu Linux 10.04 server operating system and a network with 1ms end-to-end link delay. Each node spawns transactions using up to 16 parallel threads, resulting in a maximum of 768 concurrent transactions. While this number may not seem high, we focused on high-contention scenarios by only allowing a low number of objects in the system.

4.3.3 Experimental Results

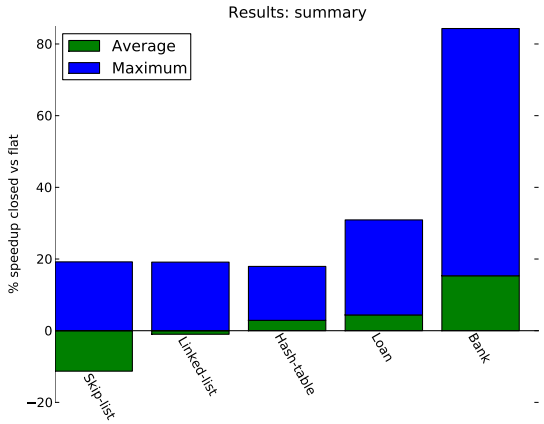


Figure 4.2: Performance change by benchmark.

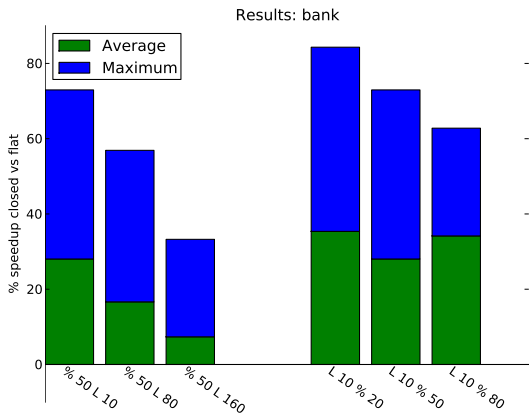


Figure 4.3: Bank monetary application. First group varies transaction length while keeping read-ratio constant. Second group varies the read-ratio for short transactions.

The results of our experiments are shown in Figures 4.2-4.10. Figure 4.2 shows a summary view of the improvement for each of our benchmarks. Figures 4.3-4.9 provide details on each of the benchmarks. Finally, Figure 4.10 looks at the scalability of N-TFA.

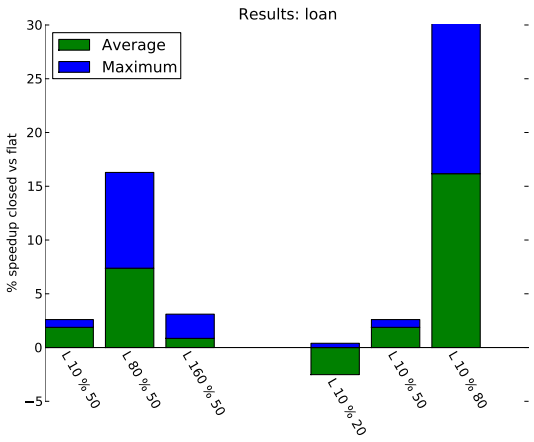


Figure 4.4: Loan monetary application. First group varies transaction length while keeping read-ratio constant. Second group varies the read-ratio for short transactions.

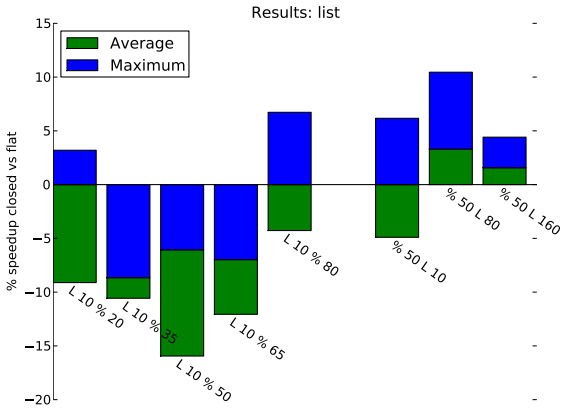


Figure 4.5: Linked-list micro-benchmark. First group varies read-ratio for short transactions. In the second group, transaction length is varied.

The performance of closed nesting varies significantly compared to flat nesting (see Figure 4.2). The single worst slowdown recorded was 42%, while the best speedup was 84%. Across all experiments, closed nesting proved to be on average 2% faster than flat nesting. However, the performance improvements depend strongly on the workload. Within our benchmarks, closed nesting performed worst for Skip-list (10.4% average slowdown) and best for Bank (15.3% average speedup).

These results lead us to believe that in workloads where each transaction accesses many different objects (like in Linked-list and Skip-list), closed nesting will be slower than flat transactions. On the other hand, in workloads where transactions access few objects (like Bank, Loan and Hash-table), greater benefit can be obtained from closed nesting.

The most reliable parameter to influence the behavior of closed nesting appears to be the number of calls. In both Hash-table (Figure 4.6 groups 1 and 3) and Skip-List (Figure 4.8 between groups), we observe that the best performance is achieved with around 2-5 calls per transaction (workload dependent), after which it declines.

The other parameters that we observed (read ratio and transaction length) did not lead to any consistent trends. In some cases, increased read-ratio lead to better performance (e.g. Loan in Figure 4.4 group 2 and Hash-table with $c = 5, o = 7$ in Figure 4.7 group 3). Other cases showed a sweet spot in the middle of the range (Hash-table with $c = 3, o = 7$ in Figure 4.7 groups 2 and 3). Yet other cases show the opposite effects: performance negatively correlated with read-ratio on Skip-list (see Figure 4.9 groups 1 and 3), or worst performance in the middle of the range (Skip-list in Figure 4.9 group 2, Bank in Figure 4.3 group 2

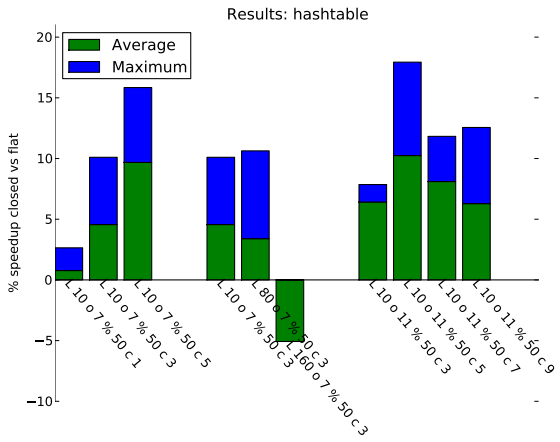


Figure 4.6: Hash-table micro-benchmark. First group shows increasing number of calls on hash-tables with 7 buckets. Second group shows the effect of increasing transaction length. Third group shows increasing number of calls on hash-tables with 11 buckets.

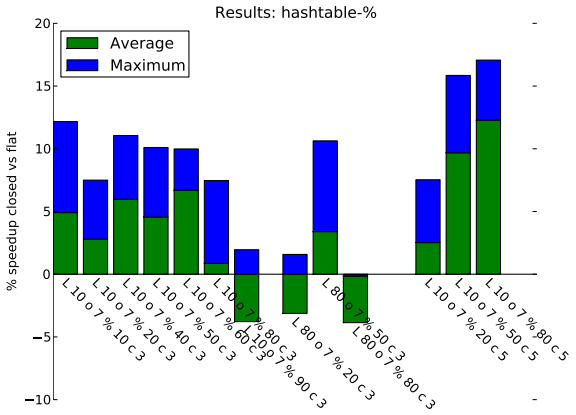


Figure 4.7: Hash-table read-ratio plot. First group shows the effect of increasing read-ratio on short transactions with 3 calls. Second group shows the effect of the same parameter on longer transactions. Third group targets transactions with 5 calls.

and, most obviously, Linked-list in Figure 4.5 group 1). Transaction length has a similar unpredictable influence: negative correlation on Bank (Figure 4.3 group 1) and Hash-table (Figure 4.6 group 2), middle range peak on Loan (Figure 4.4 group 1) and middle range dip on Skip-list (Figure 4.9 group 1).

The *number of objects* parameter was only varied in one benchmark (Hash-table), so we cannot formulate any trends. This parameter did not apply in other benchmarks such as as Linked-list and Skip-list. In our particular case we observe that closed nesting seems to benefit somewhat from the reduced contention enabled by more hash buckets (Figure 4.6 between groups 1 and 3).

From the experiment to evaluate closed nesting's scalability (Figure 4.10), we observe that the performance drops with increasing nodes until about 19 concurrent transactions per object (as seen on Bank in group 1: 12 nodes $\times$ 16 threads / 10 objects). After that threshold, closed nesting performance increases relative to flat nesting.

4.4 Conclusion

We presented N-TFA, an extension of the Transactional Forwarding Algorithm that implements closed nesting in a Distributed Software Transactional Memory system. N-TFA

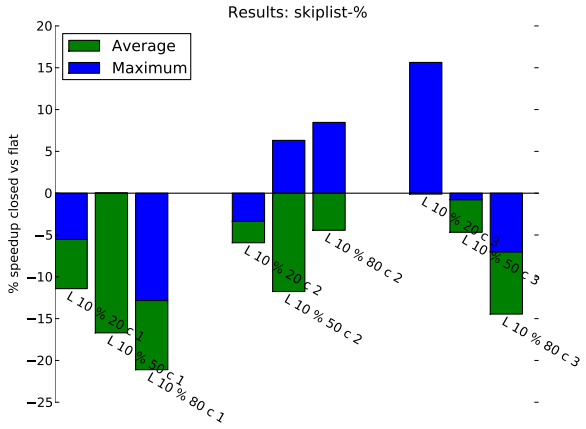


Figure 4.8: Skip-list micro-benchmark. Shows the effect of increasing read-ratio on tests with one, two and three operations performed in a transaction, respectively.

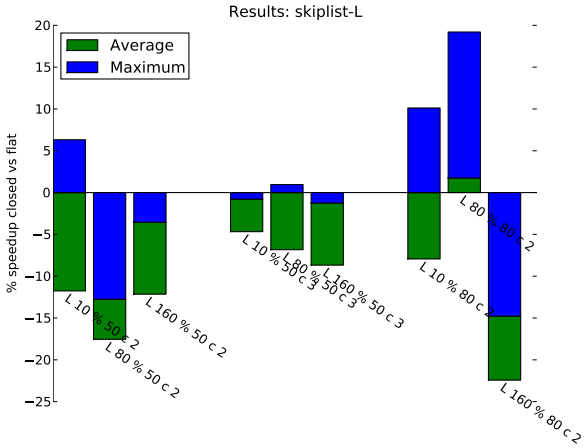


Figure 4.9: Skip-list micro-benchmark. Shows the effect of increasing transaction length. First group contains transactions with 2 calls. For the second group, the number of calls is 3. The last group has 80% reads.

guarantees opacity and strong progressiveness. We implemented N-TFA in the HyFlow DTM framework, thus providing, to the best of our knowledge, the first DTM implementation to support closed nesting. Our N-TFA implementation, although is on average only 2% faster than flat transactions, enables up to 84% speedup in limited cases.

We determined that closed nesting applies better for simple transactions that access few objects. The number of simple sub-transactions is important for the performance of closed-nesting, and we found that N-TFA performs best with 2-5 sub-transactions. N-TFA scales somewhat better than TFA, although the performance dips at around 19 concurrent transactions per object.

Closed nesting however fails as a simple, general purpose method for increasing DTM performance. This is because in two out of five benchmarks, closed nesting is on average slower than flat transactions. In another two benchmarks the average speedup was less than 5%.

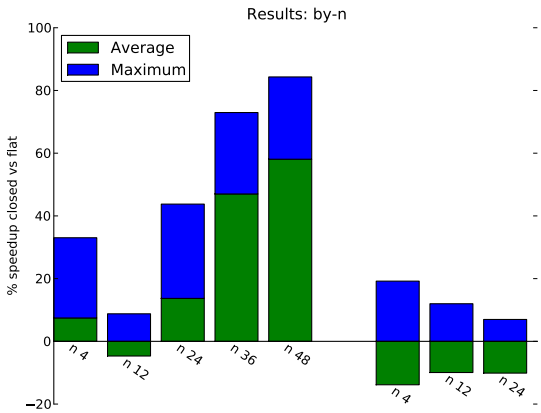


Figure 4.10: Effect of number of nodes participating in the experiment. First group shows the Bank benchmark with 16 threads/node. Second group shows the Skip-list micro-benchmark, with 4 threads/node.

Chapter 5

Open Nesting

We first describe the Transactional Forwarding Algorithm with Open Nesting (TFA-ON), TFA's extension to support open nesting. We then describe key details of its implementation in the HyFlow DTM framework. We proceed to evaluate it experimentally and analyze the results.

5.1 TFA with Open Nesting (TFA-ON)

We describe TFA-ON with respect to the TFA algorithm and N-TFA (Section 4.1), its closed-nesting extension. The low-level details of TFA were summarized in Section 3.3, and we omit them here. In TFA-ON, just as in TFA, transactions are immobile. They are started and executed to completion on the same node. Furthermore, all children of a given transaction T_k are created and executed on the same node as T_k.

Open-nested sub-transactions in TFA-ON are similar to top-level, root transactions, in the sense that they commit their changes directly to the globally committed memory. This affects the behavior of their closed-nested descendants. Under TFA and N-TFA, only the start and commit of root transactions were globally important events. As a result, the node-local clocks were recorded when root transactions started, and the clocks were incremented when root transactions committed. Also, transactional forwarding was performed upon the root transaction itself.

Under TFA-ON, open-nested sub-transactions are important as well: their starting time must be recorded and the node-local clock incremented upon their commit. Closed-nested descendants treat open-nested sub-transactions as a *local root*: they validate read-sets and perform transactional forwarding with respect to the closest open-nested ancestor. Simplified source code of the important TFA-ON procedures is given in Figure 5.1.

When transactional forwarding is performed, all the read-sets up to the innermost open-

```
class Txn {

 // TFA-ON read-set validation routine
 validate() {
  // validate readsets from self until
  // innermost open ancestor
  Txn t = this;
  do {
   if (! t.ReadSet.validate(
          innerOpenAncestor.startingTime ))
    abort(); //validation failed
   t = t.parent;
  } while (t != innerOpenAncestor);
  // validation successful
 }

 forward(int remoteClk) {
  if(remoteClk>innerOpenAncestor.startingTime))
  {validate(); // aborts txn on failure
   innerOpenAncestor.startingTime = remoteClk;
  }
 }

 // TFA-ON commit procedure
 commit() {
  if (nestingModel == OPEN) {
   if ( checkCommit() ) {
    writeSet.commitAndPublish();
    handlers.onCommit();
    parent.handlers += myCommitAbortHandlers;
   } else handlers.onAbort();
  } else if (nestingModel == CLOSED) {
   // merge readSet, writeSet, lockSet and
   // handlers into parent's
  }
 }

 // Called when aborting a transaction due to
 // early-validation/commit failure, etc
 abort() {
  if (! committing)
   handlers.onAbort();
  throw TxnException;
 }

 // acquires locks, validates read-set
 checkCommit() {
  try {
        writeSet.acqLocks();
        lockSet.acqAbsLocks();
        validate();
        return true;
  } catch(TxnException) {
        lockSet.release();
        writeSet.release();
        return false;
  }
 }
```

Figure 5.1: Simplified source code for supporting Open Nesting in TFA's main procedures.

nested boundary must be early-validated. Validating read-sets beyond this boundary is unnecessary, because the transactional forwarding operation that is currently underway poses no risk of erasing information about the validity of such read-sets.

5.1.1 SCORe with Open Nesting (SCORe-ON)

For the most part, SCORe-ON transactions (both parents and open-nesting children) behave similarly to normal SCORe transactions, as described in Section 3.4. However, due to snapshot reads (MVCC) and the fact that SCORe commits read-only transactions differently from read-write transactions, special treatment is needed for the various parent/child combinations. We discus how SCORe-ON handles these combinations bellow:

Read-write parent, read-write child. This is the normal behavior where both parent and child undergo the distributed commitment protocol. The child acquires any needed abstract locks, which get passed to the parent upon the sub-transaction's commit.

Read-only parent, read-write child. In this situation, the parent must be treated as a read-write transaction and undergo the distributed commitment protocol. More specifically, the read-set must be validated at commit time. Failure to do so may allow a sub-transaction to make changes based on stale data, thus breaking serializability.

Read-write parent, read-only child. To ensure correctness in this case, SCORe-ON must acquire abstract locks for all read-only sub-transactions. This guarantees a higher-level read operation can not become stale, potentially leading the parent transaction execute an incorrect write operation. A simple way to implement lock acquisition is as normal DTM read-write operations, effectively transforming the child into a read-write sub-transaction that must undergo commit-time validation. Thus, the snapshot reads optimization can not be applied to any sub-transaction that requires abstract locks. This again is needed for maintaining correctness.

Read-only parent, read-only child. This case is essentially a whole read-only transaction. In SCORe, read-only transactions are executed using snapshot reads and never need to abort. Applying open-nesting semantics to this case would negate this optimization. To avoid this, the programmer should instead use normal flat nesting. If this case is not spotted at design time, the system would unnecessarily acquire abstract locks for read-only sub-transactions, slowing transaction execution and reducing concurrency.

5.2 Mechanisms and implementation

Beyond the necessary protocol modifications as described in TFA-ON and SCORe-ON, several additional mechanisms are needed in order to support open nesting in an actual implementation. These mechanisms relate to dealing with abstract lock management and the

execution of commit and compensating actions, as explained in Section 3.2.2.

5.2.1 Abstract locks

Abstract locks are acquired only at commit time, once the open-nested sub-transaction is verified to be free of conflicts at the lower level. Since abstract locks are acquired in no particular order and held for indefinite amounts of time, deadlocks are possible. Thus, we choose not to wait for a lock to become free, and instead abort all transactions until the innermost open ancestor. This releases all locks held at the current abstraction level.

We implemented two variants of abstract locking: read/write locks and mutual exclusion locks. Locks are associated with objects, and each object can have multiple locks. Our data-structure designs typically delegate one object as the *higher level* object, which services all locks for the data-structure, and its value is never updated (thus never causing any low-level conflicts).

5.2.2 Defining Transactions and Compensating Actions

Commit and compensating actions are registered when an open-nested sub-transaction commits. They are to be executed as open-nested transactions by the innermost open-nested ancestor, when it commits, or respectively, aborts. Closed-nested ancestors simply pass these handlers to their own parents when they commit, but they have to execute the compensating actions in case they abort.

We chose to use anonymous inner classes for defining transactions and their optional commit and compensating actions. Compared to automatic or manual instrumentation, this approach enables rapid prototyping as the code for driving transactions is simple and resides in a single file. Thus, for using open-nested transactions, one only needs to subclass our Atomic<T> helper class and override up to three methods (atomically, onCommit, onAbort). The desired nesting model can be passed to the constructor of the derived class; otherwise a default model will be used. The performance impact of instantiating an object for each executed transaction is insignificant in the distributed environment, where the main factor influencing performance is network latency.

We aimed to make the mechanism for defining open nested transactions consistent across implementations. Specifically, the Atomic<T> acts as a compatibility layer above both Infinispan and Hyflow, and abstracts away the API differences between frameworks — Infinispan uses a map-like interface for accessing data (i.e, get and set), while Hyflow has a directory for keeping track of objects (i.e., open and register). Hyflow's directory implementation was reused in TFA/Infinispan, in order to support object migrations, as required by TFA and the data-flow model. Furthermore, our Atomic<T> layer relieves the user from having to know the actual model currently in use (data-flow or control-flow).

```
new Atomic<Boolean>(NestingModel.OPEN) {
 private boolean inserted = false;
 @Override boolean atomically(Txn t) {
  BST bst = (BST) t.open("tree-1");
  inserted = bst.insert(7, t);
  t.acquireAbsLock(bst, 7);
  return inserted;
 }
 @Override onAbort(Txn t) {
  BST bst = (BST) t.open("tree-1");
  if (inserted) bst.delete(7, t);
  t.releaseAbsLock(bst, 7);
 }
 @Override onCommit(Txn t) {
  BST bst = (BST) t.open("tree-1");
  t.releaseAbsLock(bst, 7);
 }
}.execute();
```

Figure 5.2: Simplified transaction example for a BST insert operation. Code performing the actual insertion is not shown.

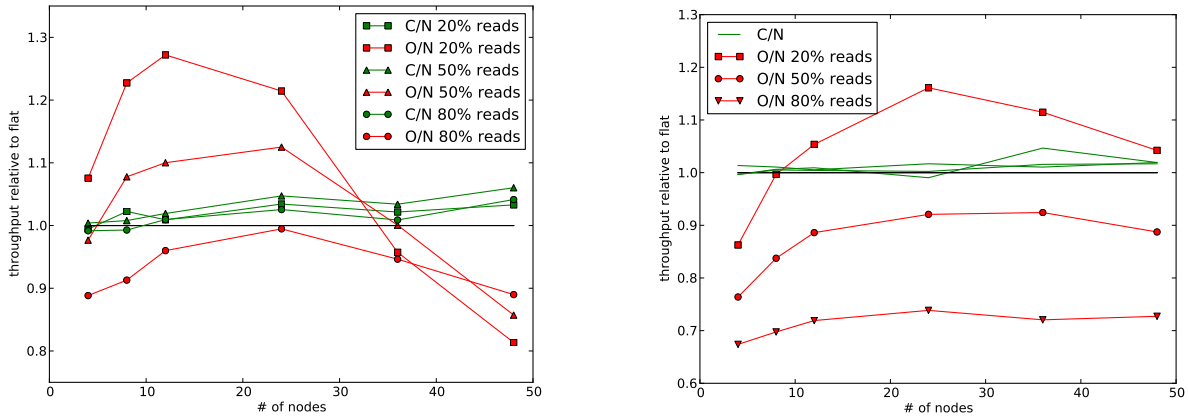
Figure 5.2 shows how a transaction would look in our implementations. Notice how the onAbort and onCommit handlers must request (open) the objects they operate on. They cannot rely on the copy opened by the original transaction, as this copy may be out-of-date by the time the handler executes (automatic re-open may be a way to address this issue).

5.2.3 Transaction Context Stack

Meta-data for each transaction (such as read and write-sets, starting time, etc.) is stored in Transaction Context objects. While originally in HyFlow and Infinispan each thread had its own context object, in order to support nesting, we arrange the context objects in thread-local stacks. Each sub-transaction (closed or open) has a context object on the stack. For convenience, we additionally support flat-nested sub-transactions, which reuse an existing object from the stack instead of creating a new one for the current sub-transaction.

5.3 Evaluation

The goals of our experimental study are finding the important parameters that affect the behavior of open nesting, and based on those, identifying which workloads open nesting performs best in. We evaluate and profile open nesting in our implementation. We quantify any improvements in transactional throughput relative to flat transactions and compare these with the improvements enabled by closed nesting alone. We focus in our study on micro-benchmarks with configurable parameters.



(a) Skip-list

(b) Hash-table

Figure 5.3: Performance relative to flat transactions, with $c = 3$ calls per transaction and varying read-only ratio. Both closed nesting and open nesting are included. (TFA-ON/Hyflow)

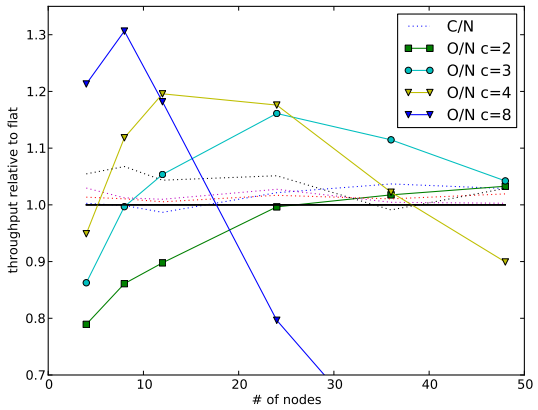
5.3.1 Experimental Settings

The performance of TFA-ON and SCORe-ON was experimentally evaluated using four distributed micro-benchmarks including three distributed data structures (skip-list, hash-table, binary search tree), and an enhanced counter application. Each protocol was implemented in both Hyflow and Infinispan, for a total of four implementations.

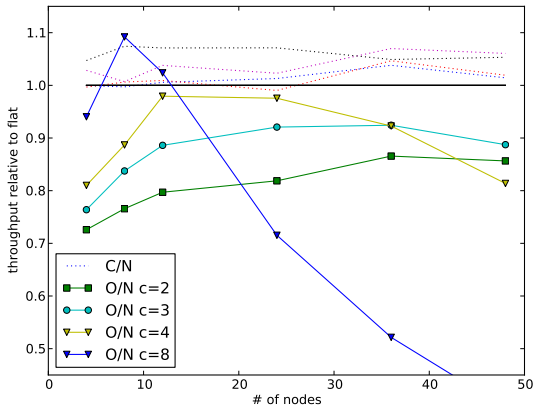
Our evaluation is focused mostly on TFA-ON/Hyflow. Given that Hyflow is our DTM framework research prototype, we were able to easily collect a variety of metrics that allowed us to perform a comprehensive analysis of open nesting behavior. The remaining three implementations (TFA-ON/Infinispan, SCORe-ON/Infinispan and SCORe-ON/Hyflow) were evaluated at a higher level, to confirm that our findings are still valid across different base algorithms and different software frameworks. Unfortunately, we cannot compare our results with any competitor DTM, as none of the two competitor DTM frameworks that we are aware of support open nesting [11, 16].

For TFA-ON/Hyflow, we ran the benchmarks under flat [99], closed [115], and open nesting for a set of parameters. We measured transactional throughput relative to TFA's flat transactions. Each measurement is the average of nine repetitions. Additionally, we quantify how much time is spent under each nesting model executing the various components of a transaction execution:

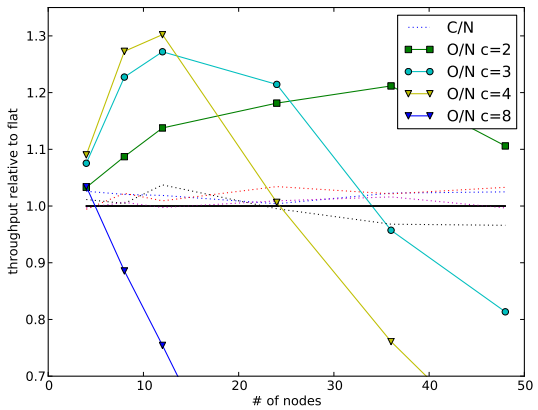
- Committed/aborted transactions.
- Committed/aborted sub-transactions (closed and open nesting).



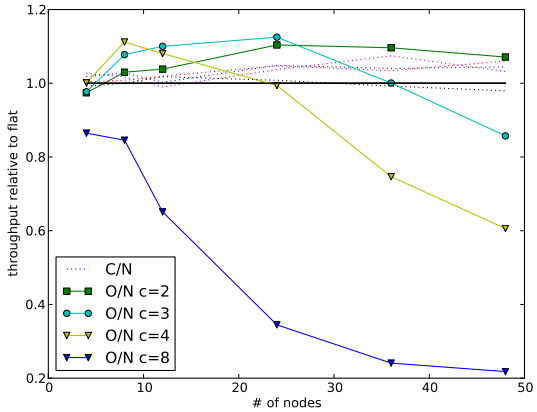
(a) Hash-table 20% reads



(b) Hash-table 50% reads



(c) Skip-list 20% reads



(d) Skip-list 50% reads

Figure 5.4: Performance relative to flat transactions at a fixed read-ratio with varying number of calls. Closed-nesting is depicted, but the individual curves are not identified to reduce clutter. (TFA-ON/Hyflow)

- Committed/aborted compensating/commit actions (open nesting only).
- Waiting time after aborted (sub-)transactions (for exponential back-off).

Other data that we recorded includes:

- Number of objects committed per (sub-)transaction.
- Which sub-transaction caused the parent transaction to abort.

Unfortunately, we cannot compare our results with any competitor DTM, as none of the two competitor DTM frameworks that we are aware of support open nesting [11, 16].

The skip-list, hash-table, and BST benchmarks instantiate three objects each, then perform a fixed number of random set operations on them using increasing number of nodes. Three important parameters characterize these benchmarks:

- Read-only ratio (r) is the percentage of the total transactions which are read-only. We used $r \in \{20, 50, 80\}$.
- Number of calls (c) controls the number of data-structure operations performed per test. Each operation is executed in its own sub-transaction. We used $c \in \{2, 3, 4, 8\}$.
- Key domain size (k) is the maximum number of objects in the set. Lower k values lead to increased semantic conflicts. Unless otherwise stated, we used $k = 100$.

The fourth benchmark (*enhanced counter*) was designed as a targeted experiment where the access patterns of a transaction are completely configurable. Transactions access counter objects which they read or increment. Transactions are partitioned into three stages: the preliminary stage, the sub-transaction stage, and the final stage. The first and last stages are executed as part of the root transaction, while the middle runs as a sub-transaction. Each stage accesses objects from a separate pool of objects. The number of objects in the pool, the number of accesses, and the read-only ratio are configurable for each stage. We also enable operation without acquiring abstract locks, thus emulating fully commutative objects.

Our experiments were conducted on a 48-node testbed. Each node is an AMD Opteron processor clocked at 1.9GHz. We used the Ubuntu Linux 10.04 server OS and a network with $1ms$ end-to-end link delay.

5.3.2 Experimental Results

We start with our TFA-ON/Hyflow experimental study before we move on to the other implementations. For all the data-structure micro-benchmarks, we observed that open nesting's best performance improvements occur at low read-only ratio workloads. Figure 5.3

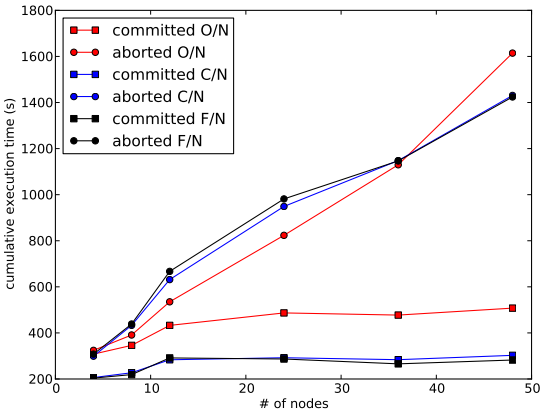
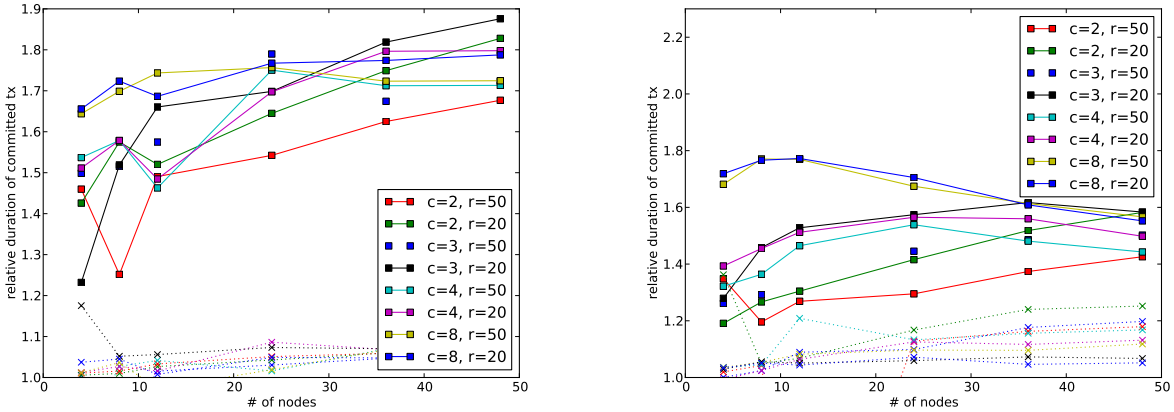
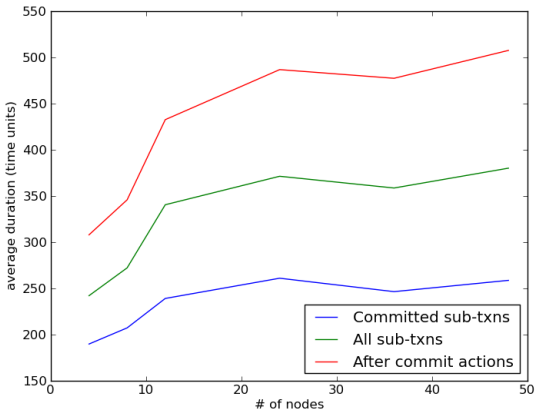


Figure 5.5: Time spent in committed vs. aborted transactions, on hash-table with $r = 20$ and $c = 4$. Lower lines (circle markers) represent time spent in committed transactions, while the upper lines (square markers) represent the total execution time. The difference between these lines is time spent in aborted transactions. (TFA-ON/Hyflow)

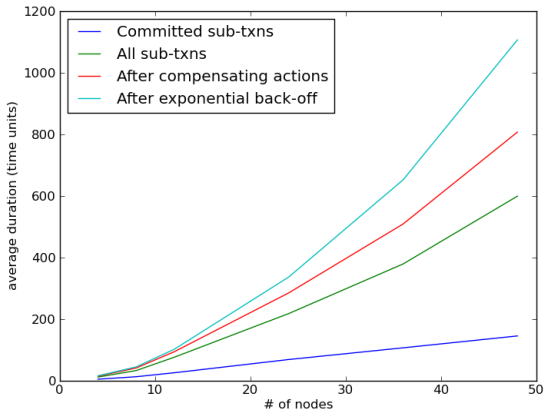


(a) Hash-table (b) Skip-list

Figure 5.6: Overhead of successful open-nested transactions. Plotted is the relative ratio of the average time taken by successful open-nested transactions to the average time taken by successful flat transactions. Closed-nested transactions are also shown, with dotted markers and without identification. (TFA-ON/Hyflow)

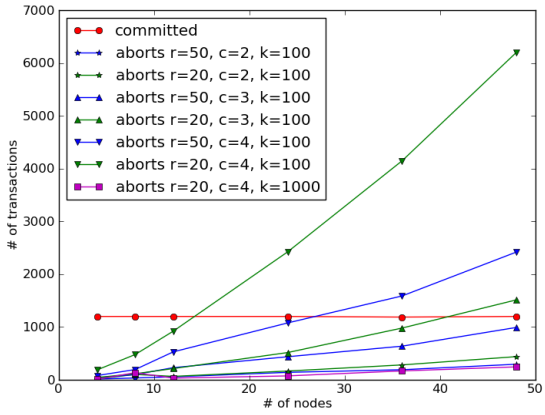


(a) Committed transactions

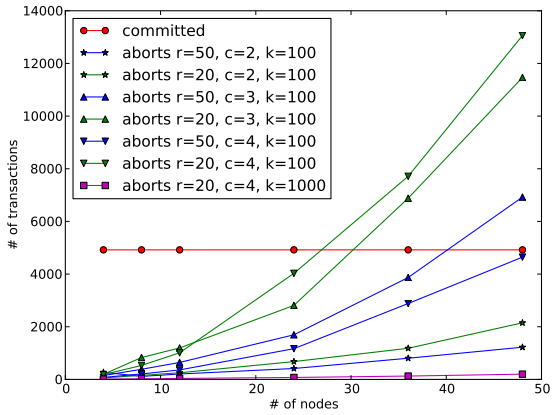


(b) Aborted transactions due to abstract lock acquisition failure

Figure 5.7: Breakdown of the duration of various components of a transaction under open nesting, on hash-table with $r = 20$ and $c = 4$. (TFA-ON/Hyflow)

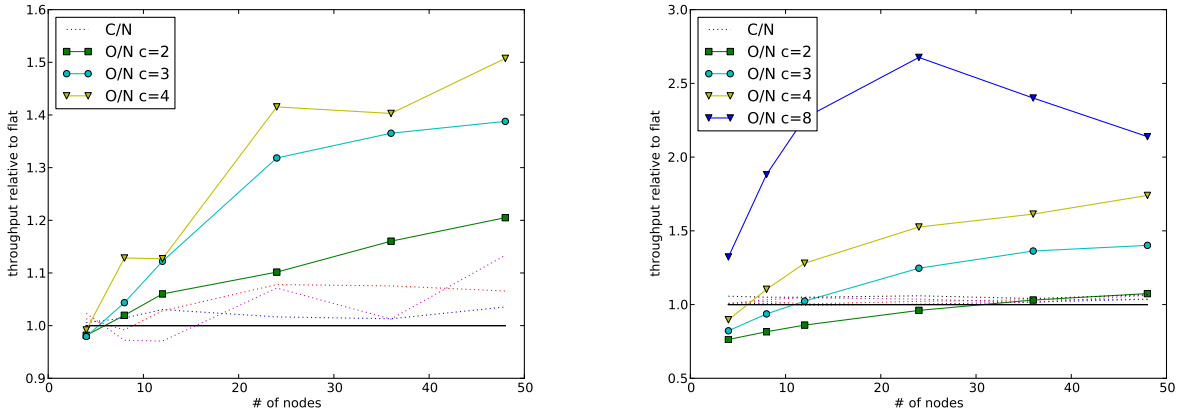


(a) Hash-table



(b) Skip-list

Figure 5.8: Number of aborted transactions under open nesting, with various parameters. The figure shows the effect of read-only ratio, number of calls, and key domain size. Note that all aborts depicted in this plot are full aborts due to abstract lock acquisition failure. The number of committed transactions is fixed for each experiment. (TFA-ON/Hyflow)



(a) Skip-list

(b) Hash-table for r=20

Figure 5.9: Throughput relative to flat nesting with increased key space $k = 1000$ and write-dominated workloads $r = 20$. (TFA-ON/Hyflow)

shows how open nesting throughput climbs up to a maximum and then falls off faster than either flat or closed nesting as contention increases due to more nodes accessing the same objects. Figure 5.3 also shows the effect that read-only ratio has on the throughput. It is noticeable that on read-dominated workloads, open nesting actually degraded performance. Closed-nesting constantly stayed in the 0-10% improvement range throughout our experiments (closed nesting behavior is uninteresting and will henceforth be either omitted from the plots or shown without identification markers to reduce clutter).

