Supporting Software Transactional Memory in Distributed Systems: Protocols for Cache-Coherence, Conflict Resolution and Replication

Bo Zhang

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
Robert P. Broadwater
Paul E. Plassmann
Anil Vullikanti
Yaling Yang

November 4, 2011
Blacksburg, Virginia

Keywords: Software Transactional Memory, Cache-Coherence, Distributed Queuing, Contention Management, Quorum System, Replication
Copyright 2011, Bo Zhang
Supporting Software Transactional Memory in Distributed Systems: Protocols for Cache-Coherence, Conflict Resolution and Replication

Bo Zhang

(ABSTRACT)
Lock-based synchronization on multiprocessors is inherently non-scalable, non-composable, and error-prone. These problems are exacerbated in distributed systems due to an additional layer of complexity: multinode concurrency. Transactional memory (TM) is an emerging, alternative synchronization abstraction that promises to alleviate these difficulties. With the TM model, code that accesses shared memory objects are organized as transactions, which speculatively execute, while logging changes. If transactional conflicts are detected, one of the conflicting transaction is aborted and re-executed, while the other is allowed to commit, yielding the illusion of atomicity. TM for multiprocessors has been proposed in software (STM), in hardware (HTM), and in a combination (HyTM).

This dissertation focuses on supporting the TM abstraction in distributed systems, i.e., distributed STM (or D-STM). We focus on three problem spaces: cache-coherence (CC), conflict resolution, and replication. We evaluate the performance of D-STM by measuring the competitive ratio of its makespan — i.e., the ratio of its makespan (the last completion time for a given set of transactions) to the makespan of an optimal off-line clairvoyant scheduler. We show that the performance of D-STM for metric-space networks is $O(N^2)$ for $N$ transactions requesting an object under theGreedy contention manager and an arbitrary CC protocol. To improve the performance, we propose a class of location-aware CC protocols, called Lac protocols. We show that the combination of the Greedy manager and a Lac protocol yields an $O(N \log N \cdot s)$ competitive ratio for $s$ shared objects.

We then formalize two classes of CC protocols: distributed queuing cache-coherence (DQCC) protocols and distributed priority queuing cache-coherence (DPQCC) protocols, both of which can be implemented using distributed queuing protocols. We show that a DQCC protocol is $O(N \log D_\delta)$-competitive and a DPQCC protocol is $O(\log D_\delta)$-competitive for $N$ dynamically generated transactions requesting an object, where $D_\delta$ is the normalized diameter of the underlying distributed queuing protocol. Additionally, we propose a novel CC protocol, called RELAY, which reduces the total number of aborts to $O(N)$ for $N$ conflicting transactions requesting an object, yielding a significantly improvement over past CC protocols which has $O(N^2)$ total number of aborts. We also analyze RELAY’s dynamic competitive ratio in terms of the communication cost (for dynamically generated transactions), and show that RELAY’s dynamic competitive ratio is $O(\log D_0)$, where $D_0$ is the normalized diameter of the underlying network spanning tree.

To reduce unnecessary aborts and increase concurrency for D-STM based on globally-consistent contention management policies, we propose the distributed dependency-aware (DDA) conflict resolution model, which adopts different conflict resolution strategies based on transaction types. In the DDA model, read-only transactions never abort by keeping a set of versions for each object. Each transaction only keeps precedence relations based on its local knowledge of precedence relations. We show that the DDA model ensures that 1) read-only transactions never abort, 2) every transaction eventually commits, 3) supports invisible reads, and 4) efficiently garbage collects useless object versions.

To establish competitive ratio bounds for contention managers in D-STM, we model the
distributed transactional contention management problem as the traveling salesman problem (TSP). We prove that for D-STM, any online, work conserving, deterministic contention manager provides an $\Omega(\max\{s, \sqrt{sD}\})$ competitive ratio in a network with normalized diameter $D$ and $s$ shared objects. Compared with the $\Omega(s)$ competitive ratio for multiprocessor STM, the performance guarantee for D-STM degrades by a factor proportional to $sD$. We present a randomized algorithm, called Randomized, with a competitive ratio $O(s \cdot (C \log n + \log^2 n))$ for $s$ objects shared by $n$ transactions, with a maximum conflicting degree $C$. To break this lower bound, we present a randomized algorithm Cutting, which needs partial information of transactions and an approximate TSP algorithm $A$ with approximation ratio $\phi_A$. We show that the average case competitive ratio of Cutting is $O(s \cdot \phi_A \cdot \log^2 m \log^2 n)$, which is close to $O(s)$.

Single copy (SC) D-STM keeps only one writable copy of each object, and thus cannot tolerate node failures. We propose a quorum-based replication (QR) D-STM model, which provides provable fault-tolerance without incurring high communication overhead, when compared with the SC model. The QR model stores object replicas in a tree quorum system, where two quorums intersect if one of them is a write quorum, and ensures the consistency among replicas at commit-time. The communication cost of an operation in the QR model is proportional to the communication cost from the requesting node to its closest read or write quorum. In the presence of node failures, the QR model exhibits high availability and degrades gracefully when the number of failed nodes increases, with reasonable higher communication cost.

We develop a prototype implementation of the dissertation’s proposed solutions, including DQCC and DPQCC protocols, Relay protocol, and the DDA model, in the HyFlow Java D-STM framework. We experimentally evaluated these solutions with respective competitor solutions on a set of microbenchmarks (e.g., data structures including distributed linked list, binary search tree and red-black tree) and macrobenchmarks (e.g., distributed versions of the applications in the STAMP STM benchmark suite for multiprocessors). Our experimental studies revealed that: 1) based on the same distributed queuing protocol (i.e., Ballistic CC protocol), DPQCC yields better transactional throughput than DQCC, by a factor of $50\% - 100\%$, on a range of transactional workloads; 2) Relay outperforms competitor protocols (including Arrow, Ballistic and Home) by more than $200\%$ when the network size and contention increase, as it efficiently reduces the average aborts per transaction (less than 0.5); 3) the DDA model outperforms existing contention management policies (including Greedy, Karma and Kindergarten managers) by upto $30\% - 40\%$ in high contention environments; For read/write-balanced workloads, the DDA model outperforms these contention management policies by $30\% - 60\%$ on average; for read-dominated workloads, the model outperforms by over $200\%$.

This work was supported in part by NSF CNS 0915895, NSF CNS 1116190, NSF CNS 1130180, and US NSWC under Grant N00178-09-D-3017-0011.
To my parents and Yan
Acknowledgments

I would like to gratefully acknowledge the supervision of my advisor, Dr. Binoy Ravindran during my Ph.D. study, for his enthusiasm, inspiration, and his great efforts to guide my research from the start. Particularly, when I experienced the hardest time during my study, I wouldn’t overcome the frustration and stress without his encouragement and support. This work will never happen without him.

Many thanks to the rest of my committee: Dr. Robert P. Broadwater, Dr. Paul E. Plassmann, Dr. Anil Vullikanti and Dr. Yaling Yang, for their invaluable advice and comments during my preliminary and defense exams. It is a great honor to have them serving in my committee. I am also grateful to Dr. E. Douglas Jensen, who gave me great advice and guidance in my first two years’ study.

In addition, I would like to thank all my colleagues in Real-Time Systems Laboratory, who provided my great environment for collaboration and discussion. They include: Kai Han, Jonathan Anderson, Shouwen Lai, Bo Jiang, Guanhong Pei, Fei Huang, Sherif Fahmy, Piyush Garyali, Junwhan Kim, Mohamed Saad, Peng Lu, and Alex Turcu. Their warm suggestions and help made me never feel alone in this long journey.

Last but not least, thank all my family members for their love and support. I am grateful to my parents, who always did their best in supporting my education from the childhood, and suggested me the correct direction to make my dream come true. My cousin, Lu Xu and cousin-in-law, Xinyu Zhang, helped me a lot on every aspect throughout my study. Finally, my wife, Yan Zhou, devoted her love and support to me through the ups and downs over the past years. It is difficult to overstate my gratitude to her for being such a wonderful wife.

This dissertation is dedicated to all the people who helped me and are helping me all the way.
# Contents

1 Introduction 1

1.1 Motivation and Problem Statement ................................. 1
1.2 The Dissertation Problem Space ................................... 4
1.3 Summary of Research Contributions ................................. 5
1.4 Organization .................................................................. 9

2 Past and Related Work 10

2.1 Transactional Memory: Overview ..................................... 10
2.2 Distributed Software Transactional Memory ....................... 11
2.3 Distributed Cache-Coherence Protocols ............................ 13
2.4 Conflict Resolution Strategies ...................................... 14
2.5 Replication for Software Transactional Memory ................ 15

3 Cache-Coherence D-STM Model 17

3.1 System Model and Problem Statement ............................... 17
  3.1.1 Metric-Space Network Model. .................................. 17
  3.1.2 cc D-STM Model. .................................................. 17
  3.1.3 Problem Statement. ............................................... 20
  3.1.4 Static Analysis .................................................... 20
  3.1.5 Conclusion ......................................................... 22

4 Location-Aware Cache-Coherence Protocols for D-STM 23
4.1 Motivation and Challenge .............................................. 23
4.2 Competitive Ratio Analysis of GREEDY .......................... 25
4.3 Cache Responsiveness ................................................. 30
4.4 Location-Aware CC Protocols ...................................... 30
4.5 makespan(GREEDY, LAC) for Multiple Objects ................. 33
4.6 Conclusion ............................................................. 35

5 Distributed Queuing and Distributed Priority Queuing Cache-Coherence Protocols 36
5.1 DQCC Protocols ......................................................... 36
  5.1.1 Distributed Queuing Protocols ................................. 36
  5.1.2 Protocol Description ........................................... 37
  5.1.3 Distributed-Queuing-Based Implementation .................. 39
5.2 DPQCC Protocols ....................................................... 41
  5.2.1 Protocol Description ........................................... 41
  5.2.2 Distributed-Queuing-Based Implementation .................. 43
5.3 Analysis ............................................................... 47
  5.3.1 Cost Measures ................................................... 47
  5.3.2 Cost Metrics ..................................................... 49
  5.3.3 Order Analysis .................................................. 51
  5.3.4 Comparison ....................................................... 52
5.4 Conclusion ............................................................. 55

6 RELAY Cache-Coherence Protocol 56
6.1 Rationale .............................................................. 56
6.2 Protocol Description .................................................. 58
  6.2.1 Correctness ..................................................... 61
6.3 Static Competitive ratio CR_i(RELAY) .............................. 63
6.4 Dynamic Analysis ...................................................... 64
6.4.1 Cost Measures .............................................................. 64
6.4.2 Dynamic Analysis of RELAY .............................................. 68
6.4.3 Dynamic Competitive Ratio $\rho^i$ of RELAY ......................... 70
6.5 Conclusion ................................................................. 73

7 Distributed Dependence-Aware Model with Non-Conservative Conflict Resolution Strategy 74
7.1 Motivation ................................................................. 74
7.2 Key Techniques ........................................................... 78
  7.2.1 Multi-versioning ......................................................... 78
  7.2.2 Precedence Graph ...................................................... 78
  7.2.3 Distributed Commit Protocol ....................................... 79
  7.2.4 Real-time Order Detection ......................................... 81
7.3 Expected Properties ....................................................... 83
7.4 Read/Write Algorithms .................................................. 85
  7.4.1 Description ............................................................. 85
  7.4.2 Analysis ............................................................... 88
7.5 Conclusion ................................................................. 91

8 D-STM Contention Management Problem 92
8.1 The D-STM Contention Management Problem ......................... 93
  8.1.1 Conflict Graph ........................................................ 93
  8.1.2 Problem Measure and Complexity .................................. 93
  8.1.3 Lower Bound .......................................................... 99
8.2 Algorithms ................................................................. 101
  8.2.1 Algorithm RANDOMIZED ........................................... 101
  8.2.2 Algorithm CUTTING .................................................. 103
  8.2.3 Analysis ............................................................... 105
8.3 Conclusion ................................................................. 107
9 A Quorum-Based Replication D-STM Framework 108

9.1 Motivation ......................................................... 108

9.1.1 Limited Support of Concurrent Reads .................. 109

9.1.2 Limited Locality ............................................. 110

9.2 Quorum-Based Replication D-STM Model ................ 111

9.2.1 Overview ..................................................... 111

9.2.2 Quorum Construction: FLOODING Protocol ........ 115

9.2.3 Analysis ...................................................... 117

9.3 Conclusion ....................................................... 121

10 Prototype Implementation and Experimental Results 122

10.1 HyFlow D-STM framework .................................... 122

10.2 Benchmarks ..................................................... 123

10.3 Candidate CC Protocols and Conflict Resolution Strategies . 125

10.3.1 CC Protocols ................................................. 125

10.3.2 Conflict Resolution Strategies ........................... 126

10.4 Testbed .......................................................... 127

10.5 Experimental Results ......................................... 127

10.5.1 Evaluation of DQCC and DPQCC Protocols ........... 127

10.5.2 Evaluation of RELAY ........................................ 130

10.5.3 Evaluation of the DDA Model ......................... 135

11 Conclusions and Future Work 139

11.1 Summary of Contributions ................................. 143

11.2 Future Work .................................................... 144

Bibliography 146
List of Figures

3.1 cc D-STM Model ......................................................... 18

4.1 Example: a 3-node network ........................................... 27
4.2 Link path evolution of the directory hierarchy built by BALLISTIC for Figure 4.1 28

5.1 The DQCC protocol: queue $Q_i$ at time $t$ for object $o_i$ .......... 38
5.2 Example 1: $T_j = \{write(o_1), write(o_2), read(o_3)\}$. At $t + \epsilon$, $T_j$ receives a request from $T_{j(1,2)+1}^i$ for $o_2$. .................................................. 39
5.3 $C.order(i)(r_j)$ .............................................................. 40
5.4 $P.movesuc(i)(T_j)$ ............................................................ 40
5.5 The DPQCC protocol: priority queue $Q^*_i$ enqueues $x^{th}$ invocation of $T_j$ at time $t$: $T^{x^*}_k \prec T^{x^*}_{k+1} \prec \ldots \prec T^{x^*}_{j(x,i)[-1](t)} \prec T^{x^*}_{j(x,i)} \prec T^{x^*}_{j(x,i)[1](t)} \prec \ldots$ 42
5.6 Example 2: $T^1_{j(1,1)}$ terminates at time $t$ .......................... 43
5.7 $P'.insert(i)(T_j)$ ........................................................... 44
5.8 $P'.dequeue(i)(T^*_k)$ ...................................................... 45
5.9 $P'.head(i)(T_j)$ ............................................................. 46

6.1 The ARROW Protocol ..................................................... 57
6.2 find message ................................................................. 60
6.3 $T_1 \prec T_{tail}$. .............................................................. 60
6.4 $T_{tail} \prec T_1$. .............................................................. 60
6.5 Example: $T_{tail} \prec T_2 \prec T_1$. Node $v_{tail}$ receives $find(v_2)$ after $find(v_1)$ and forwards $find(v_1)$ to $v_2$. .............................. 61
6.6 The complete execution of $T_j$ with respect to $o_i$: $T_j$ is aborted by the transaction on $v_{i,j}(m), m \geq 2$. ................................................. 65

7.1 Example 1: The Greedy manager is adopted and transaction $T_i$ has the $i^{th}$ oldest timestamp. $T_i$ can only commit at its $i^{th}$ execution. ......................... 75

7.2 Dining philosophers problem: transactions’ write version requests may be interleaved. ................................................................. 77

7.3 The commit operation which inserts object versions by traversing each object 79

7.4 The commit operation implemented by INSERTVERSION algorithm ......... 81

7.5 $T_4$ detects that $T_1 \prec_H T_4$ at $t_2$. Then at $t_3$, $T_4$’s commit is postponed after $t_4$. 82

7.6 $T_5$ detects that $T_1 \not\prec_H T_5$ at $t_3$. Then at $t_3$, $T_5$ cannot read the version written by $T_4$. ......................................................... 83

7.7 Transactions are serialized in order $T_1T_3T_2T_5T_4T_6$, where $T_6$ aborts. .... 87

7.8 Transactions are serialized in order $T_1T_2T_3T_4$, where $T_1$ and $T_2$ abort. .... 88

8.1 Conflict graph $G_c$ ................................................................. 94

8.2 Ordering conflict graph $G_c(\phi_{\kappa})$ ............................................. 95

8.3 Conflict graph $G_c$ ................................................................. 98

8.4 $G^*_c(\phi_{\kappa_1})$ and $G^*_c(\phi_{\kappa_2})$ ............................................. 98

10.1 HyFlow Architecture ................................................................. 123

10.2 A bank transaction using an atomic TransferTransaction TM method. .... 124

10.3 Hierarchical clustering of BALLISTIC ........................................ 126

10.4 The directory hierarchy of BALLISTIC ........................................ 126

10.5 Performance of Bank benchmark under HOME, B-DQCC, and B-DPQCC protocols. .......................................................... 127

10.6 Performance of Loan benchmark under HOME, B-DQCC, and B-DPQCC protocols. .......................................................... 128

10.7 Execution time for a single node to commit 100 transactions under Bank and Loan benchmarks. .......................................................... 129

10.8 Throughput of Vacation .......................................................... 129

10.9 Throughput of distributed linked list ........................................... 129

xii
10.10 Throughput of distributed binary search tree and red-black tree microbenchmarks. ....................................................... 130
10.11 Bank Throughput under HOME, ARROW, BALLISTIC and RELAY protocols. 131
10.12 The execution time for a single node to commit 100 transactions of Bank Benchmark. ................................................. 132
10.13 Average number of aborts per transaction of Bank benchmark under HOME, ARROW, BALLISTIC and RELAY protocols. 134
10.14 Bank Throughput under RELAY; 10 accounts ......................... 135
10.15 Bank Throughput under RELAY; 10ms link delay ................... 135
10.16 Bank throughput of the DDA model .................................... 136
10.17 Average number of aborts per transaction of Bank benchmark under GREEDY, KARMA, KINDERGARTEN and the DDA model. 137
10.18 The execution time for a single node to commit 100 transactions of the Bank Benchmark with 90% reads, and 5 accounts are distributed. ................. 138
# List of Algorithms

1. Data structure of an object version \texttt{Version} ............................... 78
2. Algorithm \texttt{INSERT\textsc{Version}} ............................................. 80
3. \texttt{UPDATE\textsc{Rt}(o)} algorithm for $T_i$ to update real-time order when accesses $o$ ................................. 83
4. Algorithms for read operations .................................................. 86
5. Algorithms for write operations ................................................ 87
6. Algorithm \texttt{RANDOMIZED} .................................................... 102
7. Algorithm \texttt{CUTTING} ............................................................. 104
8. QR model: read and write ......................................................... 112
9. QR model: request-commit ......................................................... 113
10. QR model: commit and abort ...................................................... 114
11. \texttt{FLOODING} protocol ....................................................... 116
Chapter 1

Introduction

1.1 Motivation and Problem Statement

Over the last few decades, much of the gain in software performance can be attributed to increases in CPU clock frequencies. However, the last few years have seen processor frequency leveling out and the focus shifting to multi-core CPUs, i.e., chips that integrate multiple processors, as a way to provide increasing computing power [1]. Nowadays, chip companies are producing multi-processors with more and more cores. As multi-core computer architectures are becoming mainstream, it is widely believed that the biggest challenge facing computer scientists and engineers today is learning how to exploit the parallelism that such architectures can offer. Only concurrent (multi-threaded) programs can effectively exploit the potential of such multi-core processors. As a result, the study of concurrency control and synchronization algorithms has become increasingly important in the new paradigm of the computing world.

The increasing parallelism in hardware offers a great opportunity for improving application performance by increasing concurrency. Unfortunately, exposing such concurrency comes at a cost: programmers now need to design programs, using existing operating system and programming language features, to deal with shared access to in-memory data objects and program synchronization. The de facto standard for programming concurrency and synchronization is threads, locks, and condition variables. Using these abstractions, programmers have been attempting to write correct concurrent code ever since multitasking operating systems made such programs possible.

However, writing correct and scalable multi-threaded programs is far from trivial. While it is well understood that shared objects must be protected from concurrent accesses to avoid data corruption, guarding individual objects is often not sufficient. Sets of semantically related actions may need to execute in mutual exclusion to avoid semantic inconsistencies. Currently, most multi-threaded applications use lock-based synchronization, which is not al-
ways adequate. Conventional synchronization methods for single and multiprocessors based on locks, semaphores, and condition variables suffer from drawbacks such as non-scalability, non-composability, potential for deadlocks/livelocks, lack of fault tolerance, and most importantly, the difficulty to reason about their correctness and the consequent programming difficulty. For example, coarse-grained locking, in which a large data structure is protected using a single lock is simple and easy to use, but permits little concurrency. In contrast, with fine-grained locking, in which each component of a data structure (e.g., a bucket of a hash table) is protected by a lock, programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Both these situations are highly prone to programmer errors. In addition, lock-based code is non-composable. For example, atomically moving an element from one hash table to another using those tables’ lock-based atomic methods (e.g., insert, delete) is not possible in a straightforward way: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the methods (of the two tables); if the methods were to export their locks, that will compromise safety. For these and other reasons, lock-based concurrent code is difficult to reason about, program, and maintain [2].

Concurrency control has been well studied in the field of database systems, where transactions have been a highly successful abstraction to make different operations access a database simultaneously without observing interference. Transactions guarantee the four so-called ACID properties [3]: atomicity, consistency, isolation, and durability. This behavior is implemented by controlling access to shared data and undoing the actions of a transaction that did not complete successfully (i.e., roll-back).

The concept of transactions has been proposed as an alternative synchronization model for shared in-memory data objects that promises to alleviate the difficulties for lock-based synchronization. Transactional memory (TM) provides programmers with constructs to delimit transactional operations and implicitly takes care of the correctness of concurrent accesses to shared data. A transaction is an explicitly delimited sequence of steps that is executed atomically by a single thread. Transactions read and write shared objects. A transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). If a transaction aborts, it is typically retried until it commits. Two transactions conflict if they access the same object and one access is a write. TM utilizes a conflict resolution approach to guarantee the correctness and make the system progress.

There have been significant recent efforts to apply the concept of transactions to shared memory data objects in multiprocessor systems. Such an attempt originated as a purely hardware solution, called hardware transactional memory (or HTM) [4, 5], was later extended to software, called software transactional memory (or STM) [6, 7, 8], and subsequently as a hardware/software combined solution (called HybridTM) [9, 10, 11, 12].

With TM, programmers express concurrency using threads, but code that accesses shared data objects are organized as transactions. Transactions execute speculatively. While doing
so, they keep track of the objects that they read and write (in a read set and a write set), and log changes to the objects that they modify (e.g., either by copying the object into an undo-log and modifying the object [13] [14]; or by copying the object into a transaction-local write-buffer and modifying the copy [15] [16][2]). Read/write or write/write conflicts between transactions are detected by checking the intersection of the logged read and write sets, either eagerly (at each read/write), or lazily (at a commit step). If a conflict is detected, a contention manager [17] is used to resolve the conflict: one of the conflicting transactions is rolled-back by aborting it, and its changes are discarded (by copying the undo-log back into the object, or by discarding the write-buffer), while the other transaction is allowed to commit by making its changes permanent (by discarding the undo-log, or by copying the write-buffer back into the object), thereby yielding the illusion of atomicity.

Aborted transactions are re-issued for execution, often immediately. A transactional scheduler [18] may delay the re-issue of an aborted transaction (to avoid future conflicts) [19], or order transaction executions to avoid or minimize potential conflicts in the first place [20][21]. Additionally, objects may be replicated for increased concurrency [22, 23, 24]: if multiple replicas of an object exist, concurrent read operations can proceed without any conflict. When a read/write or write/write conflict occurs, conflicting transactions can proceed as long as the consistency is not violated by tracking the dependencies among running transactions.

The difficulties of lock-based synchronization are exacerbated in distributed systems due to the additional layer of distributed concurrency, causing distributed versions of their centralized problem counterparts. For example, in the widely used remote procedure call (RPC) model [25], RPC calls, while holding locks, can become remotely blocked on other calls for locks, causing distributed deadlocks. Distributed livelocks, lock convoys, composability, and programmability challenges similarly occur.

These challenges have similarly motivated research on supporting the TM abstraction in distributed systems – i.e., distributed STM (or D-STM) as an alternative distributed concurrency control abstraction. Two execution models for D-STM have been studied in the past: data flow [26] and control flow [25][27][28]. In the data flow model, transactions are immobile and objects migrate to invoking transactions. A distributed cache-coherence (CC) protocol [26, 29, 30] is needed in this model: when a transaction requests a read/write operation on an object, the object may be located at a remote node. Thus, a CC protocol locates the object using a lookup protocol, and moves a copy of the object to the requesting transaction, while ensuring that only one writable copy of the object exists at any given time.1 A contention manager is also needed. When a transaction (possibly remote) requests an object, the object may currently be in use. The contention manager must decide whether to abort the transaction accessing the object and transfer the object, or delay the (remote)

1Note that, CC protocols are not needed for multiprocessor STM, because they exploit CC protocols of modern multiprocessor architectures (e.g., Intel SMPs’ MESI protocol [31]). As a matter of fact, the original HTM proposal [5] was based on extending hardware CC protocols to support transactional synchronization and recovery.
request until the current transaction commits.

In contrast, in the control flow model, objects are immobile, and transactions move from node to node (e.g., via RPCs). A two-phase distributed commit protocol [32] is needed in this model. As transactions read/write objects at a set of nodes during its execution, transactional read/write sets are distributed. Thus, detecting a conflict and deciding on commit requires coordination among the nodes – e.g., at the commit step, a coordinator requests commit votes from each transactional node and decides to commit if all nodes indicate no conflict [33].

1.2 The Dissertation Problem Space

In this dissertation, we study three problems on supporting the D-STM abstraction in the data flow model: cache-coherence, conflict resolution, and replication. We focus on the data flow D-STM model, because, past work on multiprocessor STM [17] illustrate that the data flow model can provide better performance than the control flow model on exploiting locality, reducing communication overhead, and supporting fine-grained synchronization. Our choice of these three problems is based on their fundamental nature in data flow D-STM: some form of cache-coherence is needed when objects are migrated to nodes, while (immobile) transactions concurrently request read/write operations on them. Conflict resolution is central to any STM. We study replication to achieve provable fault-tolerance in D-STM. As we show, fundamental theoretical properties of solutions to these problems and how they impact D-STM performance were previously open.

Inherited from multiprocessor STM, the performance of a D-STM is evaluated by its competitive ratio, which is the ratio of its makespan, i.e., the last completion time of a given set of transactions to the makespan of an optimal off-line clairvoyant scheduler $OPT$ [34, 35]. Therefore, the performance of a D-STM system with a CC protocol $C$ and a conflict resolution strategy $A$ is evaluated by its competitive ratio:

$$CR(A, C) = \frac{makespan(A, C)}{makespan(OPT)}.$$

The first problem that we focus is the design of cache-coherence protocols for D-STM. The communication costs for CC protocols to locate an object and move an object copy in distributed systems are orders of magnitude larger than that in multiprocessors and are often non-negligible. Such costs are often determined by the physical locations of nodes that invoke transactions, as well as that of the inherent features of the CC protocol. These costs directly affect D-STM performance. We would like to understand what is the competitive ratio of D-STM under a given contention manager and a CC protocol, and whether this ratio can be improved through CC protocols that are aware of nodes’ physical locations. Additionally, we would like to understand the dynamic competitive ratio of such CC protocols – i.e., competitive ratio for transactions generated during a time period.
Secondly, we are interested in the design of conflict resolution strategies for D-STM. This problem is two-fold. At first, we are interested in the performance of directly extending existing multiprocessor conflict resolution strategies for D-STM. Various conflict resolution strategies have been proposed in the past for multiprocessor STM (e.g., the class of online, work conserving deterministic contention managers) with provable competitive ratio bounds. It is important to understand what their bounds are for D-STM. Intuitively, such strategies cannot guarantee similar bounds for D-STM, since extending from multiprocessors to distributed systems only increases the problem complexity. If indeed, they do not guarantee similar bounds for D-STM, how can their performance be improved? On the other hand, it is desirable to design conflict resolution strategies that take into account unique characteristics of D-STM (e.g. higher communication costs) with improved performance compared with those inherited from multiprocessor STM. How to design such strategies? What is their performance?

The last problem we consider is the design of fault-tolerant D-STM. Obviously, keeping only one copy for each shared object in D-STM is inherently vulnerable in the presence of node failures. If a node failure occurs, the objects held by the failed node will be simply lost and all transactions requesting such objects would never commit. Object replication is a promising approach for building fault-tolerant D-STM. In a replicated D-STM, each object has multiple (writable) copies. A suite of replication protocols are required to manage transactional operations and guarantee the consistency among replicas of an object. How to design such protocols with provable fault-tolerance properties and efficient communication costs?

Finally, we would like to understand the performance of the proposed CC protocols and conflict resolution strategies in a practical setting. In particular, we would like to understand the relative performance of the proposed solutions over competitor solutions, on microbenchmarks (e.g., data structures) and macrobenchmarks (e.g., STAMP benchmark [36] equivalents for distributed systems).

### 1.3 Summary of Research Contributions

In this dissertation, we answer these questions. The dissertation’s solution space includes:

- Design of CC protocols with provable performance properties;
- Analysis of existing conflict resolution strategies, and the design of conflict resolution strategies for D-STM with provable performance properties;
- Design of replication protocols for D-STM with provable fault-tolerance properties; and
• Implementation of proposed protocols and their experimental evaluations on a suite of micro and macrobenchmarks.

The dissertation makes the following contributions.