Focusing on write-dominated workloads ($r = 20$ and $r = 50$), Figure 5.4 shows how the maximum performance benefit of open nesting generally increases as the number of sub-transactions increases. For more sub-transactions however, the benefit of open nesting occurs at fewer nodes and falls off much faster with increasing number of nodes. The maximum improvements we have observed (with reduced key-domain, $k = 100$) are 30% on skip-list with $r = 20$ and $c = 4$, 31% on hash-table with $r = 20$ and $c = 8$, and 29% on BST with $r = 20$ and $c = 8$ [114]. On skip-list it is noticeable that at high contention ($c = 8$) the region of maximum benefit disappears and the performance decreases monotonously.

These observations can be explained by examining how is the time spent when using open nesting. Figure 5.5 shows how the time taken by successfully committed transactions under open nesting and closed nesting increases at a similar rate. However, open nesting has a significant overhead, caused by the increased rate of commits. This effect is more pronounced in read-dominated workloads, where object updates are rare, and as a result, read-set early-validations under flat-nesting are also rare (early-validations are performed when a commit is detected at another node). In open nesting however, the read-set must be validated for every sub-transaction commit, thus adding multiple network accesses to the cost of success-

ful transactions. Figure 5.6 shows that the average overheads of open nesting relative to flat transactions (50-80% on hash-table and 40-50% on skip-list) are significant and higher than that of closed nesting (3-7% on hash-table and 5-16% on skip-list). We observe the overheads are benchmark dependent, and are lower for workloads which access more objects in every sub-transaction. This is apparent when comparing Figures 5.6(b) and 5.6(a), and further experiments we have performed with higher nodal levels on skip-list [114] confirm our observation.

On the other hand, the time taken by aborted transactions in open nesting (Figure 5.5) is much lower at low node-counts, but increases rapidly for higher node-counts. Examining the average time taken by the various stages of a transaction (Figures 5.7(a) and 5.7(b)), we see that the duration of transactions (committed or aborted) does increase with increasing number of nodes, but this increase is relatively small. Moreover, individual failed transactions consistently take less time than committed ones. Thus, the rapid increase in total time taken by aborted transactions (and therefore a decrease in overall throughput) can only be explained if there is a significant increase in the number of aborts. The data upholds this hypothesis, as shown in Figure 5.8. Note that in our data-structure benchmarks under open nesting, all transaction (full) aborts are caused by abstract lock acquisition failure. With respect to the top-level transactions, abstract locks are acquired eagerly – when the sub-transaction which performed the access commits. When semantic conflicts are frequent, this strategy will cause more aborts and lower performance compared to TFA's strategy, which defers all lock acquisitions to the end of each top-level transaction.

Intuitively, the number of aborts is lower when there are fewer sub-transactions competing for the same number of locks, or when the number of available abstract locks is increased. These effects are also illustrated in Figure 5.8. Increasing the number of calls leads to a rapid increase in the number of aborts. However, the key space k has a more pronounced effect. Setting $k = 1000$ reduced the frequency of semantic conflicts and abstract lock contention. As a result, the number of aborts as compared to other configurations in Figure 5.8 became negligible, and thus the performance increase of open nesting is more stable and more significant than for the cases we previously discussed. In Figure 5.9, we show throughput increase up to 51% on Skip-list (at $c = 4$ and $r = 20$) and up to 167% on Hash-table (at $c = 8$ and $r = 20$). Benefits for open nesting become possible even in non-write-dominated workloads: with $c = 3$ on skip-list, we have found 12% improvement at $r = 80$ and 21% improvement at $r = 50$ [114].

In our enhanced counter micro-benchmark we observed improvements consistent with our previous findings (plot in [114]). However, these improvements only manifested if the root transaction does not experience significant contention after the open-nested sub-transaction commits. Any increase in contention at this stage quickly leads to performance degradation. This result is in agreement with the theory, as open nesting releases isolation early, optimistically assuming the parent will commit. Increased contention after the open-nested sub-transaction contradicts this assumption.

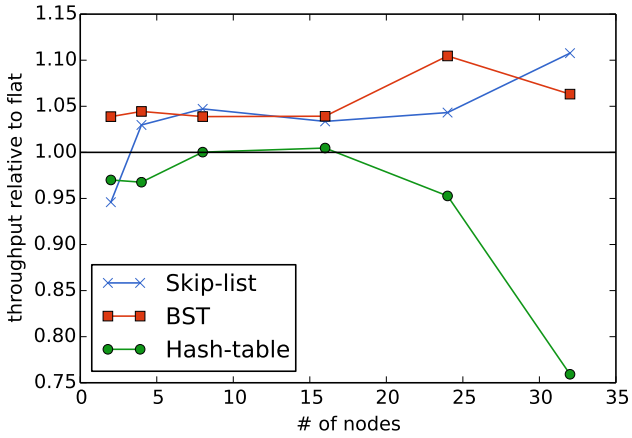


Figure 5.10: Relative throughput for TFA-ON implementation in Infinispan.

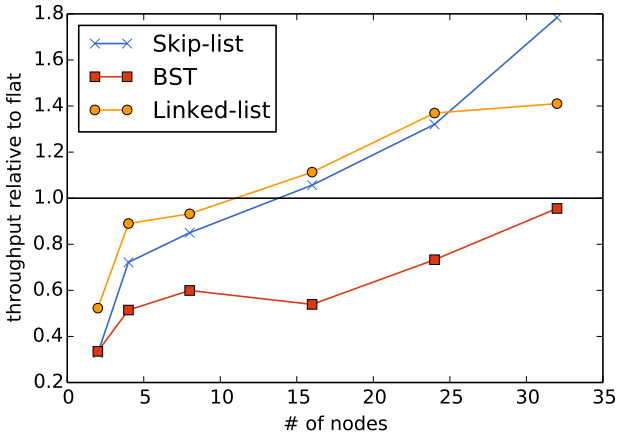


Figure 5.11: Relative throughput for SCORe-ON implementation in Infinispan.

In the context of this benchmark we also briefly experimented with fully commutative objects, by not acquiring abstract locks at all. For our particular case, this resulted in a further 20-30% performance benefit for open nesting. Better improvements are however entirely possible if the post-sub-transaction contention is even lower (in our test, a majority of aborts were caused by post-sub-transaction contention).

The evaluation of our other three implementations are presented in Figures 5.10, 5.11, and 5.12. The absolute numbers differ due to differences in the underlying architecture and benchmark configurations, but the general trends are consistent to those in our comprehensive evaluation of TFA-ON/Hyflow.

More specifically, in TFA-ON/Infinispan (Figure 5.10) the relative throughput sees an initial increase, followed by a drop. The peak throughput is however wider and the slopes in the graph are much gentler. This test was configured with $c = 3$ and $r = 0$.

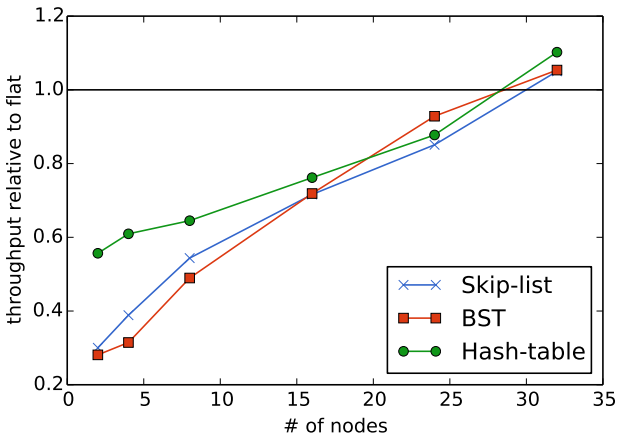


Figure 5.12: Relative throughput for SCORe-ON implementation in Hyflow.

For both SCORe-ON implementations (Figures 5.11, and 5.12), the drop in throughput did not manifest within the range of our experiments. Furthermore, open nesting performs significantly worse at low contention (fewer nodes), which can be attributed by the inherent differences between TFA and SCORe algorithms — since SCORe orders commit operations using a commit queue, the overhead of extra commits in the case of open nesting is greater. SCORe was configured with the same settings as TFA, to make the comparison fair. Objects were distributed across nodes using a consistent hash function (as is standard in Infinispan). The benchmark parameters were $c = 7$ and $r = 0$. We purposefully avoided read-only transactions in these tests, as SCORe-ON would not need to employ open nesting due to the consistent snapshot reads (See Section 5.1.1).

We presented TFA-ON and SCORe-ON, extensions to two DTM algorithms with support for open nesting. We implemented our extensions in the HyFlow DTM framework and the Infinispan data-grid, thus providing (to the best of our knowledge) the first-ever DTM implementations to support open nesting. Our implementations enabled up to 30% speedup when compared to flat transactions, for write-dominated workloads and increased semantic conflicts. Under reduced semantic conflicts workloads, speedup was as high as 167%.

We determined that open nesting performance is limited by two factors: commit overheads and semantic conflict rate. Semantic conflicts limit the scalability of open nesting at higher node-counts, and depend on the available key space for abstract locking. Commit overheads determine the baseline performance of open nesting, at lower node counts, under reduced contention. Commit overheads are significant under read-dominated workloads, and are also influenced by the number of objects accessed in sub-transactions. Furthermore, we confirm that open nesting does not apply well to workloads which incur significant contention after the open-nested sub-transaction commits.

Chapter 6

Hyflow2: A High-Performance DTM Framework in Scala

In this chapter we introduce Hyflow2, our high-performance DTM framework for the JVM, written in Scala. We start by looking at Hyflow and describing our reasons for choosing to rewrite it from scratch. We then present Hyflow2's programming interface, and finally, we evaluate its performance relative to its predecessor.

6.1 The Hyflow DTM framework. Motivation

Hyflow is the original DTM prototype implementing TFA [100]. It was built on top of the Deuce STM library and the Aleph communication framework. Hyflow's modular design attempts to allow for pluggable network transports, transactional algorithms, directory protocols and contention managers. However its interfaces were not abstract enough to allow

```
@Atomic
void transfer(Account a1, Account a2, int amount)
{       withdraw(a1, amount);
        deposit(a2, amount);
}
@Atomic
void withdraw(Account a, int amount) {
        a.value -= amount;
}
@Atomic
void deposit(Account a, int amount) {
        a.value += amount;
}
```

Figure 6.1: Example of the original Hyflow API. Transactions are marked using the @Atomic annotation.

the implementation of more complex algorithms and any possible work around resulted in a source code difficult to maintain.

Hyflow (just like the underlying Deuce STM) relies on automatic byte-code rewriting to provide an API based on annotations. As seen in Figure 6.1, the user marks the methods to be executed transactionally as *@Atomic*. A Java Agent rewrites such methods into two polymorphic copies: the first copy has the same signature as the original method, and it initiates a new transaction (or reuses an already running transaction, if available) and then calls the second copy within the context of this transaction. The second copy is a transacted version of the original method's byte-code. It takes an additional argument (a transaction context), and replaces all field reads and writes with transactional read and write operations. Any method calls within transacted code are modified to also pass the transaction context argument.

The automatic instrumentation also touches on methods not marked as *@Atomic*, by creating an additional transacted copy of the method as described above. When such method is called outside any transaction, the original byte-code is executed. When methods are called within a transaction (by transacted code), the addition of the transaction context argument leads to executing the transacted versions of the methods.

This approach works particularly well for a simple multiprocessor transactional memory system because the instrumented byte-code can be made very fast: no extra objects need to be instantiated (the transactional context object can be reused), method calls can be kept to a minimum (the transactional read and write operations can be inlined), and only one thread-local variable lookup needs to be performed at the beginning of the transaction. However, this model is particularly poor for rapid prototyping, essential for researchers, because of the low-level nature of byte-code instrumentation. Moreover, the potential speed benefits of this model become negligible when dealing with distributed systems, where network accesses are the most costly operations. Modern JVMs with state-of-the-art Just-in-Time (JIT) compilation and garbage collection further minimize the benefits of the byte-code rewriting approach.

Being dissatisfied with Hyflow, we decided to design and implement a better DTM framework, Hyflow2. Our aims for Hyflow2 are as follows:

- High-performance. In order for DTM to have any traction, it needs to be at least similar in speed with the existing systems it aims to replace. Thus, performance is paramount.
- Rapid prototyping. We want our framework to be accessible for DTM researchers, in order to encourage progress in this exciting field. As a side-note, our chosen language for implementing Hyflow2, Scala, is excellent for the purpose of rapid-prototyping.
- Easy to use. Besides being a vehicle for DTM research, we also want Hyflow2 to be used by regular programmers. Thus, an easy, clean and familiar API is desirable.

```
val ctr = Ref(0)
atomic { implicit txn ⇒
  ctr() = ctr() + 1
}
```

Figure 6.2: An example transaction in ScalaSTM (common usage).

```
val ctr: Ref[Int] = Ref[Int](0)
atomic.apply(new Function1[InTxn,Unit] {
  def apply(implicit txn: InTxn): Unit = {
    ctr.update(ctr.apply(txn) + 1)(txn)
  }
})
```

Figure 6.3: A more verbose version of the code in Figure 6.2, with several Scala syntactic shortcuts written explicitly.

6.2 Hyflow2 API

Hyflow2 API is based on the excellent ScalaSTM API. In fact, Hyflow2 tries to reuse ScalaSTM's interfaces wherever possible, and partially implements a back-end for the ScalaSTM API.

6.2.1 ScalaSTM

ScalaSTM is an STM API for Scala under consideration to be included in the Scala standard library in an upcoming release. The API allows for pluggable back-end implementations, and it ships with a reference implementation, CCSTM[13]. Hyflow2 inherits all features described in this section.

Transactions in ScalaSTM are defined using *atomic blocks*, as shown in Figure 6.2. To enable programming using this syntax, *atomic* is a *TxnExecutor* object whose *apply* method takes a function as its only argument and executes this function as a transaction. The "implicit txn =>" construct denotes that the function passed to *apply* takes one implicit argument, the transaction context object.

ScalaSTM uses *transactional references* (*Ref*s) as a container for the values that are to be accessed using transactional semantics. The Ref containers mediate all access to the data within. To access a value of a Ref *ref1* within a transaction, one would use *ref1()* – i.e., call ref1.apply() – or ref1.get() as an alternative syntax. To change the value of the Ref inside a transaction, one should use *ref1() = v* – i.e., call ref1.update(v) – or alternatively, *ref1.set(v)*.

All of these methods (apply, get, update and set in class Ref) take a transaction context object (i.e., an instance of the class *InTxn*) as an additional, implicit argument. Implicit

```
def takeFirst(): T = atomic {
  implicit txn =>
    val old_head = this.head()
    if (old_head == null)
      retry // do not proceed if empty
    this.head() = old_head.next
    return old_head.value
}
```

Figure 6.4: Conditional synchronization using retry. Transaction can only proceed once there is at least one item in the list.

arguments in Scala code may be omitted, as long as the compiler can find in scope a variable of the appropriate type marked with the *implicit* keyword. In Figure 6.2, the *txn* object is automatically passed to the apply() and update() methods. Figure 6.3 shows how Scala interprets the code in Figure 6.2.

This mechanism using implicit arguments and Refs leads to a clean syntax with relatively little redundant code (only the "implicit txn =>" construct and the function call "()" characters are superfluous). Another benefit of this mechanism is protecting against concurrent access of a memory location from both transactional code and non-transactional code. This property is highly desirable in TM systems because in such scenarios, the behavior of interleaving transactional with non-transactional operations is undefined. Accesses to a Ref's contents via the *apply* or *update* methods require an implicit transaction context object to be in scope, otherwise compilation fails. This requirement is satisfied inside an atomic block as explained in the previous paragraph. Outside atomic blocks however, no transaction context value is implicitly available, so calls to *apply* or *update* would lead to compilation errors. Single-operation transactions are used to allow accessing Refs outside atomic blocks. *ref1.single.get()* would, for example, spawn a transaction for the sole purpose of retrieving ref1's value.

ScalaSTM allows temporarily aborting a transaction using the *retry()* method. This is usually used for enforcing preconditions. Suppose for example the *takeFirst* operation on a queue (Figure 6.4). When the queue is empty, this operation may invoke retry, effectively blocking until at least one element is available. This behavior is called *conditional synchronization*. After calling *retry*, the transaction should only execute again once any of the values it has read is updated, otherwise it will follow the same execution path and call *retry* again. A simple implementation may, however, blindly restart the transaction after an exponential back-off, resulting in good performance.

6.2.2 Hyflow2 Objects

While in ScalaSTM transactions operate on Refs directly, Hyflow2 introduces an additional layer – the Hyflow2 Object – as a container for Refs (see Figure 6.5). This layer is needed

```
class Account(val _id: String) extends HObj {
  val type = field("")  // a string field
  val value = field(0)  // an integer field
  Hyflow2.dir.register(this) // Register with the directory manager
}
```

Figure 6.5: Hyflow2 Object example for a bank account.

```
def deposit(accId: String, amount: Int) = atomic {
  implicit txn =>
    val acc = Hyflow2.dir.open[Account](accId)
    val newVal = acc.value() + amount
    acc.value() = newVal
    returm newVal
}
```

Figure 6.6: Hyflow2 transaction example. Transaction must open an object before operating on it.

in order to solve the data distribution problem. On a single node objects can be referred to using a JVM reference, but for multiple nodes, this extra mechanism is required for identifying objects.

An Hyflow2 Object (henceforth referred to as HObj) mixes in the HObj Scala trait [1] and is Hyflow2's basic unit of data. Each Hyflow2 Object has a unique identifier, which Hyflow2 uses to locate the object. This key is usually specified by the user at the object's creation, by passing it as an argument to the constructor.

Each HObj is composed from one or more fields. Fields are specialized Refs that maintain their association with the enclosing HObj and their order number within that object. Fields are created by calling the *HObj.field* method inside the object's constructor, and passing it an initial value.

6.2.3 Hyflow2 Directory Manager

The Directory Manager (DM) is Hyflow2's module that keeps track of the objects' location. When an HObj instance is created, it registers itself with the DM (Figure 6.5). If the object later migrates to a different node, it updates its registration with the DM.

The Directory Manager also handles retrieving objects from their owner nodes over the network. This operation is called *opening* (see Figure 6.6). It requires the identifier of the requested object and it generally caches a copy of the requested object on the local node.

[1] A Scala trait is similar to a Java interface. A class can therefore mix in (i.e., implement) multiple traits. However unlike interfaces, Scala traits may contain implementation.

```
// Simple open-nested transaction without abstract locks or commit or abort handlers
openAtomic { implicit txn =>
  val ctr = Hyflow.dir.open[Counter]("id")
  ctr.value() += 1
}
// Open-nested transaction that acquires a single abstract lock
openAtomic("abslock0") { implicit txn =>
  val ctr = Hyflow.dir.open[Counter]("id")
  ctr.value() += 1
}
// More complex usage case, with abort and commit handlers. Lock is held after commit.
openAtomic { implicit txn =>
  acquireAbsLock("absLock0")
  val ctr = Hyflow2.dir.open[Counter]("id")
  ctr.value() += 1
} onAbort { implicit txn =>
  val ctr = Hyflow2.dir.open[Counter]("id")
  ctr.value() -= 1
} onCommit { implicit txn =>
  holdAbsLock("absLock0")
}
```

Figure 6.7: Open nesting in Hyflow2

6.3 Transaction Nesting

Hyflow2 includes support for nested atomic blocks. In this section we first briefly describe the three nesting models previously studied in TM [44, 81]: flat, closed and open. Next we introduce the API support for nesting in Hyflow2, and explain how it works.

6.3.1 Nesting API

Flat and closed nesting are semantically equivalent and can be used interchangeably (this does not consider parallel nested transactions, which we do not support). Unlike in the original Hyflow, we decided not to expose the decision of which of the two models to use in the regular user-facing API. Hyflow2 may use any of these models to handle nested atomic blocks. Currently, the decision is fixed based on a configuration value, but in the future it could be made adaptively at runtime.

Open nesting on the other hand requires API support. Following the style of ScalaSTM, in Hyflow2 we propose the following syntax (see Figure 6.7):

- An open nested transaction should be started with *openAtomic*. The body of the transaction follows the syntax of regular transactions.
- Following the transaction's body two optional blocks may be specified. These blocks are introduced by *onCommit* and *onAbort*, and represent the transaction's commit and abort handlers, respectively. The handlers themselves are executed as open-nested

```
STM.atomic(new Runnable {
  public void run() {
    Counter ctr = Hyflow2.dir().<Counter>open("ctr")
    ctr.set(ctr.get() + 1);
} } );
```

Figure 6.8: ScalaSTM Java compatibility API.

transactions, so they must accept the implicit transaction context argument. If both handlers are present, their order is not important.

- If an open-nested transaction requires the acquisition of an single abstract lock which is known in advance, the lock's identifier can be passed as a string argument to *openAtomic*. The lock will be acquired before the open-nested transaction can commit, and will be released automatically as part of the transaction's abort and commit handlers. These handlers do not need to be present in the code, the lock will be released anyway (see Figure 6.7).

- For any other abstract lock scenarios, the locks must be acquired within the sub-transaction's body using *acquireAbsLock*. These locks too will be automatically released as part of the sub-transaction's abort and commit handlers.

- If for any reasons an abstract lock should be kept beyond the sub-transaction's commit or abort, *holdAbsLock* must be called in the commit and/or abort handler. Any such lock will be propagated to the innermost open-nested ancestor transaction and will be released upon its commit or abort.

6.4 Java Compatibility API

Scala provides excellent interoperability with Java. As a result, many of the operations described above will just work when invoked from Java code either directly, or in a slightly different form (for example, methods *ref1.get*, *ref1.set*, *Hyflow2.dir.open*, *retry* becomes *Txn.retry*, etc.). Several of the more advanced Scala features that we use in the Hyflow2 API are however not supported from Java code, so we need to provide additional mechanisms to obtain the same results.

6.4.1 Defining Transactions

ScalaSTM already provides a way for starting transactions from Java which uses the *Callable* and *Runnable* interfaces for defining the transaction's body (Figure 6.8). The transaction context argument isn't used anymore – instead, all transactional operations need to dynamically determine the context object at run-time. If no transaction exists for the current

```
new Atomic<Boolean> {
  public Boolean atomically(InTxn txn) {
    Counter ctr = Hyflow2.dir().<Counter>open("ctr");
    ctr.value.set(ctr.value.get() + 1);
    return true;
  }
  public void onCommit(InTxn txn) {
    // Commit handler, omit if not needed
  }
  public void onAbort(InTxn txn) {
    // Abort handler, omit if not needed
  }
}.execute();
```

Figure 6.9: Hyflow2 Java compatibility API using the Atomic class.

```
public class Counter extends jHObj {
  Ref<Integer>  value = field(0);
  public Counter() {
    Hyflow2.dir().register(this);
  }

  // This method is an example transaction. It is not part of the Hyflow2 Object definition.
  public static void increment(final String id) {
    new Atomic {
      public void atomically(InTxn txn) {
        Counter ctr = Hyflow2.dir().<Counter>open(id);
        // The first way of accessing Refs works only from an Atomic class
        // due to the txn parameter
        ctr.set(ctr.get(txn) + 1, txn);
        // The second way of accessing Refs also works using a Runnable
        ctr.single.set(ctr.single.get() + 1);
      }
    }.execute();
} }
```

Figure 6.10: Scala-style Hyflow2 Object definition in Java. Notice how accessing Refs in this style is more verbose.

thread, a single-operation transaction is created automatically. This mechanism, however, does not define the abort and commit handlers required for open-nesting.

To support open-nesting, Hyflow2 provides an Atomic abstract class with three methods: *atomically*, *onCommit* and *onAbort*. User code must subclass it and provide at least the implementation for *atomically* (see Figure 6.9). If implementations are provided for the other two methods, they will be used as commit and abort handlers. Unlike ScalaSTM's Java API, a transactional context object is passed to the transaction as an argument. Our reasons for doing so will become clear in Section 6.4.2.

```
public class Counter extends jHObj {
  Ref.View<Integer>  value = jfield(0);
  public Counter() {
    Hyflow2.dir().register(this);
  }
  // Example transaction
  public static void increment(final String id) {
    STM.atomic(new Runnable {
      public void run() {
        Counter ctr = Hyflow2.dir().<Counter>open(id);
        ctr.set(ctr.get() + 1);
    } } );
} }
```

Figure 6.11: Java-style Hyflow2 Object definition in Java. Compact Ref access.

6.4.2 Defining Hyflow2 Objects

Inheriting from a Scala trait in Java code is non-trivial. To allow a simpler way of defining Hyflow2 Objects in the Java API, we provide an abstract class called *jHObj*, which users must subclass.

Fields may be declared in two ways, which we named the Scala and the Java styles. This decision influences how the fields are later accessed from both Scala and Java code. The Scala way of declaring fields was already described in Section 6.2.2, and only differs cosmetically (see Figure 6.10). However, choosing to declare fields the Scala way makes Java code accessing that field more verbose: either the transaction context object needs to be passed explicitly to each *Ref.get / Ref.set* call (this object is available by sub-classing the *Atomic* abstract class as mentioned in Section 6.4.1), or *Ref Views* must be used to determine the context at run-time by calling *Ref.single.get* or *Ref.single.set* instead of simply *Ref.get* or *Ref.set*. The Scala style of declaring Refs is thus recommended when the application is predominantly written in Scala.

For applications written mostly in Java (or even Java-only), the Java style of declaring fields makes Java code more compact. Fields are declared using *jfield* instead of *field* and their type becomes *Ref.View* instead of *Ref* (see Figure 6.11). Java code can now access the fields using the shorter *ref1.get()*, etc. Note that the actual method invoked is now *Ref.View.get()* and determines the transaction context object dynamically at run-time. When using the Java style, the Scala compiler will not complain if a Ref.View is accessed outside an atomic block. Instead, it would fire a single-operation transaction.

6.5 Mechanisms and Implementation

Our implementation uses the actor model using Akka.

6.5.1 Actors and Futures

Akka is a very efficient actor model implementation for the JVM. The actor model can lead to very fast implementations because it reduces the need for thread context switching. Actor libraries generally do their own user-space scheduling, as opposed to relying on the OS scheduler, and prohibit blocking function calls (such as disk access. etc). Instead, actors send messages to each other and respond to the messages they receive – it is an event-based programming model.

An important part of Akka's interface are *Futures*. Futures represent the result of a computation that is expected to complete at some later time. Futures can be used when a thread sends a request to an actor and expects a response. Instead of waiting for the response to arrive, the method sending the request immediately returns a Future object. The thread can register a callback to be executed when the response is received, query the Future periodically, or even block for the result. Computations can also be composed by chaining or aggregating Futures, thus reducing the number of times a thread needs to block and improving performance. Futures, as well as actors, receive and process messages and events using a configurable thread-pool.

6.5.2 Network Layer

Akka actors provide network transparency. They can seamlessly communicate across JVM and machine boundaries. Actor instances are identified using *ActorRef* objects. *ActorRef*s can be sent across the network while still maintaining their association with the correct actor. *ActorRef*s can then be used on the remote machine to communicate to the original actor. In conjunction with Futures, this makes it easy for developers to handle communication in an efficient manner.

Internally, Akka uses Netty for communicating over the network. Netty is a fast, asynchronous event-driven network application framework. It uses the non-blocking, high performance Java New I/O API. Netty also uses a configurable thread-pool for servicing received messages.

6.5.3 Serialization

Serialization is the process of converting an object to a format that can be sent through the network, and back. Traditionally, Java objects must implement a *Serializable* interface in order to enable this functionality. The standard Java serializer however is notorious for its poor performance. Fortunately, Akka provides an API for custom serializers, so we implemented an adapter for the Kryo library[2]. Kryo is one of the fastest JVM serialization

[2]http://code.google.com/p/kryo/

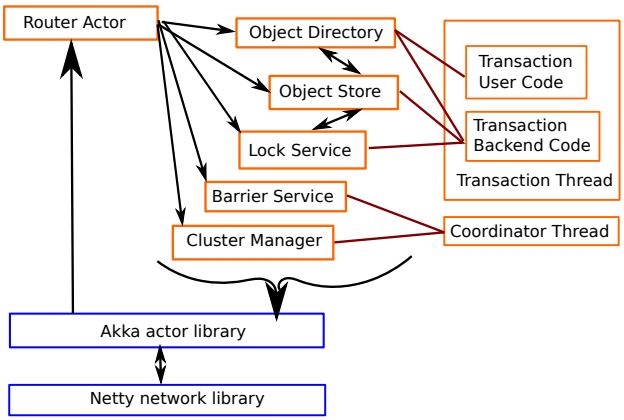


Figure 6.12: Hyflow2 system diagram

frameworks, and is compatible with Scala.

6.5.4 Hyflow2 Architecture

Hyflow2 has a modular architecture. Depending on their function, module implementations need to comply to certain interfaces. Hyflow2 currently provides the following interfaces: lock service, object store, object directory, barrier service and cluster manager. A module implementation consists of a singleton object that complies to one of these interfaces and is used for sending requests to the module and an actor which services such requests. Modules communicate between each other and with the transactions' threads using message passing and Futures.

The lock service module handles acquiring, releasing and verifying the status of object and/or field locks. The object store module holds the objects themselves and handles queries, updates and validations (version checks). Due to their tight coupling, the lock service and object store can be combined in a single module. The object directory tracks object locations: it handles queries, updates, and it can also send notifications to interested transactions when an object is updated. The cluster manager tracks which nodes participate in Hyflow2 transactions, and is currently implemented by delegating a coordinator node (a gossip protocol could be easily integrated for decentralizing the control). The barrier service lets multiple nodes coordinate their execution and is used mostly for benchmarking. An additional module is tasked with gathering statistics from all participating nodes. Figure 6.12 shows a system diagram including Hyflow2 modules and their interactions with the user-code and underlying libraries.