**LAC protocols** [37]. We establish the competitive ratio of the GREEDY contention manager [35] with an arbitrary CC protocol in a metric-space network, where communication cost between nodes forms a metric. In the worst case, the competitive ratio is $O(N_i^2)$, where $N_i$ is the number of transactions that request an object. Based on this observation, we present location-aware CC protocols, called LAC protocols. We show that the worst-case performance of the GREEDY manager with an efficient LAC protocol is improved and predictable. We prove an $O(N \log N \cdot s)$ competitive ratio for GREEDY/LAC combination, where $N$ is the maximum number of nodes that request an object, and $s$ is the number of objects.

**DQCC and DPQCC protocols** [38]. We formalize the class of distributed queuing cache-coherence (DQCC) protocols, which encompass all CC protocols based on distributed queuing. We present the implementation of a DQCC protocol based on a distributed queuing protocol $C$. We then formalize the class of distributed priority queuing cache-coherence (DPQCC) protocols. We show that a DPQCC protocol can also be implemented based on the same distributed queuing protocol $C$. We evaluate DQCC and DPQCC protocols by measuring their competitive ratios. Given the same underlying distributed queuing protocol, we show that DPQCC protocols guarantee a worst-case competitive ratio, which is a factor proportional to $N$ better than that of DQCC protocols for $N$ conflicting transactions in a metric-space network.

**Relay protocol** [29, 39]. We present a CC protocol, called Relay, which works on a network spanning tree in a metric-space network. The Relay protocol efficiently reduces the worst-case number of total aborts to $O(N)$ for $N$ conflicting transactions requesting an object. We analyze the dynamic competitive ratio of Relay, which captures Relay’s performance when nodes initiate transactions at arbitrary times. We show that when the local execution cost for transactions is relatively small compared with the communication cost for objects to migrate in the network (up to $O(\log D)$, where $D$ is the diameter of the underlying network spanning tree), the overall communication cost is mainly determined by the choice of the distributed CC protocol. As our analysis shows, D-STM transactions involve two more variables than simple ordering requests: the local execution time and the worst-case number of aborts, which makes the dynamic analysis complex. We show that the dynamic competitive ratio of Relay is $O(\log D_0)$, if the maximum local execution time of the transactions is $O(\log D)$, where $D_0$ and $D$ are the normalized diameter and the diameter of the underlying network spanning tree, respectively.

**DDA model** [40]. We propose the distributed dependency-aware (or DDA) D-STM model, which utilizes different conflict resolution strategies targeting different types of transactions: read-only, write-only and update (involving both read and write operations) transactions. The key idea is to relax the restriction of simultaneous accesses to shared objects by setting
up precedence relations between conflicting transactions. A transaction can commit as long as its established precedence relations with other transactions are not violated. However, establishing transactional precedences in D-STM may involve large amount of communication cost between transactions.

Therefore, in the DDA model, we design a set of algorithms to avoid frequent inter-transaction communications. Read-only transactions never abort by keeping a set of versions for each object. Each transaction only keeps precedence relations based on its local knowledge of precedence relations. The proposed algorithms guarantee that, when a transaction reads from or writes to an object based on its local knowledge, the underlying precedence graph remains acyclic. In addition, we use a randomized algorithm to assign priorities to update/write-only transactions. This strategy ensures that an update transaction is efficiently processed when it potentially conflicts with another transaction, and ensures system progress.

We prove that the DDA model satisfies the opacity correctness criterion [41]. We define starvation-free multi-versioned (SF-MV)-permissiveness, which ensures that: 1) read-only transactions never abort and 2) every transaction eventually commits. The DDA model satisfies SF-MV-permissiveness with high probability. The DDA model uses a real-time useless-prefix (RT-UP)-garbage-collection (GC) mechanism, which enables it to only keep the shortest suffix of versions that might be needed by live read-only transactions. The DDA model also supports invisible reads, which is a desirable property for D-STM.

Competitive ratio bounds for D-STM contention managers. We study the D-STM contention management problem, where transactions are dynamically generated over space and time. To find an optimal scheduling algorithm, we construct a dynamic ordering conflict graph \( G_c^*(\phi(\kappa)) \) for an offline algorithm \((\kappa, \phi)\), which computes a \(k\)-coloring instance \(\kappa\) of the dynamic conflict graph \(G_c\) and processes the set of transactions in the order of \(\phi(\kappa)\). We show that the makespan of \((\phi, \kappa)\) is equivalent to the weight of the longest weighted path in \(G_c^*(\phi(\kappa))\). Therefore, finding the optimal schedule is equivalent to finding the offline algorithm \((\phi, \kappa)\) for which the weight of the longest weighted path in \(G_c^*(\phi(\kappa))\) is minimized. We illustrate that, unlike the one-shot scheduling problem (where each node only issues one transaction), when the set of transactions are dynamically generated, processing transactions according to a \(\chi(G_c)\)-coloring of \(G_c\) does not lead to an optimal schedule, where \(\chi(G_c)\) is \(G_c\)'s chromatic number. We prove that for D-STM, any online, work conserving deterministic contention manager provides an \(\Omega(\max[s, s^2/D])\) competitive ratio for \(s\) shared objects in a network with normalized diameter \(D\). Compared with the \(\Omega(s)\) competitive ratio for multiprocessor STM, the performance guarantee for D-STM degrades by a factor proportional to \(s/D\). This motivates us to design a randomized contention manager that has partial knowledge about the transactions in advance.

We present an algorithm RANDOMIZED, a randomized algorithm motivated by existing randomized algorithms for multiprocessor STM (which obviously, do not optimize the cost of moving objects in the network). RANDOMIZED uses a random initial interval for each transaction so that a transaction switches to high priority after the random initial interval expires.
A high priority transaction can only be aborted by other high priority transactions. By randomly selecting an initial interval, conflicting transactions are shifted and may be executed at different time slots so that potentially many conflicts are avoided. We show that the competitive ratio of Randomized is $O(s \cdot (C \log n + \log^2 n))$ for $s$ objects shared by $n$ transactions, with a maximum conflicting degree of $C$. The competitive ratio of Randomized is $O(C)$ factor worse than existing randomized algorithms for multiprocessor STM ([42] [43]).

We then develop an algorithm called Cutting, a randomized algorithm based on an approximate TSP algorithm $A$ with an approximation ratio $\phi_A$. Cutting divides the nodes into $O(C)$ partitions, where $C$ is the maximum degree in the conflict graph $G_c$. The cost of moving an object inside each partition is at most $\frac{\text{ATSP}_A}{C}$, where ATSP$_A$ is the total cost of moving an object along the approximate TSP path to visit each node exactly once. Cutting resolves conflicts in two phases. In the first phase, a binary tree is constructed inside each partition, and a transaction always aborts when it conflicts with its ancestor in the binary tree. In the second phase, Cutting uses a randomized priority policy to resolve conflicts. We show that the average case competitive ratio of Cutting is $O(s \cdot \phi_A \cdot \log^2 m \log^2 n)$ for $s$ objects shared by $n$ transactions invoked by $m$ nodes, which is close to the multiprocessor bound of $O(s)$ [34].

QR model [44]. We present the QR model, a quorum-based replication model for D-STM, which provides provable fault-tolerance in a failure-prone metric-space network subject to node failures. In distributed systems, a quorum is a set of nodes such that the intersection of any two quorums is non-empty if one of them is a write quorum [45]. By storing replicated copies of each object in an overlay tree quorum system, motivated by the one in [46], the QR model supports concurrent reads of transactions, and ensures the consistency among replicated copies at commit-time. The QR model has bounded communication cost for its operations, which is proportional to the communication cost from node $v$ to its closest read/write quorum, for any operation starting from $v$. Compared with directory-based CC protocols, the communication cost of operations in the QR model does not rely on the stretch of the underlying overlay tree (i.e., the worst-case ratio between the cost of direct communication between two nodes $v$ and $w$ and the cost of communication along the shortest tree path between $v$ and $w$). Thus, the QR model allows D-STM to tolerate node failures with communication cost comparable with that of the single copy D-STM model.

Implementation. We implemented the DQCC and DPQCC protocols, the Relay CC protocol, and the DDA model in the HyFlow D-STM framework [47]. (HyFlow is a Java D-STM framework that provides pluggable support for CC protocols, transactional synchronization and recovery mechanisms, contention management policies, and network communication protocols.) We experimentally evaluated the performance of the protocols with competitor protocols on a suite of micro and macrobenchmarks.

Our evaluation shows that DPQCC yields better transactional throughput than DQCC, by a factor of 50% – 100%, on macrobenchmarks (distributed version of Bank, Loan and Vacation of STAMP benchmark [36]) and micorbehmarks (distributed linked list, binary search tree
and red-black tree). DPQCC outperforms DQCC and HOME (a centralized CC protocol similar to Jackal [48]) as it better optimizes remote communication cost.

Our experimental studies show that RELAY outperforms ARROW [49], BALLISTIC [26] and HOME protocols by more than 200% when the network link delay increases. Our experimental results also illustrate the inherent advantage of RELAY: RELAY keeps the average number of aborts per transaction at a very low level (less than 0.4), which guarantees that more than 60% of transactions commit during their first execution.

Our experimental studies with the DDA model’s implementation show that, DDA outperforms competitor contention management policies including GREEDY [35], KARMA [50] and KINDERGARTEN [50] by upto 30%-40% during high contention. For read/write-balanced workloads, the DDA model outperforms these contention management policies by 30%–60% on average. For read-dominated workloads, the model outperforms by over 200%

### 1.4 Organization

The rest of the dissertation is organized as follows.

In Chapter 2, we overview the TM literature and discuss past and related efforts. We describe the D-STM model in Chapter 3. In Chapter 4, we present Lac protocols. We present and compare the DQCC and DPQCC protocols in Chapter 5. The RELAY CC protocol is detailed in Chapter 6. We describe the DDA model in Chapter 7, and study the distributed contention management problem as the traveling salesman problem (for establishing competitive ratio bounds) in Chapter 8. In Chapter 9, we present the QR D-STM model. The details of our prototype implementation and experimental results are discussed in Chapter 10. We conclude the dissertation and identify future research directions in Chapter 11.
Chapter 2

Past and Related Work

2.1 Transactional Memory: Overview

TM is motivated by database transactions — unit of work of a database system, with ACID properties [3]. In TM, concurrent threads synchronize via transactions when they access shared memory. A TM transaction is an explicitly delimited sequence of steps executed atomically by a thread [51]. Atomicity implies all-or-nothing: the sequence of steps (i.e., reads and writes) logically occur at a single instant in time; intermediate states are invisible to other transactions.

The semantic difficulty of locks and the resulting high development and maintenance costs have been the driving motivation for seeking alternate concurrency control abstractions. The design of lock-free, wait-free, or obstruction-free data structures are one such alternative. These approaches are highly performant, but significantly complex to write and reason about, and therefore, have largely been limited to a small set of basic data structures — e.g., linked lists, queues, stacks [52, 53, 54, 55, 56, 7]. For example, to the best of our knowledge, there is no lock-free implementation of a red-black tree that does not use STM (this does not imply that it is impossible to do so; it is indeed possible, but it merely indicates that the difficulty of designing such a complex data structure from basic principles has discouraged researchers from attempting it). Note that lock-freedom, wait-freedom, and obstruction-freedom are non lock-based progress guarantees and, as such, can encompass non lock-based solutions like STM. However, we use these terms here to refer to hand-crafted code that allows concurrent access to a data structure without suffering from race conditions.

The first idea of providing hardware support of transactions is due to Knight [57]. The term “transactional memory” was proposed by Herlihy and Moss [5], where they presented hardware support for lock-free data structures. The idea was soon popularized and has been the focus of research efforts since then. Following these early HTM attempts [5, 58], Shavit and Touitou proposed STM [8], which provides TM semantics in a software library. Since
then, TM APIs for multiprocessors have been proposed for hardware [4, 59, 60, 61] and software [62, 63, 6, 64, 7, 65, 66, 67, 68, 69, 70], with impressive results. Hybrid TM, which allows STM to exploit any available HTM support to improve performance, have also been proposed [9, 10, 11, 12].

One basic problem in STM is how to correctly and efficiently resolve conflicts when multiple threads access one shared object simultaneously. Generally, a conflict resolution module is responsible to ensure that all transactions eventually commit by choosing which transaction to delay or abort and when to restart the aborted transaction in case of conflicts.

The major challenge of a transactional scheduler is guaranteeing progress. Dynamic STM (DSTM) was proposed for dynamic data structures [7], which suggests that an STM would exhibit high performance if it satisfies obstruction-free property. A concurrent algorithm is obstruction-free if it guarantees that any thread, if run by itself for long enough, will make progress. One attractive property of obstruction-free STM is that supports a clean separation of concerns: correctness and progress can be addressed by different modules. In a such STM, the system seeks advice from the contention management [17] module to either wait or abort a transaction at the time of conflict.

2.2 Distributed Software Transactional Memory

Concurrency control is difficult in distributed systems — e.g., distributed race conditions are complex to reason about. The problems of locks, which are also the classical concurrency control solution for distributed systems, only exacerbate in distributed systems. Distributed deadlocks, livelocks, and lock convoys are significantly more complex to detect and cope with. The success of multiprocessor STM has therefore similarly motivated research on D-STM.

Inspired by database transactions, distributed control-flow programming models have been proposed for D-STM, such as [25][27][28]. In these systems, data objects are typically immobile, but computations move from node to node, usually via remote procedure calls (RPCs) or remote method invocation (RMI) calls. Synchronization is often provided by two-phase locking mechanism, and atomicity is ensured by a two-phase commit protocol. Thus, the difficulties of lock-based synchronization also appear in control-flow D-STMs.

D-STM can be classified based on the system architecture: cache-coherent D-STM (cc D-STM) [26], where a number of nodes are interconnected using message-passing links, and a cluster model (cluster D-STM), where a group of linked computers works closely together to form a single computer ([71, 72, 73, 74, 75, 76]). The most important difference between the two is communication cost. cc D-STM assumes a metric-space network, whereas cluster D-STM differentiates between local cluster memory and remote memory at other clusters.

Herlihy and Sun proposed data-flow distributed STM model. In this model, transactions are
immobile, and objects are dynamically migrated to invoking transactions. Object conflicts and object consistency are managed and ensured, respectively, by contention management and distributed CC protocols. In the data-flow STM model, a directory-based CC protocol is often adopted such that the latest location of the object is saved in the distributed directory and the cost to locate and move an object is bounded. Such CC protocols include BALLISTIC [26] and COMBINE [30].

While cc D-STM proposals mainly focused on theoretical properties of D-STM, several concurrent and subsequent cluster D-STM efforts developed implementations. In [71], Bocchino et al. decompose a set of existing cache-coherent TM designs into a set of design choices, and select a combination of such choices to support TM for commodity clusters. They provide a low-level API which is supposed to get integrated into some domain specific language for high productivity computer systems, and thus poses a great burden on the programmer. They show how remote communication can be aggregated with data communication to obtain excellent D-STM scalability on up to 512 processors. However, each processor is limited to one active transaction at a time. Also, in their implementation, no progress guarantees are provided, except for deadlock-freedom.

In [72], Manassiev et al. present a page-level distributed concurrency control algorithm for cluster D-STM, which automatically detects and resolves conflicts caused by data races for distributed transactions accessing shared in-memory data structures. They adopt distributed multiversioning, which uses replicas of the shared memory on each network node in combination with a distributed shared memory consistency protocol. Their implementation yields near-linear scaling for common e-commerce workloads. In their algorithm, page differences are broadcast to all other replicas, and a transaction commits successfully upon receiving acknowledgments from all nodes. A central timestamp is employed, which allows only a single update transaction to commit at a time.

Kotselidis et al. present the DiSTM D-STM framework for easy prototyping of TM CC protocols [73]. They evaluated several coherence protocols on benchmarks for clusters, one of them decentralized (Transactional Coherence and Consistency, TCC [4]). They show that, under the TCC protocol, DiSTM induces large traffic overhead at commit time, as a transaction broadcasts its read/write sets to all other transactions, which compare their read/write sets with those of the committing transaction. Using lease protocols [77], this overhead is eliminated. However, they also show that an extra validation step is added to the master node, as well as bottlenecks are created when the application entails high contention because of acquiring and releasing the leases.

Couceiro et al. present the DSTM for distributed systems [74]. Here, STM is replicated on distributed system nodes, and strong transactional consistency is enforced at transaction commit time by a non-blocking distributed certification scheme.

In [78], Kim and Ravindran develop a transactional scheduler for D-STM, called Bi-interval, that optimizes the execution order of transactional operations to minimize conflicts. Their implementation shows throughput improvement of up to 200% over cc D-STM.
Romano et al. extend D-STM for Web services [75], and Cloud platforms [76]. In [75], they present a D-STM architecture for Web services, where application’s state is replicated across distributed system nodes. Distributed TM ensures atomicity and isolation of application state updates, and consistency of the replicated state. In [76], they show how D-STM can increase the programming ease and scalability of large-scale parallel applications on the on-demand, pay-only-for-what-you-use pricing model of Cloud platforms.

2.3 Distributed Cache-Coherence Protocols

The data-flow D-STM model allows objects to move through the network to request transactions, and use distributed CC protocols to locate, move and manage the consistency of objects. Usually, these protocols employ a directory that can be tightly coupled with its registered objects, or permits objects to change their directory.

Arrow [49] is a distributed directory protocol, maintaining a distributed queue, using path reversal. The protocol runs on a fixed spanning tree. Each node keeps an “arrow” or a pointer to itself or to one of its neighbors in the tree, indicating the direction towards the node that owns the object. The path formed by all the pointers indicates the location of a sink node either holding the object or that is going to own the object. A move request redirects the pointers as it follows this path to find the sink, so the initiator of the request becomes the new sink. The communication cost of Arrow for each request operation is proportional to the stretch of the underlying spanning tree, where the stretch of a tree is the worst case ratio between the cost of direct communication between two nodes \( p \) and \( q \) in the network, and the cost of communicating along the shortest tree path between \( p \) and \( q \).

In [26], Herlihy and Sun presented a distributed CC protocol, called Ballistic, for metric-space networks, where the communication costs between nodes form a metric. The protocol’s performance is evaluated by measuring its stretch, which is the ratio of the protocol’s communication cost for obtaining a cached copy of an object to that of the optimal communication cost. Ballistic operates on a hierarchical clustering of the network: nodes are organized as clusters at different levels, where clusters are built upon maximal independent sets. For constant-doubling metrics, their protocol has amortized stretch \( O(\log D) \) in sequential executions, where \( D \) is the diameter of the metric-space network. Ballistic is a distributed queuing protocol and the worst-case queue length is \( O(n^2) \) for \( n \) concurrent conflicting requests. The hierarchical structure degrades its scalability — e.g., whenever a node joins or departs the network, the whole structure has to be rebuilt and maximal independent sets have to be recalculated.

Attiya et al. proposed Combine [30] which runs on an overlay tree, whose leaves are the nodes of the system. The authors claimed that Combine does not require fifo communication links, and avoids race conditions of Ballistic. The stretch of Combine is proportional to the stretch of the embedded overlay tree. Thus, in the worst case the stretch of Combine
is as much as the network diameter.

## 2.4 Conflict Resolution Strategies

Contention managers were first proposed in [17], and were widely applied in recent software transactional memory proposals for multiprocessors [19, 79, 80, 81]. For an STM with the obstruction-free property, a contention manager is responsible to ensure that the system as a whole makes progress. Experimental studies of contention managers can be found in [50, 82], where contention managers are evaluated based on different benchmarks. Generally, for multiprocessors, randomized algorithm yields best performance in most cases, with leaving a small chance of arbitrary large completion time. Apart from that, the choice of best contention manager varies with the used benchmarks.

The main advantage of obstruction-free synchronization algorithm is due to its clean separation of concerns of correctness and progress. The core of an obstruction-free algorithm must maintain data invariants (guaranteeing correctness), and guarantee progress when only one thread is running. On the other hand, guaranteeing progress is the responsibility of the contention manager, since transactions are often restarted.

Although transactional memory has long been the research interest, relatively fewer works have been devoted to its theoretical ramifications [41, 83, 84]. The first theoretical analysis of contention management in multiprocessors is due to Guerraoui et al. [35], where an $O(s^2)$ upper bound of competitive ratio is given for the Greedy contention manager for $s$ shared objects, compared with an optimal clairvoyant offline algorithm. Guerraoui et al. further studied in [79] the impact of transaction failures on contention management and proved the $O(ks^2)$ competitive ratio when some running transaction may abort $k$ times and then eventually commits.

Attiya et al. [34] formulated the contention management problem as a non-clairvoyant job scheduling paradigm [85, 86]. They improved the bound in [35] to $O(s)$, and the result in [79] to $O(ks)$. Furthermore, a matching lower bound of $\Omega(s)$ is given for any deterministic (and also randomized) contention manager in [34], where the adversary can alter resource requests of waiting transactions.

To obtain alternative and improved formal bounds, recent works have focused on randomized contention managers [42, 43]. Schneider and Wattenhofer [42] presented a deterministic algorithm called COMMIT_rounds with a competitive ratio $\Theta(s)$ and a randomized algorithm called RANDOMIZED_rounds with a makespan $O(C \log M)$ for $M$ concurrent transactions in separate threads with at most $C$ conflicts with high probability. In [43], Sharma et al. consider a set of $M$ transactions and $N$ transactions per thread, and present two randomized contention managers: OFFLINE-GREEDY and ONLINE-GREEDY. By knowing the conflict graph, OFFLINE-GREEDY gives a schedule with makespan $O(\tau \cdot (C + N \log MN))$ with high probability, where each transaction has the equal length $\tau$. ONLINE-GREEDY is only
Contention managers guarantee consistency by making sure that whenever there is a conflict, i.e. two transactions access a same resource and at least one writes into it, one of the transactions involved is aborted. Such transactional schedulers are called conservative [18]. While easy to implement, it is argued in [18] that such a conservative transactional scheduler may lead to significant number of unnecessary aborts, especially when high concurrency is preferred — e.g., for read-dominated workloads. On the other hand, non-conservative transactional schedulers [22, 23, 24] have been proposed to enhance concurrency by aborting a transaction only when consistency is violated. For this purpose, non-conservative transactional schedulers requires tracking conflicts and dependencies among all transactions to make a correct decision in resolving conflicts. Therefore, non-conservative approach brings more overhead compared with conservative approach in making each individual decision for the sake of reducing unnecessary aborts. Depending on the workload — the set of transactions and their characteristics, for example, their arrival times, duration, and (perhaps most importantly) the resources they read or modify, the choice of conflict resolution strategy may be different for the best achievable performance.

2.5 Replication for Software Transactional Memory

To achieve high availability in the presence of network failures, keeping only one copy of each object in the system is not sufficient. Inherited from database systems, replication is a promising approach to build fault-tolerant D-STM systems, where each object has multiple (writable) copies. D²STM [74] is the first fault-tolerant D-STM which provides cluster-wide consistency and availability guarantees in scenarios of failures. D²STM adopts an optimistic certification method, which relies on a commit-time atomic broadcast based distributed validation to ensure global consistency. Motivated by database replication schemes, distributed certification based on atomic broadcast [87] avoids the costs of replica coordination during the execution phase and runs transactions locally in an optimistic fashion.

Carvalho et al. proposed Asynchronous Lease Certification (ALC) D-STM replication scheme in [88], which overcomes some drawbacks of the atomic broadcast based replication scheme in [74]. ALC reduces the replica coordination overhead and avoid unnecessary aborts due to conflicts originated on remote nodes by employing the notion of asynchronous lease. ALC relies on Uniform Reliable Broadcast [87] to disseminate exclusively the writesets, which reduces the inter-replica synchronization overhead.

Aforementioned cc D-STM proposals assume that only a single (writable) copy is kept in the system [26, 30]. Therefore, these solutions are inherently vulnerable in the presence of node and link failures. If a node failure occurs, the objects held by the failed node will be simply lost and all following transactions requesting such objects would never commit. Hence, they cannot afford any node failures. BALLISTIC also assumes a reliable and fifo logical link
between nodes, since they may not perform well when the message is reordered [30]. On the other hand, COMBINE can tolerate partial link failures and support non-fifo message delivery, as long as a logical link exists between any pair of nodes. However, similar to other directory-based CC protocols, COMBINE does not permit network partitioning incurred by link failures, which may make some objects inaccessible from outer transactions. In general, single-copy D-STM model is not suitable in a network environment with aforementioned node/link failures.

Most existing replicated D-STM solutions are solely proposed for cluster-based D-STM ([71, 72, 73, 74, 88]). As we mentioned before, these solutions all require some form of broadcast to maintain consistency among replicas and assume a uniform communication cost across all pairs of nodes. Directly applying these solutions to cc D-STM may not lead to the similar performance as cluster-based D-STM scenarios.
Chapter 3

Cache-Coherence D-STM Model

In this chapter, we describe the system model of cc D-STM. We present the distributed cache-coherence problem for D-STM and the performance measures of CC protocols. We then give the static analysis of a general CC protocol $C$, which provides performance bounds of $C$ given a fixed set of transactions.

3.1 System Model and Problem Statement

3.1.1 Metric-Space Network Model.

We consider the metric-space network model of distributed systems, similar to the one proposed in [26]. We consider a distributed system with $n$ nodes. Let $G = (V, E)$ be a weighted connected graph, where $|V| = n$ and an edge $(u, v) \in E$ if $u$ and $v$ are connected. For two nodes $u$ and $v$ in $V$, let $d(u, v)$ denote the distance between them in $G$, i.e., the length of the shortest path between $u$ and $v$ provided by the underlying network protocols. We define the normalized diameter of $G$ as:

$$\text{Diam} = \max_{u, v, x, y \in V} \left\{ \frac{d(u, v)}{d(x, y)} \right\}.$$

All $n$ nodes are assumed to be contained in a metric space of normalized diameter $\text{Diam}$.

3.1.2 cc D-STM Model.

A transaction is a sequence of requests, each of which is a read or write operation request to an individual object. Given a set of $s \geq 1$ objects, $\{o_1, \ldots, o_s\}$, we use the tuple $T_j = (v_j, t_j, o_j, \tau_j)$ to describe a transaction $T_j$, where:
- $v_j$: node that initiates $T_j$.
- $t_j$: time at which $T_j$ is initiated.
- $o(j)$: set of units of objects requested by $T_j$. Let $o(j) = \{o_1(j), \ldots, o_s(j)\}$, where $o_i(j) \in \{0, 1, \frac{1}{n}\}$ represents the units of object $o_i$ required by $T_j$. If $T_j$ does not require access to $o_i$, $o_i(j) = 0$. If $o_j$ updates $o_i$, i.e., a write operation, $o_i(j) = 1$. If it reads $o_i$ without updating, $o_i(j) = \frac{1}{n}$, i.e., the object can be read by at most $n$ nodes simultaneously. Suppose there are two transactions $T_j$ and $T_k$, and $o_i(j) + o_i(k) > 1$. Then $T_j$ and $T_k$ are said to conflict at $o_i$.
- $\tau_j$: duration of $T_j$'s successful local execution. An execution of a transaction is a sequence of timed actions. Generally, there are four action types that may be taken by a transaction: write, read, commit, and abort. An execution ends by either a commit (success) or an abort (failure). A successful local execution of $T_j$ is a successful execution when all objects requested by $T_j$ are in the local cache of node $v_j$, i.e., there is no need to fetch those objects from the network.

Recall that in Herlihy and Sun’s data-flow D-STM model [26] that we consider, transactions are immobile and objects migrate to invoking transactions.\footnote{By “migrating” the object in the network, we assume that there is a shared code base such that application logic is available at all participating nodes. For example, in current multiprocessor STM implementations such as DSTM2 [2], the “state space” of an object is defined in terms of a simple Java interface that provides corresponding methods for each field. Moving the object simply means moving the state of the object (i.e., the object’s internal data structures). The local JVM will create a class definition that supports that state space when such information is passed from one node to another.} Transactional synchronization is optimistic: a transaction commits only if no other transaction has executed a conflicting access.

![Figure 3.1: cc D-STM Model](image-url)
We illustrate cc D-STM model in Figure 3.1. Each node has a TM proxy that provides interfaces to the TM application and to proxies of other nodes. When a node $v_A$ initiates a transaction $A$ that requests a read/write access to an object $o$, its TM proxy first checks whether $o$ is in the local cache (step 1); if not, the TM proxy invokes a CC protocol $CC$ to locate $o$ in the network (steps 2 & 3). Assume that object $o$ is in use by a transaction $B$ initiated by node $v_B$. When $v_B$ receives the request $CC.locate(o)$ from $v_A$, its TM proxy checks whether $o$ is in use by an active local transaction (step 4); if so, the TM proxy invokes a conflict resolution module to compare the priorities of transactions $A$ and $B$ (steps 5 & 6), where transactional priorities are assigned according to an application-specific policy (e.g., FIFO). Based on the result ($CR(A, B)$) of the conflict resolution module (step 7), $v_B$’s TM proxy decides whether to abort $B$ immediately, or postpone $A$’s request and let $B$ proceed to commit. Eventually, $v_B$ invokes $CC$ to move $o$ to $v_A$ (step 8).

Therefore, cc D-STM requires a CC protocol and a conflict resolution strategy.

**CC protocol.** When a transaction attempts to access an object, the CC protocol must locate the current cached copy of the object, move it to the requester’s cache, and invalidate the old copy. A CC protocol must perform the following functions:

- When a transaction $A$ attempts to access an object in the network, the protocol is invoked to carry $A$’s request to the node which holds the object in a finite time period.
- When a transaction $B$, which receives a read/write access request for an object it holds, has made the decision whether to abort the local transaction or to postpone the response, the protocol is invoked to move the object either immediately or after some time. In either case, the protocol must guarantee that the object is moved to the requester in a finite time period.
- At any given time, the protocol must guarantee that there exists only one writable copy of each object in the network — i.e., each object can only be written by one transaction at any time.

**Conflict resolution strategy.** Generally, a conflict resolution strategy is responsible for mediating conflicts over any shared object. Past works usually adopt a contention management policy, which aborts (or at least postpones) one transaction whenever two transactions conflict over a shared object. An efficient contention management policy should guarantee progress — i.e., at any given time, there exists at least one transaction that proceeds to commit without interruption. For cc D-STM, we require a conflict resolution strategy that satisfies the work conserving [34] and pending commit [35] properties:

**Definition 1.** A conflict resolution strategy is work conserving if it always lets a maximal set of non-conflicting transactions to run. A conflict resolution strategy obeys the pending commit property if, at any given time, some running transaction will execute uninterrupted until it commits.

For example, GREEDY contention manager in [35], which uses a globally consistent priority policy that issues priorities to transactions, is shown in [34] to satisfy both these properties.
3.1.3 Problem Statement.

We evaluate the performance of a distributed transactional memory system by measuring its \textit{makespan}. Given a set of transactions accessing a set of objects under a conflict resolution strategy \( A \) and a CC protocol \( C \), \( \text{makespan}(A, C) \) denotes the duration that the given set of transactions are successfully executed under the conflict resolution strategy \( A \) and CC protocol \( C \).

It is well-known that optimal off-line scheduling of tasks with shared resources is NP-complete [89]. While an online scheduling algorithm does not know a transaction’s object demands in advance, it does not always make optimal choices. An optimal clairvoyant off-line algorithm, denoted \( \text{Opt} \), knows the sequence of object accesses of the transaction in each execution.

We now consider an optimal, clairvoyant, offline ordering algorithm, denoted \( \text{Opt} \), which has the complete knowledge of all the transactions in \( T \). Now, the performance of a CC protocol \( C \) can be evaluated by measuring its competitive ratio:

\textbf{Definition 2.} 
\[
CR(C) = \frac{\max_A \text{makespan}(A, C)}{\text{makespan(OPT)}} = \frac{\text{makespan}(C)}{\text{makespan(OPT)}},
\]

where \( \max_A \text{makespan}(A, C) = \text{makespan}(C) \) for any \( A \) satisfying work conserving and pending commit properties.

We say a competitive ratio is \textit{static} if the makespan evaluates the performance of a CC protocol in a fixed way, i.e., the set of transactions is fixed and no new transaction joins the system during the execution. In the next section, we identify the critical factors that affect the makespan of any CC protocol.

3.1.4 Static Analysis

We first analyze the makespan of the optimal CC protocol, denoted \( \text{makespan(OPT)} \). Let the makespan of a set of transactions which require accesses to an object \( o_i \), be denoted as \( \text{makespan}_i \). It is composed of three parts:

1) \textbf{Traveling Makespan} (\( \text{makespan}_i^d \)): the total communication cost for \( o_i \) to travel in the network.
2) \textbf{Execution Makespan} (\( \text{makespan}_i^\tau \)): the duration of the transactions’ execution involving \( o_i \), including all successful and aborted executions; and
3) \textbf{Waiting Makespan} (\( \text{makespan}_i^w \)): the time that \( o_i \) waits for a transaction request.

Generally, a CC protocol performs two functions: 1) locating the up-to-date copy of the object and 2) moving it in the network to meet transactions’ requests. We define these costs as follows:
Definition 3 (Locating Cost). In a given graph $G$, the locating cost $\delta_C(T_i, T_j)$ is the communication cost incurred for a request invoked by transaction $T_i$ to travel in the network, to successfully locate an object held by transaction $T_j$, under a CC protocol $C$.

Definition 4 (Moving Cost). In a given graph $G$, the moving cost $\zeta_C(T_i, T_j)$ is the communication cost incurred for an object held by transaction $T_i$ to travel in the network to transaction $T_j$, which invokes a request to access the object, under a CC protocol $C$.

Let the node that holds object $o_i$ at the start of the system be denoted as $v_{o_i}^0$. Let $T^i = \{T_j \in T : o_i(j) > 0\}$, i.e., $T^i$ is the set of transactions that require accesses to $o_i$. For the set of nodes $V_{T^i}$ that invoke transactions for accessing object $o_i$, we build a complete graph $G_i = (V_i, E_i)$, where $V_i = \{V_{T^i} \cup v_{o_i}^0\}$ and $d_i(u, v) = d(u, v)$. We use $H(G_i, v_{o_i}^0, v_j)$ to denote the cost of the minimum-cost Hamiltonian path that visits each node from $v_{o_i}^0$ to $v_j$ exactly once. Now, we have:

**Theorem 1.**

$$\text{makespan}^d_i(\text{OPT}) \geq \min_{v_j \in V_{T^i}} H(G_i, v_{o_i}^0, v_j)$$
$$\text{makespan}^v_i(\text{OPT}) \geq \sum_{v_j \in V_{T^i}} \tau_j$$
$$\text{makespan}^w_i(\text{OPT}) \geq \sum_{v_j \in V_{T^i}} \min_{v_k \in V_i} d(v_k, v_j)$$

**Proof.** The execution of the given set of transactions with the minimum makespan schedules each transaction exactly once, which implies that $o_i$ only has to visit each node in $V_{T^i}$ once. In this case, the node travels along a Hamiltonian path in $G_i$ starting from $v_{o_i}^0$. Hence, we can lower-bound the traveling makespan by the cost of the minimum-cost Hamiltonian path and the execution makespan by the sum of $\tau_j$. For the optimal CC protocol, each object is located via the shortest path. The theorem follows. \hfill $\square$

Let $\lambda^*(j)$ denote $T_j$’s worst-case number of aborts under CC protocol $C$, and $\Lambda^*_C$ denote the worst-case number of total transaction aborts under $C$. Let $N_i = |V_{T^i}|$, i.e., the number of transactions that request accesses to object $o_i$. We have the following theorem for a general CC protocol $C$:

**Theorem 2.**

$$CR_i(C) \leq \max_{v_j \in V_{T^i}} \{ \max_{v_j \in V_{T^i}} (\lambda^*_C(j) + 1), \frac{\Lambda^*_C + N_i}{N_i} \cdot \max_{u,v \in V} \{ \frac{\zeta^C(u,v)}{d(u,v)} \cdot \max_{u,v \in V} \frac{\delta^C(u,v)}{d(u,v)} \} \cdot \text{Diam} \}$$
$$= \max_{v_j \in V_{T^i}} \lambda^*_C(j), \frac{\Lambda^*_C}{N_i} \cdot \max \{ \text{max-str}^\delta_C, \text{max-str}^\lambda_C \} \cdot \text{Diam},$$

where $\lambda^*_C(j) = \lambda^*_C(j) + 1$, $\Lambda^*_C = \Lambda^*_C + N_i$, $\text{max-str}^\delta_C = \max_{u,v \in V} \{ \frac{\delta^C(u,v)}{d(u,v)} \}$ and $\text{max-str}^\lambda_C = \max_{u,v \in V} \{ \frac{\zeta^C(u,v)}{d(u,v)} \}$. 
Proof. From Theorem 1 we know that

\[
\text{makespan}^d_i(C) \leq \Lambda_C \cdot \max_{v_j \in V_{i\tau}} \{ \max_{v_k \in V_i} \zeta^C(v_j, v_k) \}
\]

\[
\text{makespan}^\tau_i(C) \leq \sum_{v_j \in V_{i\tau}} \{ \lambda_C(j) \cdot \tau_j \}
\]

\[
\text{makespan}^w_i(C) \leq \Lambda_C \cdot \max_{v_j \in V_{i\tau}} \{ \max_{v_k \in V_i} \delta^C(v_j, v_k) \}.
\]

Hence, we have the following relationships for \( CR_i^d(C) \), the static competitive ratio of the CC protocol \( C \) for transactions requesting accesses to object \( o_i \):

\[
CR_i^d(C) \leq \frac{\Lambda_C \cdot \max_{v_j \in V_{i\tau}} \{ \max_{v_k \in V_i} \zeta^C(v_j, v_k) \}}{\sum_{v_j \in V_{i\tau}} \{ \min_{v_k \in V_i} d(v_j, v_k) \}}
\]

\[
CR_i^\tau(C) \leq \max_{v_j \in V_{i\tau}} \lambda_C(j)
\]

\[
CR_i^w(C) \leq \frac{\Lambda_C \cdot \max_{v_j \in V_{i\tau}} \{ \max_{v_k \in V_i} \delta^C(v_j, v_k) \}}{\sum_{v_j \in V_{i\tau}} \{ \min_{v_k \in V_i} d(v_j, v_k) \}}
\]

Let the maximum locating stretch and maximum moving stretch with respect to \( C \) be denoted, respectively, as: \( \text{max-str}^d_C = \max_{u,v \in V} \{ \frac{\zeta^C(u,v)}{d(u,v)} \} \) and \( \text{max-str}^\delta_C = \max_{u,v \in V} \{ \frac{\delta^C(u,v)}{d(u,v)} \} \).

The theorem follows. \( \square \)

### 3.1.5 Conclusion

Theorem 2 gives the upper bound of the static competitive ratio of the CC protocol \( C \). Clearly, the design of a CC protocol should therefore focus on minimizing its worst-case number of aborts, maximum locating stretch, and maximum moving stretch.
Chapter 4

Location-Aware Cache-Coherence Protocols for D-STM

In this chapter, we propose a class of distributed CC protocols with location-aware property, called LAC protocols. In LAC protocols, the duration for a transaction requesting node to locate the object is determined by the communication delay between the requesting node and the node that holds the object. The lower communication delay implies lower locating delay. In other words, nodes that are “closer” to the object will locate the object more quickly than nodes that are “further” from the object in the network. We show that the performance of the Greedy manager with LAC protocols is improved. We prove this worst-case competitive ratio and show that LAC is an efficient choice for the Greedy manager to improve the performance of the system.

4.1 Motivation and Challenge.

The past works on transactional memory systems for multiprocessors motivate our selection of the contention manager for D-STM.\(^1\) The major challenge in implementing a contention manager is to guarantee progress: at any time, there exists some transaction(s) which will run uninterruptedly until they commit. The Greedy manager proposed in [35] satisfies this property. Two non-trivial properties are established for Greedy in [35] and [34]:

- Every transaction commits within a bounded time.
- The competitive ratio of the Greedy manager is \(O(s)\) for a set of \(s\) objects, and this bound is asymptotically tight.

\(^1\)Since we focus solely on the design of cc D-STM, from Chapter 4, we use D-STM instead of cc D-STM for brevity.
The core idea of the Greedy manager is to use a globally consistent contention management policy that avoids both deadlocks and livelocks. For the Greedy manager, this policy is based on the timestamp at which each transaction starts. This policy determines the sequence of priorities of the transactions and relies only on local information, i.e., the timestamp assigned by the local clock. To make the Greedy manager work efficiently, the local clocks must be synchronized. The sequence of priorities is determined at the beginning of each transaction and will not change over time. In other words, the contention management policy serializes the set of transactions in a decentralized manner.

At first, transactions are processed greedily whenever possible. Thus, a maximal independent set of transactions that are non-conflicting over their first-requested objects is processed each time. Secondly, when a transaction begins, it is assigned a unique timestamp which remains fixed across re-invocations. At any time, the running transaction with the highest priority (i.e., the “oldest” timestamp) will neither wait nor be aborted by any other transaction.

These good properties of the Greedy manager for multiprocessors motivate us to study its performance in distributed systems. In a networked environment, the Greedy manager still guarantees transaction progress: the priorities of transactions are assigned when they start. At any time, the transaction with the highest priority (the earliest timestamp for the Greedy manager) never waits and is never aborted due to a synchronization conflict.

However, as discussed in Chapter 1, it is much more challenging to evaluate the Greedy manager’s performance in distributed systems, due to the cost involved for locating and moving objects among processors/nodes. While for multiprocessors, this cost can be ignored due to built-in CC protocols, for distributed systems, this cost — which depends on the CC protocol used — can be high, and may constitute the major part of the makespan. Hence, in order to evaluate the Greedy manager’s performance in distributed systems, the underlying CC protocol must be taken into account.

One unique phenomenon for transactions in distributed systems is the cost of “overtaking”. Suppose there are two nodes, $v_{T_1}$ and $v_{T_2}$, which invoke transactions $T_1$ and $T_2$, respectively, that require write accesses to object $R_1$. Assume that $T_1 \prec T_2$. An overtaking may be caused due to the following reasons:

* Due to the locations of nodes in the network, the cost for $v_{T_1}$ to locate the current cached copy of $R_1$ may be much larger than that for $v_{T_2}$.

* Due to the order of the sequence of actions of each transaction, $T_2$’s request for $R_1$ may be ordered earlier than that of $T_1$’s, e.g., the write access to the object is the first action of $T_2$ and the second of $T_1$.

In both the cases, $T_2$’s request may be ordered first and $R_1$ is moved to $v_{T_2}$ first. Then, $T_1$’s request has to be sent to $v_{T_2}$ since the object has been moved to $v_{T_2}$. The success or failure of an overtaking is defined by its result:
Overtaking Success: If $T_1$’s request arrives at $v_{T_2}$ after $T_2$’s commit, then $T_2$ is committed before $T_1$.

Overtaking Failure: If $T_1$’s request arrives at $v_{T_2}$ before $T_2$’s commit, the contention manager of $v_{T_2}$ will abort the local transaction and send the object to $v_{T_1}$.

A transaction is aborted when an overtaking failure occurs. Overtaking failures are unavoidable for transactions both in multiprocessors and in distributed systems. For multiprocessors, the aborted transaction is re-invoked immediately and the cost of the re-invocation is negligible. However, in distributed systems, it may take much more time for the aborted transaction to locate the new position of the object. Such failures may significantly increase the makespan of a set of transactions. Thus, we have to design efficient CC protocols to relieve the impact of overtaking failures. We will now show the impact of such failures on the competitive ratio of the Greedy manager.

4.2 Competitive Ratio Analysis of GREEDY

We focus on the makespan of the Greedy manager with a given CC protocol $C$. As mentioned before, to implement distributed transactional memory, a distributed CC protocol is needed. We use the metric stretch to evaluate the responsiveness of a CC protocol.

**Definition 5** (Stretch). The stretch of a CC protocol $C$ for a given metric-space network $G = (V, E)$ is the maximum ratio of the locating cost to the moving cost between two nodes:

$$\text{Stretch}(C) = \max_{i,j \in V} \frac{\delta^C(i, j)}{d(i, j)}.$$  

Let $N_i = |V_{T_R}^i|$, i.e., $N_i$ represents the number of transactions that request access to object $R_i$. We have the following theorem:

**Theorem 3.**

$$CR_i(\text{Greedy}, C) = O(\max[N_i^2, N_i \cdot \text{Stretch}(C)])$$  

**Proof.** Given a subgraph $G_i$, we define its priority Hamiltonian path as follows:

**Definition 6** (Priority Hamiltonian Path). The priority Hamiltonian path for a subgraph $G_i$ is a path which starts from $v_{T_R}^0$ and visits each node from the lowest priority to the highest priority.

Formally, the priority Hamiltonian path is $v_{T_R}^0 \rightarrow v_{T_{N_i}} \rightarrow v_{T_{N_i-1}} \rightarrow \ldots \rightarrow v_{T_1}$, where $N_i = |V_{T}^{R_i}|$ and $T_1 < T_2 < \ldots < T_{N_i}$. We use $H^p(G_i, v_{R_i}^0)$ to denote the cost of the priority Hamiltonian path for $G_i$.  

We first analyze the worst-case traveling makespans of the GREEDY manager. At any time \( t \) during the execution, let set \( A(t) \) contains nodes whose transactions have been successfully committed, and let set \( B(t) \) contains nodes whose transactions have not been committed. We have \( B(t) = \{ b_i(t) | b_1(t) < b_2(t) < \ldots \} \). Hence, \( R_i \) must be held by a node \( r_t \in A(t) \).

Due to the property of the GREEDY manager, the transaction requested by \( b_1(t) \) can be executed immediately and will never be aborted by other transactions. However, this request can be overtaken by other transactions if they are closer to \( r(t) \). In the worst case, the transaction requested by \( b_1(t) \) is overtaken by all other transactions requested by nodes in \( B \), and each overtaking is failed. In this case, the only possible path that \( R_i \) can travel is \( r(i) \rightarrow b_{|B(t)|}(t) \rightarrow b_{|B(t)|-1}(t) \rightarrow \ldots \rightarrow b_1(t) \). The cost of this path is composed of two parts: the cost of \( r(i) \rightarrow b_{|B(t)|}(t) \) and the cost of \( b_{|B(t)|}(t) \rightarrow b_{|B(t)|-1}(t) \rightarrow \ldots \rightarrow b_1(t) \). We can prove that each part is at most \( H^p(G_i, v^0_{R_i}) \) by triangle inequality (note that \( G_i \) is a metric completion graph). Hence, we know that the worst traveling cost for a transaction execution is \( 2H^p(G_i, v^0_{R_i}) \). Hence, we establish the upper bound of makespan of \( \tau_i \):

\[
\text{makespan}_i^d(\text{GREEDY}, C) \leq 2N_i \cdot H^p(G_i, v^0_{R_i}),
\]

The upper bound of the execution makespan can be proved directly. For any transaction \( T_j \), it can be overtaken at most \( N_i - j \) times. In the worst case, they are all overtaking failures. Hence, the worst execution cost for \( T_j \)'s execution is \( \sum_{j \leq k \leq N_i} \tau_k \). By summing them over all transactions, we have:

\[
\text{makespan}_i^e(\text{GREEDY}, C) \leq \sum_{1 \leq j \leq N_i} j \cdot \tau_j
\]

We now prove the upper bound of the idle time. If at time \( t \), the system becomes idle for the GREEDY manager, there are two possible reasons:

1. A set of transactions \( S \) invoked before \( t \) have been committed and the system is waiting for new transactions. There exists an optimal schedule that completes \( S \) at time at most \( t \), has idle till the next transaction is released, and possibly has additional idle intervals during \([0, t] \). In this case, the idle time of the GREEDY manager is less than that of OPT.

2. A set of transactions \( S \) is invoked, but the system is idle since objects haven’t been located. In the worst case, it takes \( N_i \cdot \delta^C(i, j) \) time for \( R_i \) to wait for the invoked requests. On the other hand, it only takes \( d(i, j) \) time to execute all transactions in the optimal schedule with the ideal CC protocol. The system will not stop after the first object has been located.

The total idle time is the sum of these two parts. We now have:

\[
\text{makespan}_i^w(\text{GREEDY}, C) \leq N_i \cdot \text{Stretch}(C) \cdot \text{makespan}_i^w(\text{OPT})
\]

The theorem follows.
Figure 4.1: Example: a 3-node network

Example: in the following example, we show that for Ballistic protocol [26], the upper bound in Theorem 3 is asymptotically tight, i.e., the worst-case traveling makespan for a transaction execution is the cost of the longest Hamiltonian path.

Consider a network composed of 3 nodes $A, B$ and $C$ in Figure 4.1. Based on Ballistic, a 3-level directory hierarchy is built, shown in Figure 4.2. Suppose $\epsilon << \alpha$. Nodes $i$ and $j$ are connected at level $l$ if and only if $d(i, j) < 2^{l+1}$. A maximal independent set of the connectivity graph is selected with members as leaders of level $l$. Therefore, at level 0, all nodes are in the hierarchy. At level 1, $A$ and $C$ are selected as leaders. At level 2, $C$ is selected as the leader (also the root of the hierarchy).

We assume that an object is created at $A$. According to Ballistic, a link path is created: $C \rightarrow A \rightarrow A$, which is used as the directory to locate the object at $A$. Suppose there are two transactions $T_B$ and $T_C$ invoked on $B$ and $C$, respectively. Specifically, we have $T_B \prec T_C$.

Now nodes $B$ and $C$ have to locate the object in the hierarchy by probing the link state of the leaders at each level. For node $C$, it doesn’t have to probe at all because it has a non-null link to the object. For node $B$, it starts to probe the link state of the leaders at level 1. In the worst case, $T_C$ arrives at node $A$ earlier than $T_B$, and the link path is redirected as $C \rightarrow C \rightarrow C$ and the object is moved to node $C$. Node $B$ probes a non-null link after the object has been moved, and $T_B$ is sent to node $C$. If $T_C$ has not been committed, then $T_C$ is aborted and the object is sent to node $B$.

In this case, the traveling makespan to execute $T_B$ is $d(A, C) + d(C, B) = 2\alpha$, which is the longest Hamiltonian path starting from node $A$. On the other hand, the optimal traveling makespan to execute $T_B$ and $T_A$ is $d(A, B) + d(B, C) = \epsilon + \alpha$. Hence, the worst-case traveling makespan to execute $T_B$ is asymptotically the number of transactions times the cost of the optimal traveling makespan to execute all transactions.

Theorem 3 gives the makespan upper bound of the Greedy manager for each individual object $R_i$. In other words, they give the bounds of the traveling and execution makespans when the number of objects $s = 1$. We can now derive the competitive ratio of $(\text{Greedy}, C)$.
for \( s \) objects. Let \( N = \max_{1 \leq i \leq s} N_i \), i.e., \( N \) is the maximum number of nodes that request an object.

**Theorem 4.**

\[
CR(\text{Greedy}, C) = O\left(\max[N^2 \cdot s, N \cdot \text{Stretch}(C)]\right)
\]

**Proof.** We first derive the bounds of \( \text{makespan}^d \) and \( \text{makespan}^\tau \) in the optimal schedule. Consider the set of write actions of all transactions. If \( s + 1 \) transactions or more are running concurrently, the pigeonhole principle implies that at least two of them are accessing an object. Thus, at most \( s \) writing transactions are running concurrently during time intervals that are not idle under \( \text{Opt} \). Thus, \( \text{makespan}^\tau(\text{Opt}) \) satisfies:

\[
\text{makespan}^\tau(\text{Opt}) \geq \frac{\sum_{i=1}^{m} \tau_i}{s}.
\]

In the optimal schedule, \( s \) writing transactions run concurrently, implying that each object \( R_i \) travels independently. From Theorem 1, \( \text{makespan}^d(\text{Opt}) \) satisfies:

\[
\text{makespan}^d(\text{Opt}) \geq \max_{1 \leq i \leq s} \min_{v_{R_i}, v_{T_j} \in V} H(G_i, v_{R_i}, v_{T_j}).
\]

Hence, we bound the makespan of the optimal schedule as:

\[
\text{makespan}(\text{Opt}) \geq \text{makespan}^w(\text{Opt}) + \frac{\sum_{i=1}^{m} \tau_i}{s} + \max_{1 \leq i \leq s} \min_{v_{R_i}, v_{T_j} \in V} H(G_i, v_{R_i}, v_{T_j}).
\]

Note that, whenever the \text{Greedy} manager is not idle, at least one of the transactions that is processed will be completed. However, from Theorem 3, we know that it may be overtaken by all transactions with lower priorities, and therefore the penalty time cannot be ignored. Using the same argument of Theorem 3, we have:

\[
\text{makespan}^\tau(\text{Greedy}, C) \leq \sum_{i=1}^{s} \sum_{k=1}^{N_i} k \cdot \tau_k.
\]
The traveling makespan of transaction $T_j$ is the sum of the traveling makespan of each object that $T_j$ involves. We have:

$$\text{makespan}^d(G\text{REEDY}, C) \leq \sum_{i=1}^{s} 2N_i \cdot H_p(G_i, v^0_{R_i}) \leq s \cdot 2N^2 \cdot \max_{1 \leq i \leq s} \min_{v_{T_j} \in V^R_{R_i}} H(G_i, v^0_{R_i}, v_{T_j}).$$

Hence, the makespan of the \text{GREEDY} manager satisfies:

$$\text{makespan}(\text{GREEDY}, C) \leq \sum_{i=1}^{s} 2N_i \cdot H_p(G_i, v^0_{R_i}) + \sum_{i=1}^{s} \sum_{k=1}^{N_i} k \cdot \tau_k + s \cdot 2N^2 \cdot \max_{1 \leq i \leq s} \min_{v_{T_j} \in V^R_{R_i}} H(G_i, v^0_{R_i}, v_{T_j}).$$

The theorem follows. \qed

Theorem 4 provides an upper bound on the competitive ratio of the \text{GREEDY} manager. In other words, it describes the worst-case performance of the \text{GREEDY} manager. On the other hand, what is the best performance that we can expect with the \text{GREEDY} manager? Can we prove a lower bound on the competitive ratio of the \text{GREEDY} manager? We have:

\textbf{Theorem 5.}

$$\text{makespan}_i^d(\text{GREEDY}, C) \geq H_p(G_i, v^0_{R_i})$$

$$\text{makespan}_i^r(\text{GREEDY}, C) \geq \sum_{v_{T_j} \in V^R_{R_i}} \tau_j$$

\textit{Proof.} In the best case, no overtaking occurs for transactions requesting object $R_i$. In this case, object $R_i$ travels along the priority Hamiltonian path in graph $G_i$. Each transaction is scheduled exactly once. The theorem follows. \qed

In the best case, the CC protocol $C$ provides a constant stretch. Hence, we can establish a lower bound for the \text{GREEDY} manager:

\textbf{Theorem 6.}

$$\text{CR}(\text{GREEDY}, C) = \Omega(s)$$

\textit{Proof.} The theorem can be proved by using the same argument of Theorem 4 combined with Theorem 5. \qed


4.3 Cache Responsiveness

To implement transactional memory in a distributed system, a distributed CC protocol is needed: when a transaction attempts to read or write an object, the CC protocol must locate the current cached copy of the object, move it to the requesting node’s cache and invalidate the old copy.

The CC protocol has to be responsive so that every transaction commits within a bounded time. We prove that for the Greedy manager, a CC protocol is responsive if and only if $\delta^C(i, j)$ is bounded for any $G$ that models a metric-space network.

Let $V^R_T(T_j) = \{v_{T_k} | v_{T_k} < v_{T_j}, v_{T_k}, v_{T_j} \in V^R_T \}$ for any graph $G$. Let

$$\Delta^C[V^R_T(T_j)] = \max_{v_{R_i} \in V^R_T} \delta^C(i, j)$$

and

$$D[V^R_T(T_j)] = \max_{v_{R_i} \in V^R_T} d(v_{R_i}, v_{T_j}).$$

We have the following theorem.

**Theorem 7.** A transaction $T_j$’s request for object $R_i$ with the Greedy manager and CC protocol $C$ is satisfied within time

$$|V^R_T(T_j)| \cdot \{\Delta^C[V^R_T(T_j)] + D[V^R_T(T_j)] + \tau_j\}.$$

**Proof.** The worst case of response time for $T_j$’s move request of object $R_i$ happens when $T_j$’s request overtakes each of the transaction that has a higher priority. Then the object is moved to $v_{T_j}$ and the transaction is aborted just before its commit. Thus, the penalty time for an overtaking failure is $\delta^C(i, j) + d(v_{R_i}, v_{T_j}) + \tau_j$, where $v_{R_i} \in V^R_T(T_j)$. The overtaking failure can happen at most $|V^R_T(T_j)|$ times until all transactions that have higher priority than $T_j$ commit. The lemma follows.

Theorem 7 shows that for a set of objects, the responsiveness for a CC protocol is determined by its locating cost.

4.4 Location-Aware CC Protocols

We now define a class of CC protocols which satisfy the following property:
Definition 7 (Location-Aware CC Protocol). In a given network $G$ that models a metric-space network, if for any two edges $e(i_1, j_1)$ and $e(i_1, j_1)$ such that $d(i_1, j_1) \geq d(i_1, j_1)$, there exists a CC protocol $C$ which guarantees that $\delta^C(i_1, j_1) \geq \delta^C(i_2, j_2)$, then $C$ is location-aware. The class of such protocols are called location-aware CC protocols or Lac protocols.

By using a Lac protocol, we can significantly improve the competitive ratio of traveling makespan of the Greedy manager, when compared with Theorem 3. The following theorem gives the upper bound of $CR_i^d(\text{Greedy, Lac})$.

Theorem 8.

$$CR_i^d(\text{Greedy, Lac}) = O(N_i \log N_i)$$

Proof. We first prove that the traveling path of the worst-case execution for the Greedy manager to finish a transaction $T_j$ is equivalent to the nearest neighbor path from $v_{R_i}^0$ that visits all nodes with lower priorities than $T_j$.

Definition 8. Nearest Neighbor Path: In a graph $G$, the nearest neighbor path is constructed as follows [90]:

1. Starts with an arbitrary node.
2. Find the node not yet on the path which is closest to the node last added and add the edge connecting these two nodes to the path.
3. Repeat Step 2 until all nodes have been added to the path.

The Greedy manager guarantees that, at any time, the highest-priority transaction can execute uninterrupted. If we use a sequence $\{v_{R_i}^1, ..., v_{R_i}^{N_i}\}$ to denote these nodes in the priority-order, then in the worst case, the object may travel in the reverse order before arriving at $v_{R_i}^j$. Each transaction with priority $p$ is aborted just before it commits by the transaction with priority $p-1$. Thus, $R_i$ travels along the path $v_{R_i}^0 \rightarrow v_{R_i}^{N_i} \rightarrow \ldots \rightarrow v_{R_i}^2 \rightarrow v_{R_i}^1$. In this path, transaction invoked by by $v_{R_i}^j$ is overtaken by all transactions with priorities lower than $j$, implying

$$d(v_{R_i}^0, v_{R_i}^{N_i}) < d(v_{R_i}^0, v_{R_i}^k), 1 \leq k \leq N_i - 1$$

and

$$d(v_{R_i}^j, v_{R_i}^{j-1}) < d(v_{R_i}^j, v_{R_i}^k), 1 \leq k \leq j - 2.$$  

Clearly, the worst-case traveling path of $R_i$ for a successful commit of the transaction invoked by $v_{R_i}^j$ is the nearest neighbor path in $G_i^j$ starting from $v_{R_i}^{j-1}$, where $G_i^j$ is a subgraph of $G_i$ obtained by removing $\{v_{R_i}^0, ..., v_{R_i}^{j-2}\}$ in $G_i$ and $G_i^1 = G_i$.