Each node has a router actor which serves as a gateway for all request messages (response messages do not pass through the gateway). The router actor dispatches messages to the

appropriate module based on the message's type (Java class). This design allows every message to contain additional payload data, which can be processed in a consistent way. For example, the Transactional Forwarding Algorithm (TFA) which Hyflow2 implements needs to attach an integer (the node-local clock value) to each message sent over the network [102]. Instead of requiring every module to attach payloads to all the messages they send and receive, payloads are handled automatically in the message's base class constructor on the sender node, and is processed on the receiver node by the router actor.

6.5.5 Conditional Synchronization

Hyflow2 is the first DTM implementation to support distributed conditional synchronization. This feature was implemented by maintaining a waiting list of transactions which are blocked on each object. When they execute, transactions record all objects they access in the transaction's read-set. When a transaction calls *retry*, it adds itself to the waiting lists of all objects which it has previously read, then blocks. Waiting lists are maintained by the Object Directory. When an object is updated, the Directory is notified, and in turn notifies all transactions on that object's waiting list. Because the message adding a transaction to an object's waiting list may arrive after the object is updated, the object version is checked as well: if the transaction is waiting on an old version of the object, the notification is sent right away. Otherwise, a transaction could be waiting unnecessarily for a condition that is already satisfied.

6.5.6 Parallel Object Open

This is another feature provided by Hyflow2 that can speed up certain transactions. Since objects are usually retrieved from remote nodes, the open operation is time-consuming. When a transaction needs multiple objects and knows their identity in advance, it can use the parallel object open operation to reduce the number of network round-trips required for acquiring a copy for each required object.

6.5.7 Transaction Checkpoints

Checkpoints were proposed by Koskinen and Herlihy [58] as an alternate mechanism for partial rollback. As opposed to nesting, where execution can return only to sub-transaction boundaries, checkpoints allow resuming execution from any desired location where a checkpoint was saved. Checkpoints rely on *Continuations*, a programming language mechanism that allow saving and resuming the control state of a program. At their core, continuations work by saving and restoring the CPU registers and the activation stack.

While some languages have varying degrees of support for continuations (e.g. in C one could

use *getcontext/setcontext* or *setjmp/longjmp*), the official Java Virtual Machine does not support this feature. In order to add support for continuations in Java, a number of paths are available:

- Use a library that employs byte-code rewriting, such as JavaFlow, NightWolf or, with modifications, Kilim. Such libraries employ a user-code level activation stack (as opposed to a JVM-level stack) and modify all local variable accesses to explicitly use this stack.
- Use an alternative JVM with support for continuations, such as the Avian JVM.
- Modify the the open-source JVM to support continuations. A patch is available for this purpose in the *Da Vinci Machine Project*.

For Hyflow2 we chose the third approach, as it gives the best performance. While this requires a non-standard JVM, Hyflow2 can run on stock JVM with checkpoints disabled.

6.5.8 Performance

Thread context switches and network round-trip time are important bottlenecks. The choice of libraries we used in Hyflow2 was made with the purpose of addressing these issues. Akka and Netty are event-driven libraries and attempt to minimize thread context switches. We configured their internal thread pools to a minimum size that produces the greatest performance. Also, we specifically targeted serialization in our quest for performance because it lays on the critical path of sending a message over the network.

6.6 Experimental Evaluation

Hyflow2 was evaluated experimentally using a suite of one pseudo-macro-benchmark (bank monetary application) and four micro-benchmarks (counter and the skip-list, linked-list and hash-table data structures). Since we do not seek to evaluate the TFA algorithm but rather the framework's performance, we compare against Hyflow which also implements TFA. Comparisons between Hyflow and other distributed transactional memory libraries implementing different algorithms are available elsewhere [102], and have shown that Hyflow outperforms competitors under most circumstances. Experiments only targeted flat nesting.

Experiments were ran on a testbed consisting of one 48-core and three 24-cores AMD Opteron machines. The operating system used is Ubuntu Linux 10.04 Server. Every node communicates with every other node via TCP links above a Gigabit Ethernet connection. The network is not saturated.

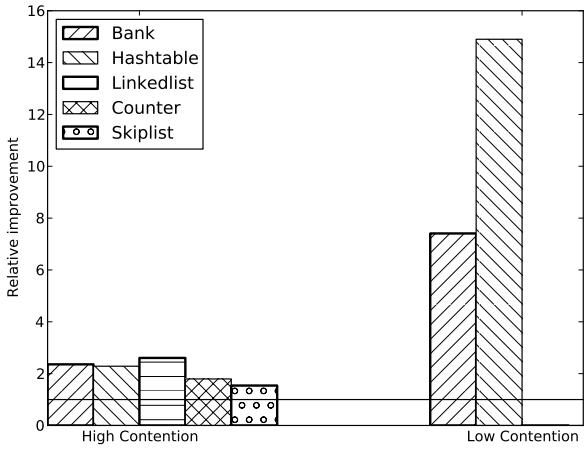


Figure 6.13: Summary of relative performance across benchmarks.

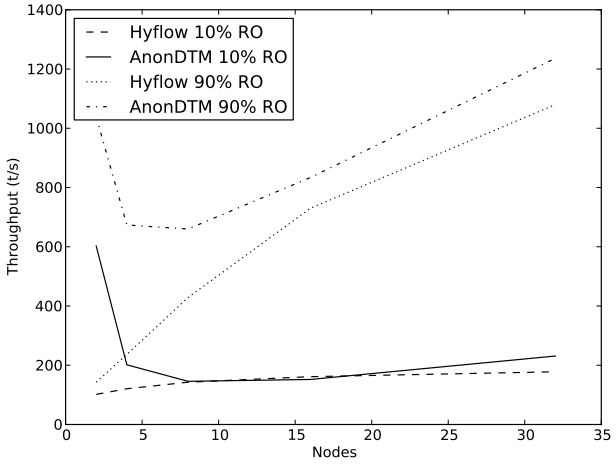


Figure 6.14: Throughput on Bank, for the high-contention workload, for different ratios of read-only transactions.

The JVM used is the 64-bit HotSpot(TM) Server VM. Benchmarks were run with Just-in-Time (JIT) compilation enabled. Each test was allowed a warm-up period to compensate for compilation and class loading overheads before measurement was started.

We evaluated Hyflow2 under two different scenarios. The first is a high-contention scenario. Up to 32 virtual nodes were spawned on a single 48-core machine. Each node is a JVM process whose execution is restricted to a single core, and transactions are spawned using a single thread for each node. Communication takes place via the *loopback* network interface. Each benchmark is configured with a minimal number of objects, such that contention is maximized. The ratio of read-only transactions was also varied between 10%, 50% and 90% reads. While this is not a genuinely distributed environment, it allows emphasizing the efficiency of the implementation.

The low-contention scenario is more realistic, with a large number of objects and many simultaneous transactions. Five nodes were spawned, with each node being allocated 24 cores, for a total of 120 cores. Each node spawns transaction using 96 threads, or 4 threads per each core.

Figure 6.14 provides details on one of the benchmarks, bank. The figure follows the throughput as the number of nodes is increased from two to 32 nodes in the high-contention scenario. Hyflow2 is very fast at a low number of nodes – up to 7 times faster than Hyflow. When the number of nodes is in middle of the range, the performance becomes comparable. Then, as more nodes are added, Hyflow2's performance benefit keeps increasing up to about 30%. Other benchmarks observed slightly different trends. For example, in hashtable, Hyflow2 is 60% faster or more compared to Hyflow throughout all our tests.

In the low-contention scenario, the results are completely different. Hyflow is barely able to maintain a reasonable throughput. After running transactions for about a minute, large pauses cause the throughput to drop significantly (we did not verify but believe this is due to the garbage collector). We therefore limited the experiment's duration such as this penalty is not incurred. Even so, Hyflow2 is about one order of magnitude faster. For example, on bank configured with 10,000 accounts and 50% read-only transactions, Hyflow managed to run 3,324 tps whereas Hyflow2 reached 24,623 tps. Results are summarized in Figure 6.13.

Chapter 7

Automated Data Partitioning in DTM

We develop a framework for automatically partitioning data in a DTM environment which employs Granola independent transactions. Our techniques build upon an existing automatic partitioning methodology, Schism.

Our implementation is based around Hyflow2 (see Chapter 6), a JVM-based DTM framework written in Scala. We implemented the Granola protocol in this DTM framework. Unlike the original Granola implementation, which relies on opaque up-calls from the framework to the application and lets the application code to handle locking and rollback mechanisms, we opted to provide a more friendly API and let the framework deal with these mechanisms.

7.1 Background

We start by giving a brief introduction to the techniques at the root of our work: Granola and Schism.

7.1.1 Granola: Independent Transactions

Granola [24] is a transaction coordination infrastructure proposed by Cowling and Liskov. Granola targets On-Line Transaction Processing (OLTP) workloads. Granola is a Transactional Memory (TM) system, as it expresses transactions in a native programming language and operates on data stored in main memory for performance reasons. Synchronization overheads are kept to a minimum by executing all transactions within the context of a single thread. This approach reduces the need for locking, and was shown to improve performance compared to conventional databases in typical OLTP workloads [109, 43, 53].

Granola employs a novel timestamp-based transaction coordination mechanism that supports

three classes of one-round transactions. *Single-Repository Transactions* are invoked and execute to completion on one repository (partition) without requiring any network stalls. *Coordinated Distributed Transactions* are the traditional distributed transactions that use locking and perform a two-phase commitment process. Additionally, Granola proposes *Independent Distributed Transactions*, which enable atomic commitment across a set of transaction participants, without requiring agreement, locking and with only minimal communication in the coordination protocol.

Single-repository and independent transactions execute in *timestamp mode*. These transactions are assigned an upcoming timestamp, and executed locally in timestamp order. Repositories participating in independent distributed transactions need to coordinate to select the same timestamp. Each participant proposes a timestamp for executing the transaction, and broadcasts its proposal (vote) to the other participants. Once all votes are received, the final timestamp is selected locally as the maximum among all proposals. This selection is deterministic, and the coordination it requires is very light-weight (needs only one messaging round). At the selected time, the transaction can execute without any stalls or network communication.

In order to execute coordinated transactions, the repository needs to switch to *locking mode*. In locking mode, all transactions must acquire locks (thus incurring overheads), and can not use the fast timestamp-based execution. Furthermore, coordinated transactions must undergo a slow two-phase commit. The repository can revert to timestamp mode when all coordinated transactions have completed.

Granola provides strong consistency (serializability) and fault-tolerance. Data is partitioned between the Granola repositories – with each repository managing one partition – although it is also possible to keep some of the data replicated among repositories to improve performance. Each repository consists of one master and several replicas. The replicas are used for fault-tolerance, not for scalability. Most transactions must be executed by the master node of each repository – the only exception is for read-only, single-repository transactions, which can run on the replicas.

In Granola, single-repository and independent distributed transactions never conflict, because they are executed sequentially using a single thread. This means mechanisms employed for rollback and aborts, such as locking and undo- or redo-logging, are not needed for these transaction classes, reducing overheads and improving performance.

Granola transactions do have restrictions that limit their applicability and place further requirements on the potential partitioning schemes: *(A)* Independent transactions must reach the same commit decision, independently, on every participating repository. This is possible when the transaction never aborts (e.g., read-only transactions), or the commit decision is based on data replicated at every participating repository. *(B)* All transactions must be able to complete using only data available at the current repository. This is a firm requirement for single-repository and independent transactions, but could potentially be relaxed for coordinated transactions.

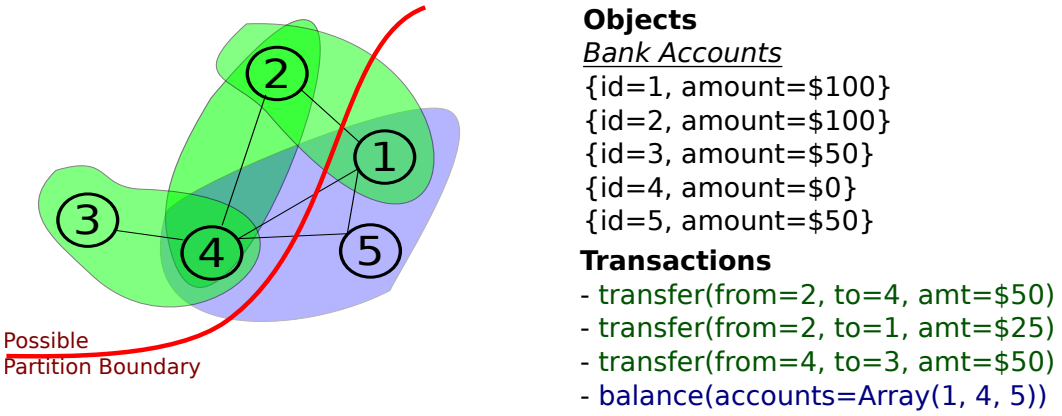


Figure 7.1: Example graph representation in Schism. The shaded areas are the transactions, which are represented in the graph by edges connecting all accessed objects.

Performance in Granola depends on how the workload and partitioning scheme are able to exploit fast single-repository and independent transactions. The user must manually define the partitioning scheme, implement the transactions using the appropriate classes, and route transactions correctly. Furthermore, the partitioning scheme must be compatible with the Granola restrictions outlined above. Our work aims to automate this partitioning process.

7.1.2 Schism: Graph-Based Partitioning

Curino et al. presented Schism [25], the approach for automated data partitioning that we build upon in this dissertation. Besides lacking support for independent transactions, Schism as is can not be applied to stored-procedure style DTM transactions, which further motivates our work. For completeness, in this section we overview Schism and describe how it works.

Schism takes as input a representative workload in the form of an SQL trace, and the desired number of partitions. It then proposes partitioning and replication schemes that minimize the proportion of distributed transactions, while promoting single-partition transactions. This is done in order to increase performance, as single-partition transactions are fast. The proportion of distributed transactions is a measure of the *partitioning quality*. The fewer distributed transactions there are, the higher the partitioning quality. The partitioning process has four phases:

First, the graph representation phase converts the SQL trace into a graph. Nodes in this graph represent data items (database tuples/transactional objects) that were encountered in the trace. Two nodes are connected by an edge if they were accessed together within the same transaction. Thus, the representation of a transaction takes the form of a clique: the tuples accessed by the transaction are all interconnected. An example is shown in Figure 7.1. A number of heuristics are applied to promote scalability, such as tuple and transaction

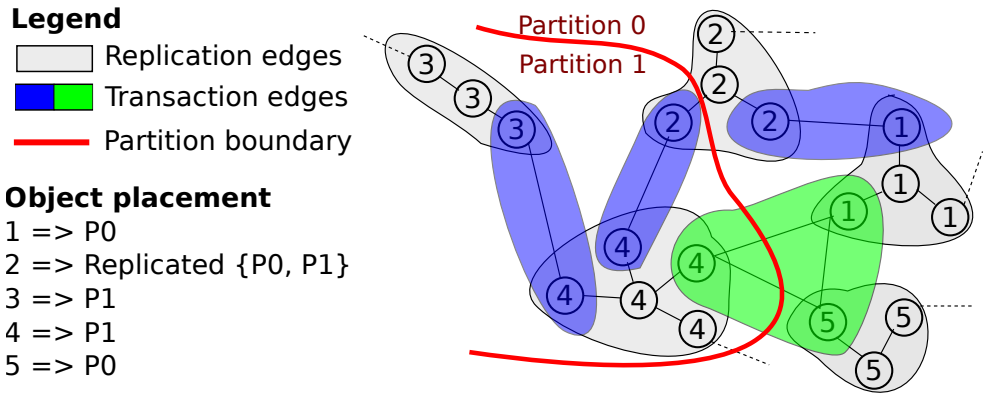


Figure 7.2: Example graph representation in Schism, with replication.

sampling, and coalescing tuples accessed by the same set of transactions into a single node.

The graph is then modified by replacing each node with a star-shaped configuration of nodes. This is done in support for data replication. A node A which previously had n neighbors, is replaced by $n+1$ nodes: one in the center, A_0, which is connected to n new nodes $(A_1...A_n)$ by edges representing the cost of replicating the original node A. Each of these new nodes is then connected by a single edge to another node representing the original neighbors. This processing can also be explained as replacing each edge in the original graph by three edges connected in sequence: the two outer edges represent the cost of replicating the data, and the middle edge represents the cost of entering a distributed transaction. An example is illustrated in Figure 7.2.

In the partitioning phase, the previously constructed graph is partitioned using a standard k-way graph partitioning algorithm. The authors used the program METIS [54] for this purpose. This is an optimization problem, where the primary target is minimizing the cumulative cost of the edges that cut across partitions. This is equivalent to minimizing the number of distributed transactions. A secondary target is balancing the partitions with respect to node weights. Node weights can be assigned based on either data size, or number of transactions, depending on whether the partitions should be balanced in storage size or load.

For small workloads, the output of the partitioning phase can be used as-is, by means of a lookup table. Newly created tuples would initially be placed on a random partition, while a separate background task periodically recomputes the lookup table and migrates data appropriately. This method however can not be applied to large datasets for two reasons: (i) creating and partitioning the graph without sampling is limited by the available memory and processing time, and (ii) the lookup table size is similarly limited by the available memory.

These reasons motivated Schism's explanation phase. In the explanation phase a more compact model is formulated to capture the tuple $\rightarrow$ partition mappings as they were produced in the partitioning phase. Schism does this by employing machine learning, or more specif-

ically, C4.5 decision trees [94] as implemented in the Weka data mining software [41]. The resulting models are essentially sets of range rules, and are useful if they satisfy several criteria: they are based on attributes present as `WHERE` clauses in most SQL queries, they do not significantly reduce the quality of the partitions by misclassification, and finally, they work for new, previously unseen queries, as opposed to being over-fitted to the training set. To satisfy these criteria, the authors employed strategies such as placing limitations on the input attributes to the classifier, using aggressive pruning and cross-validation, and discarding classifiers that degrade the partitioning quality.

Lastly, the final partitioning scheme is chosen in the final validation phase. The candidates considered are (i) the machine-learning based range rules, (ii) the fine-grained lookup table, (iii) a simple hash-based partitioning, and (iv) full-table replication. The scheme chosen is the one with the fewest distributed transactions. In case two schemes lead to similar results, the simpler of the two is chosen.

7.2 Overview

Due to our choice of environment and transaction model, Schism can not be applied directly, because:

- *(A)* Schism does not support independent transactions. Any distributed transactions in Schism would have to be 2PC-coordinated, which degrades performance.
- *(B)* Schism makes no effort to prevent data dependencies across partitions. At best, such dependencies are incompatible with independent transactions. At worst, they are incompatible with Granola's single-round transaction model, leading to unusable partitions.
- *(C)* Schism assumes transactions are expressed in SQL code, whose WHERE clauses can trivially be inspected to obtain information about the dataset of a transaction, which is then used to route each transaction to the appropriate partitions. Given that transactions in our system are not expressed in parsable query code, but are stored procedures written in a programming language, the task of routing transactions becomes significantly more complicated.

7.2.1 Generating Partitions

This section provides a brief description of the partitioning process. In a production system, this process would run periodically alongside transaction processing, and dynamically migrate objects at run-time. Our implementation however, being only a prototype, performs the partitioning off-line.

The first phase in our partitioning workflow performs static analysis and byte-code rewriting on all transactional routines in the workload. This step serves three purposes. Firstly, it collects data dependency information which is later used to ensure the proposed partitioning schemes are able to comply to our chosen one-round transactional model (no data dependencies are allowed across partitions). Secondly, it extracts summary information about what operations may be performed inside each atomic block, to determine whether an atomic block is abort-free or read-only. Finally, each transactional operation is tagged with a unique identifier to help make associations between the static data dependencies and the actual objects accessed at run-time.

The second phase is collecting a representative trace for the current workload, which includes a record for every transactional operation performed. Each record contains the transaction identifier, the type of operation, the affected object, and the operation's identifier as previously tagged.

The next three phases are similar to the corresponding phases in Schism. The graph representation phase converts the workload trace into a graph where nodes represent objects and edges represent transactions. This graph is governed by the same rules as in Schism (see Section 7.1.2). Additionally, edge weights are updated to reflect the new transaction models, along with their restrictions and desirability. The graph is then partitioned using METIS in the partitioning phase. The result from this step is a fine-grained association from object identifiers to partitions. A concise model of these associations is created using WEKA classifiers in the explanation phase.

The final phase is concerned with transaction routing and model selection. While in Schism routing information was easily extracted from the `WHERE` clause of SQL queries when available, our atomic block model for expressing DTM transactions prohibits using a similar approach. We thus introduce a machine-learning based routing phase. The data used to train this classifier is derived from the workload trace, using the object-to-partition mapping. Finally a transaction model is selected for every transaction class based on the number of partitions it needs, whether it may abort, and whether it writes any data (or is read-only).

7.2.2 Using the Partitions at Run-Time

During the previously described process, we train two sets of classifiers. The first set is tasked with object-to-partition mapping. These classifiers determine the object placement, and we will call them the *placement classifiers*. While it may reduce the quality of the resulting partitions, misclassification at this stage is mostly harmless, since it is the classifier that dictates the final object placement.

The second set of classifiers are the *routing classifiers*. They are used on the client side (i.e., in the thread that invokes the transaction) to decide which nodes to contact for the purpose of executing the current transaction. Due to the transactions being expressed as

regular executable code, this information is not readily available until the code is run. Inputs for these classifiers are the parameters passed to the transaction. Misclassification at this stage has the potential to be harmful, as a misrouted transaction may not have access to all objects needed to execute successfully. We address this situation by allowing misrouted transactions to abort and restart on a larger set of nodes.

Finally, we do not require users to be aware of the partitioning scheme or the transaction execution model when writing transaction code. Thus, users should be able to write a single atomic block, and the system would make sure the appropriate code branches will execute at the corresponding partitions. In our prototype implementation, the same code is expected to execute properly on all partitions. This requires a defensive programming style, which checks that the return value of certain object open operations is not *null*. While this is a good practice anyway for error handling, our current implementation explicitly uses *null* references to denote an object is located at another partition.

7.3 Static Analysis

Our static analysis phase is motivated by three factors: (i) determining data dependencies in order to avoid dependencies across partitions, (ii) determining which transactions can abort in order to choose the correct transaction model, and (iii) help with recording workload traces. Simply observing runtime behavior is insufficient — for instance, observing a particular transaction profile never aborted as recorded in a runtime trace does not constitute a guarantee that it can never abort.

Our static analysis phase is implemented using the Soot Java Optimization Framework [62]. Since we operate on JVM bytecode, few of the mechanisms described in this section are actually specific to Scala — transactions could just as easily be expressed in Java, with only simple changes required to the static analysis mechanisms. We make several passes over every application method.

7.3.1 First Pass

The first pass serves three purposes: (i) it identifies transactional methods, (ii) it tags transactional operations, and (iii) it records associations between the classes Scala uses for anonymous functions and their main method which contains the application code.

To identify transactional methods, we iterate over all units of each method (units are Soot's abstraction over the JVM byte-code). We look for invocations of certain methods and references to objects of certain classes that are usually associated with transactions — these are listed in Table 7.1. Methods that match are recorded as *transactional methods*.

In addition to recording transactional methods, we tag units representing invocations to the

Table 7.1: Method invocations and reference types that aid in identifying transactional methods and features (static analysis, first pass).

Method signature / *Ref type*	**Description**
TxnExecutor.apply (block: Function)	Invokes a transaction given an anonymous function as an atomic block. The block can be a top-level transaction, and thus an entry point for the subsequent analysis phases.
TxnExecutor.apply (name: String, args: Array)	Invokes a pre-registered transaction given its name.
Hyflow.registerAtomic (name: String, block: Function)	Registers an atomic block to execute as a transaction when invoked by name.
HRef.apply() HRef.update(val: X)	Reads and writes, respectively, a field of a transactional object. Used to extract data dependencies between objects. Also used to identify read-only transactions.
Txn.rollback() Txn.retry()	Permanently or temporarily aborts a transaction. Used to identify non-aborting transactions and data dependencies leading to an abort decision.
Directory.open(id: Product) Directory.delete(id: Product)	Opens and deletes a transactional object, respectively. Used to extract data dependencies between objects.
HObj, HRef	Transactional objects and fields, respectively. Any references to these types flag the containing method as transactional (and therefore, of interest).
InTxn	Transaction context type. Same as above.

methods in Table 7.1. Tags are a feature in Soot that can associate information with any unit, for easier retrieval. Within the tag we store what kind of transactional operation this invocation represents (e.g., object open, object delete, field read, field write, transaction abort, etc.), and an integer uniquely identifying each invocation site (we name this integer the *tag id*).

Scala uses classes inheriting *AbstractFunctionN*[1] to implement anonymous functions (closures). The application code is usually located in a method named *apply* which takes arguments of the appropriate types. Scala however defines another polymorphic method with the same name, but with arguments of type Object (the root base class on the JVM). This method acts as a stub — its purpose is to convert (typecast or un-box) all arguments to the correct specific type and call the *apply* method containing the application code. For the purpose of our static analysis the stub method is not interesting. We thus record the association between the *AbstractFunctionN*-derived class and the *apply* method containing application code, but only if *apply* is a transactional method as defined above.

[1]Where N is an integer standing for the number of arguments taken by the function.

7.3.2 Second Pass

Once all transactional methods and transactional anonymous function classes are known, we construct a static invocation graph. This is done in the second analysis pass. As before, we pay attention to method invocations, but targeting the previously identified transactional methods. We first add all transactional methods as nodes in the invocation graph. Any invocation of method g from within method f adds to the graph directed edge $f \rightarrow g$.

Besides direct invocations of transactional methods, we also add indirect invocations to the graph. Scala is a functional language and has support for higher-order functions (functions that take other functions as parameters). An invocation site is included in the graph when a previously identified transactional *AbstractFunctionN* object is passed to a higher-order function (either user-defined, or from the standard library: map, filter, etc.). The edge added to the static invocation graph points from the invoking function f to the *apply* method of the transactional *AbstractFunctionN* object, which is invoked indirectly by the higher-order function. Alongside constructing the static invocation graph, all invocation sites (direct and indirect) are tagged as before.

7.3.3 Third Pass

The third analysis pass extracts internal data dependency information for each transactional method. It processes each method, taking as input its bytecode as tagged in passes 1 and 2. The output is a directed graph representing data dependencies between accessed objects and external methods invoked. Firstly, nodes are created in the output graph for important transactional operations that are the targets of the dependency analysis. Such operations are object open, create, delete, transaction abort, and also external method invocations, as tagged in previous steps.

This pass is implemented as a forward data-flow analysis. Each Soot unit has an associated state data-structure that can hold a representation of its dependencies. This representation has two parts: (i) a set of node dependencies and (ii) a set of value dependencies. A node dependency occurs when the result of an important operation (i.e. a transactional object that has been opened) is used in a subsequent statement. Value dependencies occur when any other (i.e., non-node) value is used in a subsequent statement. The latter do not have a presence in the dependency graph, but help propagate dependencies between nodes.

Initially, all the state data-structures are empty. We identify the direct dependencies for every Soot unit, and categorize them into two sets, for node and value dependencies. The value dependencies are traced back to the *origin unit* that defined each of the values. The states associated with the origin units are then retrieved and merged. We further merge this state with a state object formed from the value and node dependencies. Finally, we store the resulting state for the current Soot unit. Pseudocode for this process is shown in Algorithm 1.

Algorithm 1 Forward data-flow analysis pseudocode.

```
for each unit ← allUnits do
    allDeps ← unit.getDirectDeps()
    (nodeDeps, valueDeps) ← allDeps.partition( isNodeDep _ )
    valueDeps_originUnits ← valueDeps.getOriginUnits()
    valueDeps_states ←
        getStateForAll( valueDeps_originUnits )
    merged_valueDepState ← mergeAll( valueDeps_states )
    currentState ← new DepState( valueDeps, nodeDeps )
    newState ← merge( currentState, merged_valueDepState )
    storeStateForUnit( unit, newState )
end for
```

After the data-flow analysis, we construct the dependency graph. Starting with an empty graph, we add nodes for all the units of interest. Then we iterate over all nodes A in the graph, adding edges from B to A, for all node-dependencies B of A.

We illustrate this process in Figure 7.3. The source code to be analyzed is shown in Figure 7.3(a). Notice how many intermediate values are held in variables of their own. This emulates the behavior of Soot, which will indeed use separate locations for every intermediate value, greatly simplifying the static analysis. For clarity, we show a simplified version. In Figure 7.3(b) we show the direct dependencies of each node and value in the code. Following the data-flow analysis, units have an associated state storing all their dependencies, shown in Figure 7.3(c) as the set of all edges pointing to a particular block. Finally, the dependency graph is created by discarding all non-node values (Figure 7.3(d)).

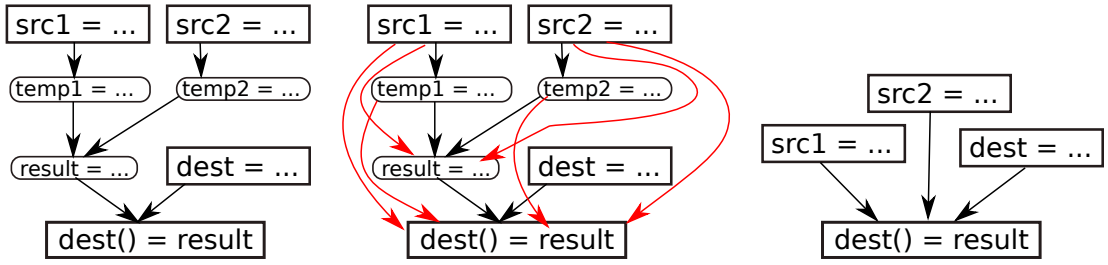
7.3.4 Byte-code Rewriting and Trace Collection

Once all transactional method invocation sites are known and tagged, we rewrite the method byte-code to make certain information available at run-time. For every invocation of a transactional operation (object open, field read/write, etc.), we change the invocation to a different method that acts as a wrapper around the desired operation. This wrapper method takes an extra argument, the *tag id* (i.e., an invocation site identifier), which it logs before passing control to the transactional operation. The tag id is filled in by the byte-code rewriter, as an integer constant.

Other outputs from the static analysis process are the static dependency graphs for all the methods, the global static invocation graph, and a few other details: Method Unique Identifier (MUID) for each transactional method; For each transactional operation invocation: tag id, type of operation; For each transactional method call: tag id, type of method call, MUID and name for the invoked method; For each type of transaction: transaction name, MUID for transaction entry point.

```
val src1 = Hyflow.dir.open[Counter]("source 1")
val src2 = Hyflow.dir.open[Counter]("source 2")
val temp1 = src1.value() * 2
val temp2 = src2.value() * 3
val dest = Hyflow.dir.open[Counter]("dest")
val result = temp1 + temp2
dest.value() = result
```

(a) Analyzed source code.



(b) Direct dependencies (c) Data-flow analysis (d) Final dep graph

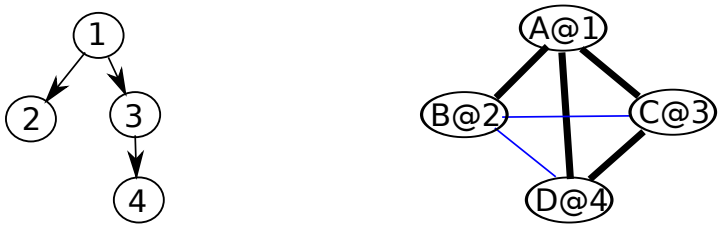
Figure 7.3: Forward data-flow analysis example for extracting the intra-method dependency graph. Rectangles represent nodes (units of interest), rounded rectangles are values.

Next, a representative trace is collected by running the workload using the modified bytecode. This will result in a log of all the transactions executed, and within those, the important transactional operations. Log entries contain:

- Transaction id. Differentiates between multiple concurrent transactions.
- Operation name, such as *atomic* (transaction request), *txn begin/commit/abort*, *obj create/open*, *field read/write*.
- Tag id. Identifies the static invocation site that generated this log entry. Available for *txn abort*, *obj create/open*, *field read/write*.
- Operation specific data. Generally, this is the run-time object id this operation acts upon. For *atomic* and *txn begin*, this is a string representing the transaction type.

7.4 Graph Representation and Partitioning

Once a trace is available, it is parsed and converted to a graph where nodes represent objects and edges represent transactions, as described in Section 7.1.2. A number of heuristics limit the size of the graph, such as object and transaction sampling, and coalescing the nodes



(a) Dependency graph (b) Resulting graph

Figure 7.4: Example partitioning graph for a txn with data deps.

that are accessed by the same set of transactions. Edges are assigned weights such that the resulting partitioning is optimized.

7.4.1 Edge Weights

We now explain the process of assigning edge weights. We aim to satisfy several conditions and optimization criteria: *(i)* Due to the Granola transaction model, we can not easily allow data dependencies between partitions. Make a best effort attempt not to allow such dependencies. *(ii)* When possible, favor independent transactions to coordinated transactions. *(iii)* Favor single-node transactions to any kind of distributed transactions.

To satisfy the first rule, we assign the highest weights to all edges that connect objects having data dependency relationships with each other (*heavy edges*). For example, in Figure 7.4(a) we show the static dependency graph for a transaction. Nodes *1*, *2*, *3* and *4* represent static invocation sites for some transactional operations. At run-time, one execution of this transaction uses objects *A*, *B*, *C* and respectively, *D*, at the four static invocation sites. The system would assign the following *heavy edges*: *A*-*B*, *A*-*C*, *A*-*D* and *C*-*D* (Figure 7.4(b)). They denote the A $\rightarrow$ B dependency, and the A $\rightarrow$ C $\rightarrow$ D chain.