We use $NN(G, v_i)$ to denote the traveling cost of the nearest neighbor path in graph $G$ starting from $v_i$. We can easily prove the following equation by directly applying Theorem 1 from [90].
\[
\frac{NN(G^j_i, v_{R_i}^{j-1})}{\min_{v_{R_i} \in G^j_i} H(G_i, v_{R_i}^{j-1}, v_{R_i}^k)} \leq \lceil \log(N_i - j + 1) \rceil + 1 \quad (4.1)
\]

Theorem 1 from [90] studies the competitive ratio for the nearest tour in a given graph, which is a circuit on the graph that contains each node exactly once. Hence, we can prove Equation 4.1 by the triangle inequality for metric-space networks. We can apply Equation 4.1 to upper-bound makespan \( d_i^{(\text{Greedy}, \text{Lac})} \):

\[
\text{makespan}^{d_i}_{i}^{(\text{Greedy}, \text{Lac})} \leq \sum_{1 \leq j \leq N_i} NN(G^j_i, v_{R_i}^{j-1})
\]

\[
\leq \sum_{1 \leq j \leq N_i} \min_{v_{R_i} \in G^j_i} H(G_i, v_{R_i}^{j-1}, v_{R_i}^k) \cdot (\lceil \log(N_i - j + 1) \rceil + 1).
\]

Note that

\[
\min_{v_{R_i} \in G^j_i} H(G_i, v_{R_i}, v_{T_j}) \geq \min_{v_{R_i} \in G^j_i} H(G_i, v_{R_i}^{j-1}, v_{R_i}^k).
\]

Combined with Theorem 3, we derive the competitive ratio for traveling makespan of the Greedy manager with a Lac protocol as:

\[
CR_i^{d_i}(\text{Greedy}, \text{Lac}) = \frac{\text{makespan}^{d_i}_{i}^{(\text{Greedy}, \text{Lac})}}{\text{makespan}^{d_i}_{i}^{(\text{OPT})}}
\]

\[
\leq \sum_{1 \leq j \leq N_i} (\lceil \log(N_i - j + 1) \rceil + 1) \leq \log(N_i!) + N_i.
\]

The theorem follows.

We now revisit the scenario in the example of Figure 4.1 by applying Lac protocols. Note that \( T_B < T_C \) and \( d(A, B) < d(A, C) \). Due to the location-aware property, \( T_B \) will arrive at \( A \) earlier than \( T_C \). Hence, the traveling makespan to execute \( T_B \) and \( T_C \) is \( d(A, B) + d(B, C) \), which is optimal in this case.

Now we change the condition of \( T_B < T_C \) to \( T_C < T_B \). In this scenario, the upper bound of Theorem 8 is asymptotically tight. \( T_C \) may be overtaken by \( T_B \) and the worst case traveling makespan to execute \( T_C \) is \( d(A, B) + d(B, C) \), which is the nearest neighbor path starting from \( A \).

Remarks: the upper bounds presented in Theorem 3 also applies to Lac protocols. However, for Lac protocols, the traveling makespan becomes the worst case only when the priority path is the nearest neighbor path.
4.5 $\text{makespan}(\text{GREEDY, LAC})$ for Multiple Objects

Theorem 8 give the makespan upper bound of the GREEDY manager for each individual object $R_i$. In other words, they give the bounds of the traveling and execution makespans when the number of objects $s = 1$. Based on this, we can further derive the competitive ratio of the GREEDY manager with a LAC protocol for the general case. Let $N = \max_{1 \leq i \leq s} N_i$, i.e., $N$ is the maximum number of nodes that requesting for an object. Now,

**Theorem 9.** The competitive ratio $CR(\text{GREEDY, LAC})$ is

$$O(\max[N \cdot \text{Stretch}(\text{LAC}), N \log N \cdot s]).$$

**Proof.** We first prove that the total idle time in the optimal schedule is at least $N \cdot \text{Stretch}(\text{LAC}) \cdot \text{Diam}$ times the total idle time of the GREEDY manager with LAC protocols, shown as Equation 4.2.

$$\text{makespan}^w(\text{GREEDY, LAC}) \leq N \cdot \text{Stretch}(\text{LAC}) \cdot \text{Diam} \cdot \text{makespan}^w(\text{OPT}) \quad (4.2)$$

If at time $t$, the system becomes idle for the GREEDY manager, there are two possible reasons:

1. A set of transactions $S$ is invoked before $t$ have been committed and the system is waiting for new transactions. There exist an optimal schedule that completes $S$ at time at most $t$, is idle till the next transaction released, and possibly has additional idle intervals during $[0,t]$. In this case, the idle time of the GREEDY manager is less than that of OPT.

2. A set of transactions $S$ are invoked, but the system is idle since objects haven’t been located. In the worst case, it takes $N_i \cdot \delta^\text{LAC}(i,j)$ time for $R_i$ to wait for invoked requests. On the other hand, it only takes $d(i,j)$ time to execute all transactions in the optimal schedule with the ideal CC protocol. The system won’t stop after the first object has been located.

The total idle time is the sum of these two parts. Hence, we can prove Equation 4.2 by introducing the stretch of LAC.

Now we derive the bounds of $\text{makespan}^d$ and $\text{makespan}^\tau$ in the optimal schedule. Consider the set of write actions of all transactions. If $s + 1$ transactions or more are running concurrently, the pigeonhole principle implies that at least two of them are accessing the same object. Thus, at most $s$ writing transactions are running concurrently during time intervals that are not idle under OPT. Thus, $\text{makespan}^\tau(\text{OPT})$ satisfies:

$$\text{makespan}^\tau(\text{OPT}) \geq \frac{\sum_{i=1}^{m} \tau_i}{s}.$$
travels independently. From Theorem 3, \( \text{makespan}^d(\text{Opt}) \) satisfies:
\[
\text{makespan}^d(\text{Opt}) \geq \max_{1 \leq i \leq s} \min_{v_{R_i} \in V_T} H(G_i, v_i^0, v_{T_j}).
\]
Hence, we bound the makespan of the optimal schedule by
\[
\text{makespan}(\text{Opt}) \geq I(\text{Opt}) + \frac{\sum_{i=1}^m \tau_i}{s} + \max_{1 \leq i \leq s} \min_{v_{R_i} \in V_T} H(G_i, v_i^0, v_{T_j}).
\]
Note that, whenever the \textit{Greedy} manager is not idle, at least one of the transactions that are processed will be completed. However, from Theorem 4 we know that it may be overtaken by all transactions with lower priorities, and therefore the penalty time cannot be ignored. Use the same argument of Theorem 4, we have
\[
\text{makespan}^r(\text{Greedy}, \text{Lac}) \leq \sum_{i=1}^s \sum_{k=1}^{N_i} k \cdot \tau_k.
\]
The traveling makespan of transaction \( T_j \) is the sum of the traveling makespan of each object that \( T_j \) involves. With the result of Theorem 8, we have
\[
\text{makespan}^d(\text{Greedy}, \text{Lac}) \leq \sum_{i=1}^s \sum_{j=1}^{N_i} NN(G_i^j, v_i^{j-1})
\]
\[
\leq s \cdot \sum_{j=1}^{N_i} \min_{v_{R_i} \in G_i^j} H(G_i^j, v_i^{j-1}, v_{R_i}^k) \cdot ([\log(N - j + 1)] + 1)
\]
\[
\leq s \cdot ([\log(N!)] + N_i) \cdot \max_{1 \leq i \leq s} \min_{v_{R_i} \in V_T} H(G_i, v_i^0, v_{T_j}).
\]
Hence, the makespan of the \textit{Greedy} manager with a \textit{Lac} protocol satisfies:
\[
\text{makespan}(\text{Greedy}, \text{Lac}) \leq I(\text{Greedy}, \text{Lac}) + \sum_{i=1}^s \sum_{k=1}^{N_i} k \cdot \tau_k
\]
\[
+ s \cdot ([\log(N!)] + N) \cdot \max_{1 \leq i \leq s} \min_{v_{R_i} \in V_T} H(G_i, v_i^0, v_{T_j}).
\]
The theorem follows. \( \square \)

\textit{Remarks:} Theorem 9 generalizes the performance of makespan of \((\text{Greedy}, \text{Lac})\). In order to lower this upper bound as much as possible, we need to design a \textit{Lac} protocol with \( \text{Stretch}(\text{Lac}) \leq s \log N \). In this case, the competitive ratio for \((\text{Greedy} \text{ manager,} \text{ Lac})\) is \( O(N \log N \cdot s) \). Compared with the \( O(s) \) bound for multiprocessors, we conclude that the competitive ratio of makespan of \textit{Greedy} degrades for distributed systems.
4.6 Conclusion

We show that the performance of a distributed transactional memory system with a metric-space network is far from optimal, under the Greedy contention manager and an arbitrary CC protocol. Hence, we propose a location-aware property for CC protocols to take into account the relative positions of nodes in the network. We show that the combination of the Greedy contention manager and an efficient Lac protocol yields a better worst-case competitive ratio for a set of transactions. This results thus facilitate the following strategy for designing distributed transactional memory systems: select a contention manager and determine its performance without considering CC protocols; then find an appropriate CC protocol to improve performance.

In this chapter we propose a class of CC protocols with location-aware property. This is the first attempt to investigate the combinative behavior of contention managers and CC protocols. For all Lac protocols, the competitive ratio upper bound is $O(N \log N \cdot s)$ when combined with the Greedy manager.
Chapter 5

Distributed Queuing and Distributed Priority Queuing Cache-Coherence Protocols

In this chapter, we formalize two classes of CC protocols: distributed queuing cache-coherence (DQCC) protocols and distributed priority queuing cache-coherence (DPQCC) protocols, both of which can be implemented using distributed queuing protocols. We analyze the two classes of protocols for a set of dynamically generated transactions and compare their time complexities with that of an optimal offline clairvoyant algorithm. We show that a DQCC protocol is $O(N \log D)$-competitive and a DPQCC protocol is $O(\log D)$-competitive for a set of dynamically generated $N$ transactions requesting an object, where $D$ is the normalized diameter of the underlying distributed queuing protocol.

5.1 DQCC Protocols

5.1.1 Distributed Queuing Protocols

We first describe the distributed queuing problem, which provides us with a starting point to understand the design of distributed CC protocols. Assume that nodes initiate ordering requests for an object at arbitrary times in the network. Formally, an ordering request $r$ can be identified by the tuple $r = (u, t)$, where $u$ is the node that initiates the ordering request, and $t$ is the time when the request is initiated. When receiving the ordering request $r$, the object is simply moved to node $u$.

A distributed queuing protocol orders all requests in the system over time, globally, in a distributed way. As a result, all ordering requests form a fixed distributed queue. Each
request will find its predecessor and will be found by its successor in the queue. Hence, the solution to the distributed queuing problem is to find an ordering algorithm for a set of requests $R$:

**Definition 9 (Ordering Algorithm).** An ordering algorithm is a distributed algorithm which defines a total order on $R$ such that in the end, each node that initiates requests knows the predecessors of all its requests.

Note that such an algorithm must be distributed. For example, Arrow protocol [49] is a simple distributed queuing protocol based on path reversal.

Assume a distributed queuing protocol $C$. We define the ordering cost of $C$ as follows:

**Definition 10 (Ordering Cost).** The ordering cost $\delta^C(r_j, r_k)$ is the communication cost to order request $r_k$ after request $r_j$ under $C$, i.e., if $v_k$ invokes the request $r_k$ at time $t_k$, then $v_j$ which invokes $r_j$ receives $r_k$ at time $t_k + \delta^C(r_j, r_k)$.

In practice, $\delta^C(r_j, r_k)$ is the communication cost for $r_k$ to realize $r_j$ so that $r_k$ can be ordered after $r_j$ under $C$. For example, for the Arrow protocol [49], $\delta^C(r_j, r_k)$ is the distance of the path from $r_k$ to $r_j$ of the underlying spanning tree in the metric space. We assume that $\delta^C(r_j, r_k)$ forms a metric, i.e., (1) $\delta^C(r_j, r_k) = 0$ if and only if $r_j = r_k$; (2) $\delta^C(r_j, r_k)$ satisfies the triangle inequality; and (3) $\delta^C(r_j, r_k)$ is non-negative and symmetric.

### 5.1.2 Protocol Description

We can design a CC protocol based on distributed queuing, called a distributed queuing cache-coherence (or DQCC) protocol. For example, Ballistic [26] is a DQCC protocol. Each transaction is considered as a sequence of ordering requests. Thus, for a set of $s$ objects, $s$ distributed queues are established. However, for a CC protocol, a distributed queue is no longer fixed — an aborted transaction may rejoin the queue and thus the length of the queue can dynamically increase.

We illustrate the class of DQCC protocols in Figure 5.1. DQCC protocols work as follows:

1. For each object, a distributed queue is formed by transactions which request read/write access to that object. Formally, for each object $o_i$, a distributed queue $Q_i$ is established. A transaction $T_j$ requests to join $Q_i$ if and only if $o_i(j) > 0$. Let $L_i(t)$ denote the length of $Q_i$ at time $t$.

2. **Enqueue operation.** At any given time $t$, a transaction joins queue $Q_i$ by sending a request to the current tail of $Q_i$, and becomes the new tail of $Q_i$ when the old tail “realizes” its request, i.e., a link from the old tail to the new tail is established. To implement this operation, for each object $o_i$, a DQCC protocol has to maintain and update a global directory which always points to the tail of $Q_i$. 
Figure 5.1: The DQCC protocol: queue $Q_i$ at time $t$ for object $o_i$

3 Dequeue operation. Let $T^i_k$ denote the $k^{th}$ transaction that joins $Q_i$. $T^i_k$ can only leave $Q_i$ after it terminates (commits or aborts) and passes $o_i$ to its successor $T^i_{k+1}$. Suppose that $T^i_k$ becomes the head of $Q_i$ at some time point (Figure 5.1). We discuss the dequeue operation case by case:

- If $T^i_k$ commits, it passes $o_i$ to its successor $T^i_{k+1}$ and leaves the queue.
- If $T^i_k$ is aborted (not necessarily by $T^i_{k+1}$), $T^i_k$ passes $o_i$ to $T^i_{k+1}$ and restarts immediately. In this case, $T^i_k$ may request to join the queue again.

Note that if no such $T^i_{k+1}$ exists when $T^i_k$ commits or aborts (i.e., $T^i_k$ is the only transaction in $Q_i$ when it terminates), $T^i_k$ does not leave $Q_i$ until a new transaction (i.e., $T^i_{k+1}$) joins the queue. At any given time, object $o_i$ is held by the head of $Q_i$. $T^i_k$ can only be dequeued from $Q_i$ after it passes $o_i$ to $T^i_{k+1}$. Hence, $L_i(t) \geq 1$ for any given $t$.

A transaction joins a sequence of distributed queues in the order of objects that it requests. For example, assume that transaction $T_j$ contains a sequence of operations \{$write(o_1), write(o_2), read(o_3)$\}. When $T_j$ is invoked, it requests to join queue $Q_1$ first. After operation $write(o_1)$, $T_j$ requests to join queue $Q_2$ for operation $write(o_2)$. In the same way, $T_j$ joins queue $Q_3$ after operation $write(o_2)$. Hence, a transaction may participate in multiple queues at the same time. If at any time during the aforementioned steps, $T_j$ is aborted by one of its successors, then it passes each object it possesses to its successor in the corresponding queue and leaves all participating queues.

We map a transaction to a set of elements of distributed queues that it participates.
Figure 5.2: Example 1: $T_j = \{\text{write}(o_1), \text{write}(o_2), \text{read}(o_3)\}$. At $t + \epsilon$, $T_j$ receives a request from $T_{j(1,2)+1}^i$ for $o_2$.

**Definition 11.** Suppose that transaction $T_j$ is invoked $\sigma_j$ times before it commits. We have $T_j \equiv T_{j(x,i)}^i$, where $x \in [1, \sigma_j]$ and $i \in [1, s]$.

In other words, when $T_j$ is invoked for the $x^{th}$ time, it is the $j(x,i)^{th}$ transaction that joins queue $Q_i$. In this example, we know that $T_j \equiv \{T_{j(1,1)}^1, T_{j(1,2)}^2, T_{j(1,3)}^3\}$.

Assume that at time $t$, $T_j$ completes $\text{write}(o_2)$ and sends a request to join $Q_3$. Assume that $T_j$ does not have a successor in $Q_2$ at $t$. Suppose at time $t + \epsilon$, $T_j$ receives a request from $T_{j(1,2)+1}^2$ for object $o_2$, which is depicted in Figure 8.3. Let $T_{j(1,2)+1}^2$ have a higher priority than $T_j$, and $T_j$ be still waiting for object $o_3$ at time $t + \epsilon$. In this case, $T_j$ is aborted by $T_{j(1,2)+1}^2$ and has to be dequeued from all queues that it participates. $T_j$ performs the following dequeue operations:

- $T_j$ passes $o_2$ to $T_{j(1,2)+1}^2$;
- if $T_{j(1,1)+1}^1$ (T_j’s successor in Q_1) exists, $T_j$ passes $o_1$ to $T_{j(1,1)+1}^1$ immediately. If not, $o_1$ is passed whenever $T_j$ learns about the existence of $T_{j(1,1)+1}^1$;
- upon receiving $o_3$ (i.e., when $T_j$ becomes the head of $Q_3$), $T_j$ passes $o_3$ to $T_{j(1,3)+1}^3$ in the same way that it passes $o_2$.

### 5.1.3 Distributed-Queuing-Based Implementation

We now describe how to implement a DQCC protocol with a given distributed queuing protocol $C$. We first define operations of a distributed queuing protocol.
Deﬁnition 12. For an ordering request \( r_j \) for object \( o_i \), a distributed queuing protocol \( C \) provides an ordering operation \( C.\text{order}(i)(r_j) \) to order \( r_j \) after \( r_{j(1,i)}^{r} - 1 \) in \( Q_i \).

Deﬁnition 13. For the \( x^{th} \) invocation of a transaction \( T_j \), a DQCC protocol \( P \) provides an ordering operation \( P.\text{order}(i)(T_j, x) \) based on a distributed protocol \( C \).

Deﬁnition 14. For the \( x^{th} \) invocation of a transaction \( T_j \), a DQCC protocol \( P \) provides a moving operation \( P.\text{movesuc}(i)(T_j, x) \) to move object \( o_i \) from \( T_j \) to its successor \( T_j^{i(x,i) + 1} \) in \( Q_i \).

Deﬁnition 15 (Moving Cost). The moving cost \( \zeta^P(T_j^{i(x,i)}, T_j^{i(x,i) + 1}) \) is the total time complexity of \( P.\text{movesuc}(i)(T_j, x) \).

In practice, the object is only moved when its destination is known. The simplest way is to move it along the shortest path between two nodes. Formally, we have the following theorem to implement \( P \) based on \( C \).

Theorem 10. Given a distributed queuing protocol \( C \), a DQCC protocol \( P \) can be implemented such that:

1. \( P.\text{order}(i)(r_j, x) \equiv C.\text{order}(i)(r_j); \) and

2. \( \zeta^P(T_j^{i(x,i)}, T_j^{i(x,i) + 1}) = O(\delta^C(r_{j_x}, r_{j(1,i)}^{i(1,i) - 1})) \).

Proof. Note that \( r_j \equiv r_{j(1,i)}^{i(1,i) - 1} \) in Deﬁnition 12, since each ordering request is invoked only once. We depict \( C.\text{order}(i)(r_j) \) in Figure 5.3. In a distributed queuing protocol, for each object \( o_i \), a global directory is maintained to always point to the tail of \( Q_i \). Each element of \( Q_i \) keeps a local pointer to its successor (Figure 5.3(a)).

We further decompose \( C.\text{order}(i)(r_j) \) into two basic operations:
1. \textit{C\.send}(i)(r_j): send \( r_j \)'s request for \( o_i \) following the global directory of \( Q_i \). Thus, a local pointer to \( T_j^i \) is created at \( r_j^{i(1,i)−1} \) (\( r_j \)'s predecessor in \( Q_i \));
2. \textit{C\.redirect}(i)(r_j): redirect the global directory of \( Q_i \) to \( r_j \). \( r_j \) becomes the new tail of \( Q_i \).

These two operations are depicted in Figure 5.3(b). We denote the time complexities of \textit{C\.send}(i)(r_j) and \textit{C\.redirect}(i)(r_j) as \( \delta_s^C(r_j, r_j^{i(1,i)−1}) \) and \( \delta_r^C(r_j, r_j^{i(1,i)−1}) \), respectively. From Definition 10, the ordering cost of the \textit{C\.order}(i)(r_j) operation is \( \delta^C(r_j, r_j^{i(1,i)−1}) \). We have the following equation:

\[
\{ \delta_s^C(r_j, r_j^{i(1,i)−1}), \delta_r^C(r_j, r_j^{i(1,i)−1}) \} = O(\delta^C(r_j, r_j^{i(1,i)−1})). \tag{5.1}
\]

Now, we can implement a DQCC protocol \( P \) based on \( C \). At first, DQCC protocols process transactions as ordering requests. Hence, \( P \) provides a similar operation as \textit{C\.order}(i)(r_j) and \textit{P\.order}(i)(r_j, x) = \textit{C\.order}(i)(r_{js}), \) where \( r_{js} \) is the ordering request equivalent to the \( x \)-th invocation of \( T_j \). From Equation 5.1, we have:

\[
\{ \delta_s^C(r_{js}, r_{js(1,i)−1}), \delta_r^C(r_{js}, r_{js(1,i)−1}) \} = O(\delta^C(r_{js}, r_{js(1,i)−1})) = O(\delta^P(T_j, T_j^{i(x,i)−1})).
\]

We depict the moving operation in Figure 6.3. Similar to the ordering operation, we decompose the moving operation into two basic operations:

1. \textit{P\.move}(i)(T_j, x): move object \( o_i \) to \( T_j^{i(x,i)−1} \)'s successor \( (T_j^{i(x,i)+1}) \) in \( Q_i \) following the local pointer of \( T_j^{i(x,i)} \) (Figure 6.3(b));
2. \textit{P\.rmvptr}(i)(T_j, x): remove the local pointer of \( T_j^{i(x,i)} \), and \( T_j^{i(x,i)+1} \) becomes the new head of the queue \( Q_i \) (Figure 6.3(a)).

Let the time complexities of \textit{P\.move}(i)(T_j, x) and \textit{P\.rmvptr}(i)(T_j, x) be denoted as \( \zeta_m^P(T_j^{i(x,i)}, T_j^{i(x,i)+1}) \) and \( \zeta_r^P(T_j^{i(x,i)}, T_j^{i(x,i)+1}) \), respectively. From the Figure 6.3, we know that

\[
\{ \zeta_m^P(T_j^{i(x,i)}, T_j^{i(x,i)+1}), \zeta_r^P(T_j^{i(x,i)}, T_j^{i(x,i)+1}) \} = O(\zeta^P(T_j^{i(x,i)}, T_j^{i(x,i)+1})) = O(\delta^C(r_{js}, r_{js(1,i)−1})).
\]

The theorem follows. \( \square \)

## 5.2 DPQCC Protocols

### 5.2.1 Protocol Description

We now present the class of \textit{distributed priority queuing cache-coherence} (or DPQCC) protocols based on distributed priority queuing, which is illustrated in Figure 5.5. Unlike DQCC protocols, DPQCC protocols maintain and update a distributed priority queue for each object, which orders transactions based on their priorities assigned by a contention manager according to an application-specific policy. DPQCC protocols work in the following way:
Similar to DQCC protocols, for each object $o_i$, a priority queue $Q_i^*$ is formed by transactions which request read/write access to $o_i$. Let $L_i^*(t)$ denote the length of $Q_i^*$ at time $t$.

**Enqueue operation.** Transaction $T_j$ joins queue $Q_i^*$ by sending a request for $o_i$ to the head of $Q_i^*$. After learning the priority of $T_j$, $Q_i^*$ “inserts” $T_j$ into a proper position such that the priority order of the queue is not violated, i.e., each element always has a higher priority than its successor. To implement this operation, for each object $o_i$, a DPQCC protocol has to maintain and update a global directory, which always points to the head of $Q_i^*$.

**Dequeue operation.** Let $T_i^k$ denote the $k^{th}$ transaction that joins $Q_i^*$. Note that in $Q_i^*$, the order of transactions is not related to the order in which they join the queue. Moreover, the order of transactions in $Q_i^*$ changes over time. Let $T_{i[m]}^*$ denote the $m^{th}$ transaction succeeding $T_i^k$ at time $t$, where $m \geq 1$. $T_i^k$ can only leave $Q_i^*$ after it terminates (i.e., commits or aborts). Suppose that $T_i^k$ terminates at some time $t$ (Figure 8.4). We discuss the dequeue operation case by case:

- If $T_i^k$ commits, it is dequeued in the same way as DQCC protocols: $T_i^k$ passes $o_i$ to $T_{i[1]}^*$ (its successor in $Q_i^*$ at time $t$) and leaves the queue.
- If $T_i^k$ aborts, it must be aborted by a newly joined transaction in one of the queues that it participates. Assume that $T_i^k$ is aborted by $T_{i'}^{\prime k}$ in queue $Q_{i'}^*$ (i.e., $T_i^k$ participates in $Q_i^*$ and $Q_{i'}^*$ simultaneously). Then:
  - if $i = i'$, $T_i^k$ passes $o_i$ to $T_{i'}^{\prime k}$ (such that $T_{i'}^{\prime k}$ becomes the new head of $Q_{i'}^*$);
  - if $i \neq i'$, $T_i^k$ passes $o_i$ to $T_{i[1]}^*$ in the same way as DQCC protocols.

In this case, $T_i^* \prime$ restarts immediately and requests to rejoin the queue.
5.2.2 Distributed-Queuing-Based Implementation

We now describe the implementation of a DPQCC protocol $P'$ based on a given DQCC protocol $P$ (which, in turn, is based on a distributed queuing protocol $C$). First, $P'$ provides an insert operation to insert each transaction into its proper place of a priority queue.

**Definition 16.** For transaction $T_j$ requesting read/write access to object $o_i$ at time $t'$, if $T_k^r$ is the head of $Q_i^r$ at $t'$ and $T_k^r < T_j$, a DPQCC protocol $P'$, which provides an insert operation $P'.insert(i)(T_j)$ to insert $T_j$ into $Q_i^r$ such that at any given time $t \geq t'$, $T_j^r(i)[l, t] < T_k^r(i)[l, t]$ for $l \geq 1$, and $T_j^r(i)[l, t] < T_j^r(i)[l, t]$ for $l \leq -1$.

We depict the operation $P'.insert(i)(T_j)$ in Figure 5.7. Interestingly, we can implement $P'.insert(i)(T_j)$ with a sequence of base operations provided by $P$ and $C$, as follows:

**Step 1.** At time $t$, queue $Q_i^r$ is in the priority order and $T_k^r$ is the head of $Q_i^r$. Let the $l^{th}$ element succeeding $T_k^r$ at time $t$ be denoted as $T_{k[l,d]}^r$. Hence, we know that $T_k^r < T_{k[l,d]}^r < \ldots T_{k[l,l+1]}^r \ldots$. The global directory always points to the head of $Q_i^r$, and object $o_i$ is always held by the head of $Q_i^r$ (Figure 5.7(a)).

**Step 2.** As shown in Figure 5.7(b), $T_j$ requests to access object $o_i$ after time $t$. $P'$ sends $T_j$’s request to $T_k^r$ following the global directory. Then $T_k^r$ establishes a local pointer to $T_j$. This step is performed by $P'.send(i)(T_j)$. It is straightforward to prove the following lemma:

**Lemma 1.**

$$P'.send(i)(T_j) \equiv C.send(i)(T_j).$$

Similar to distributed queuing protocols, DPQCC protocols enqueue each transaction once.
Step 3. After the $P'.send(i)(T_j)$ operation, $T_k^i$ has two local pointers: one points to its original successor in $Q_i^*$ and another points to $T_j$. The contention manager at $T_k^i$ compares the priorities of its successor ($T_k^i[1,t]$) and $T_j$ (we assume that $T_k^i ≺ T_j$ and will discuss the other case later). Hence, the contention manager of a transaction has to save the priority information of its successor in each queue that it participates. If $T_k^i[1,t] ≺ T_j$, then $T_j$ should be queued after $T_k^i[1,t]$; if not, then $T_j$ should be queued between $T_k^i$ and $T_k^i[1,t]$.

We define the following operation:

**Definition 17.** For a transaction $T_k^i$ with two local pointers to $T_{k_1}^i$ and $T_{k_2}^i$ in $Q_i^*$, respectively, where $T_k^i ≺ T_{k_1}^i ≺ T_{k_2}^i$, a DPQCC protocol $P'$ provides a $P'.mvptr(i)(T_k^i)$ operation to remove the pointer from $T_k^i$ to $T_{k_1}^i$ and let $T_{k_1}^i$ keep a local pointer to $T_{k_2}^i$.

The purpose of $P'.mvptr(i)(T_k^i)$ is to “pass” the pointer to $T_j$ along the priority queue to find a proper position. If we consider a pointer itself as an object, we have the lemma:
Lemma 2.

\[ P'.mvptr(i)(T^*_k) \equiv P.mvobj(i[ptr\_low])(T^*_k, 1), \]

where \( ptr\_low \) represents the “local pointer to the lower priority transaction of the two pointing transactions of \( T^*_k \).”

Note that we use \( (T^*_k, 1) \), since each transaction is inserted into \( Q^*_i \) just once. Assume that \( T^*_k[l,t] \prec T^*_j \prec T^*_k[l+1,t] \) where \( l \geq 0 \). \( P' \) performs \( l \) number of \( P'.mvptr(i)(T^*_k) \) operations to move a local pointer from \( T^*_k[l,t] \) to \( T^*_k[l+1,t] \), and \( m = 0, 1, \ldots, l - 1 \), as shown in Figure 5.7(c).