In our current implementation we use a very high weight (10,000) for heavy edges, effectively enforcing that no such edges will be broken. We should note that, with the Granola repository in locking mode, accessing remote objects would be possible, but with a penalty in performance. As such, instead of making heavy edges unbreakable, we could let the optimization process figure out if it may be, in fact, more desirable to break a small number of heavy edges instead of breaking a larger number of lighter edges. Thus our process could be extended with a heuristic that assigns weights to heavy edges based on the workload characteristics, instead of using a large constant as we do now.

The second rule refers to independent transactions as compared to coordinated transactions. These two models differ in that coordinated transactions execute a two-phase commit round, and thus allow reaching the commit/abort decision based on data not available at all repositories. Independent transactions can be used when the transaction does not need to abort,

or reaches a commit/abort decision based on data available to all participating repositories.

To encode this in the partitioning graph, we first identify abort operations. If a transaction does not have any abort operations, it may be executed using the independent transaction model. Thus all remaining edges in such a transaction receive the lightest weight possible (10, we call these *light edges*). On the other hand, if a transaction does have abort operations, we want to encourage replication of all objects that were used in the commit/abort decision, as opposed to entering a coordinated transaction. Thus we use a medium weight (500) for the edges connecting to such objects. We call these *mid-weight edges*.

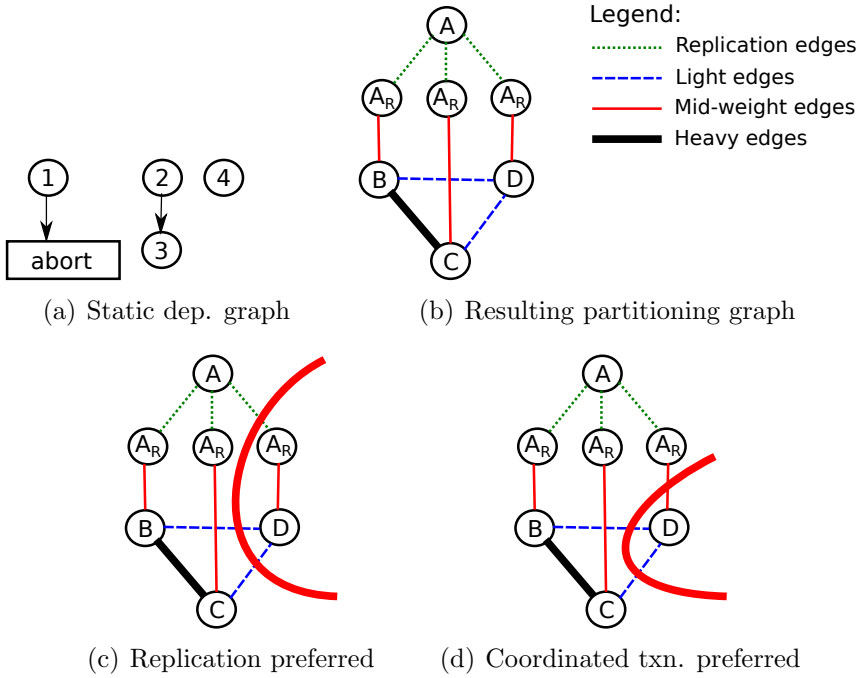
This use-case may lead to replicating an object, even if the object is only accessed by one transaction. This behavior is new to our work, and requires an adjustment to Schism's handling of replicated nodes, which was described in Section 7.1.2. Previously, a replicated node for object A was created for each transaction that accessed object A. With our use-case, it is possible that more than one replicated node is required for the same transaction. This applies for the objects that lead to a commit/abort decision and may be replicated internally.

To better explain this behavior, we provide an example in Figure 7.5. The static dependency graph is shown in Figure 7.5(a). The transaction makes an abort decision based on an object opened at invocation site *1*. Separately, it accesses three more objects (at sites *2*, *3* and *4*), with a data dependency between sites *2* and *3*. Assuming at run-time, the objects accessed are A, B, C and respectively, D, this transaction translates to the partitioning graph shown in Figure 7.5(b). Object A has three replica nodes, one for each other object in the transaction, arranged in a star-shaped configuration. The cost of replication edges are determined based on access patterns to object A throughout the workload. Because object A is used to make a commit/abort decision, its replicas connect to the other objects in the transaction using mid-weight edges $A_R - B$, $A_R - C$, and $A_R - D$. The other edges are heavy or light edges, based on the existence of dependency relationships.

Two possible partitioning schemes are shown in Figures 7.5(c) and 7.5(d). In Figure 7.5(c), object D and one replica of object A are separated from the rest of the objects. This may happen, for example, when object A is rarely written to, and the cost of replicating it is therefore low. In this case, the transaction runs as an independent transaction. Alternatively, Figure 7.5(d) shows a partitioning scheme where only object D is separated from the others. There is no replication of object A, but the transaction must be coordinated.

7.4.2 Partitioning and Explanation

Once weights are assigned, we let METIS solve the optimization problem and propose a partitioning scheme. The result is a fine-grained association from objects to partitions. This can be used as-is only for small workloads. Specifically, we can not use object sampling to keep the problem size small, because the system would not know what to do with objects that do not appear in the mapping. If the problem size increases too much, running time



(a) Static dep. graph (b) Resulting partitioning graph

(c) Replication preferred (d) Coordinated txn. preferred

Figure 7.5: Partitioning graph example in the presence of aborts. The correspondence between the static invocation sites and objects accessed at runtime is: 1-A, 2-B, 3-C, 4-D.

and memory requirements rapidly increase as well.

We thus employ an explanation phase, where we train machine learning classifiers (using the Weka library) based on the fine-grained mapping. As opposed to Schism, we do not need to restrict our classifiers to be rule-based. Instead, we can use any classifier that works best for the current workload. This is possible because we have the whole stack under our control, and thus we do not need to restrict ourselves to what could be encoded efficiently in SQL. Although the current prototype hard-codes a single classifier type, we envision training a forest of classifiers in parallel, and choosing the ones that produce the best end-to-end results.

We train one classifier for each different type of objects. As in Schism, we use virtual partition numbers to represent replicated objects. For example, if there are two partitions in the system, P=1 and P=2, we use P=3 as a virtual partition to represent objects replicated on both partitions.

7.5 Transaction Routing

Our system uses a stored procedure execution model, invoking transactions using the transaction's name and a list of arguments. Not knowing in advance the data each transaction is going to access makes it difficult to determine the partitions each transaction needs to be

routed to. Using a simple directory based approach would be impossible. In Schism, the data a transaction will access is essentially known in advance — one looks at the `WHERE` clause of the SQL query for a quick decision about where to route transactions. This approach does not work in our situation.

Instead, we need to establish a link between a transaction's input arguments and the set of partitions it needs to be routed to for execution. For this, we again turn to machine learning, and employ another set of Weka classifiers. We train these routing classifiers using a workload trace. No static knowledge is used for routing. For each transaction in the trace, we want to route to at least the following partitions:

- Partitions that replicate any object in the writeset.
- A minimal set of partitions R, such that for any object X in the read-set, at least one partition $P \in R$ replicates object X.

Finding R is known as the *hitting-set problem*, which is NP-complete. Algorithms exist that approximate R, but are exponential in time [7]. We compute an approximation of the set R using a simple heuristic (Greedy), and we use that approximation to train our classifiers. This will be the output of the classifier. The input to the classifier is the list of arguments being passed to the transaction.

In our current implementation, we let the clients route transactions as they issue them. This is acceptable in a DTM environment where clients and servers are co-located. If clients can not be trusted with the identity of the servers, or the servers are located behind a firewall, it would be possible to employ a dedicated router/gateway process.

Classifiers do not always yield 100% accuracy. Misclassification at the routing stage may mean more nodes are contacted than strictly necessary, which is a benign situation. However, it is also possible that not enough nodes are contacted to allow completing the transaction. In such a situation, the transaction should abort on all currently participating nodes, and restart on a superset of the nodes. Algorithm 2 describes how to handle this situation (our prototype does not implement this mechanism yet).

The primitive to be used when executing a transaction is decided after the transaction has been routed. If only one repository is involved, the single-repository model will be chosen. For distributed transactions that do not explicitly abort (as identified in the static analysis phase) the independent transaction model is chosen. All other transactions use the coordinated model. This approach can be further refined by determining whether the decision to abort is made based on data available at all nodes. If so, an independent transaction can be used.

To summarize, we have developed a methodology for using automatic data partitioning in a Granola-based Distributed Transactional Memory. We perform static byte-code analysis to determine transaction classes that can be executed using the independent transaction model.

Algorithm 2 Proposal for handling misrouted transactions.

```
N_CRT = the set of nodes participating in this transaction
upon open(X) = failed do
                                  ▷ find N_REPL, the set of nodes that replicate object X
    N_REPL ← placement classifier (X)
    if N_REPL ∩ N_CRT ≠ ∅ then
        return                    ▷ X can be processed on a different node and
                                  ▷ the transaction can continue normally
    end if
    if current txn may write to X then        ▷ from static analysis
        Restart txn on N_CRT ∪ N_REPL
    else
        Restart txn on N_CRT ∪ ANY(N_REPL)
    end if
```

We also use the analysis results to propose partitions that promote independent transactions. Due to our DTM focus, we take a machine-learning approach for routing transactions to the appropriate partitions.

Chapter 8

Evaluating Automated Data Partitioning in DTM

8.1 Experimental Setup

We evaluate our partitioning process using several popular On-Line Transaction Processing (OLTP) benchmarks: TPC-C [23], TPC-W, AuctionMark, EPinions and ReTwis. These workloads (especially TPC-C) have been employed in many recent works [111, 53, 24], but while these works assume a manual partitioning, we employ our system in order to automatically derive the partitioning schemes. In our evaluation, we run transactions back-to-back using the Granola model and compare the manual partitioning against our automatic partitioning for each benchmark. We reiterate that the Granola model requires data partitioning and can not function without it. Additionally, we compare against PaxosSTM [56], a modern state-machine replication based DTM system that does not support partitioning, and Horticulture [86] (the automatic partitioner of H-Store [53]), where available.

Throughput measurements were obtained on the NSF PRObE testbed [35] with up to 20 physical machines. Each machine has 64 AMD Opteron 6272 cores running at 2.1GHz, and

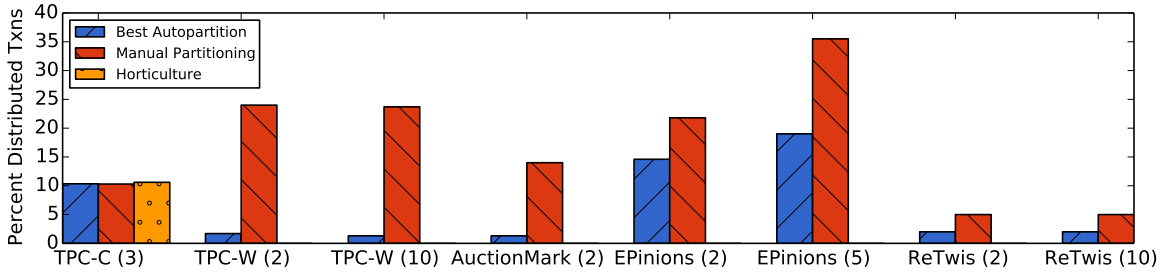


Figure 8.1: Partition quality on the various benchmarks (lower is better).

128GB RAM. Machines communicate over a 40GB Ethernet network. Experiments were allowed sufficient time for warming-up before the measurement was started. Data points represent the average across eight measurements. Experiments were conducted without batching to avoid differences in throughput due to details in the batching implementation (batching is an orthogonal feature).

PaxosSTM supports snapshot reads, an optimization in the execution of read-only transactions. This optimization depends on the use of multi-versioning and full replication in the underlying protocol, and can negatively affect the freshness of the data available to read-only transactions. Snapshot reads are not available in Granola because Granola does not offer multi-versioning or full replication.

8.1.1 Benchmarks

Our work is the first to apply Granola independent transactions to any other benchmark besides TPC-C. In doing so, we show that independent distributed transactions can be used in a wide range of OLTP workloads. In the rest of this section we discuss the benchmarks we used in our evaluation and why the Granola transaction model is appropriate in each case. We also describe the manual partitioning schemes that we used in our evaluation. In deriving the manual partition schemes, we took inspiration from the TPC-C partitioning, which is known to be optimal. While we can not claim our partitions are optimal, they are still effective at minimizing distributed transactions.

TPC-C emulates a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. It consists of 9 kinds of objects (i.e., *tables*, in database terminology) and 5 transaction profiles (i.e., stored procedures). All objects can be grouped based on the parent warehouse, except items, which are global. The manual partitioning scheme we used is known to be optimal. It groups each warehouse and all related objects, distributing the groups across partitions and replicating all items at all partitions. Using this scheme, Granola is able to execute all transaction profiles as either single-node or independent transactions [24]. This is because the only transactions that ever abort (NewOrder) do so based on data available at all partitions (the Item objects), and thus distributed coordination is never required.

TPC-W emulates an e-commerce website. It consists of 8 kinds of objects and 14 transaction profiles. None of the transaction profiles ever abort, and can thus be trivially executed using single-node or independent transactions. One of the frequently used transaction profiles (BestSellers) is very long and would appear poorly suited for Granola's single-thread execution model. However, in a production setting, the results of such a transaction would be cached outside the repository, amortizing the impact of long transactions. The manual partitioning scheme we adopted replicates *item*, *author* and *country* objects at all partitions, and groups each *customers* with all related objects, distributing (sharding) groups across all partitions. This is similar to the TPC-C scheme, with the customers treated like the TPC-C

Table 8.1: Per phase running time, on TPC-C with 15 warehouses and a 89MB input trace containing 42k txns.

Tuple-level sampling rate	Creating graph from txn trace	METIS partitioning	Train placement classifiers	Compute partitions & train routing classifiers
5%	1m56	26s	22s	2m51s
10%	3m55	1m01s	37s	7m30s
20%	9m49	1m44s	1m02s	6m18

warehouses.

AuctionMark simulates the workload of a popular auction website. The benchmark is complex and consists of 13 kinds of objects and 14 transaction profiles. Only one of the transaction profiles (UpdateItem, representing 2% of the workload) needs to abort. This transaction would need to execute using 2PC coordination when it accesses data from multiple partitions. This leaves the remaining distributed transactions (12% of the workload) able to execute using the independent transaction model. The manual partitioning scheme we used groups each *user* together with all the related objects (including the *items* a user sells) – users are then sharded across all partitions.

The EPinions benchmark simulates the popular website with the same name. This workload manages *users* and *products*, as well as the many-to-many relationships between them (*trust* and *reviews* encode the relationships between two users, and between an user and a product, respectively). There are 5 kinds of objects and 9 transaction profiles. None of the transaction profiles abort, thus independent transactions can trivially be used for all distributed transactions. The manual partitioning scheme we adopted for this workload replicates everything everywhere, thus executing all write transactions as distributed transactions, while executing read transactions locally. For a read-dominated workload, this scheme results in fewer distributed transactions when compared to a scheme that uses sharding. With sharding, many read-only transactions would need to execute distributed, because they involve two users (a buyer and a seller).

Our final workload, ReTwis, is a Twitter clone originally designed to run on a non-transactional key-value store. We converted all its operations to transactional stored procedures, thus strengthening consistency and eliminating many round-trips between the application and the data-store. None of the transactions abort – all distributed transactions can execute using the independent transaction model. The manual partitioning we adopted for this benchmark groups *users* with all related objects, then shards users across all partitions. It is hard to define a more effective partitioning scheme without a deeper knowledge of the social graph in the workload.

8.2 Evaluation

Figure 8.1 is a summary of our experimental evaluation and shows that our tool is able to match or exceed the manual partitioning schemes in terms of the number of distributed transactions on all benchmarks. In TPC-C, accesses to the various warehouses are uniform and each warehouse corresponds to its own partition. Our tool matches the manual partition scheme for TPC-C, which is known to be optimal. Horticulture also arrives at this optimal partitioning scheme on TPC-C.

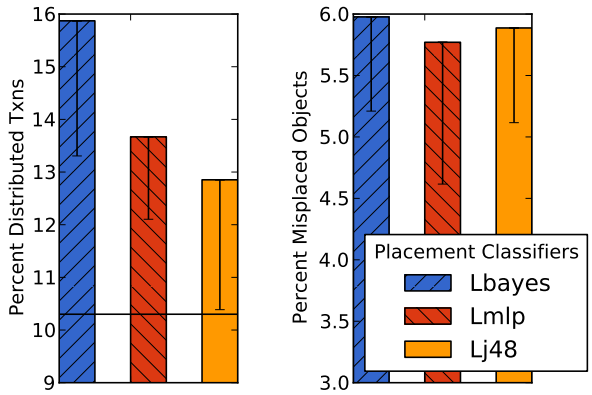
The remaining benchmarks are characterized by significantly more complex data access patterns, and the automated tool proposes better partitions than our manual partitioning schemes. Good partitioning schemes for these benchmarks must take workload details into account, e.g., which users are most active, what are the social graphs between users, etc. Such details are complex and usually ignored in manual partitioning, which leads to sub-optimal partitions.

TPC-W and AuctionMark observed the largest improvements, with distributed transactions reduced from 24% (and respectively 14%) to only 2%. This reduction was possible because the objects observed to be frequently accessed together were placed in the same partition. EPinions and ReTwis also benefited from automatic partitioning, but to a lesser extent. This can be attributed to the difficulty to partition the many-to-many relationships from these workloads.

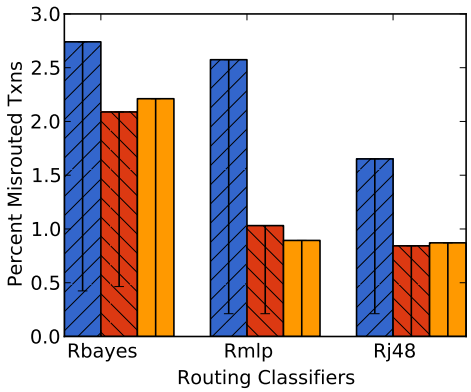
We used three classifier types (Naive Bayes [51], Multilayer Perceptron [45] and C4.5 decision trees [94]) for both object placement and transaction routing. Figure 8.2 shows results for a sample TPC-C workload. In this workload, approximately 10.3% of all issued transactions span more than one warehouses. These transactions would be executed as distributed transactions under the best known manual partitioning for TPC-C, i.e., each warehouse in its own partition, and all *item* objects replicated at all partitions. We find that using C4.5 decision trees for placement and routing gives the best results, both in terms of minimizing distributed transactions and in terms of avoiding misrouted transactions. This applied in all the workloads that we have tested.

Our system proposes high quality partitions. By manual inspection of the resulting decision trees, we determined that many of our best partitions were identical to the best known manual partitioning scheme for TPC-C. The same conclusion is also supported by Figure 8.3, which compares the ratio of distributed transaction between our best partitions and the optimal manual partitioning, as the data size (number of warehouses) is increased.

We scope out a direct comparison against Schism — both our system and Schism essentially propose the same partitions (optimal) on TPC-C. Unlike Schism, our system is able to use the independent transaction model for most distributed transactions in our workloads. Instead, Schism would use all 2PC-coordinated transactions, leading to lower performance. A direct comparison between independent transactions and 2PC-coordinated transactions



(a) Distributed Txns (b) Misplaced Objects



(c) Misrouted Transactions

Figure 8.2: Results on TPC-C for a workload configured with 3 warehouses, with different classifiers. The trace used contains approx. 1200 transactions. Boxes show average values. Black markers show best classifier. Horizontal line in 8.2(a) shows theoretical best.

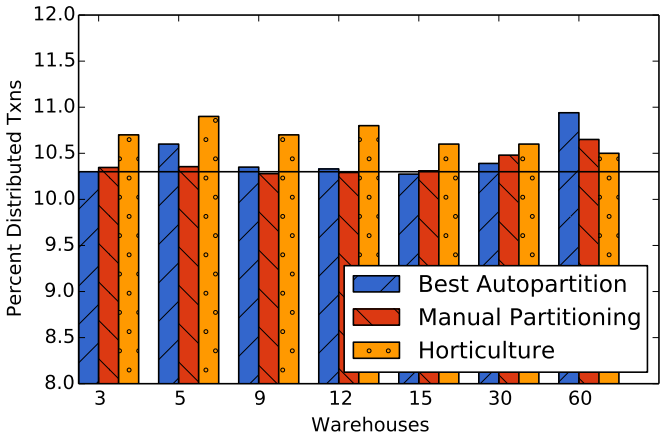
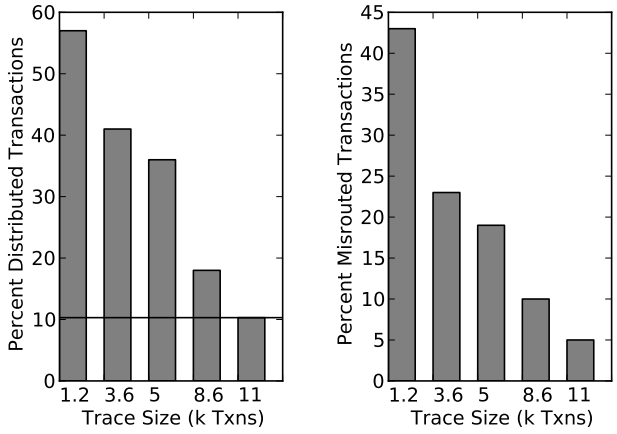


Figure 8.3: Best partition quality on TPC-C with increasing number of warehouses (lower is better). Horizontal line is theoretical minimum. Differences are minimal (less than 0.4%)



(a) Partitioning (b) Routing

Figure 8.4: Quality of partitioning and routing with respect to increasing trace size, on TPC-C (15 warehouses). In 8.4(a), horizontal line is optimal manual partitioning.

was reported in [24].

Due to the random sampling of tuples and transactions, not every partitioning attempt had the same optimal result. This can be observed in Figure 8.2(a), where the best cases match the theoretical minimum of distributed transactions, but the average case is a few percentage points away. Several of the trained classifiers reached 100% routing accuracy on our test set, as seen in Figure 8.2(c). To deal with the inherent variability of random sampling, we repeated the partitioning process several times and chose the best result.

As the data size is increased, however, the size of the trace that is the input to the system must also increase, otherwise the partition quality decreases. For example, if 3 warehouses only needed a trace with 1.2k transactions to give good partitions, 7 warehouses required 3.5k transactions and 15 warehouses needed 11k transactions. Figure 8.4 shows how the quality of partitioning and routing evolves with increasing the trace size, for 15 warehouses. In practice, one would likely start with a short trace (which can be evaluated faster) and progressively increase the trace size until the partition quality stops improving.

To show how our process scales as we increase the graph size, we present running times for the various phases in Table 8.1. We varied the graph size by adjusting the tuple-level sampling factor (i.e., the ratio of data items present in the transaction trace that we represent as nodes in the graph, the remaining data items are ignored). We notice that a majority of the time is spent in the graph representation and evaluation phase. In the evaluation phase, most time is spent computing routing information for each transaction in the input trace (training the routing classifiers is relatively fast). We believe these two most time-consuming operations could be further optimized.

While our prototype only supports off-line partitioning, we can estimate a live system would exhibit a trade-off between three factors: system performance during partitioning, the duration of partitioning and spare CPU capacity. Since Granola executes all transactions using a single thread, its maximum throughput is limited by the performance of a single core, and not by the number of CPU cores available. Thus, as long as spare CPU cores are available towards partitioning, there will be no interaction between the transaction execution system and the separate partitioning tool. Transactional throughput can still be affected by runtime logging, but the impact can be arbitrarily reduced through sampling, at the expense of needing more time to complete a trace. If spare CPU capacity is still available, certain phases of the partitioning process can be parallelized to further speed up partitioning.

Figures 8.5 and 8.6 show transactional throughput measurements. PaxosSTM is faster than our approach on a few of the workloads (AuctionMark-2, EPinions-2). This happens when contention is low and PaxosSTM benefits from running multiple transactions concurrently. On higher contention workloads however, Granola's single-threaded approach performs better. This effect is strongest on TPC-C, where all transactions relating to the same warehouse conflict.

Furthermore, PaxosSTM does not scale with increasing number of nodes, as all transactions

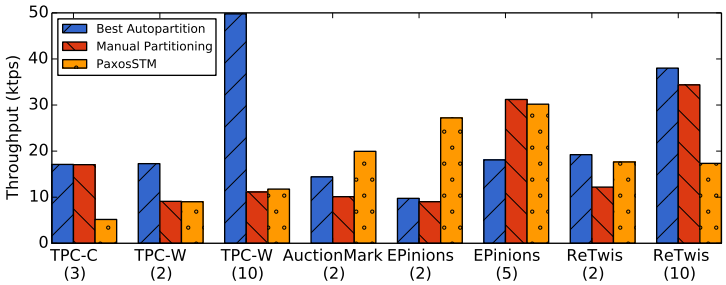


Figure 8.5: Total transactional throughput by benchmark.

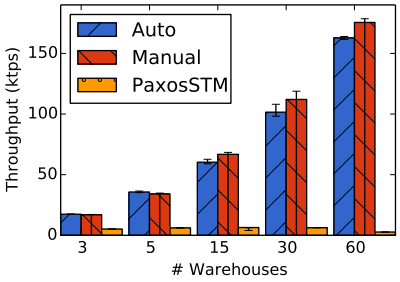


Figure 8.6: Total throughput on TPC-C. The bar is the average. The lower error bar is the standard deviation. The upper error bar is the maximum.

must be coordinated across all participants. Our partitioning approach on the other hand is scalable, especially when the ratio of distributed transactions is low. This can be seen on EPinions and ReTwis, as the number of nodes is increased from 2 to 5 (and 10 respectively). With 2 nodes, PaxosSTM is similar or faster than the Granola approach, but as more nodes are added, throughput under the Granola model increases while on PaxosSTM it stagnates.

Our throughput measurements also reflect the benefits of automated partitioning as compared to manual partitioning. TPC-W observed the best improvement, of almost 4.5x (measured with 10 nodes). The throughput with manual partitioning did not increase as more nodes were added to the system, because of the large ratio of distributed transactions. With automatic partitioning however, throughput increased significantly as the experiment was scaled up.

AuctionMark and ReTwis showed more modest improvements. In ReTwis the manual partitioning scheme is already good – further improvements are difficult. In AuctionMark, despite having fewer distributed transactions, the transactions span more partitions and are thus slower. In TPC-C, both the manual and automatic partitions were already optimal, so the throughput only reflects the overheads of classifying objects at runtime. This effect is more pronounced at higher node-counts, when the thread executing Granola transactions becomes saturated.

In EPinions however, automated partitioning became significantly slower than the manual partitions as the number of partitions in the system was increased from 2 to 5, despite still having a lower ratio of distributed transactions. This situation is caused by a skew in the workload towards a single partition, which then becomes a bottleneck. The issue can be resolved by balancing the partitions in terms of offered workload instead of data size. The workload would thus be evenly distributed across all partitions, although the data distribution may become skewed instead (some partitions hold more data than others).

We additionally varied the fraction of distributed transactions in a TPC-C workload to

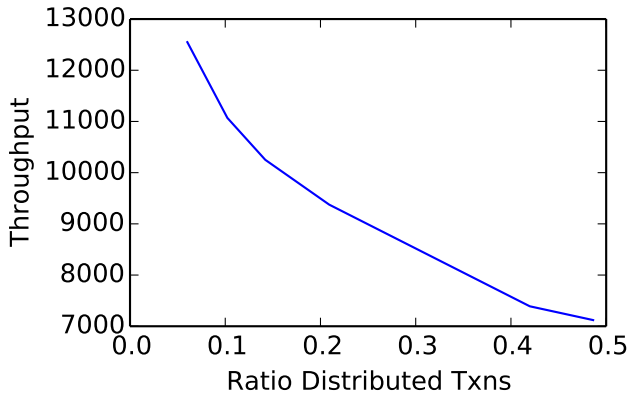


Figure 8.7: Total transactional throughput (TPC-C, 3 warehouses), with a varying fraction of distributed txns.

simulate the effect that partition quality has on throughput. Results are shown in Figure 8.7. Fewer distributed transactions clearly lead to better performance. This effect is strongest when distributed transactions account for less than about 10-15% of the total workload. Thus, optimizing the quality of partitioning can be very beneficial and is especially important for workloads with less than 10-15% distributed transactions.

To summarize, we evaluated our system on 5 benchmarks and in most cases observe improvements in both the ratio of distributed transactions and transactional throughput. The largest improvements (up to 4.5x in throughput) were observed on benchmarks where different objects were frequently accessed together in non-trivial patterns. We additionally provided the first investigation of single-repository and independent transaction models on several benchmarks other than TPC-C.

Chapter 9

Alvin: A General, Consistent, Geo-Replicated Transactional System

Alvin is a geo-replicated transactional system built around a novel Partial Order Broadcast protocol (POB) that globally orders only conflicting transactions while minimizing the number of communication steps for non-conflicting transactions and avoids relying on a single designated leader.

The idea to consider message semantics when defining consensus originates from Generalized Consensus [63] and Generic Broadcast [88]. POB however introduces a novel approach for ordering transaction commits, that does not rely on a single designated leader and thus avoids the limitations associated with most existing solutions (e.g., Generalized Paxos [63]) when deployed in a geographically distributed system (GDS). A leader is not needed for neither of ordering transactions or conflict resolution.

POB draws inspiration from modern multi-leader state machine replication protocols that specifically target GDS, such as *Mencius* [74] and *EPaxos* [77]. POB inherits their benefits while at the same time avoiding their drawbacks. Like Mencius, POB decides the final order of commands directly on the node that issues them, but benefits from employing a quorum instead of requiring a information from all nodes in the system. This is especially beneficial in a GDS setting, with high latencies and unreliable communication, or when some nodes have a much lower rate of issued commands.

In a similar way to EPaxos, POB may adjust the order of commands after their proposal. This is done based on command dependencies, and helps reduce the number of communication steps when no conflicts are present. POB however does not need to process dependency graphs as EPaxos does. This helps maintain a speedy execution as the graph size increases due to more complex workloads with a higher transaction size and increased contention.

In POB, each node has a predefined exclusive subset of positions that it can use for proposing new commands. The node proposing a command becomes the command's leader, and is

thus responsible for selecting the position which will define the ordering. After the order is stabilized, commands can be delivered according to their position number. Commands are also associated with a set of dependencies, allowing POB to reach agreement while communicating with only a quorum of nodes, as opposed to Mencius which needs acknowledgement from every other node in the system. The dependencies of a transaction T_2 are the set of transactions conflicting with T_2 that precede T_2 in the order defined by POB. POB further ensures that when T_1 is in dependency set of T_2, its position number will be lower than T_2's position number and T_2 will only be delivered on a node after T_1 was also delivered.

ALVIN uses POB in conjunction with P-CC, a local parallel concurrency control layer able to fully exploit all of POB's advantages. P-CC commits non-conflicting transactions in parallel and only needs to serialize conflicting transactions, thereby increasing parallelism.

ALVIN guarantees that transactions always observe a consistent view of the memory. This includes transactions that will eventually be aborted. In DTM systems this property is important, as observing an inconsistent state may lead to crashes or other unexpected behavior [37]. This guarantee is provided thanks to ALVIN's transaction processing model. Transactions execute optimistically on the local site, employing a timestamp-based multi-versioned scheme for all read operations (i.e., snapshot reads). Upon completion, POB disseminates a record of the transactions' execution, and then P-CC is tasked to validate and commit each transaction at every replica in the system. P-CC's task can be completed without any further remote communication, because POB guarantees conflicting transactions are ordered identically on all nodes.

ALVIN is flexible and allows programmers to configure aspects of both the POB and P-CC, according the the application requirements. Specifically, ALVIN offers the choice of two consistency criteria, Serializability (SR) [10] and Extended Update Serializability (EUS) [3, 90] (i.e., PL-3U [3]). Both are considered as strong consistency. To guarantee serializability, ALVIN broadcasts all transactions through POB, including read-only transactions. Conversely, if EUS is desired, ALVIN POB only needs to order transactions that contain at least one write operation. Read-only transactions can be processed locally, at the expense of allowing some non-serializable executions, typically silent to the application. Another choice ALVIN offers to the programmer is whether to enable fast decisions or not. Fast decisions may lead to agreement after only two communication delays, but require a larger quorum than the quorum for a classic decision.

We implement ALVIN in the Go programming language, and we evaluate it on Amazon EC2 infrastructure using up to 7 geographically distributed sites, and benchmarks including Bank [50] and TPC-C [23]. We compare against two certification-based transactional systems [87] with Multi-Paxos [64] and EPaxos [77] as the ordering layer. ALVIN with EUS proves to be significantly faster than EPaxos, up to 4.8× on TPC-C with a 7 datacenter deployment. With SR, ALVIN is only 26% faster than EPaxos, due to the absence of a complex graph processing phase. On Bank, the smaller transactions with significantly lower contention led to smaller dependency graphs and thus similar behavior between ALVIN and

EPaxos. The performance of Multi-Paxos was significantly lower than the other competitors, showing how disadvantageous it is to have a single leader in GDS.

9.1 Assumptions and System Model

Our work assumes a set of geographically distributed sites $\Pi = \{P_1, P_2, \ldots, P_N\}$ executing operations on some common shared data. They are organized as an asynchronous distributed system, and use a wide area network for communication between sites. No specific network delay distribution or upper bound is assumed, thus every message may experience arbitrarily large but finite delays.

Each node logically represents a datacenter. In practice, multiple nodes may be deployed in a datacenter. Managing their synchronization is an orthogonal issue which we scope out for our work. Thus each site stores the entire shared data set, and transactions running at any site can access any datum locally.

Let N be the total number of sites, and at most $f < \lceil \frac{N}{2} \rceil$ of them can experience faults at any time. At least a majority of nodes is thus always correct. For our work we scope out Byzantine faults (malicious behavior), and we assume the crash-stop failure model [10]. For any phase of the ordering protocol, a node contacts all sites and waits for a *quorum* Q of replies. We employ two quorum types: a *classic quorum* CQ, and a *fast quorum* FQ. Both CQ and FQ at least equal to $\lfloor \frac{N}{2} \rfloor + 1$, such that any two quorums have a non-empty intersection. This means that upon at most f failures, there is at least one correct site with sufficient information as required for system recovery. The actual values for CQ and FQ may vary according to configuration, and will be specified later in this chapter.