Then, \( T^*_k[l,t] \) keeps a local pointer to \( T^*_j \) and knows that \( T^*_j \prec T^*_k[l+1,t] \). Hence, \( P' \) performs a \( P'.mvptr(i)(T^*_k) \) operation to let \( T^*_j \) keep a local pointer to \( T^*_k[l+1,t] \). As a result, \( T^*_j \) is correctly inserted between \( T^*_k[l,t] \) and \( T^*_k[l+1,t] \) (Figure 5.7(d)).

This completes the sequence of steps needed to implement \( P'.insert(i)(T_j) \).

Definition 18 (Inserting Cost). The inserting cost \( \kappa^{P'}(T_j, T^*_k) \) is the total time complexity of \( P'.insert(i)(T_j) \), where \( T^*_k \) is the head of \( Q^*_i \) when \( T_j \) requests to join \( Q^*_i \). Specifically,

\[ \kappa^{P'}(T_j, T^*_k) = \delta^C(T_j, T^*_k) + \Sigma_{m=0}^{l} \zeta^{P'}(T^*_k[m], T^*_k[m+1]) \]

\( P' \) provides a dequeue operation when a committed transaction leaves the queue.

Definition 19. For a transaction \( T^*_k \) committed at time \( t_c \), if at any time \( t (t \geq t_c) \), \( T^*_k \) learns about the existence of its successor \( T^*_k[l+1,t] \), \( P' \) provides a dequeue operation \( P'.dequeue(i)(T^*_k) \) to move an object \( o_i \) from \( P'.dequeue(i)(T^*_k) \) to \( T^*_k[l+1,t] \) and redirect the global directory of \( Q^*_i \) to \( T^*_k[l+1,t] \).

Hence, \( P'.dequeue(i)(T^*_k) \) contains two basic operations: \( P'.movesuc(i)(T^*_k) \) and \( P'.redirect(i)(T^*_k[l+1,t]) \), as shown in Figure 5.8. Both operations can be implemented by the existing operations provided by \( C \) and \( P \). Then we have the following lemma.
**Lemma 3.**

\[
P'.\text{movesuc}(i)(T^*_k) \equiv P.\text{movesuc}(i)(T^*_k, 1) \\
P'.\text{redirect}(i)(T^*_{k[1,t]}) \equiv C.\text{redirect}(i)(T^*_{k[1,t]})
\]

**Definition 20 (Dequeueing Cost).** The dequeueing cost \(\xi'_{P}(T^*_k, T^*_{k[1,t]})\) is the total time complexity of \(P'.\text{dequeue}(i)(T^*_k)\) performed at time \(t\), where \(T^*_k\) is the head of \(Q^*_i\) at \(t\). Specifically,

\[
\xi'_{P}(T^*_k, T^*_{k[1,t]}) = \delta^C_C(T^*_{k[1,t]}, T^*_k) + \xi^P_{P'}(T^*_k, T^*_{k[1,t]})
\]

Recall that the \(P'.\text{insert}(i)(T_j)\) operation only considers the case \(T^*_k \prec T_j\), where \(T^*_k\) is the head of \(Q^*_i\) and receives \(T_j\)'s request of \(o_i\). On the other hand, it is possible that \(T_j \prec T^*_k\). In this case, the \(P'.\text{insert}(i)(T_j)\) operation does not work and we need to define a heading operation to place \(T_j\) as the head of \(Q^*_i\).

**Definition 21.** For a transaction \(T_j\) requesting read/write access to object \(o_i\) at time \(t'\), if \(T^*_k\) is the head of \(Q^*_i\) at time \(t'\) and \(T_j \prec T^*_k\), a DPQCC protocol \(P'\) provides a heading operation \(P'.\text{head}(i)(T_j)\) to place \(T_j\) as the head of \(Q^*_i\). At any given time \(t \geq t'\), \(T_j^*(i) \prec T_j^*(i)[l,t]\) for \(l \geq 1\), and \(T_j^*(i)[l,t] \prec T_j^*(i)\) for \(l \leq -1\).

We depict \(P'.\text{head}(i)(T_j)\) in Figure 5.9. Similar to \(P'.\text{insert}(i)(T_j)\), \(P'.\text{head}(i)(T_j)\) can also be implemented with the basic operations provided by \(P\) and \(C\).

**Step 1.** At time \(t\), \(T^*_k\) is the head of \(Q^*_i\) (Figure 5.9(a)).

**Step 2.** The first step is similar to \(P'.\text{insert}(i)(T_j)\): a \(P'.\text{sendi}(T_j)\) operation is performed and \(T^*_k\) establishes a local pointer to \(T_j\) (Figure 5.9(b)).

**Step 3.** The contention manager at \(T^*_k\) detects that \(T_j \prec T^*_k\). A basic operation \(P'.\text{movepre}(i)(T^*_k)\) is performed to move an object \(o_i\) from \(T^*_k\) to \(T_j\).
Definition 22. For a transaction $T_k^*$ with two local pointers to $T_{k_1}^*$ and $T_{k_2}^*$ in $Q_i^*$, respectively, where $T_{k_1}^* \prec T_k^* \prec T_{k_2}^*$, a DQCC protocol $P'$ provides a $P'.movepre(i)(T_k^*)$ operation to move an object $o_i$ from $T_{k_1}^*$ to $T_{k_2}^*$, and delete $T_{k_1}^*$ 's local pointer pointing to $T_j$.

Lemma 4. $P'.movepre(i)(T_k^*) \equiv P.moveSuc(i[high])(T_k^*)$, where high represents the successor with the highest priority.

Hence, we can implement $P'.movepre(i)(T_k^*)$ with the basic operation provided by $P$ (Figure 5.9(c)). Note that $T_j$ can establish a local pointer to $T_{k_1}^*$ as long as it receives $o_i$.

Step 4. At last, a $P'.redirect(i)(T_j)$ operation is invoked to redirect the global directory to $T_j$. From Lemma 3, we know that it can be implemented by $C redirect(i)(T_j)$ (Figure 5.9(d)).

This completes the sequence of steps needed to implement $P'.head(i)(T_j)$.

Definition 23 (Heading Cost). The heading cost $\eta'(T_j, T_{k_1}^*)$ is the total time complexity of $P'.head(i)(T_j)$, where $T_{k_1}^*$ is the head of $Q_i^*$ when $T_j$ requests to join $Q_i^*$. Specifically,

$$\eta'(T_j, T_{k_1}^*) = \delta_s(T_j, T_{k_1}^*) + \zeta^P(T_{k_1}^*, T_j) + \delta_r(T_j, T_{k_1}^*) = \delta_c(T_j, T_{k_1}^*) + \zeta^P(T_{k_1}^*, T_j).$$

We have the following theorem to generalize the implementation of $P'$:

Theorem 11. Given a distributed queuing protocol $C$ and a DQCC protocol $P$ implemented based on $C$ according to Theorem 10, a DPQCC protocol $P'$ can be implemented such that Lemmas 1, 2, 3, and 4 hold.

Proof. The theorem follows from our 3-step description to implement $P'.insert(i)(T_j)$, the description to implement $P'.dequeue(i)(T_{k_1}^*)$, and the 4-step description to implement $P'.head(i)(T_j)$.

5.3 Analysis

5.3.1 Cost Measures

DQCC protocols. Consider a set of $N$ transactions $\mathcal{T} := \{T_1, T_2, \ldots, T_N\}$ that require access to an object. We can construct a distributed queue $Q_i := \{Q_0, Q_1, Q_2, \ldots, Q_L\}$ under a DQCC protocol $P$, where $L \geq N$. Specifically, we can arrange each element $Q_i$ in the order such that $Q_i$ is the predecessor of $Q_{i+1}$. Note that $Q_0$ is the dummy transaction at the object’s initial location. Hence, each transaction in $\mathcal{T}$ is mapped to at least one element in $Q$, since each transaction may join the queue multiple times. Arrange the set of transactions $\mathcal{T}$ in the priority-order. Specifically, let $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$, where $T_j \prec T_k$ if $j < k$. We have the following theorem:
Theorem 12.

\[ \sigma_j \leq j \]

where \( \sigma_j \) is the number of times that transaction \( T_j \) joins the queue.

Proof. The theorem is straightforwardly proved by the combination of work conserving and pending commit properties of the contention manager. \( \square \)

From Theorem 12, we have the following corollary:

**Corollary 1.**

\[ L \leq \frac{N(N-1)}{2} \]

Theorem 12 and Corollary 1 give the upper bounds of the enqueue times for a single transaction and the total enqueue times for a set of transactions, respectively. It tells us that the enqueue times of a transaction depends on its priority. Intuitively, a transaction with a higher priority implies fewer enqueue times, and less cost to commit. Let the cost to commit a transaction \( T_j \) be denoted as \( M_j \). Hence, a transaction \( T_j \) invoked at time \( t_j \) will commit at time \( t_j + M_j \). We can further define the total cost for a set of transactions \( T \) as:

\[ M = \max_{j=1}^{N} (t_j - t^1 + M_j), \]

where \( t^1 = \min_{j=1}^{N} t_j \).

A more useful cost measure is the amortized cost of a single transaction \( T_j \), i.e., the contribution made by transaction \( T_j \) to the total cost \( M \).

**Theorem 13.** Let the amortized cost of a transaction \( T_i \) under \( P \) be defined as:

\[ M(j) = \sum_{k=1}^{\sigma_j} [\delta^C(Q_{j(k)}, Q_{j(k)}-1) + \zeta^P(Q_{j(k)}-1, Q_{j(k)}) + \tau_j], \]

where \( Q_{j(k)} \) is the element mapped to transaction \( T_j \), which joins in the \( k^{th} \) time.

Then we have:

\[ M \leq \sum_{j=1}^{N} M(j). \]

Proof. The total cost \( M \) describes the time complexity to commit all transactions in \( T \). The total cost is composed of three parts: the cost for all the objects accessed by transactions in \( T \) to travel in the network, the cost for executing operations on that set of objects by the transactions in \( T \), and the cost for that set if objects to wait for requests. From the definition of amortized cost, the theorem follows. \( \square \)
**DPQCC protocols.** For a DPQCC protocol, a transaction can only be aborted when it becomes the queue’s head and a higher priority transaction joins the queue. We have the following theorem:

**Theorem 14.**

\[ \sum_{j=1}^{N} \mu_j \leq 2N - 1, \]

where \( \mu_j \) is the number of times that transaction \( T_j \) joins the queue.

**Proof.** In a set of transactions \( T = \{ T_1, T_2, \ldots, T_N \} \) in the priority order, there are \( j - 1 \) transactions with higher priority than \( T_j \). When \( T_j \) joins the queue, there are \( \lambda_i \) transactions with higher priority in the queue. \( T_i \) can only commit after those \( \lambda_i \) transactions commit. Hence, \( T_i \) will be aborted at most \( i - \lambda_i - 1 \) times.

A transaction can only be aborted when a new transaction joins the queue. Therefore, the second part of the theorem is proved directly. Theorem follows.

Now we focus on the total cost \( M' \) for a set of transactions \( T \) under \( P' \) to under \( P' \). Similar to a DQCC protocol, we have the following theorem:

**Theorem 15.** Let the amortized cost of \( T_j \) under \( P' \) be defined as:

\[
M'(j) = \begin{cases} 
\xi'_{jH}(T_j, Q_{jsuc}) + \tau_j, & Q_{jH} \prec T_j \\
\xi'_{jH}(T_j, Q_{jsuc}) + \tau_j + \eta'_{jH}(T_j, Q_{jH}) + \tau_{jH}, & T_j \prec Q_{jH} 
\end{cases}
\]

(5.2)

where \( Q_{jH} \) is the head of the queue when \( T_j \) joins the queue, and \( Q_{jsuc} \) is \( T_j \)'s successor when \( T_j \) leaves the queue. Then we have: \( M' \leq \sum_{j=1}^{N} M'(j) \).

**Proof.** Whenever a transaction \( T_j \) is inserted into the queue, the cost of \( insert(i)(T_j) \) does not increase the total cost \( M' \). Assume that \( T_j \) is inserted into the queue. At first, its request is sent to \( Q_{jH} \), which holds the object. Then \( Q_{jH} \) moves \( T_j \)'s request following its local pointer down the queue. Clearly, the object will only be moved from \( Q_{jH} \) after \( T_j \)'s request arrives at \( Q_{jH} \). Otherwise, the global directory would be redirected. In other words, the cost of \( insert(i)(T_j) \) is covered by the local execution cost and the moving cost of other transactions. Hence, the total cost \( M' \) is at most the sum of all possible local execution costs and moving costs, which is \( \sum_{j=1}^{2N-1} M'(j) \). Theorem follows.

### 5.3.2 Cost Metrics

**Cost Metric of \( P \).** We first analyze the total cost of \( P \). Given a queue \( Q = \{ Q_0, Q_1, \ldots, Q_L \} \), we define the cost metric to order \( Q_k \) after \( Q_j \) under \( P \) as

\[
c_{q}(Q_k, Q_j) := \delta^{C}(Q_k, Q_j) + \]
\( \zeta^P(Q_j, Q_k) \). From Theorem 13, we have:

\[
\sum_{j=1}^{N} M(j) = \sum_{l=1}^{L} c_q(Q_{l-1}, Q_l) + \sum_{k=1}^{N} \sigma_k \tau_k. \tag{5.3}
\]

**Cost Metric of** \( P' \). We\( \text{ } \) decompose \( \text{each transaction} \) \( T_i \) into a set of sub-transactions \( \{T_j(1), T_j(2), \ldots, T_j(\mu_j + 1)\} \). Specifically, we have \( T_j = (v_j, t_j, \tau_j) \) and \( T_j(l) = (v_j, t_j(l), \tau_j) \), where \( t_j(1) = t_j \) and \( t_j(l) \) is the time when \( T_i \)'s \((l-1)\)th abort occurs. For each single object, we arrange all the sub-transactions in the order of obtaining the object: \( T' = \{T(0), T(1), T(2), \ldots, T(L')\} \), where \( N \leq L' \leq 2N - 1 \), since each transaction may obtain the object multiple times. The second inequality stems from Theorem 14. Each sub-transaction \( T_j(l) \) is mapped to one element in \( T' \). \( T(j) = (v(j), t(j), \tau(j)) \) denotes the \( j \)th transaction that receives the object in \( P' \)'s order. We define the cost metric to order \( T(k) \) after \( T(j) \) under \( P' \) as:

\[
c_i(T(j), T(k)) := 2\delta^C(T(k), T(j)) + \zeta^P(T(j), T(k)). \tag{5.4}
\]

Note that \( P' \) is based on the distributed queuing protocol \( C \). From 5.2, we have:

\[
\sum_{j=1}^{N} M'(j) \leq \sum_{l=1}^{L'} c_i(T(l-1), T(l)) + \sum_{k=1}^{N} (\mu_k + 1) \tau_k.
\]

**Cost Metric of** \( \text{OPT} \). To evaluate the cost of \( P \) and \( P' \), we now consider the cost of an optimal clairvoyant offline ordering algorithm, denoted \( \text{OPT} \), that has complete knowledge of all the transactions \( T \). Clearly, for each object, an optimal offline algorithm has to order each transaction to receive the object just once to commit. Let \( \phi_o \) be the order of \( \text{OPT} \). For the cost of \( \text{OPT} \), we have to take into account the complete knowledge of all transactions. For a transaction \( T_j = ((v_j, t_j, \tilde{R}(j), \tau_j)) \), the algorithm \( \text{OPT} \) already knows the succeeding transaction \( T_k = ((v_k, t_k, \tilde{R}(k), \tau_k)) \). When an object is available at \( v_j \), the algorithm can immediately send the object to \( v_k \). Hence, we define the transaction \( T_j \)'s completion time in the order \( \phi_o \) as \( t^O_{j} \). We therefore define the moving cost \( c_O(T_j, T_k) \) of ordering \( T_k \) after \( T_j \) in the order \( \phi_o \) as:

\[
c_O(T_j, T_k) := d(v_j, v_k) + \max\{0, t^O_j - t_k + d(v_j, v_k)\} + \tau_k
\]

\[
\geq d(v_j, v_k) + \max\{0, t_j - t_k + d(v_j, v_k)\} + \tau_k.
\]

The total cost of an optimal algorithm with respect to \( R_i \) therefore becomes:

\[
\text{cost}_{\text{OPT}} = \min_{\phi} \left\{ \sum_{j=1}^{N} c_O(T_{\phi_o(j-1)}, T_{\phi_o(j)}) \right\} \tag{5.4}
\]

Hence, \( \phi_o \) is an order which minimizes the sum of 6.6. Now, we can define the **competitive ratio** of \( P \) and \( P' \):

**Definition 24** (Competitive Ratio). \( \rho_P = \frac{M}{\text{cost}_{\text{OPT}}} \), \( \rho_{P'} = \frac{M'}{\text{cost}_{\text{OPT}}} \).
5.3.3 Order Analysis

We now focus on the orders in \(Q\) and \(T'\) produced by \(P\) and \(P'\), respectively. Motivated by the method in [91], we prove that the orders produced by \(P\) and \(P'\) correspond to two nearest neighbor traveling salesman paths (TSPs) by defining two new comparable cost metrics.

**Definition 25.**

\[
c_Q(Q_j, Q_k) := t_{Q_k} - t_{Q_j} + \delta^C(Q_j, Q_k),
\]

where \(t_{Q_k}\) is the time that \(Q_k\) sends a request to join the queue.

**Definition 26.**

\[
c_T(T(j), T(k)) := \begin{cases} t(k) - t(j) + \eta^{P'}(T(j), T(k)), & T_k \prec T_j \\ t(k) - t(j) + \xi^{P'}(T(j), T(k)), & T_j \prec T_k \end{cases}
\]

We have the following theorem.

**Theorem 16.** The orders of \(Q\) and \(T'\) are defined by the two nearest neighbor TSPs on metrics \(c_Q(Q_j, Q_k)\) and \(c_T(T(j), T(k))\), starting with \(Q_0\) and \(T(0)\), respectively. Further, \(c_Q(Q_j, Q_k) \geq 0\) for all pairs of \(Q_j\) and \(Q_k\), and \(c_T(T(j), T(k)) \geq 0\) for all pairs of \(T(j)\) and \(T(k)\).

**Proof.** We prove the theorem by induction. The object is initialized at \(T_0\), which corresponds to the dummy transaction. Hence, we have \(t_{Q_0} = t(0) = t_0\). For the order of \(Q\), the transaction \(Q_k\) that minimizes \(t_{Q_k} - t_0 + \delta^C(Q_0, Q_k)\) arrives at \(Q_0\) first. The same case holds for the order of \(T'\), for which the transaction \(T(k)\) that minimizes \(t(k) - t(0) + \eta^{P'}(T(j), T(k))\) arrives at \(T(0)\) first. Clearly, \(\{c_Q(Q_0, Q_1), c_T(T(0), T(1))\} \geq 0\).

We now focus on the order of \(Q\). Assume that \(Q_{v}\) is the transaction that minimizes \(c_Q(Q_{v-1}, Q_{l})\) for all \(Q_{m} \in Q\{Q_{0}, Q_{1}, \ldots, Q_{l-1}\}\). From the definition of \(Q\), we know that \(Q_{v+1}\) will receive the object from \(Q_{v}\). Note that at time \(t_{Q_{v'}} + \delta^C(Q_{v}, Q_{v-1})\), the object is moved from \(Q_{v-1}\) to \(Q_{v'}\). Hence, the transaction that minimizes \(c_Q(Q_{v'}, Q_{m'})\) for all \(Q_{m'} \in Q\{Q_{0}, Q_{1}, \ldots, Q_{l}\}\) is \(Q_{v+1}\), which is the first transaction that was ordered after \(Q_{v}\).

Note that \(c_Q(Q_{v-1}, Q_{v}) \leq c_Q(Q_{v-1}, Q_{v+1})\). Then:

\[
0 \leq c_Q(Q_{v-1}, Q_{v+1}) - c_Q(Q_{v-1}, Q_{v}) \\
\leq t_{Q_{v+1}} - t_{Q_{v-1}} + \delta^C(Q_{v-1}, Q_{v+1}) - (t_{Q_{v'}} - t_{Q_{v'-1}} + \delta^C(Q_{v-1}, Q_{v+1})) \\
\leq t_{Q_{v+1}} - t_{Q_{v'}} + \delta^C(Q_{v}, Q_{v+1}) = c_Q(Q_{v}, Q_{v+1})
\]

Similar induction steps hold for the order of \(T'\). Theorem follows. \(\square\)

Let \(C_Q = \sum_{k=0}^{L} c_Q(Q_{k-1}, Q_k)\) and \(C_T = \sum_{l=0}^{L'} (T(l-1), T(l))\). We have the following lemma:
Lemma 5.

\[ C_Q \geq \frac{1}{2} \sum_{l=1}^{L} c_q(Q_{l-1}, Q_l) \]

\[ C_T \geq \frac{1}{2} \sum_{l=1}^{L'} c_t(T(l-1), T(l)) \]

Proof. The lemma follows from Theorem 16. □

In the following theorem, we give the upper bounds of \( c_Q(Q_j, Q_k) \) and \( c_T(T(j), T(k)) \):

Theorem 17.

\[ c_Q(Q_j, Q_k) \leq D_\delta + D_\zeta + \max_{j=1}^{N} \tau_j \quad \text{and} \quad c_T(T(j), T(k)) \leq 2D_\delta + D_\zeta + \max_{j=1}^{N} \tau_j \]  

(5a, b)

where

\[ D_\delta = \max_{Q_j, Q_k \in \mathcal{Q}} \delta^C(Q_j, Q_k) = \max_{T(j), T(k) \in \mathcal{T}'} \delta^C(T(j), T(k)) \],

and

\[ D_\zeta = \max_{Q_j, Q_k \in \mathcal{Q}} \zeta^P(Q_j, Q_k) = \max_{T(j), T(k) \in \mathcal{T}'} \zeta^P(T(j), T(k)) \].

Proof. When the transactions are sparse enough — i.e., in a relatively long time period, only one transaction is invoked, \( P, P' \), and \( \text{OPT} \) produce the same ordering. We can shift the transactions as much as possible without increasing the costs of \( P, P' \), and \( \text{OPT} \).

Let \( Q_l \) and \( Q_{l+1} \) be two consecutive transactions in the order of \( \mathcal{Q} \). Let \( \epsilon := c_Q(Q_l, Q_{l+1}) - \delta^C(Q_{l-1}, Q_l) - \tau' \). If \( \epsilon > 0 \), for all transactions \( Q_m \), where \( m \geq l + 1 \), \( t_{Q_{l+1}} \) can be replaced by \( t_{Q_{l+1}} - \epsilon \) without increasing the costs of \( C \) and \( \text{OPT} \). By applying this method as many times as possible, we have 5.5a.

The same argument holds for the order of \( \mathcal{T}' \), resulting in 5.5b. Theorem follows. □

5.3.4 Comparison

We first define the Manhattan metric \( c_M \), which is comparable to \( c_Q \) and \( c_T \).

Definition 27 (Manhattan Metric). The Manhattan metric \( c_M(T_j, T_k) \) is defined as:

\[ c_M(t_j, t_k) := d(v_j, v_k) + |t_j - t_k| + \tau_j + \tau_k. \]

Lemma 6. Let \( \phi \) be an ordering, and \( C_O \) and \( C_M \) be the costs for ordering all transactions in order \( \phi \) with respect to \( c_O \) and \( c_M \), respectively. The Manhattan cost \( C_M \) is bounded by:

\[ C_M \leq 2C_O + t_{\phi(N)}. \]
Proof. We can lower bound the optimal cost of $c_O$ as:

$$c_O(T_j, T_k) \geq d(v_j, v_k) + \max\{0, t_j - t_k\} + \tau_k.$$  

Let $D_O = \sum_{j=1}^{N} \{d(v_{\phi(j-1)}, \phi(j)) + \tau_j + \tau_{j-1}\}$. Then we have:

$$2C_O \geq D_O + 2N \sum_{j=1}^{N} \max\{0, t_{\phi(j-1)} - t_j\} = D_O + \sum_{j=1}^{N} [0, t_{\phi(j-1)} - t_j] - t_{\phi(N)} = C_M - t_{\phi(N)}$$

Now, we can measure the competitive ratio of $P$ and $P'$. We have the following theorem:

**Theorem 18.**

$$\rho_P = O\left(\max[N \cdot \log_2 \left(\frac{2D_\delta + \max_{i=1}^{N} \tau_i}{\min_{v_j, v_k \in V} d(v_j, v_k)}\right) \cdot N \cdot \frac{\max_{j=1}^{N} \sigma_j \tau_j}{H}\right)$$  \hspace{1cm} (5.6)

$$\rho_P' = O\left(\max[\log_2 \left(\frac{3D_\delta + \max_{i=1}^{N} \tau_i}{\min_{v_j, v_k \in V} d(v_j, v_k)}\right) \cdot \frac{\max_{j=1}^{N} \mu_j \tau_j}{H}\right)$$  \hspace{1cm} (5.7)

where $H$ is the total cost of the TSP path for $\mathcal{T}$ with respect to the metric $d(v_j, v_k)$.

Proof. We use the following lemma from [91]:

**Lemma 7.** Let $c'_M(T_j, T_k) := d(v_j, v_k) + |t_j - t_k|$, and $C'_M$ be the cost of ordering all requests in order $\phi$ with respect to $c'_M$. Then,

$$C'_M \geq \frac{3}{2} t_N$$

where $t_N = \max_{i=1}^{N} t_i$.

Hence, we have the following theorem, which allows $C_M$ to be comparable to $C_O$:

**Theorem 19.**

$$C_M \leq 6C_O$$

Proof. The theorem can be proved by the Lemmas 10 and 11. Note that we have $c_M \geq c'_M$, and $t_N \geq t_{\phi(N)}$. Then the theorem follows. \hfill \Box

We now compare $C_M$, $C_Q$, and $C_T$ with the help of the following lemma from [91]:
Lemma 8. Let $V$ be a set of $N := |V|$. Let $d_n : V \times V \to \mathbb{R}$ and $d_o : V \times V \to \mathbb{R}$ be the distance functions between the nodes of $V$. For $d_n$ and $d_o$, the following conditions hold:

\[
\begin{align*}
  d_o(u, v) &= d_o(v, u), \quad d_n(u, v) = d_n(v, u) \\
  d_o(u, v) &\geq d_n(u, v) \geq 0, \quad d_o(u, u) = 0 \\
  d_o(u, w) &\leq d_o(u, v) + d_o(v, w)
\end{align*}
\]

Let $C_N$ be the length of a nearest neighbor TSP tour with respect to the distance function $d_n$. Let $C_O$ be the length of an optimal TSP tour with respect to the distance function $d_o$. Then,

\[
C_N \leq \frac{3}{2} \left\lfloor \log_2 \left( \frac{D_{NN}}{d_{NN}} \right) \right\rfloor \cdot C_O,
\]

where $D_{NN}$ and $d_{NN}$ are the lengths of the longest and the shortest non-zero edge on the nearest neighbor tour with respect to $d_n$, respectively.

Now we can compare $C_Q$, $C_T$, and $C_M$ based on Lemma 8.

Theorem 20.

\[
\begin{align*}
C_Q &\leq \frac{3L}{2N} \left\lfloor \log_2 \left( \frac{2D_\delta + \max_{i=1}^N \tau_i}{\min_{v_j, v_k \in V} d(v_j, v_k)} \right) \right\rfloor \left( C_M - 2 \sum_{j=1}^N \tau_j \right) \\
C_T &\leq \frac{3L'}{2N} \left\lfloor \log_2 \left( \frac{3D_\delta + \max_{i=1}^N \tau_i}{\min_{v_j, v_k \in V} d(v_j, v_k)} \right) \right\rfloor \left( C_M - 2 \sum_{j=1}^N \tau_j \right)
\end{align*}
\]

Proof. This theorem follows from Theorem 17 and Lemma 8. Note that $c_Q$ and $c_T$ comply with the condition for $d_n(u, v)$, and $c_M$ complies with the condition for $d_o(u, v)$. In addition, the triangle inequality holds for $c_M$. Finally, we can bound the shortest value of $c_T$ by $\min_{v_j, v_k \in V} d(v_j, v_k)$. Theorem follows. \qed

Now we can prove Theorem 18. We have:

\[
M \leq \sum_{i=1}^N M(i) = \sum_{k=1}^L c_q(Q_{k-1}, Q_k) + \sum_{j=1}^N \sigma_j \tau_j \leq 2C_Q + \sum_{j=1}^N \sigma_j \tau_j,
\]

where the first inequality follows from Theorem 13, the second equality follows from 5.3, and the third inequality follows from Lemma 5. On the other hand,

\[
\text{cost}_{\text{Opt}} \geq H + \sum_{j=1}^N \tau_j.
\]

Then 5.6 holds. We can prove 5.7 in the same way. Theorem 18 follows. \qed
From Theorem 18, we know that the competitive ratio is determined by the maximum $\tau_j$. We have the following corollary that gives a possible range for the value of the maximum $\tau_j$.

**Corollary 2.**

$$\rho_P = O(N \log \overline{D}_{\delta})$$
$$\rho_{P'} = O(\log \overline{D}_{\delta})$$

where $\overline{D}_{\delta}$ is the normalized diameter $\frac{D_{\delta}}{\min_{v_j, v_k \in V} d(v_j, v_k)}$, if $\max_{j=1}^{N} \tau_j = O(\log D_{\delta})$.

In other words, if the maximum local execution time of a set of transactions $T$ is sufficiently small (up to the logarithmic order of $D_{\delta}$), then the competitive ratio $\rho_P$ is $O(N \log \overline{D}_{\delta})$, and $\rho_{P'}$ is $O(\log \overline{D}_{\delta})$. Hence, a DPQCC protocol guarantees a worst-case competitive ratio that is better than that of a DQCC protocol by a factor proportional to $N$, when both the protocols are based on the same distributed queuing protocol $C$.