Furthermore, we assume the presence of a weak unreliable failure detector [40], as is also needed for implementing a leader election service [39]. This is needed to guarantee agreement is eventually reached when sites become faulty, such as a datacenter becoming unreachable.

9.2 Alvin: Geo-Replicated Transactional System

Transactions in ALVIN follow a simple object-oriented interface. Accesses are performed on shared objects, and can be either READ or WRITE. Transactions are delimited using BEGIN and COMMIT commands.

ALVIN exploits partial ordering of transactions to improve performance when compared to a total order approach. Totally ordering all commits on all nodes would certainly guarantee that the exact same state transitions are executed across the system, but this is too strong of a condition for a GDS because it unnecessarily delays the commits of all transaction until all previous transactions (including non-conflicting transactions) are finalized. In ALVIN on

the other hand, transactions only need to wait for conflicting transactions before they are allowed to commit. With this approach, the shared state across nodes is still guaranteed to converge quickly, while at the same time allows a much higher degree of parallelism especially when inter-datacenter conflicts are rare, a common situation in GDS.

ALVIN follows a certification-based approach [87] for executing transactions. Its architecture includes two fundamental layers: the Partial Order Broadcast layer (POB) and the Parallel Concurrency Control layer (P-CC). POB is responsible for ordering commit requests making sure conflicting transactions are delivered in the same order at all nodes. P-CC on the other hand optimistically executes the transactions, making sure to always provide a consistent view of the shared state. After the commit request is ordered by POB, P-CC also verifies the optimistic execution is still valid before finally installing the writes to the shared state.

ALVIN supports transactional applications consisting of multiple threads distributed among all nodes. According to the certification-based replication scheme [87] which we employ, a thread spawns transactions on the same node the thread itself is running. Objects read and written are recorded in a per transaction private space named read-set ($T.RS$) and respectively, write-set ($T.WS$).

To execute a transaction T, two phases must be completed. In the first phase, the transaction executes locally, optimistically, to build its read-set and write-set under the control of P-CC. No changes are made to the shared state at this stage. When complete, the executing thread broadcasts T (including $T.RS$ and $T.WS$) using the POB layer and waits so T can be validated globally and thus committed or aborted. In case of an abort, T's records expunged, and the application thread must restart the transactions. In case of a commit, T waits for any conflicting transactions ordered before itself, and them proceeds to apply its updates to the globally shared state.

In order to process a transaction T, the POB layer uses the following interface: POBROADCAST(T) is used to broadcast T including its read-set and write-set; PODELIVER($T, deps_T$) is a callback invoked by POB to deliver T on each node. Besides T, also delivered is a set of transactions $deps_T = \{T_1, \cdots, T_m\}$ which contains all transactions T_k conflicting with T which must be processed (i.e., certified and committed or aborted) before processing T. We say two transactions T and T' are conflicting if at least one of the following conditions is true: (i) $T.WS \cap T'.WS \neq \emptyset$, (ii) $T.WS \cap T'.RS \neq \emptyset$, (iii) $T.RS \cap T'.WS \neq \emptyset$ (i.e., they access a common object, and at least one of the accesses to the common object is a write).

9.2.1 Partial Order Broadcast Layer

The POB layer guarantees that conflicting transactions are delivered in the same order on all nodes. This is motivated by the fact that, two transactions that do not conflict can be processed in any order, while arriving at the exact same state on all nodes. ALVIN prefers using POB because it is cheaper than using a Total Order Broadcast (TOB) service which

ensures a total order among all transactions.

POB provides the following properties:

- *P1: Strong Uniform Conflicting Order.* If a node delivers message $m = [T, deps_T]$ before message $m' = [T', deps_{T'}]$ and transactions T and T' conflict, then every node delivers m before m'.
- *P2: Local Dependency.* If a node delivers message $m = [T, deps_T]$ before message $m' = [T', deps_{T'}]$ and transactions T and T' conflict, then $T \in deps_{T'}$ and $T' \notin deps_T$.

Property *P1* protects against the omission of messages — therefore it is defined as *strong*. It comes in contrast with the weak order property that, despite preserving the order of delivery on all nodes, it would allow some messages to be delivered on some nodes but not on others. Transaction processing requires the strong order property, as the state of the system can irrevocably diverge if any transactions are skipped on some nodes.

The second property, *P2*, ensures that the layer processing a transaction T post-delivery from POB (i.e., P-CC) has a reliable view of conflicting transactions delivered before T, i.e., $deps_T$. All P-CC needs to do to process transactions according to the partial order defined by POB is to wait for all $T_k \in deps_T$ before processing T. POB also guarantees the typical properties of a reliable broadcast service (i.e., validity, integrity, uniform agreement [29]).

Overview. POB enforces an order only among conflicting commands, as previously specified by the problems of Generalized Consensus [63] and Generic Broadcast [88], for which a number of solutions have been proposed (e.g., Generalized Paxos [63], EPaxos [77]). POB improves upon these previous solutions by employing a decentralized design without a stable leader to dictate the order while avoiding expensive processing on the critical path leading to the delivery of messages.

POB relies on a deterministic scheme for assigning position numbers in the final order (also called *delivery slots*) to submitted transactions. The delivery order of transactions is determined by the proposing nodes, in a similar way to *communication history*-based total order broadcast protocols [29, 74]. The proposing node for transaction T acts as its unique leader tl_T which applies the rules bellow to determine the final delivery position $pos_T \in \mathbb{N}$.

- *Rule 1.* If transaction T was proposed by node P_i (i.e. $tl_T = P_i$), then T can only be delivered in a free position labeled pos_T, such that $pos_T \bmod N = i$.
- *Rule 2.* Transaction T is delivered in position pos_T iff for every conflicting transaction T_0 ordered ahead of T (i.e., $pos_T < pos_{T_0}$), T_0 is a dependency of T_0 and T is not a dependency of T_0 (i.e., $T_0 \in deps_T$ and $T \notin deps_{T_0}$).

Following *Rule 1* POB can guarantee two transactions proposed by different nodes would never occupy the same position. It is however possible for two transactions issued by the

same node to be assigned the same position number. Such a situation is defused by deterministically ordering the two transactions based on their unique numeric identifier. *Rule 2* is specifically defined such that POB can satisfy the property *P2*.

In the absence of suspected failure, the transaction leader tl_T for a transaction T is the node that issued T. Any other node may however suspect the original leader to have failed and subsequently start a vote to be elected the new leader for T as part of a recovery process.

Protocol. A transaction T is submitted to POB via the POBROADCAST(T) interface, and goes through the following four phases: *Proposal phase*, *Decision phase*, *Accept phase* and *Delivery phase*.

In the Proposal phase, node P_i leader of transaction T selects the next position number available to be used as pos_T. pos_T must be greater than every other position that P_i observed to be in use, but at the same time is the smallest value allowable under *Rule 1*. Also selected is the dependency set $deps_T$, the set of all transactions that conflict with T which have the position number currently lower than pos_T.

After collecting this data, P_i broadcasts a PROPOSE$\langle T, pos_T, deps_T, e\rangle$ to all nodes. At this stage, all know information about T is broadcast, namely its identifier, read-set and write-set ($T.id$, $T.RS$, and respectively, $T.WS$).

To support failure recovery, also included with the PROPOSE message and any other message associated with transaction T is an epoch number e. The epoch is advanced when a node suspects the old leader to have failed and initiates the recovery procedure. In a similar way to the Paxos algorithm, a node must ignore all messages with a lower epoch number e_0 lower than its current known epoch number e for transaction T. If on the other hand a message with a greater epoch number e_1 arrives, e is locally updated to this value. The initial epoch number for all PROPOSE messages is 0, representing the initial epoch for which T's sender is also T's initial leader tl_T.

Upon receiving a PROPOSE$\langle T, pos_T, deps_T, e\rangle$ message, a node P_j confirms receipt by sending an ACKPROPOSE$\langle T.id, newPos_T^j, newDeps_T^j, e\rangle$ message back to P_i. $newPos_T^j$ represents a request to update pos_T in the situation when P_j observed another transaction T_1 conflicting with T for which $pos_{T_1} > pos_T$. In this case, $newPos_T^j$ will be the next available position for T such that $newPos_T^j > pos_{T_1}$ (and still compatible with *Rule 1* for node P_i). If no such conflicting transaction exists, then simply $newPos_T^j = pos_T$.

Similarly, $newDeps_T^j$ is the set of conflicting transactions T_k with $pos_{T_k} < newPos_T^j$.

POB stores in-flight transactions in a data structure called *Delivery Queue* (DQueue). At each node P_j, DQueue holds a tuple $\langle T, pos_T, deps_T, status, e\rangle$ for each transaction issued or received by P_j. DQueue is sorted according to the pos_T value of each entry. Valid values for *status* are PENDING, ACCEPTED, STABLE. Upon sending or receiving a PROPOSE message for transaction T, an entry is added to DQueue with its $status$ = PENDING.

In the Decision phase, P_i, the leader of transaction T, waits for a quorum of FQ ACKPRO-

POSE replies from the previous phase. The final position pos_T and dependencies $deps_T$ are then computed as follows:

- pos_T is the maximum among all $newPos_T^j$ position proposals in the quorum.
- $deps_T$ is the union of all $newDeps_T^j$ dependency set proposals in the quorum.

P_i then broadcasts an ACCEPT$\langle T.id, pos_T, deps_T, e\rangle$ message for T with the final position and dependency set. In the basic POB configuration, $FQ = f + 1$. We show in Section 9.2.1 how *fast decisions* allow avoiding two communication steps if no contention is encountered by increasing the value of FQ.

In the Accept phase, upon node P_j receiving message ACCEPT$\langle T.id, pos_T, deps_T, e\rangle$ it updates its corresponding DQueue entry setting $status$ = ACCEPTED and updating pos_T and $deps_T$ according to the values received.

Node P_j then replies with an ACKACCEPT$\langle T.id, pos_T, newDeps_T^j, e\rangle$, where $newDeps_T^j = deps_T \cup deltaDeps_T^j$. This set $deltaDeps_T^j$, contains all the transactions T_Δ that conflict with T, have arrived at P_j after P_j processed the initial PROPOSE message and have a position pos_{T_Δ} between the old $newPos_T^j$ and the new (final) pos_T.

In the Delivery phase, T's leader node P_i waits for a quorum of $CQ = f + 1$ ACKACCEPT replies from the previous phase. The value of CQ is the same with the classic Paxos quorum size and ensures that even in the case of f failures, at least one node in the system remains having recorded $status$ = ACCEPTED and the final pos_T. After receiving the CQ replies, tl_T computes the final $deps_T = \cup_j newDeps_T^j$ from those replies and broadcasts a message STABLE$\langle T.id, pos_T, deps_T, e\rangle$ to inform all nodes of its decision.

Node P_j, upon receiving STABLE$\langle T.id, pos_T, deps_T, e\rangle$, updates its corresponding DQueue entry by setting $state$ = STABLE and updating pos_T and $deps_T$ if needed. Finally, after all transactions in $deps_T$ have been delivered, T can also be delivered by triggering PODELIVER$(T, deps_T)$.

Following this protocol, a situation may arise at this stage such that $T' \in deps_T$ and $T \in deps_{T'}$, with $pos_T < pos_{T'}$. In order to avoid a deadlock, the dependency set for the later transaction T' is modified to expunge T, but only when $status_T$ =STABLE and $status_{T'}$ =STABLE.

When a node P_k suspects that T's current leader P_i has crashed and $status_T \neq$ STABLE, P_k starts a Failure Recovery process. P_k tries to become T's new leader by executing a classic Paxos *Prepare phase* [64]. First, P_k broadcasts a PREPARE$\langle T.id, e_2\rangle$, where e_2 is greater than the last observed epoch for T. Upon receiving this request, nodes send a reply PROMISE$\langle T, pos_T, deps_T, status_T, e_2\rangle$, promising to not participate in any round with an epoch number $e < e_2$. Also included in the reply are the last observed details for transaction T, as recorded in the node's DQUEUE. Upon receiving a quorum of $CQ = f + 1$ such PROMISE replies, P_k is acknowledged as T's new leader. P_k can analyze the replies and take

a decision for T that is guaranteed to be identical to the decision taken by T's old leader P_i, if any.

Based on the replies received above, three situations may arise:

- At least one of the replies contains $status_T = \text{STABLE}$. This reply contains the final details $\langle pos_T, deps_T \rangle$ for transaction T, which P_k must enforce. Thus P_k only needs to do a *Delivery phase* by broadcasting a $\text{STABLE}\langle T.id, pos_T, deps_T, e_2 \rangle$ message.
- None of the replies have $status_T = \text{STABLE}$, but at least one reply has $status_T = \text{ACCEPTED}$. This reply contains the details $\langle pos_T, deps_T \rangle$ for transaction T, which P_k must use. Thus, P_k starts an *Accept phase* and broadcasts an $\text{ACCEPT}\langle T.id, pos_T, deps_T, e_2 \rangle$ message with these details.
- None of the replies have $status_T \in \{\text{STABLE}, \text{ACCEPTED}\}$. P_k must start a new *Proposal phase* and may select the next available position for T.

Fast Transaction Decisions

Besides the base behavior described in the previous section, POB can also be configured with a fast-path for taking a decision in only two communication steps. When enabled, *fast decisions* help avoid the two message delays of the *Accept phase* for transactions that do not encounter contention. In order to support fast decisions, POB requires waiting for a larger quorum $FQ > f + 1$ in the *Decision phase* and has a more complex recovery procedure. With GDS, we can certainly imagine deployments where fast decisions do not bring any performance benefits, especially for higher contention workloads. Enabling this optimization should thus be done after considering the trade-off between the lower number communication steps and the larger quorum size.

The fast path works as follows: after broadcasting the PROPOSE message, the leader waits for a fast quorum of FQ identical replies. If these replies are received, the leader makes the decision immediately and broadcasts the corresponding STABLE message with the same $\langle pos_T, deps_T \rangle$. Otherwise, the regular path is taken by proceeding to the *Decision phase* as described in the previous section.

With fast decisions, FQ must be adjusted such that, in the event of f faults including a transaction T's leader, the system can safely recover. More specifically, T's new leader must be able to determine the decision of the failed leader, if any, even when this decision was taken on the fast path.

In the recovery procedure, T new leader $tl_T = P_k$ broadcasts a $\text{RECOVER}\langle T.id, e_2 \rangle$, and waits for a quorum of $CQ = f + 1$ replies. If no transaction T' conflicting and concurrent with T exists, then the recovery is trivial. If such a T' exists and its leader $tl_{T'}$ has not failed, the previous decision for T may be inferred from the information sent by $tl_{T'}$. Otherwise,

assuming tl_T, $tl_{T'}$ and up to $f-2$ other nodes have failed, we must ensure a majority of the remaining nodes agree with the previous fast-path decision. Therefore, $N-FQ-1$, representing the nodes that may have voted for T' but excluding $tl_{T'}$ which is known failed, must be less than a majority of replies, $\lfloor \frac{CQ}{2} \rfloor + 1$, or:

$$N - FQ - 1 < \left\lfloor \frac{CQ}{2} \right\rfloor + 1$$

A separate condition that needs to be enforced is disallowing two nodes attempting to recover two different but conflicting transactions T and T' to both believe they had a fast quorum. Supposing each new leader collects half the votes in the recovery phase, i.e., $\frac{N-f}{2}$, and assumes it had all the votes of the failed nodes, except the leader of the other conflicting transaction, i.e., $f-1$, it should not add up to make a fast quorum:

$$\frac{N-f}{2} + f - 1 < FQ$$

Solving for the two inequations above, and aiming for a minimal number of nodes $N = 2f+1$, gives us $CQ = f+1$ and $FQ = f + \lfloor \frac{f+1}{2} \rfloor$. These are the same values as the quorums adopted by EPaxos. Moreover, the full recovery procedure is a trivial extension to the recovery procedure of EPaxos [77].

9.2.2 Parallel Concurrency Control Layer

The Parallel Concurrency Control layer (P-CC) sits between POB and the application code on each node in the system. Its two main tasks are *(i)* optimistically executing the transactions submitted by the clients and *(ii)* performing the final validation and commit of transactions after their delivery from POB.

P-CC maintains a node-local variable *clock* which is incremented with every transaction commit. Each object retains a list of recent versions. Each object version is tagged with a pair of integers $\langle clk, pos \rangle$ — clk represents the value of *clock* when the transaction was committed and pos is the pos_T value of the transaction that committed it.

The first stage in a transaction's lifetime is handled by the P-CC and is called *execution phase*. The application thread issues READ and WRITE operations one-at-a-time, which get serviced by P-CC as part of a transaction. Through the use of Multi-Version Concurrency Control (MVCC) and the node-local *clock*, P-CC guarantees transactions always observe a consistent view of the memory, even if they later abort. Upon its first read operation, a transaction records *clock* in a private variable named tc_T. Future reads will return object values tagged with the greatest value clk no bigger than tc_T. All reads are also recorded in the transaction's read-set, $T.RS$. Similarly, writes are buffered in the transaction's write-set

$T.WS$, without modifying the shared memory. When the transaction completes its optimistic execution and attempts to commit, PC-C submits it to the POB layer for global ordering.

The second task of P-CC begins when POB delivers transaction T thus beginning the *commit phase*. In the commit phase, P-CC first waits for transactions in $deps_T$ to also complete their commit phase before attempting to process T. After that, T's read-set must be validated. Read-set validation makes sure no objects in $T.RS$ have been updated since T originally read them, by comparing the *pos* tag of the latest version of the object with the *pos* of the version stored in $T.RS$. Since the validations must have the same outcome at all replicas, the globally consistent *pos* values must be used. If the validation succeeds, P-CC can install the writes in $T.WS$ to the shared memory and increment the node-local *clock*.

By employing snapshot reads, P-CC ensures that transactions always observe a consistent state. Additionally, P-CC can be configured to guarantee one of two consistency criteria for the set of committed transactions: serializability (SR) or Extended Update Serializability (EUS). SR is the reference consistency criterion for transactional applications and ensures every committed transaction appears to have executed atomically. P-CC provides serializability by submitting all transactions (including read-only transactions) to POB for ordering and subsequent validation. We should note that SR can also be achieved by totally ordering all update transactions (thus replacing POB with a Total Order Broadcast layer, or TOB), and locally committing read-only transactions by default, without any certification, as show in [104].

EUS [3] is an alternative consistency criterion weaker than SR, but still strong enough for most transactional applications [90]. Under EUS, the system never transitions to an invalid state, and all operations observe a consistent state. With respect to update transactions, EUS and SR are identical. EUS however allows certain executions of read-only transactions that would otherwise be rejected under SR. More specifically, two read-only transactions executing at different nodes may observe two non-conflicting update transactions with different orders. P-CC provides EUS by only submitting update transactions for ordering via the POB layer, while locally committing read-only transactions by default, without any certification.

Pseudocode and proofs for ALVIN are provided in Appendix A.

9.2.3 Garbage Collection

ALVIN periodically removes old transactions from the delivery queue DQUEUE. This is done to keep dependency sets short, which have to be transmitted over the network. A transaction may only be considered for deletion when it is known to have completed on all nodes in the system. This requires disseminating information using a mechanism such as gossip. ALVIN currently uses an approach where a node periodically collects and disseminates the pos_T of the oldest transaction still not completed. Transactions in positions older than this pos_T may thus be collected.

We now show it is insufficient to simply collect transactions locally as soon as they complete. Assume a transaction T has completed at all nodes except P_1, which does not yet know T is stable. Thus all nodes except P_1 remove T from their DQUEUE. A transaction T' conflicting with T arrives at P_2 and since T was already collected, T' does not acquire a dependency on T. T' can become stable without communicating with P_1 (e.g., due to dropped or delayed messages), broadcasting a STABLE$\langle T' \rangle$ message. If this message gets delivered to P_1 before STABLE$\langle T \rangle$, the P_1 can deliver T and T' in the opposite order compared to all other nodes, bringing the system in an invalid state.

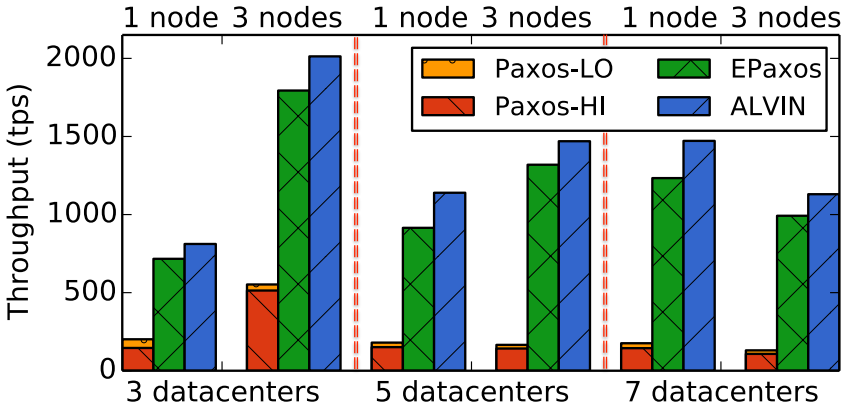
Finally, garbage collection is also required for removing old object versions when they are no longer readable by any live transaction. For this, the knowledge of the oldest live transaction in the system (its tc_T value) must be disseminated throughout the cluster.

9.3 Evaluation

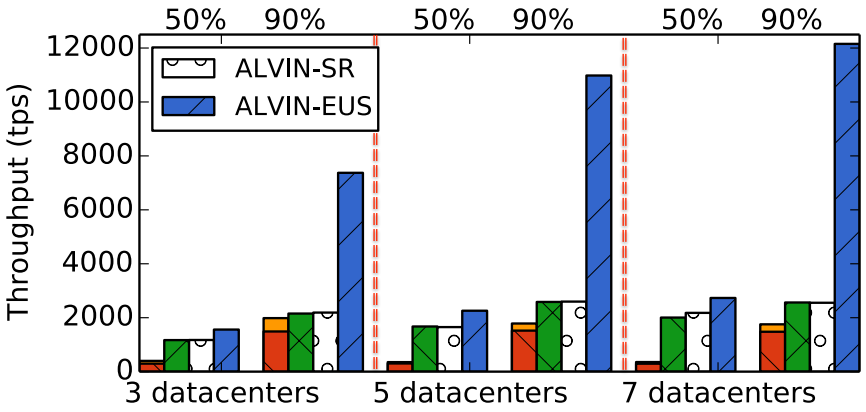
We evaluate ALVIN by comparing it against two certification-based transaction execution protocols [87] that rely on MultiPaxos [64] and EPaxos [77] for their ordering layer. Multi-Paxos ensures serializability by totally ordering the commit requests for all write transactions, while serving read-only transactions locally through multi-versioning. However, MultiPaxos is sequencer-based, thus the location of the node designated as the leader significantly affects its performance. In order to conduct a fair comparison, we used two versions of MultiPaxos: one with the leader located at a node with a point-to-point latency to other nodes that is higher than the average (*Paxos-HI*), and another where the connection latency is lower (*Paxos-LO*). We implemented ALVIN and the competitors in the same transaction processing framework, using *Go* as the programming language.

We used two benchmarks in the evaluation: TPC-C [23] and Bank [50]. The former is a well known benchmark representative of on-line transaction processing workloads; the latter mimics operations of a monetary application where each transaction transfers an amount of money between some bank accounts. We ran our experiments on the Amazon EC2 infrastructure, using *r3.2xlarge* nodes in up to 7 geographically distributed sites (three in Asia, two in North America, one in South America and Europe). Each node has 8 CPU cores and 61GB RAM. Results are the average of 7 samples.

Figure 9.1 reports ALVIN's throughput on the TPC-C benchmark by varying the number of geographically distributed sites in {3,5,7}. In Figure 9.1(a) we also changed the number of nodes per site as {1,3}, using a write intensive workload (<3% read-only). Results on read-dominated workloads are showed in Figure 9.1(b). Here we change the percentage of read-only transactions from 50% to 90% while using one node per datacenter. In this read dominated scenario we explore both versions of ALVIN, ensuring SR (ALVIN-SR) and EUS (ALVIN-EUS), with the purpose of assessing the effectiveness of EUS. In all depicted scenarios, we configured ALVIN to run with fast decisions enabled. We batch messages for



(a) Write-intensive workload for {3,5,7} sites and {1,3} nodes per site.



(b) 50% and 90% read-only transactions. One node per site.

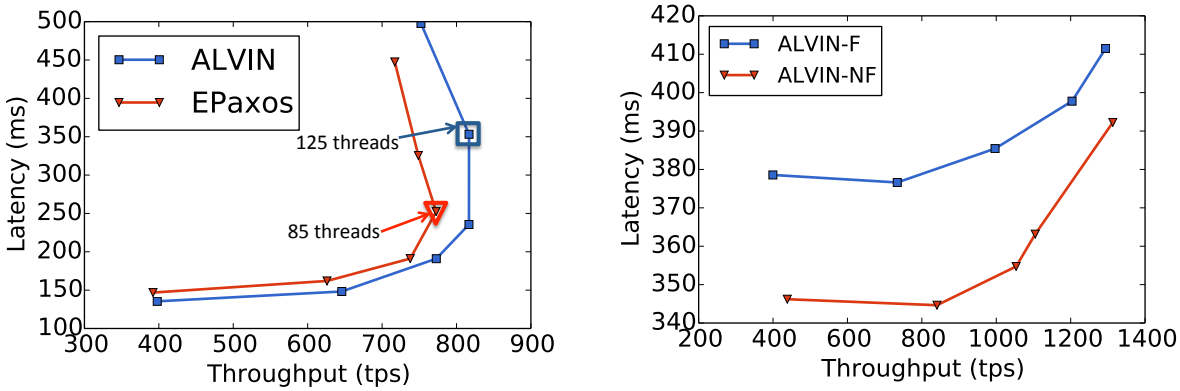
Figure 9.1: Throughput of TPC-C benchmark.

all competitors, using a window of 20 to 50 msec, according to each deployment.

TPC-C's transactions access several shared objects and have a non-negligible computation. This results in long transaction execution time and a complex dependency graph to be analyzed during the processing of commit requests in EPaxos. Therefore, ALVIN is able to improve the parallelism because of the different delivery rules of POB, overcoming EPaxos in throughput by up to 26%. Both EPaxos and ALVIN sustain their throughput while increasing the system's load until 9 nodes (3 datacenters with 3 nodes each), then the system becomes overloaded and performance degrades due to increasing contention. MultiPaxos in both its configurations performs worse than the others due to the presence of single remote leader that slows down the entire system's progress. In addition, these transactions are long thus the sequential certification limits its performance.

Figure 9.1(b) shows the effectiveness of exploiting EUS in read-dominated workloads by avoiding to broadcast read-only transactions via the ordering layer. Therefore ALVIN-EUS

provides a speed up of up to 4.8× in throughput when compared to ALVIN-SR and EPaxos. It is important to notice that in these scenarios, MultiPaxos is also able to take advantage of local computation of read-only transactions. In fact, its Paxos-LO configuration performs similar to EPaxos and ALVIN-SR for the case of 90% of read-only transactions and 3 datacenters. In other scenarios, Paxos-LO saturates its leader's resources, slowing down the ordering process. As before, Paxos-HI exposes poor performance due to the high communication latency with the faraway designated leader. Regarding the comparison between EPaxos and ALVIN-SR, they follow about the same trend observed in Figure 9.1(a) because they both process read-only transactions in the same way.



(a) ALVIN Vs EPaxos on 5 sites.

(b) Impact of fast decision.

Figure 9.2: Throughput Vs Latency using TPC-C benchmark varying application threads.

In Figure 9.2(a) we plot the latency increasing the system's load by adding application threads per node from 15 to 125. Here, we used the TPC-C benchmark deployed on 5 sites, adopting the same workload as in Figure 9.1(a). For increasing the readability of the plot we excluded MultiPaxos because its results were 3× slower than the other competitors. From the analysis of EPaxos' and ALVIN's trends we observe that ALVIN has a lower transaction latency and it sustains its throughput better than EPaxos. Specifically, with 85 threads per site EPaxos stops scaling while ALVIN is still able to serve more requests. ALVIN reaches its saturation point running 125 threads per site.

With the plot in Figure 9.2(b) we highlight the importance of configuring ALVIN without the fast decision in high contention scenarios. In these situations, the probability of taking a fast decision after having collected a fast quorum of replies is low. Therefore the POB layer always pays the maximum number of communication steps to reach a decision by contacting a fast quorum of nodes in the *Proposal phase* and then falling back to the *Accept phase*. Disabling the fast decision forces the leader to always collect replies from a classic quorum. We configured TPC-C as in Figure 9.1(a) with 7 sites and one node at each site, and we increased the load as before. ALVIN-NF (fast decisions disabled) has better latency than ALVIN-F (fast decisions enabled) by up to 30 msec, confirming that, in some scenarios, waiting for an unlikely fast decision does not pay off.

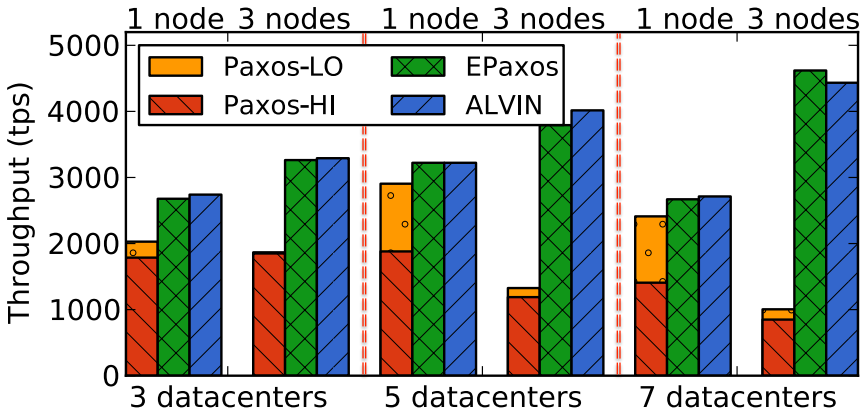


Figure 9.3: Throughput under write-intensive workload for {3,5,7} sites and {1,3} nodes per site using Bank benchmark.

The Bank benchmark has very small transactions (only few operations) and the amount of transactional work can be considered as negligible when compared to the coordination steps required for establishing the agreement on the global ordering. This makes the results of both ALVIN and EPaxos comparable in almost all configurations tested as we showed in Figure 9.3. Bank's accesses are uniformly distributed across all objects and we managed the total number of shared objects for having an average transaction's abort rate in the range of 10-20%.

EPaxos's dependency graph analysis does not slow down the transaction's critical path significantly because the strongly connected components with more than one node are only 1.7% of the total, thus the main impacting factor on the performance is the number of communication delays used for delivering transactions and, with fast decisions enabled, both ALVIN and EPaxos use the same number of communication delays for delivering. However, it is worth noting that all competitors relying on partial order instead of total order sustain their throughput when we increase the number of nodes until 7 datacenters, where they start degrading. MultiPaxos in both its configurations performs worse than others due to the presence of single remote leader that slows down the entire system's progress. The exception is Paxos-LO, which is the closest to others because it benefits from having a low latency leader when site count is low.

At its core, the design of ALVIN shows that it is possible to achieve an effective trade-off between performance and programmability in geo-replicated environments. An important insight of our work is that partial ordering of transactions can be significantly exploited to speed up local concurrency control through parallelism and that it can be determined without a unique leader, which increases scalability in a geo-replicated setting.

Chapter 10

M^2Paxos: Faster General Consensus

M^2PAXOS is our solution to the Generalized Consensus problem, and combines a number of desirable properties: (i) it allows fast decisions, in two message delays, (ii) it does not rely on a single designated leader, (iii) employs a small quorum size, equal to that of classic Paxos and (iv) does not need to track, record or exchange inter-command dependency relations.

To achieve these goals we turned to a technique often used in database systems to promote scalability and high-performance: exploiting the locality of data accesses. With this technique, the data in a system is partitioned across multiple nodes, in such way that the workload originating on a particular node will find the data it needs on that same node, with a high probability. Workload locality exploitation generally depends on optimized data placement, which can be done either manually by a developer or automatically, in a similar manner to our work in Chapter 7.

M^2PAXOS works by assigning the responsibility of ordering a certain request to specific nodes, based on the data accessed by the request. Therefore, two requests accessing the same data will be ordered by the same node, eliminating the possibility of contention between them. Furthermore, in order to best exploit workload locality, M^2PAXOS aims to delegate ordering a particular request to the same node that issued it with a high likelihood. In such cases, consensus is reached after only two message delays (which is optimal).

M^2PAXOS uses the notion of data ownership by nodes, or leases. Only the owner of a particular data is allowed to make ordering decision for that data. Ownership is asynchronous, in the sense that it is not time-limited and set to automatically expire, but it can be transferred when another node requests it. For our work, we only consider on-demand lease transfers — other techniques such as pre-fetching or statistical transfers can be implemented but are an orthogonal problem.

M^2PAXOS clearly works best when a node only issues requests for data whose leases it already owns. This applies well to workloads exhibiting a high degree of locality as is common for many On-line Transactional Processing (OLTP) situations, or where transactions access

multiple related objects. TPC-C [23] is a good example of such a workload, as customers access their *home* warehouse with a high probability.

While M^2Paxos finds its sweet spot for workloads exhibiting high locality, it remains correct for all other cases too. Moreover, if command conflicts involve only a pair of nodes (i.e., conflict on a single lease), M^2Paxos guarantees agreement within three communication delays in the absence of failures, like Multi-Paxos. This situation is handled by forwarding requests to the node currently owning all required leases. Another similarity to Paxos is the quorum size required for agreement, $CQ = \lfloor \frac{N}{2} \rfloor + 1$, which is minimal.

To promote scalability, M^2Paxos adopted a design that does not require a single designated leader. The protocol's design also promotes simplicity, which proved important to obtain high-performance in practice. Unlike other competitors, M^2Paxos does not need to track, record or exchange inter-command dependency relations. In our Go language implementation, this allowed M^2Paxos to reach performance levels 7× higher than the nearest competitor, EPaxos (at 49 nodes).

10.1 System Model and Consensus

Let N be the number of nodes in the system, with $\Pi = \{p_1, p_2, \ldots, p_N\}$ the set of nodes. Nodes communicate via message passing. Messages may be arbitrarily delayed, and up to f nodes may fail by crashing (they are not malicious). Nodes that do not crash are correct.

We assume existence of the weakest type of unreliable failure detector [40] that is necessary to implement a leader election service [39]. This is necessary due to the FLP result [33]. Additionally, due to [19] we must also assume that a majority of nodes $\lfloor \frac{N}{2} \rfloor + 1$ is correct, therefore similarly to classic Paxos, $f = \lfloor \frac{N}{2} \rfloor$.

We follow the definition and interfaces of Generalized Consensus [63]. Nodes use Propose*(Cmd c)* to propose commands. After agreement, the consensus layer delivers *C-structs* using the Decide*(C-struct cs)* interface. The delivered *C-structs* satisfy the following properties:

- (a) Non-triviality: a command included in a delivered *C-struct* must have been proposed by some node.
- (b) Stability: if a node delivered *C-struct cs*, then at all later times it can only deliver $cs \circ \sigma$, where σ is a sequence of commands.
- (c) Liveness: if a command c was proposed by a correct node, then c will eventually be decided in a *C-struct* at all correct nodes.
- (d) Consistency: two *C-structs* decided by different nodes are prefixes of the same *C-struct*.

We define objects as the minimal unit for partitioning data. Data ownership (leases) is defined on a per-object basis. Commands operate on at least one such object. The set of all available object in the system is LS. The set of objects accessed by a command c is denoted by $c.LS$, where $c.LS \subseteq LS$.

Objects have a set IN of positions in (also named instances) in which commands may be decided. When a command is decided, it is associated a position in for each object $l \in c.LS$. If two commands c_1 and c_2 are delivered in positions in_1 and in_2 for the same object l and $in_1 < in_2$, then c_1 will be executed before c_2.

10.2 Protocol Overview

We now present a high-level overview of M^2Paxos' main components and the way they interact in order to solve the consensus problem, and we defer the full protocol description to Chapter 11. We first explain how M^2Paxos can reach consensus in two message delays, which is the theoretical minimum, while still employing minimal sized classic quorums of cardinality $f+1$. This is possible under optimal conditions for M^2Paxos, namely when the workload exhibits temporal locality and there are no conflicts between commands proposed by different nodes.

Conversely, in less than optimal workloads with conflicts, M^2Paxos must switch to one of two slower paths, depending on the nature of the conflicts encountered. When a conflict only involves a pair of nodes (i.e., another node holds all the leases needed for the current command), M^2Paxos can reach consensus in three communication delays. Not only does M^2Paxos meet the lower bound defined for the problem of consensus in an asynchronous system in the presence of conflicts, but it also avoids incurring any penalties due to switching to a slower path. The latter property is important for effectively exploiting fast decisions, as shown in [18].

Finally, the most general (but also rare) case is when commands proposed by a node conflict with commands issued by multiple other nodes. To reach consensus in this case, M^2Paxos takes a path that requires at least four communication delays, and is unbounded. In practice, this is not a critical problem for M^2Paxos, because any workload dominated by such conflicts will stand to benefit from using classic Multi-Paxos, instead of protocols optimized for low contention such as EPaxos, Fast/Generalized Paxos, M^2Paxos.

The following three subsections each describe one of these cases.

10.2.1 The Fastest Delivery

M^2Paxos takes a different approach from other recent scalable implementations of Generalized Consensus such as EPaxos [77] and Alvin (Chapter 9). These approaches let the

proposer node be the leader of a command and coordinate with other nodes to define an ordering. M^2Paxos on the other hand, gives exactly one node in the system the authority to directly decide the delivery positions for each commands.

We need to determine which node in the system has this authority. We define:

$$Owners \subset \Pi \times LS \times IN$$

a logic association of *nodes*, *objects* and *delivery positions*. If $\langle p_i, l, in\rangle \in Owners$ is true, then node p_i has the ownership of object l for position in. We also define:

$$Decided \subset LS \times IN$$

a logic association of *nodes* with *delivery positions*. If $\langle l, in\rangle \in Decided$ is true, then a command accessing object l has been decided in position in for that object. We say a node p_i is the owner for command c if p_i has the exclusive authority to decide an instance in for every object $l \in c.LS$:

$$p_i \text{ IsOwner } c = \forall l \in c.LS, \exists in \in IN : \text{Exclusive}(p_i, l, in)$$

A node p_i has exclusive authority to decide instance l for an object l if $Owners$ contains the correct association, no other node has competing associations, and the specific instance has not been decided yet:

$$\text{Exclusive}(p_i, l, in) = \langle p_i, l, in\rangle \in Owners \wedge \nexists p_j \neq p_i : \langle p_j, l, in\rangle \in Owners \wedge \langle l, in\rangle \notin Decided$$

In the optimal situation when node p_i is the proposer of command c and the relation p_i IsOwner c is satisfied, M^2Paxos can reach consensus in two communication delays. In this situation, upon a Propose(c) request on node p_i, the node p_i broadcasts:

$$\text{Accept}\{\langle l, in\rangle | \forall l \in c.LS \ : \text{Exclusive}(p_i, l, in) = \top \wedge in \text{ is minimal}\}$$

where in are the next available positions at each object $l \in c.LS$ for which node p_i has exclusive decision authority. Afterwards, node p_i waits for a quorum of $CQ = f + 1$ AckAccept replies acknowledging receipt, in order to make sure that in case of up to f failures, at least one node remains that has witnessed its decision.

For example, let's consider two commands c_1 and c_2 proposed to the consensus layer and accessing the pairs of objects $\{A, B\}$ and respectively, $\{B, C\}$. Furthermore, suppose c_1 was decided in position $in = 1$ for both objects A and B, and c_2 was decided in position $in = 2$

for the object B and in position $in = 1$ for the object C. Therefore the sequence delivered by the consensus so far is $c_1 \circ c_2$.

At this stage, we assume that node p_i is the owner of position 2 of object A and position 3 of object be B, namely both Exclusive$(p_i, A, 2) = \top$ and Exclusive$(p_i, B, 3) = \top$. Node p_i now proposes command c_3 such that $c_3.LS = \{A, B\}$, i.e., c_3 will access objects A and B. Then p_i can simply broadcast an Accept message for c_3 with the set $ins = \{\langle A, 2\rangle, \langle B, 3\rangle\}$, requesting to accept c_3 in the final sequence after both c_1 and c_2. The next decided sequence will be $c_1 \circ c_2 \circ c_3$.

The scenario described in this section is optimal for M^2Paxos, as commands can be decided after only two message delays while employing a classic quorum of minimal size. However, this scenario can fail if nodes refuse to accept by reply with Nack, or if the relation p_i IsOwner c is not satisfied. We will describe these situations later.

10.2.2 Forwarding Requests

In a different situation from what we described above, a node p_i may propose command c without satisfying the relationship p_i IsOwner c, i.e., not having ownership for all objects accessed by c. While this is a less-than-optimal scenario, there may still be a node p_j that has all required ownership for command c, i.e., p_j IsOwner $c = \top$. This case is handled by forwarding the request to execute command c to node p_j, and relying on node p_j for a fast delivery of c. This behavior resembles the classic Multi-Paxos algorithm, where nodes forward their commands to a stable leader, and let the leader deliver them in two additional communication delays.

More specifically, upon a Propose(c) event on node p_i, where:

$$p_i \text{ IsOwner } c = \bot$$

$$\exists p_j \in \Pi | p_j \text{ IsOwner } c = \top$$

Node p_i will then forward Propose(c) to node p_j. On node p_j, the Propose(c) event will be re-raised. Assuming p_j IsOwner c is still satisfied, p_j will then broadcast an Accept(c) message according to Section 10.2.1 and wait for a quorum of $CQ = f + 1$ AckAccept replies.

Although in this scenario we can not deliver command c in two communication delays, M^2Paxos is able to deliver in three delays, which matches the lower bound for consensus in the presence of conflicts [66].

10.2.3 Requesting Ownership

In the final and most general case, there is no node p_j that has ownership of all objects accessed by some proposed command c:

$$\nexists p_j \in \Pi | p_j \text{ IsOwner } c$$

In this case, M^2Paxos must alter the *Owners* relation in order to aggregate ownership for all objects in $c.LS$ on a single node, which can be either the proposing node p_i itself (as in Section 10.2.1) or some other node (as in Section 10.2.2). In our implementation we chose to transfer the required ownership to p_i, the node having originally proposed c. While this approach is simple, effective, and allows for delivering the command c within four communication delays, we also imagine choosing other nodes to transfer ownership to in order to optimize other metrics, e.g., minimizing the number of leases altered.

Therefore node p_i, the proposer of c, attempts to become the new owner for all objects in $c.LS$. This task can be done in multiple ways, which may exhibit different behaviors according to the conflict characteristics of the workload. We first focus on a simple but effective approach first, which however is not able to provide a bound on the number of steps required to deliver command c. We later summarize an alternative approach which can bound the number of communication delays to consensus.

The Effective Approach

With this approach, when p_i is proposing a command and no node in the system has all required leases, M^2Paxos runs a Paxos prepare phase for all objects in $c.LS$. This starts a new epoch for these objects. To proceed, a quorum of nodes must reply with Promises to not accept any further requests tagged with a previous epoch. This is the same approach used in Paxos and similar algorithms when changing the leader.

Therefore, upon Propose(c) on node p_i such that

$$\nexists p_j \in \Pi | p_j \text{ IsOwner } c$$

node p_i will broadcast a message

$$\text{Prepare } \{\langle l, in, e+1\rangle | l \in c.LS\}$$

announcing to other nodes to start the next epoch (computed at sending time as $e+1$, where e is the current epoch for object l) and stop accepting any requests with an epoch number less than that. Afterwards, node p_i waits for a quorum of $CQ = f + 1$ *Promise* replies, and, provided there are no Nacks among them, can proceed to take authoritative decisions for any object in $c.LS$. More specifically, since now condition p_i IsOwner c is true, p_i can follow the procedure described in Section 10.2.1 in order to deliver command c after a total of four communication steps.

It may happen that some other node (say p_j) sending Promises has previously responded to an Accept request for some other still unstabilized command c'. In order to maintain consistency, p_j must inform p_i of this command alongside the Promise message, and p_i must deliver c' in the first position of the new epoch.

The Bounded Approach

A more adverse scenario involves multiple nodes concurrently attempting to acquire the ownership of the same object. In such a case, only one of the nodes can succeed, while the other ones will receive Nacks and must retry the acquisition attempt after a short back-off period. If this scenario is frequent, it is indicative of high inter-node conflict. M^2Paxos was not optimized for such a situation, and may lead to unbounded retries before any particular consensus instance is decided.

Alternatively, one may totally order ownership requests using another separate consensus protocol to bound communication delays. More specifically, delegating a distinguished node for ordering all ownership requests is equivalent to using a separate Multi-Paxos layer, and eliminates the need to retry failed acquisition attempts.

10.3 Evaluation

We implemented M^2Paxos and all competitors within a unified framework, written in the Go programming language [1], version 1.4rc1. Go is compiled, garbage collected and has built-in support for concurrency.

We evaluated M^2Paxos by comparing it against three other consensus algorithms: EPaxos, Generalized Paxos and Multi-Paxos. We used up to 49 nodes on Amazon EC2 infrastructure. Unless otherwise stated, each node is a c3.4xlarge instance (Intel Xeon 2.8GHz, 16 cores, 30GB RAM) running Amazon Linux 2014.09.1. All nodes were deployed under a single placement group. Network bandwidth was measured in excess of 7900mbps.

To stress the system we injected commands in an open-loop using up to 64 client threads at each node. Commands are accompanied by a 16-byte payload. After issuing each command, a client thread goes to sleep for a configurable amount of time, i.e., think time. To prevent overloading the system, we limit the number of commands still in-flight. The limit is configured for best performance under each deployment, and when it is reached, a node will skip issuing new commands. Each datapoint represents the average of at least 5 measurements.

We first evaluated M^2Paxos under its most favorable conditions. More specifically, all commands touch a single object, and a command proposed by a node can only conflict with commands proposed by the same node. This scenario is representative for strongly partitioned data, where replication is only employed for fault-tolerance.

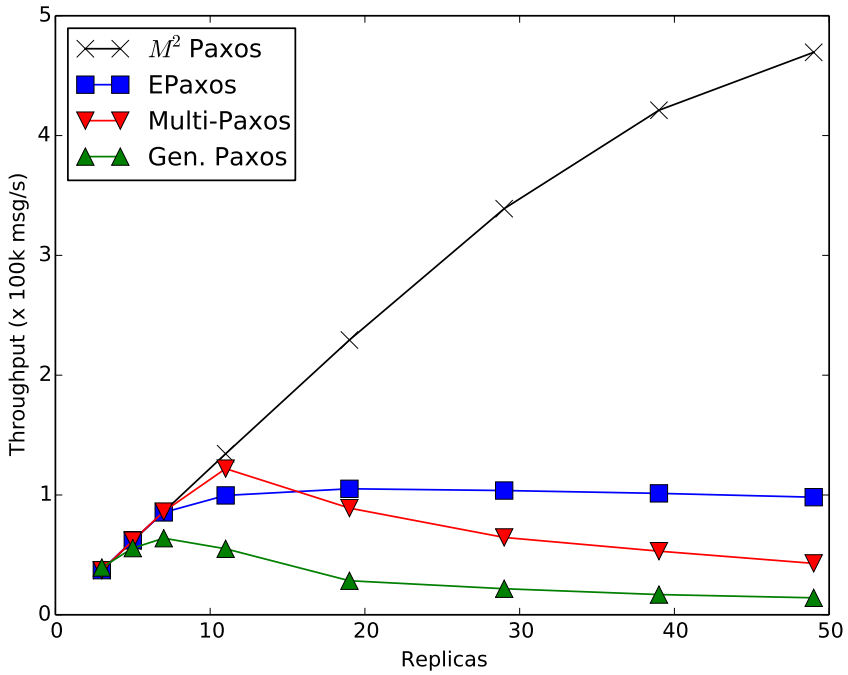
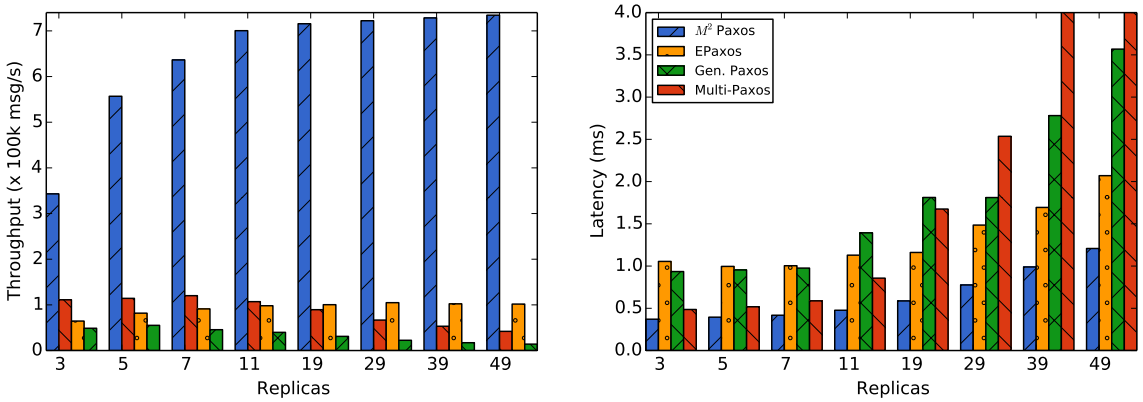


Figure 10.1: Scalability in a practical deployment. 64 client threads per node, and 5 ms think time. Command locality is 100%.



(a) Throughput (b) Latency

Figure 10.2: Scalability plot. (a) Reflects the maximum attainable throughput. (b) Shows the median latency when the system is underloaded. Command locality is 100%.

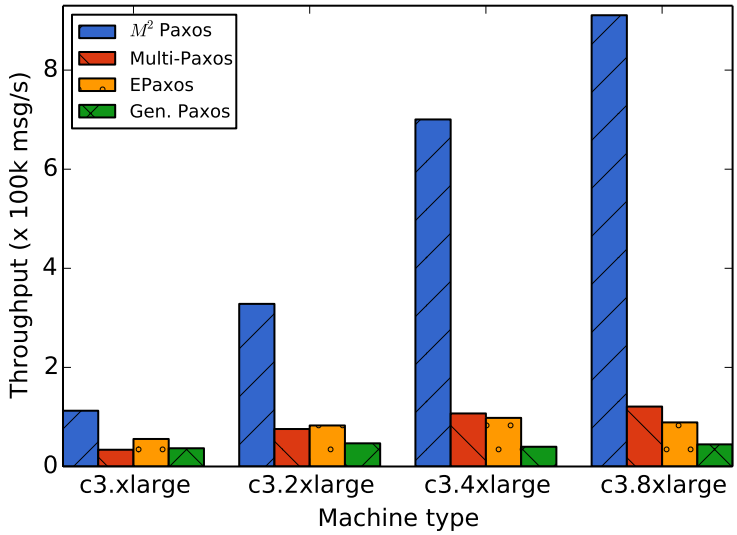


Figure 10.3: Maximum throughput for 11-nodes deployments with different machine types. The number of cores are 4, 8, 16 and 32 respectively.

We evaluated the scalability of each consensus protocol as we scaled the system up from 3 to 49 nodes. For each deployment we gradually increased the workload until the saturation point is reached. We report these results in Figure 10.2. From a throughput perspective (see Figure 10.2(a)), M^2Paxos observes a 3-11× improvement when compared to the nearest competitor. M^2Paxos exhibits great scalability up until 11 replicas. Its throughput keeps increasing past 11 nodes, albeit at a slower rate. Multi-Paxos is a distant runner-up at 11 nodes and below. After that, Multi-Paxos' performance degrades, leaving way for EPaxos, which almost manages to maintain its throughput up to the full 49 nodes.

Figure 10.2(b) shows the median command latency with an underloaded system and aggressive batching disabled. With a low number of nodes, the M^2Paxos narrowly wins over Multi-Paxos, having its latency lower by 23%. As the number of nodes is increased, M^2Paxos remains the fastest to deliver, with up to 41% better latency than EPaxos.

In practice however, a system is not easily and always maintained at full capacity. Therefore we also explored a more practical deployment with a fixed client workload at each replica, in order to assess the scalability of our proposal. The results of such a configuration are showed in Figure 10.1, where M^2Paxos, unlike all the other competitors, exhibits near-linear scalability. This is because M^2Paxos is not suffering yet from the high contention at the network layer, which becomes the bottleneck for a lightweight protocol like M^2Paxos in case of a fully loaded system.

We further evaluated how consensus protocols scale when the number of nodes in a deployment is held constant, but the CPU capacity of each node is increased from 4 to 32 cores. This is relevant for the implementations of Generalized Consensus in order to assess

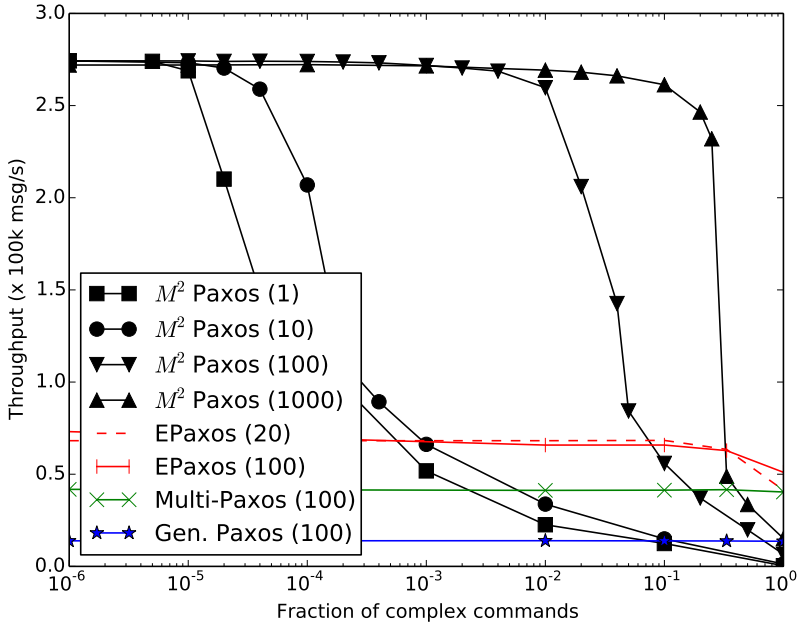
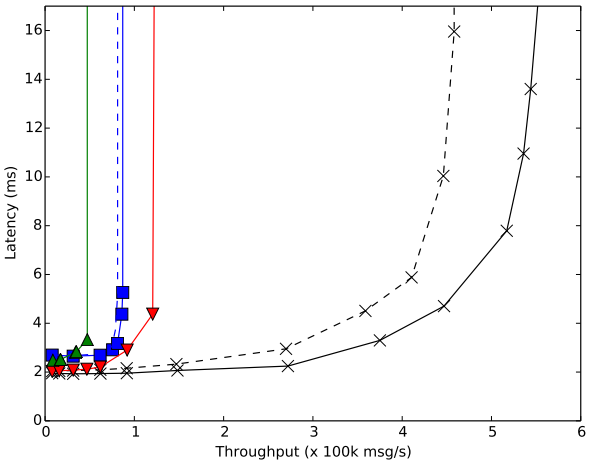


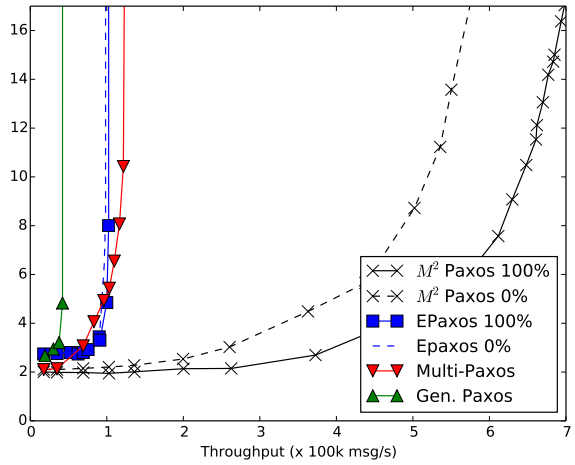
Figure 10.4: Throughput varying the fraction of complex commands. The value in paranthesis is the number of possible objects per node. The deployment consists of 49 replicas.

their ability to exploit parallelism in case of low or no conflicts among commands. To this purpose, we ran our benchmark on four classes of Amazon EC2 machines. Each class increment represents a doubling of the number of CPU cores, and an almost 2× increase in available RAM. Figure 10.3 shows the result of this experiment on four deployments of 11 nodes each. M^2Paxos exhibits great scalability up to 16 cores. Throughput still increases beyond that, but at a lower rate, as other components of the system become bottlenecked (more specifically, the networking layer). Clearly this scalability is not exhibited by single leader algorithms, and surprisingly this good trend is not observable even for EPaxos that pays the costs of dependency management.

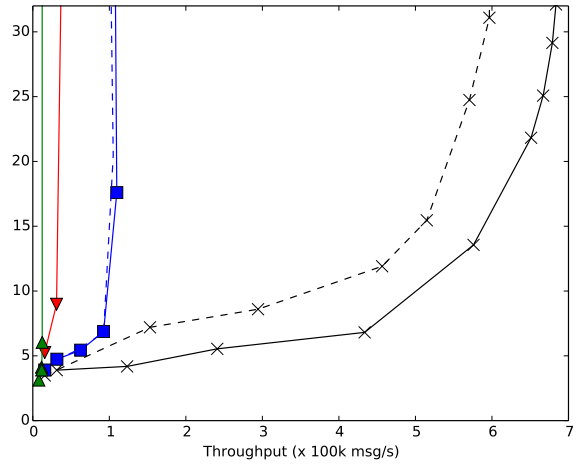
We next evaluated the behavior of M^2Paxos for workloads that do not exhibit perfect locality. Towards this purpose, we show latency vs. throughput plots for several deployments (5, 11 and 49 nodes, see Figure 10.5). For M^2Paxos and EPaxos we plot two workloads at opposite sides of the locality spectrum where commands still access one object. One workload has perfect locality (100%) and is the best case for M^2Paxos, where commands proposed by a node only conflict with commands from the same node. The other workload has no locality (0%) and is a worse case. Any other workload would fall between these two limits. Multi-Paxos and Generalized Paxos are not sensitive to locality. M^2Paxos handles non-local commands by simply forwarding them to the node that currently owns the requested object (see also Section 10.2.2). EPaxos breaks down 2-6% earlier in the workload



(a) 5 replicas



(b) 11 replicas



(c) 49 replicas

Figure 10.5: Latency vs. throughput plots, with 0% and 100% command locality for M^2Paxos and EPaxos.

with no locality. For M^2Paxos, the throughput is 13-16% lower for the non-local workload.

Finally, we introduced complex commands in our evaluation (see Figure 10.4). We define complex commands as the commands that require triggering the *Acquisition phase* because they access multiple objects, hence by potentially conflicting with commands from multiple nodes. Complex commands are similar to distributed transactions. Since M^2Paxos uses real-time backoff to ensure the progress of lease acquisitions, lease migrations are essentially rate-limited. This results in a drop in throughput as the fraction of complex commands is increased. The drop rate, corner point, and final throughput all depend on the total number of objects available in the system, because it affects the contention rate in our experiments. Multi-Paxos and Generalized Paxos are not affected by the presence of complex commands. EPaxos exhibits a small reduction in throughput as the percentage of complex commands nears 100%. However M^2Paxos is able to sustain the throughput by even using almost 50% of complex commands, in case of 1000 objects maintained per node.

10.3.1 Discussion

We evaluated M^2Paxos as a standalone protocol that solves the problem of generalized consensus. M^2Paxos is also a good candidate for using as an underlying ordering protocol in transaction execution frameworks such as Alvin. We however chose not to perform an evaluation focused on measuring transactional throughput due to the large overheads of transaction execution when compared M^2Paxos' lightweight coordination. In particular, we noticed that at maximum throughput, M^2Paxos is able to fully utilize all available cores on each node. If we additionally executed transactions, the system performance would become limited by transactional bookkeeping, while the distributed coordination through the M^2Paxos protocol would take a minimal fraction of the total CPU usage. Thus while M^2Paxos would still be the fastest competitor, the differences between M^2Paxos and its competitors would be significantly reduced.

In a geographically distributed deployment however, CPU load is not necessarily the bottleneck. M^2Paxos' fast delivery combined with the minimal quorum size directly reduce latency as one third fewer nodes need to be contacted before a decision. Lowering the latency with a fixed number of client threads, will in turn increase throughput.

Our evaluation also omitted Alvin POB, which has a communication pattern similar to EPaxos but with lighter-weight computation on the critical path. As seen in Chapter 9, POB performs slightly better than EPaxos. However, POB's performance can not approach that of M^2Paxos, because POB still tracks dependency relationships between commands and employs a relatively large fast-quorum size. For instance, the quorum size increase by itself results in a 20% drop in performance.

In this chapter we presented the M^2Paxos algorithm, a scalable and high-performance implementation of Generalized Consensus. M^2Paxos is able to decide sequences of com-

mands with an optimal cost of two communication delays, when in a conflict-free workload. M^2Paxos employs the minimal quorum size for solving consensus in asynchronous systems, i.e., $\lfloor \frac{N}{2} \rfloor + 1$, where N is the total number of nodes.

M^2Paxos thus improves upon Paxos and its variants by avoiding the bottleneck of a single designated leader to order all commands and resolve conflicts. M^2Paxos also improves upon the recent multi-leader based solutions to Generalized Consensus (e.g., EPaxos) which need to track and exchange command dependencies and also use a higher quorum size. Our evaluation study confirms these expectations, showing up to 7× better performance over state-of-the-art consensus and generalized consensus algorithms.

Chapter 11

M^2Paxos: Protocol Details

In this chapter we provide details on all aspects of M^2Paxos, including those we only briefly mentioned in Section 10.2. Given that M^2Paxos solves the problem of consensus, it provides two interfaces to interact with the application layer, namely Propose(*Cmd c*) and Decide(*C-struct cs*). The former is used by any node in the system to propose a command *c*, while the latter is used by the consensus layer to notify correct nodes about the delivery of a *C-struct cs*.

Before presenting the complete protocol, we first introduce the data structures used by M^2Paxos.

11.1 Data Structures

The following data structures are maintained at every node:

- *Decided* is a multidimensional array that provides a mapping from a tuple $\langle$*object*, *consensus instance*$\rangle$ to a command *c*. $Decided[\langle l, in\rangle] = c$ if *c* was decided in the consensus instance (position) *in* of the object *l*. Defaults to $NULL$.
- *LastDecided* is an unidimensional array that provides a mapping from an object to a consensus instance. $LastDecided[l] = in$ if the latest instance of object *l* for which the current node has observed a decision is *in*. Defaults to 0.
- *Epoch* is a multidimensional array that provides a mapping from a tuple $\langle$*object*, *consensus instance*$\rangle$ to an epoch number expressed as a non-negative integer. $Epoch[\langle l, in\rangle] = e$ if e is the current (largest) epoch number that was observed by the current node for consensus instance *in* of object *l*. Defaults to 0.

- *Owners* is a multidimensional array that provides a mapping from a tuple $\langle object,\ consensus\ instance \rangle$ to a node p_i. $Owners[\langle l, in \rangle] = p_i$ if p_i is the current owner of consensus instance in of object l. Defaults to $NULL$.

- *Rnd* is a multidimensional array that provides a mapping from a tuple $\langle object,\ consensus\ instance \rangle$ to an epoch number e. $Rnd[\langle l, in \rangle] = e$ if e is the highest epoch number in which the current node participated for the consensus instance in of object l. Defaults to 0.

- *Rdec* is a multidimensional array that provides a mapping from a tuple $\langle object,\ consensus\ instance \rangle$ to an epoch number e. $Rdec[\langle l, in \rangle] = e$ if e is the highest epoch number in which the current node has accepted a command for the consensus instance in of object l. Defaults to 0.

- *Vdec* is a multidimensional array that provides a mapping from a tuple $\langle object,\ consensus\ instance \rangle$ to a command c. $Vdec[\langle l, in \rangle] = c$ if c is the command accepted by the current node for the consensus instance in of object l. Defaults to $NULL$.

- *Acks* is a multidimensional array in which a node collects the AckAccept messages it receives in reply to any outgoing Accept requests. $Acks[\langle l, in, e \rangle] = S$, where S is a set and $\langle C, j \rangle \in S$ if the current node has received an AckAccept message with command c for consensus instance in of object l and epoch e. Defaults to $\emptyset$.

- *Cstruct* is the most recent status of the *command structures* that has been delivered by the current node. Defaults to $\perp$.

11.2 The Protocol

M^2Paxos receives commands through the Propose($Cmd\ c$) interface. Commands go through four phases before delivery: (i) coordination, (ii) acquisition, (iii) accept and (iv) decision. The following subsections describe each of the four phases.

11.2.1 Coordination Phase

The *Coordination phase* is shown in Algorithm 3 and establishes whether the command can be decided in two, three or more communication steps. This algorithm is executed whenever a command c is proposed on a node p_i using the Propose($Cmd\ c$) interface.

The first step (Line 2) in the coordination phase is to establish the set ins of instances for which we want to decide command c. The existence of a tuple $\langle l, in \rangle \in ins$ symbolizes intent to deliver c after every other command c' decided in a lower instance number in' of object l:

$$\exists \langle l, in' \rangle : Decided[\langle l, in' \rangle] = c' \wedge in' < in$$

We also say p_i intends to participate to decide c in the consensus instance in for object l. The set ins is constructed by selecting the next available instance ($LastDecided[l] + 1$) for each object $l \in c.LS$, but only if the command c was not already decided for that object.

Algorithm 3 M^2Paxos: *Coordination phase* (node p_i).

```
1:  upon Propose(Cmd c)
2:      Set ins ← {⟨l, LastDecided[l] + 1⟩ : l ∈ c.LS ∧ ∄in : Decided[⟨l, in⟩] = c}
3:      if ins = ∅ then
4:          return
5:      if IsOwner(p_i, ins) = ⊤ then
6:          Bool acc ← AcceptPhase(∅, c, ins, Epoch)
7:          if acc = ⊥ then
8:              trigger Propose(c) to p_i
9:      else if |GetOwners(ins)| = 1 then
10:         send Propose(c) to p_k ∈ GetOwners(ins)
11:         wait(timeout) until ∀l ∈ c.LS, ∃in : Decided[⟨l, in⟩] = c
12:         if ∃l ∈ c.LS, ∄in : Decided[⟨l, in⟩] = c then
13:             trigger Propose(c) to p_i
14:     else
15:         AcquisitionPhase(c)
16:
17: function Bool IsOwner(Node p_i, Set ins)
18:     for all ⟨l, in⟩ ∈ ins do
19:         if Owners[⟨l, in⟩] ≠ p_i then
20:             return ⊥
21:     return ⊤
22:
23: function Set GetOwners(Set ins)
24:     Set res ← ∅
25:     for all ⟨l, in⟩ ∈ ins do
26:         res ← res ∪ {Owners[⟨l, in⟩]}
27:     return res
```

If the resulting set ins is empty, it means the command c was already decided for every object accessed by c so the protocol can safely terminate as the delivery position for c is known. Otherwise, the algorithm branches based on the ownership relationships, with cases for a fast delivery, a three-step delivery, and a required ownership reconfiguration.