### 5.4 Conclusion

In this chapter, we formalize two classes of CC protocols for D-STM, and compare their competitive ratios. We compare the DQCC and DPQCC protocols against the optimal offline algorithm $\text{Opt}$. In practice, it is often difficult to explicitly describe the algorithm $\text{Opt}$. Hence, we adopt an analytical method similar to the methods used in [34, 35] and [91], where the optimal algorithm is implicitly described by its cost.

Our analysis thus shows that, a DPQCC protocol guarantees a much better performance bound than a DQCC protocol, given the same underlying distributed queuing protocol. Further, if the maximum local execution time is sufficiently small (up to the logarithmic order of $D_{\delta}$), a DQCC protocol is $O(N \log \overline{D}_{\delta})$-competitive and a DPQCC protocol is $O(\log \overline{D}_{\delta})$-competitive. This means that, when the network latency is the significant part of the communication cost, the selection of CC protocols determines the overall performance, since it determines the total cost for an object to travel in the network. On the other hand, when the local execution time is relatively large, the total execution cost of transactions will be the dominating part of the total time complexity. In this case, D-STM is more similar to multiprocessor STM, where the underlying contention manager determines the maximum abort times of each transaction.
Chapter 6

RELAY Cache-Coherence Protocol

In this chapter we present RELAY protocol, a novel CC protocol, which efficiently reduces the total number of aborts for a given set of transactions. We show that RELAY’s static competitive ratio is significantly improved when compared against past distributed queuing protocols like ARROW [49]. We also analyze RELAY for a set of transactions which are dynamically generated in a given time period, and measure RELAY’s dynamic competitive ratio in terms of the communication cost (for dynamically generated transactions). We show that RELAY is $O(\log D_0)$-competitive, where $D_0$ is the normalized diameter of the spanning tree.

6.1 Rationale

Our work is motivated by the ARROW protocol [49], which is a simple distributed queuing protocol based on path reversal on a network spanning tree. Distributed queuing is a fundamental problem in the management of synchronization accesses to mobile objects in a network. When multiple nodes in the network request an object concurrently, the requests must be queued in some order, and the object travels from one node to another down the queue. To manage such a distributed queue, an efficient distributed queuing protocol must solve two problems: a) how to order the requests from different nodes into a single queue; and b) how to provide the necessary information to nodes such that each node knows the location of its successor in the queue and the object can be forwarded down the queue. Note that the protocol is “distributed” in the sense that no single node needs to have the global knowledge of the queue. Each node only has to know its successor in the queue and forward the object to it.

ARROW runs on a fixed spanning tree $ST$ of $G$. Each node $v$ keeps an “arrow” or a pointer $p(v)$ to itself or to one of its neighbors in $ST$. If $p(v) = v$, then $v$ is the tail of the queue, i.e., the next request should be forwarded to $v$. In this case, the node $v$ is defined as a “sink”.

56
Clearly, at any time, there exists only one sink for each object. If \( p(v) = u \), then \( p(v) \) only knows the “direction” of the tail of the queue and the request is forwarded following that direction.

The protocol works based on path reversal, as shown in Figure 6.1. When an object is created by a node \( u \), the arrows are initialized such that following the arrows from any node leads to the object (Figure 6.1(a)). A node \( v \) requests the object by sending a find message to \( p(v) \) and flips \( p(v) \) to point to \( v \). When a node \( w' \) receives a find message from its neighbor \( w \), there are two possible cases: 1) if \( p(w') \neq w' \), then it forwards the find message to \( p(w') \) and flips \( p(w') \) to point to \( w \); and b) if \( p(w') = w' \), then the find message has arrived at the tail of the queue (Figure 6.1(b)). The object will move to \( v \) after it arrives at \( u \), and \( p(u) \) is also flipped to point to \( w' \) (Figure 6.1(c)). We only give an informal description of the protocol here, and more details can be found in [49].

Figure 6.1: The Arrow Protocol

Arrow is attractive as a candidate CC protocol. In the context of D-STM, nodes request access to mobile objects in the network. Hence, a CC protocol must arrange the requests to be ordered in a queue. If we directly apply Arrow as a CC protocol, we immediately have the following relationships: \( \text{max-str}^{\delta}_{\text{Arrow}} = \text{max-str}(ST) \) and \( \text{max-str}^{\zeta}_{\text{Arrow}} = 1 \). Hence, the maximum locating stretch of Arrow is the maximum stretch of the underlying spanning tree. The maximum moving stretch of Arrow is 1 — i.e., the object can be directly moved via the shortest path.

However, the difference between Arrow and a CC protocol is that Arrow does not consider the contention between two transactions. Hence, Arrow is not able to reduce the worst-case number of aborts \( \lambda_C(j) \). We have the following theorem:

**Theorem 21.**

\[
\max_{v_j \in V_T} \lambda_{\text{Arrow}}(j) \leq N_i
\]

\[
\Lambda_{\text{Arrow}} \leq \frac{N_i(N_i + 1)}{2}
\]

**Proof.** Note that we assume a contention manager with work conserving and pending commit properties. Hence, we know that at any given time, there exists at least one transaction in \( V_T \) that will execute uninterruptedly until it commits. Let \( T_{\text{Arrow}}^* \) denote the transaction
with the maximum worst-case number of aborts in $V_T$. We first prove that, for each time that $T^*_\text{Arrow}$ is aborted by another transaction in $V_T$, at least one transaction in $V_T$ will commit before $T^*_\text{Arrow}$ is aborted again.

Assuming that $T^*_\text{Arrow}$ is aborted by another transaction $T^\text{head}_\text{Arrow}$, we have $T^\text{head}_\text{Arrow} \prec T^*_\text{Arrow}$. According to Arrow, the “arrows” are now pointed to the tail of the queue, which is the latest transaction requesting the object. Transaction $T^*_\text{Arrow}$’s new request will be forwarded to the tail of the queue. We now focus on the set of transactions between $T^\text{head}_\text{Arrow}$ and $T^*_\text{Arrow}$ in the queue, denoted by $S$. Let $T'$ be the transaction with the highest priority in $S$. If $T' \prec T^\text{head}_\text{Arrow}$, then $T'$ will commit before it forwards the object down the queue. Otherwise, $T^\text{head}_\text{Arrow}$ will commit. In both cases, at least one transaction will commit before the object is forwarded to $T^*_\text{Arrow}$ again.

Now, it is easy to prove that $\max_{j \in V_T} \lambda_{\text{Arrow}}(j) \leq N_i$, as for each time that $T^*_\text{Arrow}$ is aborted, at least one transaction in $V_T$ commits. By induction, we can further prove that the second-maximum worst-case number of aborts in $V_T$ is at most $N_i - 1$, the third-maximum worst-case number of aborts is at most $N_i - 2$, etc. The theorem follows.

We now have the following corollary from Theorem 2:

**Corollary 3.**

$$CR_i(\text{Arrow}) \leq \frac{(N_i + 1)}{2} \cdot \max-str(S\mathcal{T}) \cdot \text{Diam}$$

Hence, a new CC protocol should focus on minimizing the number transaction aborts to improve the upper-bound of the static competitive ratio.

### 6.2 Protocol Description

**RELAY** is inspired by **Arrow**, which is based on path reversal on a network spanning tree. It is a distributed CC protocol designed for the synchronized management of transactional accesses to mobile objects (i.e., D-STM model) in a network. To manage a distributed queue formed by transactional requests instead of simple ordering requests, an efficient distributed CC protocol must solve three problems: a) how to order the transactional requests from different nodes into a single queue; b) how to provide the necessary information to nodes such that each node knows the location of its successor in the queue and the object can be forwarded down the queue; and c) how to efficiently reduce the length of the queue.

We assume a fixed-rooted spanning tree $S\mathcal{T}$ of the given network $G$. Given the spanning tree $S\mathcal{T}$, we define the distance in $S\mathcal{T}$ between a pair of two nodes, $u$ and $v$, to be the sum of the lengths of the edges on the unique path in $S\mathcal{T}$ between $u$ and $v$, denoted by $d_{\mathcal{T}}(u, v)$. 


Now, we define the *stretch* of \( u \) and \( v \) in \( ST \) as:

\[
\text{str}_{ST}(u,v) = \frac{d_{ST}(u,v)}{d(u,v)}.
\]

Let the *maximum stretch* of \( ST \) be denoted as:

\[
\text{max-str}(ST) = \max_{u,v \in V} \{\text{str}_{ST}(u,v)\}.
\]

We define the *diameter* \( D \) of \( ST \) as:

\[
D = \max_{u,v \in V} d_{ST}(u,v);
\]

and the *normalized diameter* \( D_0 \) of \( ST \) as:

\[
D_0 = \max_{u,v,x,y \in V} \left\{\frac{d_{ST}(u,v)}{d_{ST}(x,y)}\right\}.
\]

**RELAY** is initialized in the same way as **Arrow**. At the start, the node \( v_{\text{tail}} \), where the object resides, is selected to be the tail of the queue. Each node \( v \in V \) maintains a pointer \( p(v) \) and is initialized so that following the pointers from any node leads to the tail, as shown in Figure 6.1.

To request the object after the initialization, a transaction \( T_1 \) invoked by node \( v_1 \) sends a *find message* \( \text{find}(v_1) \) to node \( p(v_1) \). Note that \( p(v_1) \) is not modified when a *find message* is forwarded, which is different from **Arrow**. If a node \( w \) between \( v \) and the tail of the queue receives a *find message*, it simply forwards the *find message* to \( p(w) \). At the end, the *find message* will be forwarded to the tail of the queue without changing any pointers.

The *find message* \( \text{find}(v_1) \) keeps a *path vector* \( \tilde{\text{path}} \) to record the path it travels. Each node receiving the *find message* from \( v_1 \) appends its ID to \( \text{find}(v_1).\tilde{\text{path}} \). When the *find message* arrives at the tail of the queue, the vector \( \text{find}(v_1).\tilde{\text{path}} \) records the path from \( v_1 \) to the tail \( v_{\text{tail}} \). Such an operation is shown in Figure 6.2.

Now the tail of the queue \( v_{\text{tail}} \) receives a *find message* from node \( v_1 \). We have to examine the status of the transaction \( T_{\text{tail}} \) which also requires the object. If \( T_{\text{tail}} \) has committed, then the object is moved to \( v_1 \). This case is trivial except the way that pointers are updated (we will discuss that update process in detail later). If \( T_{\text{tail}} \) has not committed, the contention manager of \( v_{\text{tail}} \) has to compare the priorities of \( T_1 \) and \( T_{\text{tail}} \). We discuss this scenario case by case.

- **Case 1:** If \( T_1 \prec T_{\text{tail}} \), then \( T_{\text{tail}} \) is aborted and the object is moved to \( v_1 \). The pointers are updated when the object is moved. To let the pointers update correctly, node \( v_{\text{tail}} \) sends a *move message* \( \text{move}(v_{\text{tail}}) \) with a *route vector* \( \text{route} \) which records the route that \( \text{move}(v_{\text{tail}}) \) will travel. In this case, \( \text{move}(v_{\text{tail}}).\text{route} = \text{find}(v_1).\tilde{\text{path}} \). Hence,
node $v_{\text{tail}}$ sends the object with $\text{move}(v_{\text{tail}})$ to $\text{move}(v_{\text{tail}}).\text{route}[\text{max}]$ (the last element of $\text{move}(v_{\text{tail}}).\text{route}$). Meanwhile, node $v_{\text{tail}}$ sets $p(v_{\text{tail}})$ to $\text{move}(v_{\text{tail}}).\text{route}[\text{max}]$. Then $T_{\text{tail}}$ restarts and immediately sends a $\text{find}(v_{\text{tail}})$ message to $p(v_{\text{tail}})$. Suppose a node $u$ receives a $\text{move}$ message from one of its neighbors. It updates $\text{move}(v_{\text{tail}}).\text{route}$ by removing $\text{move}(v_{\text{tail}}).\text{route}[\text{max}]$ and sends the object to the new $\text{move}(x).\text{route}[\text{max}]$, setting $p(u) = \text{move}(v_{\text{tail}}).\text{route}[\text{max}]$. Finally, when the object arrives at $v_1$, $p(v_1)$ is set to $v_1$ and all pointers are updated. Such operations guarantee that at any given time, there exists only one sink in the network, and, from any node, following the direction of its pointer leads to the sink. Such an operation is shown in Figure 6.3.

Case 2: If $T_{\text{tail}} \prec T_1$, then $T_1$ will be postponed to let $T_{\text{tail}}$ commit. Node $v_{\text{tail}}$ stores a “virtual pointer” $\text{next}(v_{\text{tail}}) = v_1$. The object is moved to $\text{next}(v_{\text{tail}})$ once after $T_{\text{tail}}$ commits. Hence, $\text{next}(v_{\text{tail}})$ has to keep a route vector $\text{next}(v_{\text{tail}}).\text{route}$ to record the path from $v_{\text{tail}}$ to itself. In this case, $\text{next}(v_{\text{tail}}).\text{route} = \text{find}(v_1).\text{path}$. We show this operation in Figure 6.4.

Note that in the above scenario, if a node sends a new find message when a move message is moving in the network, RELAY guarantees that the find message will arrive at the object’s new location: although the find message may travel along the “old” direction to the object’s original location, at some node in the middle of the path, the direction must be updated (in the worst case, the node is the object’s original holder), and the find message ultimately will be directed to the object’s new location.

Since the pointers are not updated until the object is moved, and the object will only be moved unless the running transaction $T_{\text{tail}}$ has committed or it receives another transaction with higher priority, node $v_{\text{tail}}$ may receive multiple find messages. Suppose it receives another $\text{find}$ message from $v_2$. If $T_2 \prec T_{\text{tail}}$, then it falls into Case 1. If $T_{\text{tail}} \prec T_2$, then the contention manager compares the priorities of $\text{next}(\text{tail})$ (in this case it is $T_1$) and $T_2$. If $T_1 \prec T_2$, then the find message from $v_2$ is forwarded to $v_1$. If $T_2 \prec T_1$, then $v_{\text{tail}}$ sets $\text{next}(\text{tail})$ to $T_2$ and forwards the find message from $v_1$ to $v_2$.

A problem appears when $v_{\text{tail}}$ forwards find messages from other nodes to a new node,
e.g., find($v_1$) to $v_2$. In this case, the path vector should record the path from $v_1$ to $v_2$. However, since find($v_1$) is forwarded along the path $v_1 \rightarrow v_{\text{tail}} \rightarrow v_2$, the path recorded in find($v_1$).path is not the shortest path from $v_1$ to $v_2$ in the spanning tree $ST$. Hence, the path vector has to be correctly updated to record the shortest path. We illustrate this update policy with the help of an example, as shown in Figure 6.5.

Since there is only one path in a spanning tree between two nodes such that each node in the path is visited exactly once, the path vector is updated to detect and eliminate nodes that have been visited multiple times. In the example of Figure 6.5, node $v_{\text{tail}}$ has to forward find($v_1$) to $v_2$. Initially, find($v_1$).path = [v_1, v_3, v_4, v_6]. When node $v_6$ receives find($v_1$), it first checks the last two elements of find($v_1$).path, which are $v_4$ and $v_6$. Since they are different, $v_6$ simply appends its ID to the path vector, as shown in Figure 6.5(a). Now, find($v_1$) arrives at $v_4$ and the last two elements of find($v_1$).path are the same ($v_6$). Node $v_4$ has to check the third last element of the path vector ($v_4$) to see whether a loop forms. Hence, a loop forms by [v_4, v_6, v_6, v_4] and $v_6$ is deleted from the path vector since it is not on the shortest path from $v_1$ to $v_2$, as shown in Figure 6.5(b). When find($v_1$) arrives at $v_2$, it finds that the last two elements of the path vector are the same, but the third last element is not $v_2$. Hence $v_4$ should exist on the path vector, as shown in Figure 6.5(c).

### 6.2.1 Correctness

In this section, we prove the correctness of Relay. We first introduce the concept of transaction histories before we present the correctness criterion.

**Transaction histories.** A transaction history is the sequence of all events issued and received by transactions in a given TM execution, ordered by the time they are issued. In a transaction history, retrying an aborted transaction is interpreted as creating a new transaction with a new id. Hence, a transaction history describes a given TM computation
by ordering the sequence of all its events. A history \( H \) is *well-formed* if no transaction both commits and aborts, and no transaction takes any step after it commits or aborts. Two histories \( H_1 \) and \( H_2 \) are *equivalent* if they contain the same transaction events in the same order. Formally, let \( H \mid T_i \) denote the longest subsequence of history \( H \) that contains only events issued/received by transaction \( T_i \). Then histories \( H_1 \) and \( H_2 \) are equivalent if for any transaction \( T_i \in T \), \( H_1 \mid T_i = H_2 \mid T_i \). A history is *complete* if it does not contain “live” transactions, i.e., the status of each transaction is either committed or aborted. If a history \( H \) is not complete, we can build a well-formed complete history \( \text{Complete}(H) \) by aborting the live transactions in \( H \). Specifically, we can obtain \( \text{Complete}(H) \) by adding a number of abort events for live transactions in \( H \). We define \( \text{committed}(H) \) to be the subsequence of \( H \) consisting of all events of committed transactions. We assume that all histories are well-formed.

The real-time order of transactions is defined as follows: for any two transactions \( \{T_i, T_j\} \in H \), if the first event of \( T_j \) is issued after the last event of \( T_i \) (a commit event or an abort event of \( T_i \)), then we denote \( T_i \prec_H T_j \). In other words, relation \( \prec_H \) represents a partial order on transactions in \( H \). Transactions \( T_i \) and \( T_j \) are *concurrent* if the transaction events of \( T_i \) and \( T_j \) are interleaved. A history \( H \) is *sequential* if no two transactions in \( H \) are concurrent [41]. A sequential history \( H \) is *legal* if it respects the sequential specification ([92, 93]) of each object accessed in \( H \). Intuitively, a sequential history is legal if every read operation returns the value given as an argument to the latest preceding write operation that belongs to a committed transaction. For a sequential history \( H \), a transaction \( T_i \in H \) is *legal* in \( H \) if the largest subsequence \( H' \) of \( H \) is a legal history, where for every legal transaction \( T_k \in H' \), either 1) \( k = i \), or 2) \( T_k \) has committed and \( T_k \prec_H T_i \).

**Correctness criterion.** We adopt the *opacity* correctness criterion proposed by Guerraoui and Kapalka [41], which defines the class of histories that are acceptable for any TM system. Specifically, a history \( H \) is opaque if there exists a sequential history \( S \), such that:

- \( S \) is equivalent to \( \text{Complete}(H) \);
- \( S \) preserves the real-time order of \( H \); and
- Every transaction \( T_i \in S \) is legal in \( S \).

Hence, to prove the correctness of RELAY, we just need to prove that all histories produced by the D-STM system (which employs RELAY) are opaque.

**Correctness proof.**

**Theorem 22.** Any history \( H \) generated by a D-STM system that employs RELAY is opaque.

**Proof.** Let \( H \) be a history over transactions \( T \) generated by the D-STM system that employs RELAY. Note that in \( H \), retrying an aborted transaction is interpreted as creating a new transaction with a new id. Let \( H_C = \text{Complete}(H) \); i.e., \( H_C \) aborts every live (neither aborted nor committed) transaction in \( H \).

Now, we sort the transactions in \( H_C \) in the order that they are terminated (aborted or
committed). Let \( \{T_{i1}, T_{i2}, \ldots, T_{im}\} \) be the sorted order of transactions, assuming that there are \( m \) transactions in \( HC \). Let \( S \) be the sequential history \( \{T_{i1}, T_{i2}, \ldots, T_{im}\} \). Clearly, \( HC \) is equivalent to \( S \) since for every transaction \( T_i \in H \), \( HC|T_i = S|T_i \).

We now prove that every \( T_i \in S \) is legal. Assume by contradiction that there are non-legal transactions in \( S \). Let \( T_j \) be the first such transaction. If \( T_j \) is non-legal, \( T_j \) reads a value of object \( o \) that is not the latest value written to \( o \) in \( S \) by a committed transaction (note that in RELAY, only a value written by a committed transaction can be read). Assume that the value read by \( T_j \) is \( o.v_1 \), and value \( o.v_1 \) is written by transaction \( T_k \). Hence, \( T_k \) must be ordered before \( T_j \) in \( S \) (since \( T_k \) has committed when \( T_j \) reads a value of object \( o \)). If the value \( o.v_1 \) written by \( T_k \) is not the value written to \( o \) by the latest transaction in \( S \) before \( T_j \) reads \( o \), then there exists another committed transaction \( T_{k'} \) that writes to \( o \) and is ordered between \( T_k \) and \( T_j \) in \( S \). Hence, we know that \( T_{k'} \) writes to object \( o \) after \( T_j \) reads it and commits before \( T_j \) commits or aborts. Hence, \( T_j \) and \( T_{k'} \) conflicts at object \( o \).

1. Case 1: \( T_j \prec T_{k'} \), then \( T_{k'} \) is aborted.
2. Case 2: \( T_{k'} \prec T_j \), then \( T_{k'} \) is postponed to let \( T_j \) commit.

In either case, a contradiction forms: \( T_{k'} \) is either aborted or committed after \( T_j \) commits or aborts. Hence, every \( T_i \in S \) is legal.

The real-time order of \( H \) is trivially preserved in \( S \) according to our sorting method. If \( T_j \prec_H T_k \), then \( T_j \) is terminated before \( T_k \) in \( H \). Therefore \( T_j \) is ordered before \( T_k \) in \( S \).

Summing up, \( Complete(H) \) is equivalent to a legal sequential history \( S \), and \( S \) preserves the real-time order of \( H \). The theorem follows.

### 6.3 Static Competitive ratio \( CR_i(\text{RELAY}) \)

We now focus on the performance of RELAY for a fixed set of transactions, which we measure through static competitive ratio of its makespan.

We can directly derive the following relationships from the protocol description:

\[
\max\text{-str}_\text{RELAY}^\delta = \max\text{-str}_\text{RELAY}^\zeta = \max\text{-str}(ST),
\]

since the object is located and moved via a unique path on \( ST \).

To illustrate the advantage of RELAY in reducing the number of aborts, we have the following theorem:

**Theorem 23.** \( \max_{ij \in V_T} \lambda_{\text{RELAY}}(j) \leq N_i, \quad A_{\text{RELAY}} \leq 2N_i - 1 \)

**Proof.** The first part of the theorem can be proved following the same way as that of Theorem 21. To prove the second part, we first order the set of transactions in the priority
order such that \( \{T_1 < T_2 < \ldots < T_{N_i}\} \). Suppose a transaction \( T_v \) is aborted by another transaction. In this case, \( T_v \) is restarted immediately and a find message is sent to its predecessor on the queue. Finally, a node \( w \) keeps a variable \( next(w) = v \). In other words, each time that a node is aborted, a successor link \( next \) between two nodes is established. Now, assume that the next abort occurs and a successor link \( next(w') = v' \) is established. If \( T_w < \{T_w \text{ or } T_v\} < T_v \), we say that these two links are joint; otherwise, we say that they are disjoint. We can prove that, if \( next(w) \) and \( next(w') \) are joint, at least one transaction in \( \{T_w, \ldots, T_v\} \) has committed. Hence, there are only two outcomes for an abort: at least one transaction commits or a successor link disjoint to other successor links is established. Hence, we just need at most \( N_i - 1 \) abortions to let \( N_i \) transactions commit or establish a chain of links among all transactions (since they are disjoint). For the latter case, no more aborts will occur since the object is moved following that chain. The theorem follows.

From Equation 6.1 and Theorem 23, we have the following corollary:

**Corollary 4.**

\[
CR_i(\text{Relay}) \leq \max\{N_i, 2\max-str(\text{ST}) \cdot Diam\}
\]

Thus, Relay improves the static competitive ratio by reducing the number of total transaction aborts.

### 6.4 Dynamic Analysis

So far we have focused on the static analysis of Relay, which evaluates the overall performance of the protocol to execute a set of transactions, given that no new transaction joins the system during the execution. Obviously, this analysis method is not suitable for all transaction inputs. For example, when different nodes keep generating new transactions, we need to find appropriate cost metrics to measure the communication cost of the set of dynamically generated transactions. In this section, we conduct a dynamic analysis of Relay.

#### 6.4.1 Cost Measures

**Cost of Relay.** We first focus on the cost of an individual object \( o_i \). As shown in the description of Relay (Section 6.2), each transaction locates the object via the direct path in the spanning tree in the same way as Arrow. On the other hand, the object is moved along the direct path on the spanning tree because the path vector is correctly updated. The locating cost and moving cost of Relay are:

\[
\delta^C(T_j, T_k) = d_{\text{ST}}(v_j, v_k)
\]
and

\[ \zeta^C(T_j, T_k) = d_{ST}(v_j, v_k). \]

We have the following theorem:

**Theorem 24.** Assume \( v_{i,j}^\uparrow(m) \) (or \( v_{i,j}^\downarrow(m) \)) is \( T_j \)'s \( m \)th destination (or source) for locating (or moving) the object \( o_i \). The total communication cost of transaction \( T_j \) to commit with respect to object \( o_i \) under Relay is:

\[
\text{cost}_R(T_j) \leq \sum_{m=1}^{\lambda_i(j)} [d_{ST}(v_j, v_{i,j}^\uparrow(m)) + \text{dist}_{ST}(v_{i,j}^\uparrow(m), v_{i,j}^\downarrow(m)) + d_{ST}(v_j, v_{i,j}^\downarrow(m)) + \tau_j], \quad (6.2)
\]

where \( \text{dist}_{ST}(v_{i,j}^\uparrow(m), v_{i,j}^\downarrow(m)) \) is the total communication cost for the \( m \)th find message from \( v_j \) to travel along a certain path from \( v_{i,j}^\uparrow(m) \) to \( v_{i,j}^\downarrow(m) \) in the spanning tree \( ST \), including the idle time that the find message waits for other transactions’ commit.

**Proof.** Each time \( T_j \) sends a find message, it waits until the object has arrived. The \( m \)th find message first arrives at \( v_{i,j}^\uparrow(m) \) and the corresponding locating cost is \( d_{ST}(v_j, v_{i,j}^\uparrow(m)) \). Since the find message may be forwarded to other nodes, we have to take into account such costs. The path from \( v_{i,j}^\uparrow(m) \) to \( v_{i,j}^\downarrow(m) \) is not necessarily the shortest path on the spanning tree since some nodes may be visited multiple times. The idle time is the total time that the find message waits on \( v_{i,j}^\downarrow(m) \) for its transaction’s commit. Finally, the object stays at \( T_j \) for at most \( \tau_j \) time before \( T_j \) aborts or commits. The theorem follows. \( \square \)

---

**Figure 6.6:** The complete execution of \( T_j \) with respect to \( o_i \): \( T_j \) is aborted by the transaction on \( v_{i,j}^\uparrow(m) \), \( m \geq 2 \).
Note that Equation 6.2 gives the total communication cost of a single transaction \( T_j \). From another point of view, an object can start moving in the network and can later get accessed by transactions once it receives the first transaction request. The total time complexity is composed of the time that the object travels and the time that the object is accessed by transactions. Hence, a more useful cost measure is the amortized cost of a single transaction, i.e., the contribution made by a single transaction to the total cost of a set of transactions. We have the following theorem:

**Theorem 25.** Let the amortized cost of a transaction \( T_j \) with respect to \( o_i \) under Relay be denoted as \( c^j_R(T_j) \). Then,

\[
c^j_R(T_j) \leq \sum_{m=1}^{\lambda_j} [d_{ST}(v_j, v_{i,j}^\ast(m)) + \tau_j].
\]  

(6.3)

In other words, the amortized cost of a transaction \( T_j \) is at most the sum of the total moving cost, and the total local execution cost of \( T_i \).

**Proof.** The total cost of a set of transactions for accessing \( o_i \) is the sum of \( o_i \)'s traveling distance in the network and the local execution cost of transactions which require accesses to \( o_i \). From Figure 6.6, we can see that for the \( m \)th find message, such traveling cost is \( d_{ST}(v_j, v_{i,j}^\ast(m)) \) and the local execution cost is at most \( \tau_j \). We now prove that all locating costs and \( \text{dist}_{ST}(v_{i,j}^\ast(m), v_{i,j}^\ast(m)) \) are covered by other transactions’ amortized cost. When \( m \geq 2 \), the find message is sent immediately after the object is moved from \( T_j \). Hence, such locating cost is covered by the moving cost from \( v_j \) to \( v_{i,j}^\ast(m) \) and the execution cost for the transaction on \( v_{i,j}^\ast(m) \). For \( m \geq 1 \), when \( v_{i,j}^\ast(m) \) forwards the find message to \( v_{i,j}^\ast(m) \), the cost of this distance is covered by the local execution cost and the moving cost for the set of transactions on \( \{v_{i,j}^\ast(m), \text{next}(v_{i,j}^\ast(m)), \text{next}(\text{next}(v_{i,j}^\ast(m))), \ldots, v_{i,j}^\ast(m)\} \). Such cost also covers the idle time (if any) that the \( m \)th find message waits on \( v_{i,j}^\ast(m) \), since the object is moved to \( v_j \) immediately when it is available on \( v_{i,j}^\ast(m) \). The theorem follows. \( \square \)

**Transaction Decomposition.** We now decompose each transaction into a set of sub-transactions, i.e., each retry of a transaction is equivalent to an invocation of a sub-transaction. Specifically, we have \( T_j = \{T_j(1), T_j(2), \ldots, T_j(\lambda_j)\} \), where \( T_j \in T^i \). The only different field between tuples \((v_j(l), t_j(l), R(j, l), \tau_j(l))\) and \((v_j, t_j, R(j), \tau_j)\) is that \( t_j(l) \) is the \( l \)th time that \( T_j \) retries, i.e., the time that \( T_j \) retries after the \((l - 1)\)th abort.