In the first case (Lines 5-8), the current node p_i already has ownership for all required objects to deliver command c, so a fast delivery in two communication steps is possible. p_i can execute the *Accept phase* for this command without transitioning to a new epoch. The protocol terminates if this phase is successful, otherwise, the *Coordination phase* has to be retried.

In the second case (Lines 9-13), there is a single node $p_k \neq p_i$ which has the ownership for all $\langle$*object*, *consensus instance*$\rangle$ tuples in ins. Thus M^2Paxos can simply request that p_k executes the *Coordination phase* for command c, potentially deciding it in two communication delays. Also considering the forwarding of command Propose(c) from p_i to p_k, command c will have been delivered in three communication delays. Node p_i keeps watch for the delivery

of command c to avoid its loss when, for instance, node p_k crashes. Should c not be delivered within a configurable time-out, p_i will restart the *Coordination phase* locally.

In the third case (Lines 14-15), no node in the system has ownership for all objects required to decide command c. Thus M^2Paxos must reconfigure ownership, to aggregate all objects from $c.LS$ on a single node. M^2Paxos enters the *Acquisition phase*, attempting to acquire said ownership on the local node p_i, which will then have the authority to execute the *Accept phase* for command c.

11.2.2 Accept phase

The *Accept phase* is shown in Algorithm 4. In this phase, a node authoritatively requests the acceptance of a command in all positions listed in *ins*, for the epochs *eps*. An Accept message is broadcast, and replies are awaited from a quorum of nodes (Lines 8-9). If this *Accept phase* is not part of an ownership reconfiguration, namely it was invoked from the *Coordination phase*, then the command to be disseminated is the command currently being proposed, c.

Algorithm 4 M^2Paxos: *Accept phase* (node p_i).

```
 1: function Bool AcceptPhase(Array toForce, Cmd c, Set ins, Array eps)
 2:     Array toDecide
 3:     for all ⟨l, in⟩ ∈ ins do
 4:         if toForce[⟨l, in⟩] = ⟨c′, −⟩ : c′ = NULL then
 5:             toDecide[⟨l, in⟩] ← c
 6:         else
 7:             toDecide[⟨l, in⟩] ← c′
 8:     send Accept(⟨toDecide, ins, eps⟩) to all p_k ∈ Π
 9:     Set replies ← receive AckAccept(⟨ins, eps, toDecide, −⟩) from Quorum
10:     if ∃⟨ins, eps, toDecide, NACK⟩ ∈ replies then
11:         return ⊥
12:     else
13:         send Decide(⟨toDecide, ins, eps⟩) to all p_k ∈ Π
14:         return ⊤
15:
16: upon Accept(⟨Array toDecide, int ins, Array eps⟩) from p_j
17:     if ∀⟨l, in⟩ ∈ ins, Rnd[⟨l, in⟩] ≤ eps[⟨l, in⟩] then
18:         ∀⟨l, in⟩ ∈ ins, Owners[⟨l, in⟩] ← p_j
19:         ∀⟨l, in⟩ ∈ ins, Vdec[⟨l, in⟩] ← toDecide[⟨l, in⟩]
20:         ∀⟨l, in⟩ ∈ ins, Rdec[⟨l, in⟩] ← eps[⟨l, in⟩]
21:         ∀⟨l, in⟩ ∈ ins, Rnd[⟨l, in⟩] ← eps[⟨l, in⟩]
22:         send AckAccept(⟨ins, eps, toDecide, ACK⟩) to all p_k ∈ Π
23:     else
24:         send AckAccept(⟨ins, eps, toDecide, NACK⟩) to p_j
```

Otherwise, *Accept* was invoked from the *Acquisition phase*, and may be required to accept a different command according to the *toForce* argument (Lines 3-7), in order to maintain system consistency as we will later see in Section 11.2.4. This happens when a node has already accepted a concurrent and conflicting command c' the same consensus instance as one of those being proposed, $\langle l, in\rangle \in ins$. In this case, p_i must collaborate towards deciding c' in position in.

The *Accept phase* can abort if any of the nodes in the quorum has transitioned to a higher epoch number for any of the objects $l \in c.LS$. This happens when another node executed the *Acquisition phase* for a conflicting command, transferring ownership of l to itself, and thus leaving p_i without authority to decide commands for consensus instances of object l. This situation can be detected when $Rnd[\langle l, in\rangle]$, the highest epoch number in which a node has participated, is higher than $eps[\langle l, in\rangle]$, the epoch number received with the current Accept request. In this case, the contacted node $p_j \in Quorum$ replies with a Nack, and the *Accept phase* fails on p_i (Lines 10-11 and 23-24).

The *Accept phase* is otherwise successful. The remote node broadcasts a positive AckAccept after updating its local state to reflect the latest accepted command for every tuple $\langle$*object*, *consensus instance*$\rangle$ from ins (Lines 18-22). More specifically, the following are updated for every $\langle l, in\rangle \in ins$ based on the incoming arguments:

- The object's owner ($Owners$).
- The last accepted command ($Vdec$).
- The greatest epoch for which the node has accepted a value ($Rdec$).
- The greatest epoch in which the node has participated in a consensus instance (Rnd).

Finally, if none of the replies in the quorum received by p_i are a Nack, p_i can inform the other nodes a decision has been reached by broadcasting a Decide message (Lines 12-14).

11.2.3 Decision phase

The *Decision phase* (Algorithm 5) is tasked to mark a command c as decided for all objects accessed by c. As such, it sets $Decided[\langle l, in\rangle] = c$.

The decision phase can be executed in two situations. Firstly, a node p_i can receive a Decide message to decide a sequence of commands *toDecide* in all positions ins (Lines 1-4). This message is sent by the owner of the objects in ins at the end of the *Accept phase*. Alternatively, a node can take this decision independently when it observes a quorum of positive AckAccept messages (Lines 6-10), without waiting for a notice from the owner.

Finally, M^2Paxos tries to deliver a *C-struct* including command c as soon as c is decided and all earlier commands have also been delivered. This condition is satisfied when c is the next command to be delivered for every object $l \in c.LS$, i.e., $in = lastDecided[l] + 1$. M^2Paxos then updates *Cstruct* appending c to it and triggers its delivery to the application layer on the local node.

Algorithm 5 M^2*Paxos*: *Decision phase* (node p_i).

```
 1: upon Decide(⟨Set toDecide, Set ins, Array eps⟩) from p_j
 2:     for all ⟨l, in⟩ ∈ ins do
 3:         if Decided[⟨l, in⟩] = NULL then
 4:             Decided[⟨l, in⟩] ← toDecide[⟨l, in⟩]
 5:
 6: upon AckAccept(⟨Set ins, Array eps, Array toDecide, ACK⟩) from p_j
 7:     for all ⟨l, in⟩ ∈ ins do
 8:         Set Acks[⟨l, in⟩][eps[⟨l, in⟩]] ← Acks[⟨l, in⟩][eps[⟨l, in⟩]] ∪ {⟨toDecide[⟨l, in⟩], j⟩}
 9:         if |Acks[⟨l, in⟩][eps[⟨l, in⟩]]| ≥ sizeof(Quorum) ∧ Decided[⟨l, in⟩] = NULL then
10:             Decided[⟨l, in⟩] ← c : ⟨c, −⟩ ∈ Acks[⟨l, in⟩][eps[⟨l, in⟩]]
11:
12: upon (∃c : ∀l ∈ c.LS, ∃in : Decided[⟨l, in⟩] = c ∧ in = LastDecided[l] + 1)
13:     Cstructs ← Cstructs • c
14:     trigger Decide(Cstructs)
15:     for all l ∈ c.LS do
16:         p_i.lastDecided[l] + +
```

11.2.4 Acquisition phase

The *Acquisition phase* (shown in Algorithm 6) is responsible for transferring the ownership of all required objects for deciding a command c to the current node p_i.

The first step is to build a set ins containing the next available consensus instance (i.e., $lastDecided[l] + 1$) for every object $l \in c.LS$ for which command c has not been decided yet. These are the instances M^2Paxos will try to get command c decided in. Furthermore, the epoch number for every tuple $\langle l, in\rangle \in ins$ is incremented and afterwards stored in an array eps. Next, all replicas are sent a Prepare request containing the instances ins and epochs eps. Execution on p_i blocks until a quorum of AckPrepare replies (also known as *promises*) is received (Lines 2-6).

If any node p_j refuses the Prepare request, it sends back a Nack. This may happen if there is at least one object which had already transitioned to an epoch equal to or greater than $eps[\langle l, in\rangle]$, prior to receiving the Prepare request. This situation is identified by comparing $eps[\langle l, in\rangle]$ to the latest stored epoch for object l on p_j, namely $Rnd[\langle l, in\rangle]$.

If p_i receives any Nacks, the *Acquisition phase* is aborted and M^2Paxos must re-attempt command c from the *Coordination phase*. This is done by triggering a new Propose(c) event.

If on the other hand node p_j does not refuse the Prepare request, it updates its Rnd with the last epoch numbers it received in eps for every tuple $\langle$*object*, *consensus instance*$\rangle$ from ins. The node thus promises to not participate to any consensus round for position $\langle l, in\rangle$ unless the epoch number is greater than $eps[\langle l, in\rangle]$.

The node p_j then replies with an AckPrepare. With the reply it also includes information about the last command c' that may have been already accepted for all $\langle l, in\rangle \in ins$, and their associated epoch number. By doing so, it forces p_i (the node that sent the Prepare request and is currently executing the *Acquisition phase*) to decide the previously accepted

c' after the *Acquisition phase* succeeds. This is necessary to maintain consistency among all replicas in the system.

Back on p_i, after a quorum of replies have been collected and if none of them is a Nack, M^2Paxos can enter the *Accept phase*. First, the function Select is used to determine the command that will be decided in this phase. For every tuple $\langle l, in\rangle \in ins$, Select chooses the command with the highest epoch number among previously accepted commands as informed through the *decs* argument in the quorum of replies. These commands, if any, are stored in an array *toForce* which is then passed to the *Accept phase*. Due to the *Acquisition phase* being an extension to multiple objects of the Paxos prepare phase, M^2Paxos inherits the useful property such that, if *toForce* is non-empty, it is guaranteed to only contain a single command.

Finally, if the *Accept phase* is unsuccessful, p_i starts a new *Coordination phase* for command c by triggering a Propose(c) event. A new *Coordination phase* is also started if p_i was forced to run the *Accept phase* for another command $c' \neq c$ due to a non-empty *toForce* array. *Acquisition phases* can additionally be rate-limited, to make sure a successful acquisition triggered by a command c is necessarily followed by a successful *Accept phase* for the same command c.

Algorithm 6 M^2Paxos: *Acquisition Phase* (node p_i).

```
1:  function Void AcquisitionPhase(Cmd c)
2:      Set ins ← {⟨l, LastDecided[l] + 1⟩ : l ∈ c.LS ∧ ∄in : Decided[⟨l, in⟩] = c}
3:      Array eps
4:      ∀⟨l, in⟩ ∈ ins, eps[⟨l, in⟩] ← ++Epoch[⟨l, in⟩]
5:      send Prepare(⟨ins, eps⟩) to all p_k ∈ Π
6:      Set replies ← receive AckPrepare(⟨ins, eps, −, −⟩) from Quorum
7:      if ∃⟨ins, eps, NACK, −⟩ ∈ replies then
8:          trigger Propose(c)
9:      else
10:         Cmd toForce ← Select(ins, replies)
11:         Bool r ← AcceptPhase(toForce, c, ins, eps)
12:         if r = ⊥ ∨ (∃l, in : toForce[⟨l, in⟩] = ⟨v, r⟩ ∧ v ≠ c) then
13:             trigger Propose(c)
14:
15: upon Prepare(⟨Set ins, Array eps⟩) from p_j
16:     if ∀⟨l, in⟩ ∈ ins, Rnd[⟨l, in⟩] < eps[⟨l, in⟩] then
17:         ∀⟨l, in⟩ ∈ ins, Rnd[⟨l, in⟩] ← eps[⟨l, in⟩]
18:         Set decs ← {⟨l, in, Vdec[⟨l, in⟩], Rdec[⟨l, in⟩]⟩ : ⟨l, in⟩ ∈ ins}
19:         send AckPrepare(⟨ins, eps, ACK, decs⟩) to p_j
20:     else
21:         send AckPrepare(⟨ins, eps, NACK, decs⟩) to p_j
22: function Set Select(Set ins, Set replies)
23:     Array toForce
24:     for all ⟨l, in⟩ ∈ ins do
25:         Epoch k ← max({r : ⟨l, in, −, r⟩ ∈ decs ∧ ⟨−, −, −, decs⟩ ∈ replies})
26:         Cmd r ← v : ⟨l, in, v, k⟩ ∈ decs ∧ ⟨−, −, −, decs⟩ ∈ replies
27:         toForce[⟨l, in⟩] ← ⟨r, k⟩
28:     return toForce
```

11.3 Correctness Arguments

In this section we provide informal arguments about the correctness of M^2Paxos. It is easy to show that the properties of Generalized Consensus are either trivially verified, or they can be inferred directly from equivalent properties of Paxos.

The *nontriviality* property is trivially guaranteed because of the way the $Cstruct$ list is grown in the *Decision phase* (Line 13 in Algorithm 5). Only commands that have previously been proposed make their way to be appended to $Cstruct$, which is then delivered to the application layer.

Assume $\exists c_x \in Cstruct$, such that c_x was not previously proposed by any node. In order for $c_x \in Cstruct$, it means it was decided in a Decision phase and was appended to *Cstruct* in Line 13 of Algorithm 5. Therefore $\exists \langle l, in \rangle : Decided[\langle l, in \rangle] = c_x$. Consequently, at least one AckAccept or Decide message was received by the current node for which $c_x \in toDecide$. Thus, an *Accept* phase must have been executed on some node in the system with either $Cmd\ c = c_x$ or $c_x \in toForce$. It is sufficient to consider only the case where $Cmd\ c = c_x$, because the case $c_x \in toForce$ implies some node in the system, during a previous execution of the algorithm, has accepted a request with $Cmd\ c = c_x$. Since an Accept phase can only be started from a from a Propose phase or an Acquisition phase, and the Acquisition phase is in turn started from a Propose phase, it follows that some node must have executed Propose(c_x), contradicting the original assumption and thus proving the *nontriviality* property.

A similar argument can be made for the *stability* property: the $Cstruct$ list is only modified by appending commands to the end. Once M^2Paxos delivers a *C-struct* cs_1 to the application layer, cs_1 will always remain a prefix of all future *C-structs* delivered on that node.

Assume a node first delivers a *C-struct* $Cstruct = cs_0 \circ c_x$, and then at a later time delivers $Cstruct = cs_0 \circ c_y \circ \sigma$, where $c_x \neq c_y$. Since operations on *Cstruct* are serialized, it follows that an operation in the M^2Paxos algorithm must have updated $Cstruct \leftarrow Cstruct - c_x$. However, M^2Paxos only ever modifies $Cstruct$ by appending new values to it on Line 13 of Algorithm 5, which contradicts our assumptions and therefore proves the *stability* property.

Liveness can be easily ensured with the simple extensions to the M^2Paxos algorithm as described in Section 10.2.3. More specifically, using Multi-Paxos for ordering ownership acquisition requests would result in the same liveness guarantees for M^2Paxos acquisitions as for Multi-Paxos, and by extension, for M^2Paxos.

Finally, *consistency* is guaranteed because Paxos guarantees that at most one command can be decided in a specific Paxos instance. This is the case because the steps executed by Paxos towards deciding a consensus instance, are the same steps M^2Paxos executes for deciding a command c in a consensus instance of a single object $l \in c.LS$. The consistency property is further supported by the following arguments:

- M^2Paxos only attempts to make a decision for a position $\langle l, in\rangle$ if the previous position (i.e., $\langle l, in-1\rangle$) has already been decided.
- A command is delivered on a node only after it has been decided in a position in for every object $l \in c.LS$.
- Commands are delivered on a node in an order consistent with the ordering of the consensus instances for all object $l \in c.LS$.

Assume two different commands are delivered in the same consensus instance by different nodes. Thus, $Decided[\langle l, in\rangle] = c_1$ on some node p_1, and $Decided[\langle l, in\rangle] = c_2$ on another node p_2. This can result in three cases: (i) node p_1 must have received a Decide message with $toDecide = c_1$, and p_2 must have received a Decide message with $toDecide = c_2$ for the same consensus instance $\langle l, in\rangle$; (ii) both nodes p_1 and p_2 receive a quorum of $f + 1$ AckAccept replies for commands c_1 and respectively c_2; and (iii) p_1 received a Decide message with $toDecide = c_1$ and p_2 received a quorum of $f + 1$ AckAccept replies for command c_2.

In all these cases, the following two situations may apply, being mutually exclusive and comprehensive. In situation A, two other different nodes p_i and p_j must have believed they hold ownership for this instance, i.e., satisfying both conditions: Exclusive(p_i, l, in) and Exclusive(p_j, l, in), while both nodes had their latest epoch number for the instance, $Epoch[\langle l, in\rangle]$ equal to the same value. Since the epoch value associated with each object is incremented atomically in Line 4 of Algorithm 6, it follows that the acquisition phase succeeded concurrently nodes p_i and p_j. Therefore, nodes p_i and p_j both obtained $f + 1$ promises (positive replies to the Prepare request) from replicas in the system. Since there are a total of $2f + 1$ nodes in the system, at least one node must have positively replied to both Prepare requests. This is impossible due to Lines 15-21 of Algorithm 6, because upon sending the first reply, a node updates its record of the latest epoch for a particular instance (Line 17), while when handling the prepare request, the node will reject it due to the check on Line 15, as it is not larger than the previous instance. Therefore, two different commands can not be delivered in the same instance according to situation A.

In situation B, node p_i executed the accept phase for command c_1 but did not yet send its Stable message. At least one node in the system receives a quorum of Acks and delivers c_1 in instance $\langle l, in\rangle$. Node p_j starts an Acquisition phase (either due to the suspected failure of node p_i or due to an ownership reorganization), and eventually delivers command c_2 in the same position as c_1. This implies that in the Accept for command c_2, the $toForce$ parameter was not equal to c_1. According to Lines 22-28 of Algorithm 6, it follows that none of the AckPrepare replies had $c_1 \in decs$, i.e. the nodes sending those replies have not already accepted c_1 in instance $\langle l, in\rangle$. This is a contradiction, as there are only $2f + 1$ nodes in the system, while $f + 1$ have previously accepted c_1 while another $f + 1$ did not. Therefore, two different commands can not be delivered in the same instance according to situation B also.

Given that situations A and B cover all possible cases, this proves the consistency property of generalized consensus.

Chapter 12

Conclusions

In this dissertation we made several contributions aimed at improving the performance of Distributed Transactional Memory. Firstly, we brought closed and open nesting to the DTM environment and evaluated their behaviors and influencing factors. We also presented Hyflow2, our new-generation DTM framework for the Java Virtual Machine with support for nesting. Secondly, we proposed an approach for automatically partitioning data in DTM workloads, with a focus on independent distributed transactions. Thirdly, we developed Alvin, a DTM framework for the geographically distributed environment which relies on a novel partial ordering protocol. Finally, we developed M^2Paxos, a generalized consensus algorithm with a scalable design, fast delivery and minimal quorums.

Closed nesting through our TFA extension, N-TFA, proved insufficient for any significant throughput improvements. It ran on average 2% faster than flat nesting, while performance for individual test varied between 42% slowdown and 84% speedup. The observed behavior was highly workload dependent, but three of our benchmarks saw an average speedup. The workloads that benefit most from closed nesting are characterized by short transactions, with between two and five sub-transactions.

Open nesting, as exemplified by our TFA-ON and SCORe-ON implementations, showed promising results. We determined performance improvement to be a trade-off of the overhead of additional commits and the fundamental conflict rate. For write-intensive, high-conflict workloads, open nesting may not be appropriate, and we observed a maximum speedup of 30%. On the other hand, for lower fundamental-conflict workloads, open nesting enabled speedups of up to 167% in our tests. We identified that open nesting has a performance sweet spot in the middle of contention range: if contention is too low, the overheads of open nesting do not justify using it over flat nesting. If on the other hand contention is too high, the aborts caused by open nesting and the compensating actions they require cause throughput degradation.

Hyflow2 is our a new, high-performance DTM framework for the JVM with support for

nesting and checkpointing. Hyflow2 is written in Scala and has a clean Scala API and a compatibility Java API. Hyflow2 is internally implemented using the actor model, and was on average two times faster than Hyflow on high-contention workloads, and up to 16 times faster, and more stable, on high-throughput, low-contention workloads. Hyflow2 stands as proof that a clean software architecture with makes performance-conscious decisions can provide significant benefits in transactional throughput.

Our second main contribution is an automated data partitioning methodology targeted to DTM environments and independent distributed transactions. Our approach is based on Schism, but applies to in-memory transactions expressed as atomic blocks instead of traditional SQL workloads, and promotes the light-weight independent transactions coordination model. Our focus on DTM enabled us to innovate, and thus we were able to contribute a static transaction analyzer, a transaction router based on machine learning, and a byte-code rewriting based log generator.

Our approach was able to generate high quality partitions, sometimes resulting in better performance than our manual partitioning efforts. The maximum improvement we observed was 4.5×, on TPC-W. This shows that data partitioning is a complicated process that is best automated, especially on complex workloads that resemble human social graphs.

Our third contribution is Alvin, a geo-replicated transactional system, and POB, the novel partial order broadcast layer and protocol Alvin is built upon. POB is multi-leader, avoiding the bottlenecks of a single distinguished node and optimizing the delivery latency. At the same time, POB avoids complex computations on the critical path, allowing it to perform better than other similar recent proposals. Alvin is also flexible, allowing a choice of consistency criteria (serializability or EUS), and fast decisions.

With our evaluation of Alvin we show that EUS is a very attractive consistency criterion in geo-distributed workloads, enabling speed-ups of up to 4.8× in throughput when compared to serialization, and if tolerated by the application. Alvin has a similar behavior to EPaxos on Bank, but is up to 26% faster on TPC-C due to the simpler computations on the critical path of a transaction and the higher complexity of the workload.

Our fourth and final contribution is M^2Paxos, an algorithm implementing generalized consensus that was designed to enable the fastest delivery to date for high-locality workloads. This is enabled by (i) the partitioning of the ordering authority among the consensus participants, (ii) a careful design to not require tracking and communicating inter-message dependencies, and (iii) the use of minimal quorums in all of our communication steps.

Our evaluation confirms that M^2Paxos has achieved its performance goals, showing up to 7× better throughput than EPaxos on 49 nodes. M^2Paxos has proved to scale well with both the system size and the capacity of individual nodes. Additionally, the latency was consistently lower than that of EPaxos, a direct consequence of our effort to minimize quorum sizes.

12.1 Contributions

To summarize, our contributions were:

- N-TFA, the first DTM algorithm with support for closed nesting. TFA-ON and SCORe-ON, the first DTM algorithms with support for open nesting.
- The first evaluation of closed and open nesting in the context of distributed transactional memory, and an analysis of the factors that influence their performance behavior in a transactional workload.
- Hyflow2, a new and highly optimized DTM framework for the JVM, written in Scala, and with support for nesting and checkpointing.
- The first methodology for automated data partitioning in a DTM environment, with a focus on promoting independent distributed transactions. Important aspects of our contribution are: (i) assigning weights based on static program analysis, (ii) machine-learning based transaction routing, (iii) log generation based on byte-code rewriting, (iv) automatic choice of an appropriate transaction model.
- Alvin, a novel geo-replicated DTM system with support for serializability and EUS, and POB, a novel partial-order broadcast layer that is multi-leader, avoids complex graph computations, and supports fast decisions.
- M^2Paxos, the first generalized consensus implementation that: (i) avoids the bottleneck of a single designated leader, (ii) employs minimal simple-majority quorums, and (iii) supports fast decisions in two communication delays.
- Open source implementations of all of the above, at https://bitbucket.org/talex/ .

12.2 Future Work

As future work we suggest improving the automatic data partitioning methodology presented in Chapters 7 and 8 in order to get better scalability and faster speed as the data-size is further increased. We propose an algorithm that groups objects together based on access patterns before invoking the graph partitioning techniques described in this dissertation. By analyzing the keys and fields commonly accessed together, we believe we may be able to drastically increase the partitioning granularity without sacrificing the quality of the results, therefore allowing the enhanced process to scale several orders of magnitude past the present limits.

For future work on Alvin we suggest implementing a mechanism for sharding data and workload on multiple nodes within a single datacenter. This would require enhancing the

Parallel Concurrency Control layer to support coordination between multiple local node. With these features, Alvin would be able to scale past the CPU and memory limits of a single node. An additional improvement for Alvin would be to support configurable partial replication, a useful feature in wake of the recent surge of interest in privacy. Support for stored-procedure based transactions can be another useful feature for Alvin.

While M^2Paxos was designed and evaluated in a local cluster setting, its features make it an appropriate contender for the geo-distributed setting. We thus suggest that M^2Paxos be evaluated in a GDS, perhaps as a building block for Alvin. M^2Paxos can be enhanced (as mentioned at the end of Section 10.2.3) by replacing its effective but simple *Acquisition phase* with a separate ordering layer such as Multi-Paxos or EPaxos, in order to improve liveness and bound the number of communication delays before agreement.

Bibliography

[1] The Go programming language. http://golang.org/.

[2] *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010.

[3] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999. AAI0800775.

[4] Kunal Agrawal, I.-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In Daniel A. Reed and Vivek Sarkar, editors, *PPOPP*. ACM, 2009.

[5] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3):5:1–5:48, November 2009.

[6] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *EuroSys 2013*.

[7] Amir and Farrokh. New approaches for efficient solution of hitting set problem. NASA JPL, 2004.

[8] Hagit Attiya, Vincent Gramoli, and Alessia Milani. A provably starvation-free distributed directory protocol. In *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, SSS'10, pages 405–419, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. *J. Parallel Distrib. Comput.*, 72(10):1386–1396, October 2012.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] Annette Bieniusa and Thomas Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In *IPDPS* [2].

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

[13] Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. CCSTM: A Library-Based STM for Scala. Scala Days, April 2010.

[14] Joao Cachopo and Antonio Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, December 2006.

[15] Nuno Carvalho, Paolo Romano, and Luis Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 376–396, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] Nuno Carvalho, Paolo Romano, and Luis Rodrigues. A Generic Framework for Replicated Software Transactional Memories. In *NCA*. IEEE Computer Society, 2011.

[17] F. Chang et al. Bigtable: a distributed storage system for structured data. In *OSDI '06*.

[18] Bernadette Charron-Bost and André Schiper. Improving fast paxos: being optimistic with no overhead. In *PRDC 2006*.

[19] Bernadette Charron-Bost and André Schiper. Uniform Consensus is Harder Than Consensus. *J. Algorithms*, 51(1):15–37, April 2004.

[20] B. F. Cooper et al. Pnuts: Yahoo!'s hosted data serving platform. *VLDB '08*.

[21] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[22] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luis Rodrigues. D2STM: Dependable Distributed Software Transactional Memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 307–313, Washington, DC, USA, 2009. IEEE Computer Society.

[23] TPC Council. TPC-C Benchmark, Revision 5.11. Feb 2010.

[24] James Cowling and Barbara Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[25] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, September 2010.

[26] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[27] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[28] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP '07*.

[29] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36, 2004.

[30] Michael J. Demmer and Maurice Herlihy. The Arrow Distributed Directory Protocol. In *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, pages 119–133, London, UK, UK, 1998. Springer-Verlag.

[31] David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*. Springer, 2006.

[32] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[33] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[34] Hector Garcia-Molina. Using Semantic Knowledge for Transaction Processing in Distributed Database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[35] G. Gibson et al. Probe: A thousand-node experimental cluster for computer systems research. In *USENIX ;login:*, 2013.

[36] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.

[37] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPOPP*, 2008.

[38] Rachid Guerraoui and Micha Kapaka. Opacity: A Correctness Condition for Transactional Memory , 2007.

[39] Rachid Guerraoui and Luis Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[40] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, March 2001.

[41] M. Hall et al. The WEKA Data Mining Software: An Update. In *SIGKDD Explorations*, 2009.

[42] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[44] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[45] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 2nd edition, 1998.

[46] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. Exploiting Locality in Lease-Based Replicated Transactional Memory via Task Migration. In *DISC 2013*.

[47] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In Siddhartha Chatterjee and Michael L. Scott, editors, *PPOPP*, pages 207–216. ACM, 2008.

[48] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[49] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In *DISC*, 2005.

[50] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. Archie: Scaling up fault-tolerant transactions with active replication. Technical report, ECE Dept., Virginia Tech, September 2013. Available at: `http://www.hyflow.org/pubs/eurosys_2014_TR.pdf`.

[51] G. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *UAI '95*.

[52] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 603–614, New York, NY, USA, 2010. ACM.

[53] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, August 2008.

[54] G. Karypis and V. Kumar. Metis - serial graph partitioning and fill-reducing matrix ordering, version 5.1, 2013.

[55] Junwhan Kim and Binoy Ravindran. Scheduling closed-nested transactions in distributed transactional memory. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 179–188, Washington, DC, USA, 2012. IEEE Computer Society.

[56] T. Kobus et al. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *ICDCS'13*.

[57] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM.

[58] Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA*, 2008.

[59] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Lujn, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *In Proc. of the International Conference on Parallel Processing (ICPP*, pages 51–58, 2008.

[60] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. EuroSys '13.

[61] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 2010.

[62] P. Lam et al. The Soot framework for Java program analysis: a retrospective. In *CETUS '11*.

[63] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[64] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[65] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[66] Leslie Lamport. Future directions in distributed computing. chapter Lower Bounds for Asynchronous Consensus. Springer-Verlag, 2003.

[67] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[68] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 265–278. USENIX Association, 2012.

[69] S. Liwen et al. Fine-grained partitioning for aggressive data skipping. In *SIGMOD 2014*.

[70] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[71] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI13), Lombard, IL*, 2013.

[72] Hatem Mahmoud et al. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 2013.

[73] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 198–208, New York, NY, USA, 2006. ACM.

[74] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *OSDI*, pages 369–384, 2008.

[75] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing, 2012.

[76] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *in HPCA*, pages 254–265, 2006.

[77] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. SOSP, 2013.

[78] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 359–370. ACM, 2006.

[79] J. Eliot B. Moss. Nested transactions: An approach to reliable distributed computing, 1981.

[80] J. Eliot B. Moss. Open nested transactions: Semantics and support (poster). In *Workshop on Mem Perf Issues*, 2006.

[81] J. Eliot B. Moss and Antony L. Hosking. Nested TM: Model and architecture sketches. *Sci Comp Prog*, 63(2):186–201, 2006.

[82] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *SIGMOD '11*, 2011.

[83] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPOPP*, 2007.

[84] S. Papadomanolakis and A. Ailamaki. Autopart: automating schema design for large scientific databases using data partitioning. In *SSDBM '04*.

[85] Marta Patino-Martinez, Ricardo Jimenez-Peris, Bettina Kemme, and Gustavo Alonso. Scalable replication in database clusters. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 315–329, London, UK, UK, 2000. Springer-Verlag.

[86] A. Pavlo et al. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *SIGMOD*, 2012.

[87] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[88] Fernando Pedone and André Schiper. Generic broadcast. In *DISC*, 1999.

[89] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Middleware '12*.

[90] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2010.

[91] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: selective multi-versioning STM. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 125–140, Berlin, Heidelberg, 2011. Springer-Verlag.

[92] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM.

[93] Matteo Pusceddu, Simone Ceccolini, Gianluca Palermo, Donatella Sciuto, and Antonino Tumeo. A Compact Transactional Memory Multiprocessor System on FPGA. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 578–581, Washington, DC, USA, 2010. IEEE Computer Society.

[94] Ross Quinlan. C4.5: Programs for machine learning. In *Morgan Kaufmann Publishers, San Mateo, CA.*, 1993.

[95] L. Rodriguez and XiaoOu Li. A dynamic vertical partitioning approach for distributed database system. In *SMC '11*.

[96] Paolo Romano, Nuno Carvalho, Maria Couceiro, Luis Rodrigues, and Joao 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.

[97] Paolo Romano, Luís Rodrigues, Nuno Carvalho, and João P. Cachopo. Cloud-TM: harnessing the cloud with distributed transactional memories. *Operating Systems Review*, 44(2):1–6, 2010.

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

[99] Mohamed M. Saad. Hyflow: A high performance dstm framework. Master's thesis, Virginia Tech, 2011.

[100] Mohamed M. Saad and Binoy Ravindran. HyFlow: a high performance distributed software transactional memory framework. In Arthur B. Maccabe and Douglas Thain, editors, *HPDC*, pages 265–266. ACM, 2011.

[101] Mohamed M. Saad and Binoy Ravindran. Snake: Control Flow Distributed Software Transactional Memory. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *SSS*, volume 6976 of *Lecture Notes in Computer Science*, pages 238–252. Springer, 2011.

[102] Mohamed M. Saad and Binoy Ravindran. Transactional forwarding algorithm. Technical report, Virginia Tech, January 2012.

[103] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[104] Rodrigo Schmidt and Fernando Pedone. A Formal Analysis of the Deferred Update Technique. In *Proceedings of the 11th International Conference on Principles of Distributed Systems*, OPODIS'07, pages 16–30, Berlin, Heidelberg, 2007. Springer-Verlag.

[105] Daniele Sciascia and Fernando Pedone. Geo-replicated storage with scalable deferred update replication. In *DSN*, 2013.

[106] Nir Shavit and Dan 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.

[107] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. SOSP, 2011.

[108] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[109] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[110] Pierre Sutra and Marc Shapiro. Fast genuine generalized consensus. In *SRDS 2011*.

[111] A Thomson et al. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD '12*.

[112] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. In *SIGMOD '91*.

[113] Alexandru Turcu, Sebastiano Peluso, Roberto Palmieri, and Binoy Ravindran. Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems. In *OPODIS 2014*.

[114] Alexandru Turcu and Binoy Ravindran. On open nesting in distributed transactional memory. Technical report, Virginia Tech, 2012.