We index all sub-transactions \( S^i \) = \( \{S_0 = (v_0, t_0, o(0), \tau_0), S_1 = (v_1, t_1, o(1), \tau_1), \ldots\} \), where \( S_j \in T^i \), in increasing order with respect to \( t_j \), with ties broken arbitrarily, i.e., \( j < k \to t_j < t_k \). For Relay, let \( \phi_R \) be the order of obtaining the object by sub-transaction \( S^i \) which is induced by Relay, i.e., \( \phi_R(j) \) denotes the index of the \( j \)th sub-transaction that receives the object in Relay’s order. We use \( S_0 = (\text{root}, 0) \) to represent the “virtual” transaction (token) at the initial location of the object \( o_i \). Hence we have \( S_{\phi_R(0)} = S_0 \).
Proof. The cost \( d_{ST}(v_j, v_{i,j}^x(m)) \) is the moving cost from \( v_{i,j}^x(m) \) to \( v_j \). Since the object is moved along this path, we know that \( v_{i,j}^x(m) \) receives the object just before \( v_j \). From the definition of transaction decomposition, the theorem follows.

Each sub-transaction \( S_j \) locates the object just once. For brevity, let \( d_{ST}(v_j, v_{i,j}^x), \text{dist}_{ST}(v_{i,j}^x, v_{i,j}), \text{dist}_{ST}(v_j, v_{i,j}^x) \) and \( d_{ST}(v_j, v_{i,j}) \) be denoted as \( d_i^x(j), \text{dist}_i(j) \) and \( d_i(j) \), respectively.

Thus, the total cost of \( \text{RELAY} \) for accessing \( o_i \) is given by:

\[
\text{cost}_{\text{RELAY}}^i = \sum_{j=1}^{N_i} c_R^i(T_j) = \sum_{j=1}^{N_i} [d_{ST}(v_j, v_{i,j}^x(1)) + \lambda_i(j)\tau_j] + \sum_{k=1}^{|S_i^j|} c_R(S_{\phi_R(k-1)}, S_{\phi_R(k)}) \tag{6.5}
\]

Cost of \( \text{Opt} \). We now consider the cost of an optimal clairvoyant offline ordering algorithm \( \text{Opt} \) for a set of dynamically generated transactions. Clearly, an optimal offline algorithm just has to order each transaction to receive the object once to commit. Let \( \phi_O \) be the order of \( \text{Opt} \). For the cost of \( \text{Opt} \), we have to take into account its complete knowledge of all transactions. For a transaction \( T_j = ((v_j, t_j, \vec{R}(j), \tau_j)) \), the algorithm \( \text{Opt} \) already knows the succeeding transaction \( T_k = ((v_k, t_k, \vec{R}(k), \tau_k)) \). When the object is available at \( v_j \), the algorithm can immediately send the object to \( v_k \). Hence, we define the transaction \( T_j \)'s completion time in the order \( \phi_O \) as \( t_j^O \). We therefore define the moving cost \( c_O^i(T_j, T_k) \) of ordering \( T_k \) after \( T_j \) in the \( \phi_O \) order as:

\[
c_O^i(T_j, T_k) := d_{ST}(v_j, v_k) + \max\{0, t_j^O - t_k + d_{ST}(v_j, v_k)\} + \tau_k \geq d_{ST}(v_j, v_k) + \max\{0, t_j - t_k + d_{ST}(v_j, v_k)\} + \tau_k.
\]

The total cost of an optimal algorithm for accessing \( o_i \) then becomes:

\[
\text{cost}_{\text{Opt}}^i = \min_{\phi} \left\{ \sum_{j=1}^{N_i} c_O^i(T_{\phi_O(j-1)}, T_{\phi_O(j)}) \right\} \tag{6.6}
\]

Hence, \( \phi_O \) is an order which minimizes the sum of Equation 6.6.
6.4.2 Dynamic Analysis of RELAY

We now focus on the analysis of the order \( \phi_R \) produced by RELAY. As suggested in [91], the order produced by ARROW corresponding to a nearest neighbor traveling salesman path (TSP) on the set of requests by defining a new comparable cost metric. Motivated by this method, we first define a new cost metric \( c_T \). Then, we show that the cost of ordering all sub-transactions in \( \phi_R \) with respect to \( c_T \) is comparable to the cost \( c^i_{R\text{elay}} \).

**Definition 28.** Let \( S_j \) and \( S_k \) be two sub-transactions such that RELAY orders \( S_j \) before \( S_k \), i.e., \( \phi_R(S_j) < \phi_R(S_k) \). Then the cost metric \( c_T^i(S_j, S_k) \) is defined as:

\[
c_T^i(S_j, S_k) := t_k + d_i^\phi(k) + \text{dist}(k) - t_j - d_i^\phi(j) - \text{dist}(j)
\]

We have the following theorem:

**Theorem 27.** The order of \( \phi_R \) is defined by a nearest neighbor TSP path on the metric \( c_T^i(S_j, S_k) \), starting with the sub-transaction \( S_0 \). Further, \( c_T(S_j, S_k) \geq 0 \) for all pairs of transactions \( S_j \) and \( S_k \).

*Proof.* We prove Theorem 27 by induction. The object is initialized at \( S_0 \). For this dummy token, \( t_0 = d_i^\phi(0) = \text{dist}(0) = 0 \). The sub-transaction \( S_j \) which minimizes \( t_j + d_{ST}(v_j, v_0) \) arrives at \( v_0 \) first. By the definition of \( \phi_R \), this is the sub-transaction \( S_{\phi_R(1)} \). In this case, \( d_i^\phi(j) = d_{ST}(v_j, v_0) \) and \( \text{dist}(j) = 0 \). The sub-transaction \( T_j \) is the one that minimizes \( c_T^i(S_0, S_k) \) for all \( S_k \in S \setminus \{S_0\} \). Clearly, \( c_T^i(S_0, S_{\phi_R(1)}) \geq 0 \).

Assume \( S_{\phi_R(k')} \) is the sub-transaction that minimizes \( c_T^i(S_{\phi_R(k' - 1)}, S_l) \) for all \( S_l \in \{S_{\phi_R(k')}, S_{\phi_R(k' + 1)}, \ldots\} \). From the definition of \( \phi_R \), we know that \( S_{\phi_R(k' + 1)} \) will receive the object from \( S_{\phi_R(k')} \). Note that at time \( t_{k'} + d_i^\phi(k') + \text{dist}(k') \), the object is moved from \( S_{\phi_R(k' - 1)} \) to \( S_{\phi_R(k')} \). From this time point, all newly generated find messages are forwarded to \( S_{\phi_R(k')} \). Hence, the sub-transaction that minimizes \( c_T^i(S_{\phi_R(k')}, S_{\phi_R(l')}) \) for all sub-transactions \( S_l \in \{S_{\phi_R(k' + 1)}, S_{\phi_R(k' + 2)}, \ldots\} \) is \( S_{\phi_R(k' + 1)} \), which is the first sub-transaction that was ordered after \( S_{\phi_R(k')} \).

Note that \( c_T^i(S_{\phi_R(k' - 1)}, S_{\phi_R(k')}) \leq c_T^i(S_{\phi_R(k' - 1)}, S_{\phi_R(k)} \}) \). Then:

\[
0 \leq c_T^i(S_{\phi_R(k' - 1)}, S_{\phi_R(k + 1)}) - c_T^i(S_{\phi_R(k' - 1)}, S_{\phi_R(k')})) = t_{k' + 1} + d_i^\phi(k' + 1) + \text{dist}(k' + 1) - t_{k' - 1} - d_i^\phi(k' - 1) - \text{dist}(k' - 1)
\]

The theorem follows.

Let \( C_T^i \) be the cost of ordering all sub-transactions in \( \phi_R \) with respect to \( c_T^i \). We have the following theorem.
Theorem 28.

\[ C^k_T \geq \sum_{k=1}^{\lvert S^i \rvert} c_R(S_{\phi_R(k-1)}, S_{\phi_R(k)}) - D, \]

where \( D \) is the diameter of the spanning tree \( ST \).

Proof. We first show that:

\[ c^k_T(S_{\phi_R(k-1)}, S_{\phi_R(k)}) \geq c_R(S_{\phi_R(k-2)}, S_{\phi_R(k-1)}) \]

where \( k \geq 2 \). Note that

\[ c_R(S_{\phi_R(k-2)}, S_{\phi_R(1)}) = d_{ST}(v_{\phi_R(k-2)}, v_{\phi_R(k-1)}) \]

by definition. Since

\[ c^k_T(S_{\phi_R(k-1)}, S_{\phi_R(k)}) = t_k + d^\phi_i(k) + \text{dist}_i(k) - t_{k-1} - d^\phi_i(k - 1) - \text{dist}_i(k - 1), \]

note that the object arrives at \( S_{k-1} \) at time \( t_{k-1} + d^\phi_i(k-1) + \text{dist}_i(k-1) + d_{ST}(v_{\phi_R(k-2)}, v_{\phi_R(k-1)}) \).

Hence, the fastest way for \( S_{\phi_R(k)} \) to get the object is by moving the object to \( S_{\phi_R(k)} \) once it arrives at \( S_{\phi_R(k-1)} \), i.e., \( S_{\phi_R(k)} \) aborts \( S_{\phi_R(k-1)} \). In this case,

\[ t_k + d^\phi_i(k) + \text{dist}_i(k) = t_{k-1} + d^\phi_i(k - 1) + \text{dist}_i(k - 1) + d_{ST}(v_{\phi_R(k-2)}, v_{\phi_R(k-1)}), \]

which is the minimum. Equation 6.7 follows.

By summing up over \( k \), we have:

\[ C^k_T \geq \sum_{k=1}^{\lvert S^i \rvert} c_R(S_{\phi_R(k-1)}, S_{\phi_R(k)}) + t_{\phi_R(1)} + d_{ST}(v_{\phi_R(1)}, v_0) - d_{ST}(v_{\phi_R(\lvert S^i \rvert - 1)}, v_{\phi_R(\lvert S^i \rvert)}) \]

which completes the proof.

Relay and an optimal offline algorithm produce the same ordering when the transactions are sparse enough, i.e., in a relatively long time period, there is only one transaction that is invoked. We can shift the sub-transactions as much as possible without increasing the cost of Relay and an optimal offline algorithm.

Lemma 9. Let \( S_{\phi_R(k)} \) and \( S_{\phi_R(k+1)} \) be two consecutive sub-transactions in the order \( \phi_R \). Let \( \epsilon := c_T(S_{\phi_R(k)}, S_{\phi_R(k+1)}) - d_{ST}(v_{\phi_R(k-1)}, v_{\phi_R(k)}) - \tau_k \).

If \( \epsilon > 0 \), for all sub-transactions \( S_{\phi_R(l)} \) where \( l \geq k + 1 \), \( t_{\phi_R(l)} \) can be replaced by \( t_{\phi_R(l)} - \epsilon \) without increasing the cost of Relay and Opt.
Proof. The proof follows the same argument as that of the proof of Lemma 2.6 in [91].

We have the following theorem:

**Theorem 29.** The upper bound of the cost $c^i_T(S_j, S_k)$ of the longest edge on Relay’s path is:

$$c^i_T(S_j, S_k) \leq D + \max_{l=1}^{\lceil |S| \rceil} \tau_l.$$  

**Proof.** The theorem follows by applying Lemma 9 as many times as possible.

### 6.4.3 Dynamic Competitive Ratio $\rho^i$ of Relay

We first define the Manhattan metric $c_M$ which is comparable to $c^i_O$.

**Definition 29 (Manhattan Metric $c_M(T_j, T_k)$).** The Manhattan metric $c_M(T_j, T_k)$ is defined as:

$$c_M(T_j, T_k) := \begin{cases} d_{ST}(v_j, v_k) + |t_j - t_k| + \tau_j + \tau_k & j \neq k \\ 0 & j = k \end{cases}$$

**Lemma 10.** Let $\phi$ be an ordering, and $C_O^i$ and $C_M$ be the costs for ordering all transactions in the order $\phi$ with respect to $c^i_O$ and $c_M$, respectively. The Manhattan cost is bounded by:

$$C_M \leq 2C_O^i + t_{\phi(\lceil T^i \rceil)}.$$

**Proof.** We can lower bound the optimal cost of $C_O^i$ by:

$$C_O^i(T_j, T_k) \geq d_{ST}(v_j, v_k) + \max\{0, t_j - t_k\} + \tau_k$$

Let $D_{ST} = \sum_{j=1}^{N_i} \{d_{ST}(v_{\phi(j-1)}, v_j) + \tau_j + \tau_{j-1}\}$. Then we have:

$$2C_O^i \geq D_{ST} + 2 \sum_{j=1}^{N_i} \max\{0, t_{\phi(j-1)} - t_j\} = D_{ST} + \sum_{j=1}^{N_i} [0, t_{\phi(j-1)} - t_j] - t_{\phi(N_i)} = C_M - t_{\phi(N_i)}$$

The lemma follows.

We use the following lemma from [91]:

**Lemma 11.** Let $c_M'(T_j, T_k) := d_{ST}(v_j, v_k) + |t_j - t_k|$ and $C_M'$ be the cost of ordering all requests in the order $\phi$ with respect to $c_M'$. Then, $C_M' \geq \frac{3}{2}t_{N_i}$, where $t_{N_i}$ is the largest time of any request in $T^i$.

Hence, we have the following theorem to make $C_M$ comparable to $C_O^i$:
**Theorem 30.**
\[ C_M \leq 6C_O^i \]

**Proof.** The theorem can be proved by Lemmas 10 and 11. Note that we have \( c_M \geq c_{M'} \) and \( t_{N_i} \geq t_{\phi(T_i)} \). Then the theorem follows. \( \square \)

We now compare \( C_M \) and \( C_T^i \) with the help of the following theorem from [91]:

**Theorem 31.** Let \( V \) be a set of \( N := |V| \), and let \( d_n : V \times V \to \mathbb{R} \) and \( d_o : V \times V \to \mathbb{R} \) be the distance functions between nodes of \( V \). For \( d_n \) and \( d_o \), the following conditions hold:

\[
d_o(u,v) = d_o(v,u), \quad d_o(u,w) \leq d_o(u,v) + d_o(v,w)
\]

\[
d_o(u,v) \geq d_n(u,v) \geq 0, \quad d_o(u,u) = 0
\]

Let \( C_N \) be the length of a nearest neighbor TSP tour with respect to the distance function \( d_n \) and let \( C_O \) be the length of an optimal TSP tour with respect to the distance function \( d_o \). Then,

\[
C_N \leq \frac{3}{2} \left[ \log_2(D_{NN}/d_{NN}) \right] \cdot C_O
\]

holds, where \( D_{NN} \) and \( d_{NN} \) are the lengths of the longest and the shortest non-zero edge on the nearest neighbor tour with respect to \( d_n \), respectively.

Now we have the following theorem:

**Theorem 32.**
\[
C_T^i \leq \frac{3}{2} \left[ \log_2(D_0 + \frac{\max_{j=1}^{N_i} \tau_j}{\min_{v_j,v_k \in V} d(v_j,v_k)}) \right] (C_M + \sum_{k=1}^{N_i} \lambda_i(k)\tau_k)
\]

**Proof.** To prove this theorem, we need to find two metrics corresponding to \( d_n \) and \( d_o \) for the same set \( |V| \). Note that \( c_T^i \) is defined on set \( S^i \) and \( c_M \) is defined on set \( T^i \). Hence, we define the Manhattan metric \( c_M^* \) that maps \( c_M \) to set \( S^i \) as follows:

**Definition 30** (Manhattan Metric \( c_M^*(S_j,S_k) \)). The Manhattan metric \( c_M^*(S_j,S_k) \) is defined as:

\[
c_M^*(S_j,S_k) := \begin{cases} 
  c_M(T_{j'},T_{k'}) & S_j \text{ and } S_k \text{ are mapped to } T_{j'} \text{ and } T_{k'}, \ j' \neq k', \text{ respectively} \\
  |l_k - l_j|\tau_l & S_j \text{ and } S_k \text{ are mapped to } T_i(l_j) \text{ and } T_i(l_k) \text{ for } T_i \in T^i, \text{ respectively}
\end{cases}
\]

Note that \( C_M \) is the cost for ordering all transactions in the order \( \phi \) on set \( T^i \) with respect to \( c_M \). Given the order \( \phi \) on \( T^i \), we define its dual order \( \phi^* \) as:

\[
(T_{\phi_{j-1}}, T_{\phi_j}) := (S_{\phi_{j-1}}(1), S_{\phi_{j-1}}(2), \ldots, S_{\phi_{j-1}}(\lambda_i(\phi_{j-1})), S_{\phi_j}(1))
\]
In other words, if we order a transaction \( T_j \) before transaction \( T_k \) in \( \phi \), then, it is equivalent to order all sub-transactions mapped to \( T_j \) (in the order of their invocations) before the first invoked sub-transaction of \( T_k \). Let \( C^*_M \) be the cost for ordering all sub-transactions in the order \( \phi^* \) with respect to \( c^*_M \). Then we have:

\[
C^*_M = C_M + \sum_{k=1}^{N_i} \lambda_i(k)\tau_k
\]

Now \( c^*_T \) and \( c^*_M \) comply with the conditions for \( d_n(u,v) \) and \( d_o(u,v) \), respectively. By Lemma 9, we have \( c^*_T(S_j, S_k) \leq c^*_M(S_j, S_k) \). And the triangle inequality holds for \( c^*_M \). Finally, we can bound the shortest value of \( c^*_T \) by \( \min_{v_j, v_k \in V} d(v_j, v_k) \). The theorem follows.

**Theorem 33.**

\[
\rho^i = \frac{\text{cost}^i_{\text{RELAY}}}{\text{cost}^i_{\text{OPT}}} = O\left( \max[\log(D_0 + \frac{\max_{j=1}^{N_i} \tau_j}{\min_{v_j, v_k \in V} d(v_j, v_k)\lambda_i(j)\tau_j})], \frac{N_i \max_{j=1}^{N_i} \tau_j}{H^i_T} \right)
\]

where \( H^i_T \) is the total cost of the TSP path for \( T^i \) with respect to metric \( d_{ST}(v_j, v_k) \).

**Proof.** From Equation 6.5 and Theorems 28, 30 and 32, we have:

\[
\text{cost}^i_{\text{RELAY}} \leq 9 \left[ \log_2(D_0 + \frac{\max_{j=1}^{N_i} \tau_j}{\min_{v_j, v_k \in V} d(v_j, v_k)}) \right] \text{cost}^i_{\text{OPT}} + D + \sum_{j=1}^{N_i} \lambda_i(j)\tau_j.
\]

Since

\[
\text{cost}^i_{\text{OPT}} = \sum_{j=1}^{N_i} \left( d_{ST}(v_{\phi_O(j-1)}, v_{\phi_O(j)}) + \max\{0, t_{\phi_O(j)} - t_{\phi_O(j)} + d_{ST}(v_{\phi_O(j-1)}, v_{\phi_O(j)})\} + \tau_{\phi_O(j)} \right)
\]

\[
\leq H^i_{ST} + \sum_{j=1}^{N_i} \tau_j,
\]

the theorem follows.

From Theorem 33, we know that \( \rho^i \) is determined by the value of the maximum \( \tau_j \). We have the following theorem for a possible range of the values of the maximum \( \tau_j \).

**Theorem 34.**

\[
\rho^i = O(\log D_0)
\]

if

\[
\max_{j=1}^{N_i} \tau_j = O(\log D).
\]

**Proof.** The theorem follows directly from Theorem 33. In other words, if the maximum local execution time of a set of transactions \( T^i \) is sufficiently small (up to the logarithmic order of the diameter of the spanning tree), the dynamic competitive ratio \( \rho^i \) is \( O(\log D_0) \).
6.5 Conclusion

Compared with the traditional distributed queuing problem, the design of a CC protocol for D-STM systems must take into account the contention between two transactions because transaction aborts increase the length of the queue. Motivated by a distributed queuing protocol with excellent performance, i.e., Arrow, we show Arrow’s worst-case number of total aborts is $O(N_i^2)$ for $N_i$ transactions requesting an object. Based on this protocol, we design Relay which reduces the worst-case number of total aborts to $O(N_i)$. Meanwhile, Relay inherits the advantages of Arrow — i.e., the maximum locating stretch and moving stretch are exactly the maximum stretch of the underlying spanning tree. As a result, Relay yields a better static competitive ratio of its makespan.

We conclude that Relay is $O(\log D_0)$-competitive for dynamically generated transactions with sufficiently small, maximum local execution time. Hence, Relay is appropriate for distributed systems, in which the network latency plays the major role in the total time complexity. For the transactions with maximum local execution time, we can use Theorem 33 to analyze the dynamic competitive ratio. When the maximum local execution time of transactions is sufficiently large, i.e., $\Omega(D)$, the execution time will be the dominating part of the total time complexity. In this case, the performance of a D-STM system is not determined by the CC protocol, but by the underlying contention manager, which determines the maximum number of abort times of a single transaction, just like the case for multiprocessors. Relay is designed to support multiple objects. Since the protocol is totally distributed (i.e., all nodes are of the same importance in the protocol), it avoids significantly overloading some nodes in the network.
Chapter 7

Distributed Dependence-Aware Model with Non-Conservative Conflict Resolution Strategy

D-STM model based on globally-consistent contention management policies may abort many transactions that could potentially commit without violating correctness. To reduce unnecessary aborts and increase concurrency, we propose the distributed dependency-aware (DDA) model for D-STM, which adopts different conflict resolution strategies based on the types of transactions. In the DDA model, read-only transactions never abort by keeping a set of versions for each object. Each transaction only keeps precedence relations based on its local knowledge of precedence relations. The DDA model that, when a transaction reads from or writes to an object based on its local knowledge, the underlying precedence graph remains acyclic. We propose starvation-free multi-version (SF-MV)-permissiveness, which ensures that: 1) read-only transactions never abort; and 2) every transaction eventually commits. The DDA model satisfies SF-MV-permissiveness with high probability. We present a set of algorithms to support the DDA model, prove its correctness and permissiveness, and show that it supports invisible reads and efficiently garbage collects useless object versions.

7.1 Motivation

D-STM inherits the globally-consistent contention management strategy (or CM model in short) from multiprocessor STM, for resolving read/write conflicts on shared objects. In the CM model, a contention manager module is responsible for mediating between conflicting accesses to shared objects. For example, GREEDY contention manager assigns priorities based on transactions’ starting timestamps. Each transaction is assigned a unique timestamp when it starts, and it remains unchanged until commit. A running transaction could only
be aborted by another transaction with an older timestamp. Whenever two transactions concurrently need exclusive access to the same shared object, only one of these transactions is allowed to continue, and the other is immediately aborted (if it has already acquired the object needed by the older transaction) or suspended.

Although easy to implement, the CM model sometimes is too conservative in achieving high throughput. Given a specific transactional workload, the CM model may execute all transactions almost entirely sequentially even if a large number of them could run concurrently. For example, consider a workload where \( n \) transactions with duration \( 1 - \delta \) each, share \( n + 1 \) objects in a network, under a CC protocol that ensures that the duration for any transaction to acquire a remote object is \( \delta \). Assume that transaction \( T_i \) requests to write to object \( o_i \) at time 0 and object \( o_{i+1} \) at time \( 1 - \delta - i \cdot \epsilon \). Assume that the Greedy manager is used to resolve conflicts. Let \( T_i \) has the \( i \text{th} \) oldest timestamp. We have \( T_1 \prec_G T_2 \prec_G \ldots \prec_G T_n \), where “\( T_i \prec_G T_j \)” means that \( T_i \)'s priority is higher than \( T_j \)'s under the Greedy manager.

![Figure 7.1: Example 1: The Greedy manager is adopted and transaction \( T_i \) has the \( i \text{th} \) oldest timestamp. \( T_i \) can only commit at its \( i \text{th} \) execution.](image)

We depict this example in Figure 8.3. We follow the style of [94] to depict transaction histories, and extend it to D-STM. Filled circles correspond to write operations and empty circles represent read operations. Transactions are represented as polylines with circles (write or read operations). Each object \( o_i \)'s state in the time domain corresponds to a horizontal line from left to right. A commit or an abort operation is indicated by the letter C or A, respectively. An object moves between transactions which write to it. For example, in Figure 8.3, object \( o_2 \) moves from \( T_2 \) to \( T_1 \) and \( T_1 \) acquires it at \( 1 - \epsilon \). Similarly, \( o_2 \) moves from \( T_1 \) to \( T_2 \) and \( T_2 \) acquires it at \( t_1 + (n-1)\epsilon \).

We assume that initially each object \( o_i \) is located at transaction \( T_i \mod n \). Hence, transaction \( T_i \) starts without acquiring \( o_i \) remotely and writes to \( o_i \) at time 0. At time \( 1 - \delta - i \cdot \epsilon \), \( T_i \) requires a write operation to \( o_i \) and invokes the CC protocol to request the object. Hence,
$T_i$ acquires $o_{i+1}$ at time $1 - i \cdot \epsilon$ for $i \geq 2$, and is aborted by $T_{i-1}$ before $1 - (i - 1)\epsilon$. Only $T_1$ commits in its first execution.

After the first execution of each transaction, object $o_i$ is located at $T_{i-1}$ for $i \geq 2$. Transaction $T_i$ ($i \geq 2$) restarts and requests $o_i$ remotely such that $T_i$ acquires a $\epsilon$ time units before $T_{i-1}$ acquires $o_{i-1}$. Note that transaction $T_{i-1}$ acquires $o_i$ ($i \epsilon$ time units before its termination. Hence, similarly to its first execution, $T_i$ is aborted by $T_{i-1}$ for $i \geq 3$ and only $T_2$ commits in its second execution. By repeating this procedure, $T_{i-1}$ aborts $T_i$ at least $i - 1$ times and $T_i$ commits at its $i^{th}$ execution. The total time duration to commit the set of $n$ transactions is $\Sigma_{i=1}^n (1 - i \cdot \epsilon + \delta) = \Omega(n + n \cdot \delta)$.

Obviously, the schedule produced by the GREEDY manager is not optimal. By first executing all even transactions and then executing all odd transactions, an optimal schedule finishes in constant time $O(1 + \delta)$. Can we find a more efficient conflict resolution strategy to achieve high concurrency? For this specific example, the answer is trivial: all transactions can proceed even when a conflict occurs. Without assigning priorities to transactions, when transaction $T_i$ receives a request from $T_{i-1}$ for object $o_i$, which is currently in use, it simply sends $o_i$ to $T_{i-1}$ with its initial value (the value before $T_i$ writes to it), denoted by $o_i^0$. When $T_i$ commits, it sends a request to $T_{i-1}$ to write its value to $o_i$. In this way, each transaction commits at its first execution. Object $o_i$ ($2 \leq i \leq n$) is first written by $T_{i-1}$ and then by $T_i$. Let the value written to $o_i$ by the $j^{th}$ write be denoted as $o_i^j$. Then we know that $T_{i-1}$ writes $o_i^1$ and $T_i$ writes $o_i^2$. As a result, all transactions can be serialized in the order from $T_1$ to $T_n$ and the time duration to commit all transactions is $O(1 + \delta)$, which is optimal.

The example in Figure 8.3 suggests that the CM model may incur a large amount of unnecessary aborts. On the other hand, instead of aborting a transaction when a conflict occurs, letting conflicting transactions to proceed in parallel can enhance concurrency efficiently as long as the correctness criterion (i.e., opacity) is not violated. This observation motivates us to propose the distributed dependency-aware (DDA) STM model, which differs from past D-STM models in the way that it resolves read/write conflicts over shared objects.

In the DDA model, a write-only transaction commits by writing a new version to each object that it requests. Adding a new version to each object in a greedy way in some cases is the simplest and correct solution (e.g., Example 1). However, this method is problematic as it violates opacity under certain workloads. Consider the dining philosophers problem, which is similar to Example 1, except that $T_n$ writes to object $o_1$ (instead of $o_{n+1}$) at time $1 - \delta - n \cdot \epsilon$, as shown in Figure 8.4. In this scenario, transaction $T_i$ needs to write two new versions to $o_i$ and $o_{(i+1) \ mod \ n}$, respectively. Note that $T_i$ only holds $o_{(i+1) \ mod \ n}$ locally (acquired at time $1 - i \cdot \epsilon$) when it commits. Hence, $T_i$ writes a version to $o_{(i+1) \ mod \ n}$ locally at time 1 and writes to $o_i$ remotely at time $1 + \delta$, where $\delta^- < \delta$. If objects versions are added in a greedy way, then for object $o_i$, $o_i^1$ is written by $T_{(i-1) \ mod \ n}$ and $o_i^2$ is written to by $T_i$. Hence, $T_i$ can only be serialized after $T_{(i-1) \ mod \ n}$, since $T_i$ writes after $T_{(i-1) \ mod \ n}$ over object $o_i$. Obviously, a cycle forms when serializing all transactions, and opacity is violated.

This phenomenon is unique for D-STM, where objects’ versions may be written in an in-
Figure 7.2: Dining philosophers problem: transactions’ write version requests may be inter-leaved.