[115] Alexandru Turcu, Binoy Ravindran, and Mohamed M. Saad. On closed nesting in distributed transactional memory. In *TRANSACT*, 2012.

[116] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 123–147, Berlin, Heidelberg, 2009. Springer-Verlag.

[117] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Trans. Database Syst.*, 16(1):132–180, 1991.

[118] Bo Zhang and Binoy Ravindran. Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, pages 268–277, Washington, DC, USA, 2009. IEEE Computer Society.

[119] Bo Zhang and Binoy Ravindran. Dynamic analysis of the relay cache-coherence protocol for DTM. In *IPDPS* [2].

[120] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. SOSP, 2013.

Appendix A

Alvin: Pseudocode and Proofs

A.1 Partial Order Broadcast Layer

Algorithm 7 Decision phase T's Leader (node N_i)

1: **upon** $\exists T : |AckProposeSet[T]| \geq FQ \wedge Decided[T] = \bot$
2: $\langle Set\ fpos, Int\ fdeps \rangle$
3: **if** $Leader(T) = N_i$ **then**
4: $Decided[T] = \top$
5: $\langle fpos, fdeps \rangle \leftarrow CDecide(AckProposeSet[T], T)$
6: **trigger** $\langle$BROADCAST[ACCEPT $\parallel$ T, $fpos$, $fdeps$, $getEpoch(T)$]$\rangle$
7:

Algorithm 8 Decision policy (node N_i)

1: $\langle Int, Set \rangle$ $CDecide(Set\ AckProposeSet, Transaction\ T)$
2: $Set\ fdeps \leftarrow \{T' \in dep' : \langle T, -, dep' \rangle \in AckProposeSet\}$
3: $Int\ fpos \leftarrow -1$
4: $Int\ maxPos \leftarrow max\{pos' : \langle T, pos', - \rangle \in AckProposeSet\}$
5: **if** $fpos = -1$ **then**
6: $fpos \leftarrow maxPos$
7: **return** $\langle fpos, fdeps \rangle$
8:

Algorithm 9 Proposal phase (node N_i)

1: **upon event** ⟨POINIT⟩ **do**
2: $Int\ N_i.mclock \leftarrow i$
3: $Array\ Decided[] \leftarrow \{\perp, \ldots, \perp\}$
4: $Array\ AckProposesSet[] \leftarrow \{\emptyset, \ldots, \emptyset\}$
5: $Array\ Uniformed[] \leftarrow \{\perp, \ldots, \perp\}$
6: $Array\ AckAcceptSet[] \leftarrow \{\emptyset, \ldots, \emptyset\}$
7: $Array\ Delivered[] \leftarrow \{\perp, \ldots, \perp\}$
8: $Array\ Promises[] \leftarrow \{\emptyset, \ldots, \emptyset\}$
9: $Array\ Promised[] \leftarrow \{\perp, \ldots, \perp\}$
10:
11: **upon event** ⟨POBROADCAST[$Transaction\ T$]⟩ **do**
12: $N_i.mclock \leftarrow N_i.mclock + N$
13: $Set\ deps_T \leftarrow DQueue.getDeps(T)$
14: **trigger** ⟨BROADCAST[PROPOSE ‖ T, $N_i.mclock$, $deps_T$, 0]⟩
15:
16: **upon event** ⟨DELIVER[PROPOSE ‖ $Transaction\ T$, $Int\ pos_T$, $Set\ deps_T$, $Int\ e$]⟩ **from** $N_j \wedge e \geq getEpoch(T)$ **do**
17: $setEpoch(T, e)$
18: $Set\ newDeps \leftarrow DQueue.getDeps(T) \cup deps_T$
19: $Int\ posLB \leftarrow max(\{pos_{T'} : T' \in newDeps\} \cup \{pos_T\})$
20: $Int\ newPos \leftarrow min(\{h : h \bmod N = j \wedge h > posLB\})$
21: $DQueue.insert(newPos, T, newDeps, PENDING)$
22: $N_i.mclock \leftarrow min(\{h : h \bmod N = i \wedge h > newPos \wedge h \geq N_i.mclock\})$
23: **trigger** ⟨SEND[ACKPROPOSE ‖ N_j, T, $newPos$, $newDeps$, e]⟩
24:
25: **upon event** ⟨DELIVER[ACKPROPOSE ‖ $Transaction\ T$, $Int\ newPos$, $Set\ deps$, $Int\ e$]⟩ **from** $N_j \wedge Decided[T] = \perp \wedge e = getEpoch(T)$**do**
26: $AckProposeSet[T] \leftarrow AckProposeSet[T] \cup \{\langle T, newPos, deps\rangle\}$
27:

Algorithm 10 Accept phase (node N_i)

1: **upon event** $\langle$DELIVER[ACCEPT $\parallel$ *Transaction* T**,** *Int pos***,** *Set deps***,** *Int* e]$\rangle$ **from** $N_j \wedge e \geq getEpoch(T)$ **do**
2: $\quad setEpoch(T, e)$
3: $\quad Set\ deltadeps \leftarrow DQueue.getDeps(T) \setminus deps$
4: $\quad DQueue.update(pos, T, deps \cup deltadeps, ACCEPTED)$
5: $\quad N_i.mclock \leftarrow min(\{h : h \bmod N = i \wedge h > pos \wedge h \geq N_i.mclock\})$
6: $\quad$ **trigger** $\langle$SEND[ACKACCEPT $\parallel$ N_j, T, pos, $deps$, $deltadeps$, e]$\rangle$
7:
8: **upon event** $\langle$DELIVER[ACKACCEPT $\parallel$ *Transaction* T**,** *Int pos***,** *Set deps***,** *Set deltadeps*]$\rangle$ **from** $N_j \wedge Uniformed[T] = \bot \wedge e = getEpoch(T)$ **do**
9: $\quad AckAcceptSet[T] \leftarrow AckAcceptSet[T] \cup \langle T, pos, deps \cup deltadeps \rangle$
10:

Algorithm 11 Delivery phase (node N_i)

1: **upon** $\exists T : |AckAcceptSet[T]| \geq CQ \wedge Uniformed[T] = \bot$
2: $\quad Uniformed[T] \leftarrow \top$
3: $\quad Set\ fdeps \leftarrow \{T' \in dep' : \langle T, -, dep' \rangle \in AckAcceptSet[T]\}$
4: $\quad Int\ fpos \leftarrow pos' : \langle T, pos', - \rangle \in AckAcceptSet[T]$
5: $\quad$ **trigger** $\langle$BROADCAST[STABLE $\parallel$ T, $fpos$, $fdeps$, $getEpoch(T)$]$\rangle$
6:
7: **upon event** $\langle$DELIVER[STABLE $\parallel$ *Transaction* T**,** *Int fpos***,** *Set fdeps***,** *Int* e]$\rangle$ **from** $N_j \wedge e \geq getEpoch(T)$ **do**
8: $\quad setEpoch(T, e)$
9: $\quad DQueue.update(fpos, T, fdeps, STABLE)$
10: $\quad N_i.mclock \leftarrow min(\{h : h \bmod N = i \wedge h > fpos \wedge h \geq N_i.mclock\})$
11:
12: **upon** $\exists T \in DQueue : Delivered[T.tid] = \bot \wedge T.status = STABLE \wedge \forall T' \in deps_T, Deliver[T'.tid] = \top$
13: $\quad$ **trigger** $\langle$PODELIVER[T, $deps_T$]$\rangle$
14: $\quad Delivered[T.tid] = \top$
15:
16: **upon** $\exists T', T'' \in DQueue : T'.status = STABLE \wedge T''.status = STABLE \wedge pos_{T''} > pos_{T'}$
17: $\quad$ **if** $T'' \in deps_{T'}$ **then**
18: $\quad\quad deps_{T'} \leftarrow deps_{T'} \setminus \{T''\}$
19:

Algorithm 12 Failure recovery (node N_i)

upon event ⟨FDSUSPECT[$Node\ N_k$]⟩ ∧ ∃$T \in DQueue : T.status \neq STABLE \wedge N_k = getLeader(T)$ **do**
  $setEpoch(T, getEpoch(T) + 1)$
  trigger ⟨BROADCAST[PREPARE ‖ T, $getEpoch(T)$]⟩

upon event ⟨DELIVER[PREPARE ‖ $Transaction\ T$**,** $Int\ e$]⟩ **from** N_j **do**
  if $e > getEpoch(T)$ **then**
    $setEpoch(T, e)$
    $[mark, pos, deps] \leftarrow DQueue.getStatus(T)$
    trigger ⟨SEND[PROMISE ‖ N_j, T, $mark$, pos, $deps$, e]⟩

upon event ⟨DELIVER[PROMISE ‖ $Transaction\ T$**,** $Status\ mark$**,** $Int\ pos$**,** $Set\ deps$**,** $Int\ e$]⟩ **from** $N_j \wedge Promised[T] = \perp \wedge e = getEpoch(T)$ **do**
  $Promises[T] \leftarrow Promises[T] \cup \langle T, mark, pos, deps, epoch\rangle$

upon $\exists T : |Promises[T]| \geq f + 1 \wedge Promised[T] = \perp$
  $Promised[T] \leftarrow \top$
  if $\exists\langle T, STABLE, pos, deps, e\rangle \in Promises[T]$ **then**
    trigger ⟨BROADCAST[STABLE ‖ T, pos, $deps$, e]⟩
  else if $\exists\langle T, ACCEPTED, pos, deps, epoch\rangle \in Promises[T]$ **then**
    trigger ⟨BROADCAST[ACCEPT ‖ T, pos, $deps$, e]⟩
  else
    $N_i.mclock \leftarrow N_i.mclock + N$
    trigger ⟨BROADCAST[PROPOSE ‖ T, $N_i.mclock$, e]⟩

Algorithm 13 Fast Decision phase T's Leader (node N_i)

upon $\exists T : |AckProposeSet[T]| \geq FQ \wedge Decided[T] = \perp$
  ⟨$Set\ fpos$, $Int\ fdeps$⟩
  if $Leader(T) = N_i$ **then**
    $Decided[T] = \top$
    $\langle fpos, fdeps\rangle \leftarrow FDecide(AckProposeSet[T], T)$
    if $fpos \neq -1$ **then**
      trigger ⟨BROADCAST[STABLE ‖ T, $fpos$, $fdeps$, $getEpoch(T)$]⟩
    else
      $\langle fpos, fdeps\rangle \leftarrow CDecide(AckProposeSet[T], T)$
      trigger ⟨BROADCAST[ACCEPT ‖ T, $fpos$, $fdeps$, $getEpoch(T)$]⟩

Algorithm 14 Fast Decision policy (node N_i)

```
1: ⟨Int, Set⟩ FDecide(Set AckProposeSet, Transaction T)
2:     Set fdeps ← {T' ∈ dep' : ⟨T, −, dep'⟩ ∈ AckProposeSet}
3:     Int fpos ← max{pos' : ⟨T, pos', −⟩ ∈ AckProposeSet}
4:     if ∀⟨T, pos', deps'⟩ ∈ AckProposeSet, pos' = fpos ∧ fdeps = deps' then
5:         return ⟨fpos, fdeps⟩
6:     else
7:         return ⟨−1, ∅⟩
8:
```

A.1.1 Correctness of POB

First we prove that the POB layer ensures properties *P1* and *P2*.

Lemma A.1.1. *For any node that delivers message $m = [T, deps_T]$ before message $m' = [T', deps_{T'}]$ and T and T' conflict, then $T \in deps_{T'}$.*

Proof. The proof follows by contradiction. We assume that there exists a pair of conflicting transactions T and T' and a node N_i such that message $m = [T, deps_T]$ is delivered before message $m' = [T', deps_{T'}]$ on N_i. Then we prove that if $T \notin deps_{T'}$ then m cannot be delivered before message m' on N_i, by contradicting the hypothesis.

Therefore let us assume that $m = [T, deps_T]$ is delivered before message $m' = [T', deps_{T'}]$ on N_i and $T \notin deps_{T'}$. In addition we name pos_T and $pos_{T'}$ respectively the final position T has in the delivery queue on N_i and the final position T' has in that queue.

In that case, each quorum for T' has never seen transaction T neither in the *Proposal* nor in the *Accept* phase, thus entailing that there exists a node N_j in a quorum for T that has inserted T' in $deps_T$ (because T and T' conflict), i.e., $T' \in deps_T$, and this happened when T' was already accepted on N_j (line 8 of Algorithm 9 executed for transaction T). Therefore the final position of T on N_i is greater than the final position of T' on N_i,i.e., $pos_T > pos_{T'}$. Note that if T' is inserted in $deps_T$ on N_j at the time T' is not already accepted, there exists another node N_k in a quorum for T' such that T is inserted in $deps_{T'}$ either in the *Proposal* or in the *Accept* phase for T' (see line 8 of Algorithm 9 and line 3 of Algorithm 10 executed for T'), by violating the hypothesis such that $T \notin deps_{T'}$.

Due to the delivery rule defined at line 12 of Algorithm 11, m cannot be delivered before m' on N_i unless N_i removes T' from $deps_T$ of m (since we have just proved that $T' \in deps_T$). That can only happens if the condition at line 16 of Algorithm 11 is verified: both m and m' are STABLE on N_i and $pos_T < pos_{T'}$. Since we have just proved that $pos_T > pos_{T'}$, T' cannot be removed from $deps_T$ and therefore m cannot be delivered before m' on N_i.

As a consequence we have that for any node that delivers message $m = [T, deps_T]$ before message $m' = [T', deps_{T'}]$ and T and T' conflict, then $T \in deps_{T'}$.

□

The Lemma A.1.1 proves that POB satisfies property *P2*. On the other hand, by using the following Lemma A.1.2 and Theorem A.1.3 we prove that POB satisfies property *P1* too.

Lemma A.1.2. *For each transaction T, if node N_j is the first that delivers T, and the delivery is performed by means of message $m = [T, deps_T]$, then every other correct node eventually delivers $m^* = [T, deps^*_T]$, where $deps^*_T \supseteq \{T' : T' \in deps_T \wedge pos_{T'} < pos_T\}$, and $pos_{T'}$, pos_T are respectively the final delivery position of T' and T.*

Proof. Node N_j has delivered $m = [T, deps_T]$ because it has received a STABLE message $m_{st} = [STABLE \parallel T, pos_T, deps_T, -]$ for T from the current leader of T, i.e. N_i (which can be possibly N_j itself), where pos_T is the final position decided for T.

In case N_i is a correct node, the lower level broadcast service guarantees that every other correct node N_k eventually receives message m, entailing that it receives message $m^* = [T, deps^*_T]$, where $deps^*_T = deps_T$. In addition the delivery rule at line 12 of Algorithm 11 and the recovery procedure of Algorithm 12 ensures that N_k eventually delivers T as soon as every other transaction $T'' \in deps_T$ has been delivered.

On the contrary, in case N_i is not a correct node, N_i may crash right after N_j (or even N_i itself) has delivered message $m = [T, deps_T]$. In this case we show that every other new correct leader of T will send a STABLE message for T equal to $m^*_{st} = [STABLE \parallel T, pos_T, deps^*_T, -]$ and where $deps^*_T \supseteq \{T' : T' \in deps_T \wedge pos_{T'} < pos_T\}$, thanks to the recovery procedure of Algorithm 12.

In fact, since N_i has sent message m_{st}, there exists a quorum Q of nodes having the message $m_{ac} = [ACCEPT \parallel T, pos_T, deps^{ac}_T, -]$ and such that $deps^{ac}_T \subseteq deps_T$ (condition at line 1 of Algorithm 11). So the new correct leader of T collects either m_{st} or at least one message m_{ac} during the recovery procedure. In the former case it broadcasts m_{st} and, if it does not crash, every other correct node eventually receives m_{st} and delivers $m = [T, deps_T]$. In the latter case, it starts an *Accept phase* by using m_{ac} and if it does not crash, it finalizes the *Delivery phase* too, with a STABLE message that is equal to m^*_{st} where $deps^*_T$ contains at least $deps^{ac}_T$, by construction of the STABLE message of lines 3-4 of Algorithm 11. Moreover $deps^*_T$ contains every other dependency T' was contained in $deps_T$ and such that $pos_{T'} < pos_T$. If this is not the case, in fact, at least a quorum for transaction T' will observe m_{ac} on at least one node and it will be associated to a position number greater than pos_T. □

Theorem A.1.3. *If some node delivers message $m = [T, deps_T]$ before message $m' = [T', deps_{T'}]$ and transactions T and T' conflict, then every node delivers $m' = [T', -]$ only after $m = [T, -]$.*

Proof. If $m = [T, deps_T]$ is delivered before $m' = [T', deps_{T'}]$ on a node N_i, it follows that $T \in deps_{T'}$ by Lemma A.1.1 and the delivery logic (line 12 of Algorithm 11) ensures that m'

cannot be delivered before m on every other node N_j even in case N_j receives the STABLE message for T' before it receives the STABLE message for T. This is because T' (and so m') cannot be delivered till the delivery of T (and so m), due to the wait condition on the delivery of the dependencies in $deps_{T'}$.

Moreover, this is still true even if the leader of T' that has decided its final position crashes, because the final message $m' = [T', deps^*_{T'}]$ is such that $T \in deps^*_{T'}$ thanks to Lemma A.1.2 and because the final position of T is less than the final position of T'.

□

A.2 Parallel Concurrency Control Layer

Algorithm 15 P-CC - Execution phase (node N_i)

```
 1: upon event ⟨BEGIN[Transaction T]⟩ do
 2:     T.sid ← N_i.CommitClock
 3:     T.RS ← ∅
 4:     T.WS ← ∅
 5:     T.origin ← N_i
 6:
 7: upon event ⟨READ[Transaction T, Obj x]⟩ do
 8:     if ∃⟨x, val⟩ ∈ T.WS then
 9:         return val
10:     Transaction owner ← getLockOwner(x)
11:     wait until owner = null ∨ owner ≠ getLockOwner(x)
12:     Ver ver ← getLastVersion(x)
13:     while ver.cClock > T.sid do
14:         ver ← ver.prev
15:         if T.writeSet ≠ ∅ then
16:             trigger ⟨TXOUTCOME[T, ⊥]⟩
17:     T.RS ← T.RS ∪ {⟨x, ver.cTid⟩}
18:     return ver.val
19:
20: upon event ⟨WRITE[Transaction T, Obj x, Value val]⟩ do
21:     T.WS ← T.WS \ {⟨x, −⟩} ∪ {⟨x, val⟩}
22:
```

Each node in the system is equipped with a parallel concurrency control (P-CC) layer that is responsible for executing transactions submitted by clients as well as processing the commit of transactions delivered by POB. P-CC works completely independently from other nodes in the system, i.e., all of its activities are executed locally on the node where it is running.

We can logically split P-CC's operations into two parts. The first part, the *execution phase*, is responsible for executing transactions optimistically. The second part, the *commit phase*, is responsible for validating and committing the optimistic execution of write transactions on all the nodes in the system, in accordance with the order provided by the POB layer.

Algorithm 16 P-CC - Commit phase (node N_i)

```
1: upon event ⟨COMMIT[Transaction T]⟩ do
2:     if T.writeSet ≠ ∅ then
3:         trigger ⟨POBROADCAST[T]⟩
4:     else
5:         trigger ⟨TXOUTCOME[T, ⊤]⟩
6:
7: upon event ⟨PODELIVER[Transaction T, Set deps]⟩ do
8:     wait until ∀T' ∈ deps, T'.completed = ⊤
9:     acquireLocks(T.tid, T.WS)
10:    Bool valid ← validate(T)
11:    if valid = ⊤ then
12:        N_i.CommitClock ← N_i.CommitClock + 1
13:        for all ⟨x, val⟩ ∈ T.WS do
14:            Ver v ← getLastVersion(x)
15:            Ver new ← ⟨val, N_i.CommitClock, T.tid⟩
16:            new.prev ← v
17:            setLastVersion(x, new)
18:    releaseLocks(T.tid, T.WS)
19:    T.completed = ⊤
20:    if T.origin = N_i then
21:        trigger ⟨TXOUTCOME[T, valid]⟩
22:
23: function validate(Transaction T) do
24:    for all ⟨x, rTid⟩ ∈ T.RS do
25:        Ver ⟨val, cClock, cTid⟩ ← getLastVersion(x)
26:        if rTid ≠ cTid then
27:            return ⊥
28:    return ⊤
29:
```

P-CC maintains meta-data for supporting transaction execution. Specifically, each node N_i has a local logical clock that is shared among all the threads running on that node, called $N_i.CommitClock$, and is incremented atomically whenever a new write transaction commits. P-CC relies on a multi-versioned repository for storing shared objects so that read operations of a transaction always return the version that is consistent with the snapshot observed so far. In particular each object x keeps a list of versions corresponding to the sequence of values committed by transactions on x. Besides the committed value val, each version ver

also stores the commit clock *cClock* of the transaction that created *ver*, along with its id, called *cTid*. As a result, $ver = \langle val, cClock, cTid \rangle$. Since all the *cClock* values are generated by monotonically increasing the $N_i.CommitClock$ and P-CC serializes commits on an object, each list of versions is ordered according to the *cClock* values.

The *execution phase* (Algorithm 15) of a transaction T starts when an application thread activates T on node N_i by means of the BEGIN event: T acquires the current value of $N_i.CommitClock$ and sets it as its snapshot id $T.sid$. $T.sid$ identifies the transactional state visible to T, namely all the versions committed before the transaction begins its execution. Write operations executed by the WRITE event handler are always buffered in a per-transaction write-set so that they can be made visible only upon the transaction's commit. On the other hand, whenever T wants to read the value of object x via the READ event, P-CC returns the version of x having the maximum *cClock* less than or equal to $T.sid$, only if T has never written to x. Otherwise, to prevent T from missing its own writes, the read operation returns the last value written in T's write-set. Read operations are also logged in a per-transaction read-set by storing, for each accessed object x, the object's identifier and the transaction's id $ver.cTid$ associated with the read version *ver*.

When a local optimistic transaction T's execution is completed, the transaction undergoes the *commit phase* (Algorithm 16). Therefore the P-CC requests a distributed validation and commitment phase for T in order to: *i)* verify that no other concurrent transactions in the system have already committed a new value on an object read by T; and *ii)* guarantee that updates on the transactional state by T are applied in the same order on all nodes with respect to other concurrent and conflicting updates. Therefore transaction T is broadcast to all nodes via the POB protocol and hence it is delivered to each node via the interface $PODeliver(T, \{T_1, \cdots, T_m\})$.

Exploiting the partial order of POB, in ALVIN a transaction can be safely validated after all the transactions belonging to its $deps_T$ set have been committed or aborted, and it does not wait for the completion of other transactions (as in total order broadcast based replication schemes [29, 105]). As an example, we consider T_1, T_2, T_3 as three transactions delivered by POB and their dependency sets $deps_{T_1}=\emptyset$, $deps_{T_2}=\{T_1\}$ and $deps_{T_3}=\{T_1\}$ respectively. In this case, the order among transactions is not total, because there is no order relation between T_2 and T_3 even if both depend on T_1. This means that the validation and commit of T_2 and T_3 can be executed in parallel as soon as T_1 either commits or aborts.

At a node N_i, the certification of a transaction consists of *i)* acquiring locks on the written objects (whose ids are in T's write-set); *ii)* validating T's read-set; *iii)* atomically incrementing $N_i.CommitClock$ by 1; *iv)* applying the updates by using the new $N_i.CommitClock$ as *cClock* only if the validation succeeds; and *v)* releasing the previously acquired locks. The lock acquisition does not generate deadlocks because conflicting transactions are delivered in order by the communication layer, but it is necessary for preventing transactions that have set their snapshot (*sid*) with a value greater than or equal to *cClock* from missing the versions that are going to be committed by T. Moreover, in case T is a read-only transaction,

only step *ii)* is performed since its write-set is empty.

The validation procedure on a node N_i checks that for each version $\langle x, rTid \rangle$ in T's read-set, the last committed version for x must be the one committed by transaction $rTid$. If even one version read by T is not valid anymore, T is aborted. Due to POB, this decision is deterministic on all nodes. When the validation of T succeeds on a node N_i, it makes the values in the write-set (if any) available to any subsequent transaction executing on N_i that starts with a snapshot id at least equal to $N_i.CommitClock$ just incremented by T. Therefore P-CC adds a new version $\langle val, Ni.CommitClock, T.tid \rangle$ as the last version of x for each tuple $\langle x, val \rangle$ in T's write-set.

A.2.1 Correctness of P-CC

We prove that P-CC satisfies EUS by showing that: *(A)* the set of committed write transactions is serializable, and *(B)* every transaction T always observes a consistent state. We do not formally prove that a transaction cannot read any value written by an aborted transaction and any intermediate value written by an executing (not committed yet) transaction, since this is trivially guaranteed: in fact P-CC makes a write operation of transaction T visible only if T commits and, for each object x written by T, only the last write operation on x by T takes effect.

Prelimiary definitions. Before proving correctness of P-CC we introduce some concepts that will be used throughout the proof. A history H is a partial order on the sequence of operations Op executed by the transactions, where Op's values are in the set $\{begin, read, write, commit, abort\}$ [3]. We denote with b_h, c_h, a_h respectively *begin* operation, *commit* operation and *abort* operation executed by transaction T_h. When a transaction T_h successfully writes a value on object x, we say that T_h writes version x_h and we denote the write operation with $w_h(x_h)$; on the other hand the notation $r_h(x_k)$ is used to indicate the transaction T_h has read version x_k of x (the one that is written by transaction T_k). In addition H implicitly induces a total order $\ll$ on committed object versions [3].

Moreover we define H^{up} as the history obtained from H by removing all ongoing, aborted and read-only transactions, i.e. H^{up} only contains committed write transactions.

We define the Direct Serialization Graph $DSG(H)$ on a history H (as in [3, 10]) as a direct graph with a vertex V_{T_h} for each committed transaction in H (H contains c_h) and a directed edge from V_{T_h} to V_{T_k} if T_h and T_k are conflicting transactions, namely there are two operations Op_h and Op_k in H on a common object x and such that at least one of them is a *write* operation. We distinguish three types of edges depending on the type of conflicts between T_h and T_k:

- *Direct read-dependency* edge if there exists an object x such that both $w_h(x_h)$ and $r_k(x_h)$ are in H. We say that T_k directly read-depends on T_h and we use the notation

$V_{T_h} \xrightarrow{wr} V_{T_k}$.

- *Direct write-dependency* edge if there exists an object x such that both $w_h(x_h)$ and $w_k(x_k)$ are in H and x_k immediately follows x_h in the total order defined by $\ll$. We say that T_k *directly write-depends* on T_h and we use the notation $V_{T_h} \xrightarrow{ww} V_{T_k}$.

- *Direct anti-dependency* edge if there exists an object x and a committed transaction T_p in H, with $p \neq h$ and $p \neq k$, such that both $r_h(x_p)$ and $w_k(x_k)$ are in H and x_k immediately follows x_p in the total order defined by $\ll$. We say that T_k *directly anti-depends* on T_h and we use the notation $V_{T_h} \overset{rw}{\dashrightarrow} V_{T_k}$.

Then a history H with a version order $\ll$ is (1-copy) serializable iff the $DSG(H)$ on H does not contain any oriented cycle [10].

Lemma A.2.1. *Given H any history executed by P-CC and H^{up} obtained from H, then the $DSG(H^{up})$ graph on H^{up} does not contain any oriented cycle.*

Proof. For each V_{T_h} in $DSG(H^{up})$ we define the CVC_h as a commit vector clock associated to transaction T_h. In particular $\|CVC_h\| = N$ and $CVC_h[i] = cClock$, where $cClock$ is the value of $N_i.CommitClock$ that T_h has used to commit on node N_i.

We prove that for each dependency edge $V_{T_h} \rightarrow V_{T_k}$ in $DSG(H^{up})$, $CVC_h < CVC_k$. We recall that for any two vectors $v1$ and $v2$ we write $v1 < v2$ if and only if $\exists i : v1[i] < v2[i] \wedge \forall j, v1[j] \leq v2[j]$. Therefore we distinguish three cases depending on the type of dependency edge:

- $V_{T_h} \xrightarrow{ww} V_{T_k}$. T_h and T_k conflict on at least one object x and T_h commits before T_k. The POB layer guarantees T_h is delivered before T_k on each node while the P-CC layer guarantees that T_k can be validated only after the completion of the commit of T_h (line 8 of Algorithm 16). This means that on each node N_j the commit clock chosen for T_k is at least greater than or equal to the commit clock chosen for T_h and incremented by 1 (line 12 of Algorithm 16). Therefore $\forall j, CVC_h[j] < CVC_k[j]$, and hence $CVC_h < CVC_k$.

- $V_{T_h} \xrightarrow{wr} V_{T_k}$. There exists a node N_i where T_k has been executed and T_k has read version x_h of an object x committed by T_h on N_i. Since the commit of T_k follows the commit of T_h on N_i, then the delivery of T_k follows the delivery of T_h on N_i, because write transactions are committed upon their delivery via the POB layer. Moreover, since T_h and T_k conflict, the POB layer guarantees that T_h and T_k are delivered in the same order on each node N_j. As in the previous case, on each node N_j the commit clock chosen for T_k is at least greater than or equal to the commit clock chosen for T_h and incremented by 1 (line 12 of Algorithm 16). Therefore $\forall j, CVC_h[j] < CVC_k[j]$, and hence $CVC_h < CVC_k$.

- $V_{T_h} \overset{rw}{\dashrightarrow} V_{T_k}$. There exists a node N_i where T_h has been executed and T_h has read version x_p of an object x committed by T_p on N_i and that is later overwritten by transaction T_k.

Since validation of T_h succeeds because T_h commits, T_k is delivered after T_h on each node N_j. As in the previous two cases, on each node N_j the commit clock chosen for T_k is at least greater than or equal to the commit clock chosen for T_h and incremented by 1 (line 12 of Algorithm 16). Therefore $\forall j, CVC_h[j] < CVC_k[j]$, and hence $CVC_h < CVC_k$.

Therefore we prove by contradiction that the $DSG(H^{up})$ graph cannot contain any oriented cycle. This is because if such a cycle exists, for each node V_{T_h} in the cycle we have $CVC_h < CVC_h$, which is clearly the absurd. □

Lemma A.2.2. *Given H any history executed by P-CC and H^{up} obtained from H. Let H^{RO} the history obtained from H^{up} by adding a read-only transaction T^{RO} contained in H. Then the $DSG(H^{RO})$ graph on H^{RO} does not contain any oriented cycle.*

Proof. The proof follows the one adopted for Lemma A.2.1. For each V_{T_h} in $DSG(H^{RO})$, where T_h is different from T^{RO}, we define the CVC_h as a commit vector clock associated to transaction T_h. In particular $\|CVC_h\| = N$ and $CVC_h[i] = cClock$, where $cClock$ is the value of $N_i.CommitClock$ that T_h has used to commit on node N_i. In addition, we associate to $V_{T^{RO}}$ the value $T^{RO}.sid$ that is the value transaction T^{RO} has used as reference clock to perform read operations, i.e. the T^{RO}'s snapshot id.

As we show in the proof of Lemma A.2.1, for each dependency edge $V_{T_h} \rightarrow V_{T_k}$ in $DSG(H^{RO})$, such that both T_h and T_k are different from T^{RO}, $CVC_h < CVC_k$.

Afterwards we prove that for each pair of edges $V_{T_h} \xrightarrow{wr} V_{T^{RO}}$ and $V_{T^{RO}} \overset{rw}{\dashrightarrow} V_{T_k}$ that involve the read-only transaction T^{RO}, there exists a node N_i such that we have $CVC_h[i] < CVC_k[i]$. This is because there exists a node N_i such that the $T^{RO}.sid$ used by T^{RO} to execute read operations on node N_i (lines 6-20 of Algorithm 15) is greater than or equal to the commit clock $CVC_h[i]$ used by T_h to commit on N_i,namely $CVC_h[i] \leq T^{RO}.sid$; in addition, since T^{RO} misses a write of transaction T_k that commits on N_i, it must be $T^{RO}.sid < CVC_k[i]$. Hence $CVC_h[i] < CVC_k[i]$, where N_i is the node the executed T^{RO}.

Therefore we prove by contradiction that the $DSG(H^{RO})$ graph cannot contain any oriented cycle. This is because if such a cycle exists, given the node N_i that executed transaction T^{RO}, for each node V_{T_h} in the cycle such that T_h is different from T^{RO} we have $CVC_h[i] < CVC_h[i]$, which is clearly the absurd. □

Theorem A.2.3. *Given H any history executed by P-CC, then H is accepted by EUS.*

Proof. If H is executed by P-CC, then the $DSG(H^{up})$ on H^{up} obtained from H does not contain any oriented cycle by Lemma A.2.1. This means that the history of write transactions committed by P-CC is 1-copy serializable.

In addition, by Lemma A.2.2, the $DSG(H^{RO})$ on H^{RO} obtained from H^{RO} by adding a read-only transaction T^{RO} from H, does not contain any cycle. This means that any read-only transaction always observes a transactional state the results from a serializable execution of

write transactions, thus always observing a consistent state. This is trivially verified also for every committed update transaction by Lemma A.2.1.

Finally, we have to notice that every ongoing or aborted transaction in P-CC, can be treated as a read-only transaction that includes all the successfully executed read operations, because write operations are made visible only if a write transaction commits.

Therefore, by Lemma A.2.1 and Lemma A.2.2, P-CC guarantees that any transaction always observes a consistent transactional state.

Summarizing, given H any history executed by P-CC, H is accepted by EUS because:

- the history of committed write transaction is H is 1-copy serializable;

- any transaction in H always observes a consistent transactional state.

□

Theorem A.2.4. *Given H any history executed by P-CC where read-only transactions are validated and committed via POB, then H is accepted by SR.*

Proof. The proof follows by Lemma A.2.1 and by considering that P-CC and POB process the commit of read-only transactions as they do for update transactions. □