Assume that two transactions $T_i$ and $T_j$ which both writes to object $o_1$ and $o_2$ try to commit. Each transaction needs to send a message to $o_1$ and $o_2$, respectively, to write its own version. While this step is trivial for multiprocessor STM (the transaction which commits first always writes the versions first), for D-STM, transactions’ requests may arrive at objects’ locations in different order: for $o_1$, $T_i$’s request may precede $T_j$’s and for $o_2$, $T_j$’s request may precede $T_i$’s. Hence, simply adding versions in a greedy way may violate correctness. To guarantee correctness, the DDA model allows a transaction to insert an object version preceding an older version. How does a transaction decide the proper place to insert an object version? One simplest way is to assign priorities to transactions based on starting timestamps and the writer’s priority, as shown in Figure 8.4. A circled letter “W” represents that an object version is inserted. When transaction $T_i$ tries to insert a version (red lines) to object $o_i$, it first checks the priorities of the older versions of $o_i$; if it finds a version $o_i^k$ which is written by a lower-priority transaction (e.g., $T_1$ finds that $o_1$ has a version written by $T_n$ at time $1 + \delta^{-}$), $T_i$ inserts its version preceding $o_i^k$ (the green dotted arc). As a result, the time duration to commit all transactions is $O(1 + \delta)$ and the history is opaque (transactions can be serialized in the priority order). We will study how transactions of different types insert the object versions in detail in Section 8.2.
7.2 Key Techniques

7.2.1 Multi-versioning.

The examples of Figure 8.3 and 8.4 illustrate that the DDA model can avoid unnecessary aborts that stem from the inherent limitation of the CM model, given that a transaction can access multiple versions of each object. Moreover, past D-STM proposals assume that each object only keeps a single version, which may be too conservative and lead to unnecessary aborts. The DDA model allows D-STM to manage multiple versions of shared objects.

Each object \( o \) maintains two object version lists: a pending version list \( o.v_p \) and a committed version list \( o.v_c \) based on the status of a version’s writer. At any given time, the versions of each list is numberer in increasing order, e.g., \( o.v_p[1], o.v_p[2], \ldots \), etc. The data structure of an object version is described in Algorithm 1. An object version \( \text{Version} \) includes the data \( \text{Version}.data \), the writer transaction ID \( \text{Version}.writer \), a set of readers \( \text{Version}.readers \), a set of detected predecessors \( \text{Version}.preSet \), and \( \text{Version}.sucSet \), a set of detected successors writing after \( \text{Version} \). A read operation of object \( o \) returns the value of one of \( o \)'s committed version list. When transaction \( T_i \) accesses \( o \) to write a value \( v(T_i) \), it appends \( v(T_i) \) to the tail of \( o.v_p \) (note that before this operation, \( T_i \) must guarantee that writing to \( o \) does not violate correctness), e.g., \( v(T_i) = o.v_p[\text{max}] \). When \( T_i \) tries to commits, \( v(T_i) \) is removed from \( o.v_p \) and inserted into \( o.v_c \). Each transaction data structure keeps a \( \text{readList} \) and \( \text{writeList} \). An entry in a \( \text{readList} \) points to the version that has been read by the transaction. An entry in a \( \text{writeList} \) points to the version written by the transaction.

**Algorithm 1:** Data structure of an object version \( \text{Version} \)

<table>
<thead>
<tr>
<th>Data: data</th>
<th>// actual data written to the object</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: writer</td>
<td>// transaction ID of the writer</td>
</tr>
<tr>
<td>int: versionNum</td>
<td>// ordered version number</td>
</tr>
<tr>
<td>TxnDsc [[]]: readers</td>
<td>// set of readers</td>
</tr>
<tr>
<td>id [[]]: sucSet</td>
<td>// set of successors detected writing after ( \text{Version} )</td>
</tr>
<tr>
<td>id [[]]: preSet</td>
<td>// set of predecessors detected precedes ( \text{Version} )</td>
</tr>
</tbody>
</table>

7.2.2 Precedence Graph

In dependence-aware D-STM systems, the basic idea to guarantee correctness is to maintain a precedence graph of transactions and keep it acyclic, which has been adopted by some recent STM efforts in multiprocessor systems [41, 22, 23]. Generally, transactions form a directed labeled precedence graph, \( PG \), based on the dependencies created during the transaction history. The vertices of \( PG \) are transactions. There exists a directed edge \( T_i \to T_j \) in \( PG \) due to following cases:

1. Real-time order: \( T_i \) terminates before \( T_j \) starts;
2. Read after Write ($W \rightarrow R$): $T_j$ reads the value written by $T_i$;
3. Write after Read ($R \rightarrow W$): $T_j$ writes to object $o$, while $T_i$ reads the version overwritten by $T_j$; or
4. Write after Write ($W \rightarrow W$): $T_j$ writes to object $o$, which was previously written to by $T_i$.

### 7.2.3 Distributed Commit Protocol

The advantages of the DDA model motivates us to design a framework to support it in D-STM. Unfortunately, past similar approaches for multiprocessor STM systems cannot be directly applied into D-STM. Particularly, a transaction has to first locate (for read/write) and fetch (only for write) the objects before it performs a read/write operation. Since the DDA model allows multiple conflicting transactions to proceed concurrently, when a transaction attempts to commit after a sequence of operations, some objects in its $writeList$ may be already moved to other transactions. Intuitively, for each object in its $writeList$, the transaction commits by finding a proper “place” in the object’s version list to insert the new version without violating correctness. As the result, in D-STM, it is unavoidable for a transaction to insert an object version remotely. In this case, directly employing the idea from multiprocessor STM systems by iteratively traversing the written objects to correctly insert all object versions is too complicated and likely to incur high communication cost.

![Figure 7.3: The commit operation which inserts object versions by traversing each object](image)

For example, consider the scenario depicted in Figure 7.3. At time $t_1$, both $T_1$ and $T_2$ attempt to commit. Note that at $t_1$, both $o_1$ and $o_2$ are moved to $T_2$ for its write operations.
Hence, $T_2$’s commit operation can be done locally. A circle filled with letter W indicates the operation to insert a version to the object’s version list. In this scenario, $T_2$ inserts object versions to $o_1$ and $o_2$ one after another. Note that $T_2$ is the first transaction to insert objects versions. Hence, it simply inserts a new version to each object. After the two versions are inserted, $T_2$ can successfully commit. A circle with letter C indicates that the transaction which inserts the new version can commit. Hence, the new version can be safely read by other transactions.

Algorithm 2: Algorithm INSERTVERSION

\begin{algorithm}
\begin{algorithmic}
  \Procedure{InsertVersion}{o, v($T_i$)} when $T_i$ inserts object version $v(T_i)$ to o
  \State remove $v(T_i)$ from $o.v_p$
  \State insert $v(T_i)$ after $o.v_c[\text{max}]$
  \For{$Version \leftarrow o.v_c[\text{max}] \text{ to } o.v_c[\text{min}]$}
    \State // scan the committed version list of o from the latest one
    \If{$T_i \in Version.preSet$}
      \State remove $v(T_i)$ from $o.v_p$
      \State move $v(T_i)$ before $Version$
      \State break
    \EndIf
  \EndFor
  \State copy $T_i.preSet$ to $v(T_i).preSet$
  \EndProcedure

  \Procedure{UpdatePre}{$T_i, o.v_c$} when $T_i$ writes to object version $o.v_c$
  \For{$Version \leftarrow o.v_c[\text{max}] \text{ to } o.v_c[\text{min}]$}
    \If{$Version.writer \prec H$}
      \ForEach{reader $\in Version.readers$}
        \State add reader to $T_i.preSet$
      \EndForEach
      \State add $T_i$ to $Version.sucSet$
    \EndIf
  \EndFor
  \EndProcedure
\end{algorithmic}
\end{algorithm}

The commit operation performed by $T_2$ follows the commit protocol in [22]. Since all operations are done locally, no communication cost between transactions is involved. On the other hand, when $T_1$ conducts similar operations, such cost is induced, as shown in Figure 7.3. Note that $T_1$ reads $o_2$’s initial value $o_2^0$ and $T_2$ writes to $o_2$. Hence, $T_1$ should be serialized before $T_2$. As the result, the versions written to $o_1$ and $o_2$ by $T_1$ can only be inserted before the versions written by $T_2$, represented by the dotted arc lines. Since $o_1$ and $o_2$ are not located at $T_1$ when $T_1$ tries to commit, $T_1$ can only perform its commit operations remotely. Such operations induce several iterations of communication between $T_1$ and the object holders until the all object versions can be correctly inserted (commit) or not (abort).

The commit operation illustrated in Figure 7.3 requires frequent coordinations between object holders. Furthermore, since a transaction traverses each object one after another, a transaction may need several iterations of traversing to find a proper place for each object without violating correctness, as suggested in [22]. Apparently, such operation introduces large potential communication cost, which makes itself not appropriate for D-STM. Such drawbacks motivates us to design INSERTVERSION algorithm (Algorithm 2), which enables each transaction to insert object version in distributed way and avoid inter-transaction communications, as shown in Figure 7.4. At time $t_1$, $T_2$ learns that it can only serialized after $T_1$ by check the readers of $o_2^0$ (lines 10-15). Hence, $T_2$ can only commit if and only if all the versions written by it are inserted after the versions written by $T_1$ (if any). Since $T_2$
Figure 7.4: The commit operation implemented by **INSERTVERSION** algorithm

inserts its versions before \( T_1 \) does (by default, an new object version is inserted to the end of committed version list, as shown in lines 2-3), the “places” of the versions written by \( T_1 \) are “reserved” at the time \( T_2 \) insert its versions. When \( T_2 \) inserts its versions, it just checks each version list to find its “reserved” places and inserts its own versions (lines 4-9). In this way, no communications between object holders are involved to make each version correctly inserted.

### 7.2.4 Real-time Order Detection

The definition of the real-time order inherits the widely-adopted definition in multiprocessor STM. However, when an update transaction \( T_i \) commits in D-STM, it inserts a new version for each object in its \( writeList \) in a distributed way. As the result, each object in \( T_i \)’s \( writeList \) may “observe” \( T_i \)’s commit at different time points. As the result, other transactions may get different information about \( T_i \)’s commit when accessing different objects. To clarify this, we must first have the clear definition of the transaction termination for D-STM.

**Definition 31** (Transaction termination). In D-STM, a transaction \( T_i \) terminates if and only if: 1) \( T_i \) aborts; or 2) \( T_i \) successfully inserts a new version for each object in its \( writeList \).

When a transaction \( T_i \) accesses an object \( o \) with a version inserted by another transaction \( T_j \), \( T_i \) needs to determine its real-time order with \( T_j \). The only information about the time of
Figure 7.5: \( T_4 \) detects that \( T_1 \prec_H T_4 \) at \( t_2 \). Then at \( t_3 \), \( T_4 \)'s commit is postponed after \( t_4 \).

\( T_j \)'s commit that \( T_i \) can get is the time that \( T_j \)'s version for \( o \) is inserted. Obviously, \( T_i \) may make a wrong decision when it uses this information as \( T_j \)'s terminating time, since \( o \) may not the last object that \( T_j \) inserts a new version. Therefore we present UPDATERT algorithm (Algorithm 3) to let transactions correctly update real-time orders and revise wrong real-time order detections.

Consider the scenario depicted in Figure 7.5. When \( T_1 \) tries to commit, it has to insert new versions to \( o_1 \) and \( o_2 \), which was moved to \( T_2 \) and \( T_3 \), respectively. We omit the insert operations of \( T_2 \) and \( T_3 \) since they are done locally. As the result, when \( T_1 \) successfully inserts a new version to \( o_1 \) at \( t_1 \), \( o_2 \) is still waiting for \( T_1 \) to insert its new version, which will be done at \( t_4 \). When transaction \( T_4 \) starts at \( t_2 \), it first accesses \( o_1 \) to read a value. By comparing its starting time and the insertion time of \( o_1 \), \( T_4 \) wrongly detect that \( T_1 \prec_H T_4 \), although in fact they are concurrent transactions since \( T_1 \) terminates at \( t_4 \). When \( T_4 \) tries to insert a version to \( o_2 \) at \( t_3 \), it can only insert a version after the version written by \( T_1 \). Furthermore, since \( T_4 \) establishes a real-time order between \( T_1 \) and itself, it has to postpone its termination until \( T_1 \) commits to comply with the real-time order it detects.

The example of Figure 7.5 illustrates that when a transaction makes a wrong decision about the real-time order, its execution should comply with the real-time order to avoid unnecessary abort. Moreover, other transactions’ execution should also accommodate the established (although wrong) real-time order. Consider, for example, the scenario depicted in Figure 7.6. When \( T_4 \) commits at \( t_4 \), it wrongly detects that \( T_1 \prec_H T_4 \) and inserts the version \( o_1^3 \) to \( o_1 \). When \( T_5 \) starts at \( t_3 \), it detects that \( T_3 \prec_H T_5 \) and reads \( o_1^1 \). As the result, \( T_5 \) establishes a \( R \rightarrow W \) order with \( T_1 \). When \( T_5 \) accesses \( o_1 \) to read a value, it detects that \( T_1 \prec_H T_4 \prec_H T_5 \).
Figure 7.6: $T_5$ detects that $T_1 \not\prec_H T_5$ at $t_3$. Then at $t_3$, $T_5$ cannot read the version written by $T_4$.

$T_5$. Now the contradiction forms since $T_5$ already knows that $T_1$ is concurrent with itself. Therefore, $T_5$ knows that $T_4$ makes a wrong detection. The solution is that $T_4$ “postpones” its termination until $T_5$ commits. As the result, the real-time order $T_4 \prec_H T_5$ is not held and $T_5$ can read the value $o_1$.

**Algorithm 3:** UPDATERT($o$) algorithm for $T_i$ to update real-time order when accesses $o$

```c
foreach Version $\in o.v_c$ do
    if Version.writer $\notin T_i.rtPre$ then
        // for each committed version inserted to $o$
        if Version.writeTime < $T_i.timeStamp$ && $T_i \rightarrow Version.writer$ then
            // check if $T_i$ and Version.writer are concurrent
            add Version.writer to $T_i.rtPre$;  // Version.writer $\prec_H T_i$
    foreach Version $\in o.v_p$ do
        if Version.writer $\in T_i.rtPre$ then
            // the detected real-time order Version.writer $\prec_H T_i$ is wrong
            wait until Version.writerstatus = committed
```

### 7.3 Expected Properties

To evaluate the effectiveness of the DDA model, we propose a set of desirable properties for an effective D-STM supporting multi-versioned objects.
**Permissiveness.** For multi-versioned STM, the key advantage compared with the CM model is its ability to reduce the number of aborts. The criterion of transaction histories accepted by an STM is captured by the notion of *permissiveness* [83], which restricts the set of aborted transactions by defining such criterion. Informally, an STM satisfies \( \pi \)-permissiveness for a correctness criterion \( \pi \), if every history that does not violate \( \pi \) is accepted by the STM. However, \( \pi \)-permissiveness is proposed based on an STM model only supporting single-versioned objects and is not sufficient for multi-versioned STM. Keidar and Perelman proposed *online \( \pi \)-permissiveness* [22] which does not allow aborting any transaction if there is a way to continue the run without violating \( \pi \). Later, Perelman *et. al.* proposed *multi-versioned (MV)-permissiveness* in [24]. In an STM that satisfies MV-permissiveness, read-only transactions never abort and an update transaction is only aborted when it conflicts with another update transaction.

\( \pi \)-permissiveness is argued to be too strong [24]: it aims to avoid all spurious aborts, but is too complicated to achieve and requires keeping a large amount of object versions. On the other hand, we argue that MV-permissiveness may not be strong enough since it does not guarantee that each transaction eventually commits (after a finite number of aborts). For example, an update transaction \( T \) may be aborted by infinite times if for every time it restarts, one object in its readset has been overwritten by another update transaction after being read by \( T \) and before \( T \) commits. In other words, under a certain workload a transaction may starve in an MV-permissive STM. Therefore, our permissive condition captures the starvation-free property in addition to MV-permissiveness.

**Definition 32.** An STM satisfies starvation-free multi-versioned (SF-MV)-permissiveness if: 1) a transaction aborts only when it is an update transaction that conflicts with another update or write-only transaction; 2) every transaction commits after a finite number of aborts.

Informally, in an STM that satisfies SF-MV-permissiveness, read-only transactions never abort and never cause other transactions’ aborts. Furthermore, transactions never starve in SF-MV-permissive STM.

**Garbage collection.** For multi-versioned STM, old object versions have to be efficiently garbage collected (GC) to save as much space as possible. Perelman *et. al.* [24] argued that no STM can be online space optimal and proposed *useless-prefix (UP)-GC*, a GC mechanism which removes the longest possible prefix of versions for each object at any point of time and keeps the shortest suffix of versions that might be needed by read-only transactions. However, for D-STM, while transactions may insert its versions before an old version, it may not be safe to always remove the longest possible suffix of versions, since a transaction may not be able to find the proper place to insert its versions. Hence, we define a more practical GC mechanism for SF-MV-permissive STM.

**Definition 33.** A SF-MV-permissive STM satisfies real-time useless-prefix (RT-UP)-GC if at any point in a transactional history \( H \), an object version \( o^k_j \) is kept only if there exists an extension of \( H \) with a live transaction \( T_i \), such that \( o^k_j \) is the latest version of \( o_j \) satisfying \( o^k_j.writer \prec_H T_i \).
**Read visibility.** The STM implementation uses invisible reads if no shared object is modified when a transaction performs a read-only operation on a shared object, i.e., a read-only transaction leaves no trace the external system about its execution. If a D-STM supports invisible reads, a traffic between nodes can be greatly reduced for read-dominated workloads and the overall throughput of operations is potentially larger.

### 7.4 Read/Write Algorithms

#### 7.4.1 Description

Before a transaction performs each read/write operation, it must guarantee that the correctness criterion is not violated. Applying the precedence graph in D-STM introduces some unique challenges. The key challenge is that, in distributed systems, each transaction has to make decisions based on its local knowledge. A centralized algorithm (e.g., assigning a coordinating node to maintain the precedence graph and make decisions whenever a conflict occurs) involves frequent interactions between different nodes, and is impractical due to the underlying high communication cost. For the same reason, it is also impractical to maintain a global precedence graph on each individual node. Thus, we propose a set of policies to handle read/write operations such that the acyclicity of the underlying precedence graph is not violated and without frequent inter-transaction communications for each individual transaction.

**Principle 1.** In the DDA model, a read-only transaction never aborts, i.e., it commits at its first execution.

The pseudo code of read operation for read-only transactions is shown in Algorithm 4. Consider a transaction $T_i$ reading object $o$. If $T_i$ is a read-only transaction, it reads the latest committed version $o.v_c[j]$ where $o.v_c[j].writer \prec_T T_i$, i.e., the writer of $o.v_c[j]$ precedes $T_i$ in real-time order (lines 2-5). In this way, a read-only transaction is always serialized before other concurrent update/write-only transactions. On the other hand, each object must keep proper object versions to satisfy that each read-only transaction can find the latest committed object version which precedes it in real-time order.

**Principle 2.** In the DDA model, a transaction aborts if: 1) it is not a read-only transaction; and 2) it conflicts with another transaction and at least one of the conflicting transaction is an update transaction.

Therefore, a transaction aborts in two cases: 1) two update transactions read the same object; and 2) one update transaction and another update/write-only transaction writes to the same object. We discuss them case by case.
Algorithm 4: Algorithms for read operations

procedure Read(o) for read-only transaction $T_i$
for Version ← o.vc[max] to o.vc[min] do
    // scan the committed version list of o from the latest one
    if Version.writer ≺ $H$ then
        return Version.data
        break
procedure Priority Assignment
On (re)start of update/write-only transaction $T_i$:
    $x_{T_i}$ ← random integer in $[1,m]$;
procedure Read(o) for update transaction $T_i$
    abortList ← ∅
    foreach suc ∈ o.vc[max].sucSet do
        if suc.status == live then
            if $x_{suc} < x_{T_i}$ then
                ABORT;
            else
                add suc to abortList;
        if $T_i.status = live$ then
            if o.vc[max].writer.timestamp ≥ $T_i.timestamp$ then
                $T_i.timestamp$ ← o.vc[max].writer.timestamp + $\epsilon$
    foreach abortWriter ∈ abortList do
        send abort message to abortWriter
    add $T_i$ to o.vc[max].sucSet
    return o.vc[max].data // return the latest version

Case 1. The pseudo code of read operation for update transactions is shown in Algorithm 4. If $T_i$ is an update transaction, it checks the known successors (which must be serialized after the versions’s writer) of the latest committed version $o.vc[max]$ and applies a randomized algorithm to make the decision. To assign the priority randomly, each update/write-only transaction $T$ selects an integer $x_T \in [1,m]$ uniformly, independently and randomly when starts or restarts (lines 6-8).

If there exists a live update/write-only transaction $T_j \in o.vc[max].sucSet$ (line 12), then either one of $T_i$ and $T_j$ can proceed. The transaction with smaller $x_T$ has the higher priority (lines 13-17). After examines all transactions in $o.vc[max].sucSet$, if $T_i$ is still alive, it sends an abort message to each transaction aborted by $T_i$ (lines 20-21). $T_i$ reads $o.vc[max]$ and adds itself to $o.vc[max].sucSet$ (lines 22-23).

An update transaction’s timestamp is updated when it reads an object. When an update transaction $T_i$ reads an object version $o.vc[max]$, it checks the timestamp of its writer ($o.vc[max].writer.timestamp$). If it has the larger timestamp than $T_i$, then $T_i$’s timestamp is increased to $o.vc[max].writer.timestamp + \epsilon$, which is slightly greater than $o.vc[max].writer.timestamp$.

For example, in the scenario depicted in Figure 7.7, the sequence of versions read by $T_2$ is $\{o_1^1, o_2^1\}$. Update transaction $T_4$ checks the successors of $o_2^1$ (written by $T_5$) when reads
Figure 7.7: Transactions are serialized in order $T_1T_3T_2T_5T_4T_6$, where $T_6$ aborts.

$O_2$. Hence, $T_4$ compares $x_{T_4}$ with $x_{T_6}$. Assume that $x_{T_4} < x_{T_6}$, $T_4$ aborts $T_6$ by sending it an abort message (the dotted line). Now the set of transactions can be serialized in order $T_1T_3T_2T_5T_4T_6$, where $T_6$ aborts. Similar analysis also applies if $x_{T_6} < x_{T_5}$ and $T_5$ aborts. Note that after reads $O_2$, $T_1$’s timestamp is updated from $t_1$ to $t_2 + \epsilon$. Later in this section, we will show that by doing this, the versions written by $T_4$ (e.g., $T_4$’s version of $O_3$) can be correctly after the versions written by $T_5$ (e.g., $T_5$’s version of $O_3$) by simply comparing their timestamps.

Algorithm 5: Algorithms for write operations

<table>
<thead>
<tr>
<th>Procedure Priority Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>On</strong> (re)start of update/write-only transaction $T_i$:</td>
</tr>
<tr>
<td>$x_{T_i} \leftarrow$ random integer in $[1,m]$;</td>
</tr>
<tr>
<td><strong>procedure</strong> WRITE($O,v(T_i)$) for update/write-only transaction $T_i$</td>
</tr>
<tr>
<td>abortList $\leftarrow \emptyset$;</td>
</tr>
<tr>
<td>for $Version \leftarrow o.v.[max]$ to $o.v.[min]$ do</td>
</tr>
<tr>
<td>if $Version.writer \prec_H T_i$ then</td>
</tr>
<tr>
<td>foreach $suc \in Version.sucSet$ do</td>
</tr>
<tr>
<td>if $suc.status = live &amp; {T_i</td>
</tr>
<tr>
<td>if $x_{suc} &lt; x_{T_i}$ then</td>
</tr>
<tr>
<td>ABORT;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>add suc to abortList;</td>
</tr>
<tr>
<td>if $T_i.status == live$ then</td>
</tr>
<tr>
<td>foreach abortSuc $\in$ abortList do</td>
</tr>
<tr>
<td>send abort message to abortSuc</td>
</tr>
<tr>
<td>add $T_i$ to $Version.sucSet$;</td>
</tr>
<tr>
<td>$o.v_p[max + 1] \leftarrow v(T_i)$;</td>
</tr>
<tr>
<td>break</td>
</tr>
</tbody>
</table>

**Case 2.** We present the pseudo code of write operations in Algorithm 5. When requests
to write value $v(T_i)$ to an object $o$, an update/write-only transaction $T_i$ checks the latest committed version $o.v_c[j]$ such that the writer of $o.v_c[j]$ precedes $T_i$ in real-time order (lines 6-7). For each live transaction $suc \in o.v_c[j].sucSec$, if one transaction in the pair $\langle suc, T_i \rangle$ is an update transaction, then either one of $T_i$ and $suc$ can proceed. The similar random algorithm as the read operations is applied to compare priorities (lines 8-13). If $T_i$ eventually proceeds, it sends a message to each aborted transactions (lines 15-16). $T_i$ writes to $o$ by appending $v(T_i)$ to the end of the pending committed list $o.v_p$ and adds itself to $o.v_c[j].sucSet$ (lines 17-18).

![Diagram](image)

Figure 7.8: Transactions are serialized in order $T_1T_2T_3T_4$, where $T_1$ and $T_2$ abort.

For example, consider the scenario depicted in Figure 7.8. $T_3$ conflicts with $T_1$ and $T_2$ at $t_1$ and $T_2$, respectively. Assume that $x_{T_3} < \{x_{T_1}, x_{T_2}\}$, then $T_3$ commits and sends an abort message to $T_1$ and $T_2$, respectively (the dotted lines). $T_4$ does not conflict with $T_3$ since $T_3 \prec_H T_4$. The set of transactions can be serialized in order $T_1T_2T_3T_4$, where $T_1$ and $T_2$ abort.

### 7.4.2 Analysis

In this section, we prove that the DDA model satisfies opacity and satisfies following properties: 1) it is SF-MV-permissive with high probability; 2) it supports RT-UP-GC; and 3) it supports invisible reads.

**Correctness.**

**Lemma 12.** In the DDA model, a transaction does not generate any cycle in the precedence graph $PG$ before it tries to commit.

**Proof.** We prove this theorem case by case. Consider an update/write-only transaction $T_i$. If $T_i$ reads object version $o_j^k$, then it only adds a $W \rightarrow R$ edge from $o_j^k.writer$ to $T_i$ to $PG$.
since \( o_j^k \) is the latest committed version of \( o_j \). If \( T_i \) writes to object \( o_j \), it first finds the latest committed version \( o_j.v_c[k] \) where \( o_j.v_c[k].writer \prec_H T_i \), i.e., the writer of \( o_j.v_c[k] \) precedes \( T_i \) in real-time order. It only adds an \( R \rightarrow W \) edge from \( T_i \in o_j.v_c[k].sucSet \) to \( T_i \) in two cases: 1) \( T_i \) is a read-only transaction which reads \( o_j.v_c[k] \); 2) \( T_i \) is a committed update transaction which reads \( o_j.v_c[k] \). Note that the operations of \( T_i \) only introduce incoming edges to \( T_i \) in \( PG \). Hence, \( T_i \) does not generate any outgoing edge before it tries to commit and no cycle forms.

Consider a read-only transaction \( T_i \). From the description of read operations, we know that \( T_i \) can always find an object version \( o_j^k \) to read for object \( o_j \), where \( o_j^k.writer \prec_H T_i \). Hence, for each object \( o_j^k \) read by \( T_i \): 1) no new incoming edge to \( T_i \) is added to \( PG \); 2) an \( R \rightarrow W \) outgoing edge from \( T_i \) to \( T_i \) is added to \( PG \) for each \( T_i \in o_j^k.writer \) where \( T_i \) writes to \( o_j \).

Suppose a cycle is generated by \( T_i \)’s operation. Then we can find a cycle \( T_{i_1} \rightarrow T_i \rightarrow T_{i_2} \rightarrow \ldots \rightarrow T_{i_k} \) where \( T_{i_1} \prec_H T_i \) and \( T_i \rightarrow T_{i_2} \) is an \( R \rightarrow W \) edge. Then a path exists from \( T_{i_2} \) to \( T_i \) before \( T_i \)’s operation. Note that \( T_{i_2} \) is an update transaction. There are two cases based on \( T_{i_2} \)’s status. If \( T_{i_2} \) is a live transaction, from the first part of the proof we know that no outgoing edge from \( T_{i_2} \) exists in \( PG \). If \( T_{i_2} \) is a committed transaction, a path forms from \( T_{i_2} \) to \( T_i \) if and only if \( T_{i_1} \) commits after \( T_{i_2} \) commits. In both cases, a contradiction forms. The lemma follows.

Lemma 12 guarantees the acyclicity of \( PG \) from the time a transaction starts to the time it tries to commit. Obviously, the commit of a read-only transaction does not make any change to \( PG \). For transactions with write operations, a new version is inserted in the committed version list for each object in its \texttt{writeList}. Such operation brings new edges to \( PG \).

**Lemma 13.** In the DDA model, the \texttt{InsertVersion} operation of a transaction with write operations does not generate any cycle in the precedence graph \( PG \).

**Proof.** Consider an update transaction \( T_i \) which inserts a new version \( v(T_i) \) to the committed version list \( o_j.v_c \) of object \( o_j \). From Lemma 12, we know that before \( T_i \) tries to insert object versions, it does not bring any new outgoing edge to \( PG \). If \( v(T_i) \) is inserted to the tail of \( o_j.v_c \), then a \( W \rightarrow W \) edge from \( o_j.v_c[\text{max}].writer \) to \( T_i \) and a set of \( R \rightarrow W \) edges from \( T_i \) to \( T_i \) for each \( T_i \in o_j.v_c[\text{max}].\text{readers} \) are added to \( PG \). Hence, no new outgoing edge from \( T_i \) is added to \( PG \).

If \( v(T_i) \) is inserted to the place preceding \( o_j.v_c[k] \), then a \( W \rightarrow W \) edge from \( o_j.v_c[k].writer \) to \( T_i \) and a set of \( R \rightarrow W \) edges from \( T_i \) to \( T_i \) for each \( T_i \in o_j.v_c[k].\text{readers} \) are added to \( PG \). Additionally, a \( W \rightarrow W \) edge from \( T_i \) to \( o_j.v_c[k].writer \) is added to \( PG \). However, from the description of \texttt{InsertVersion} we know that \( v(T_i) \) is inserted before \( o_j.v_c[k] \) if and only if there preexists an edge from \( T_i \) to \( o_j.v_c[k] \) in \( PG \). Hence, the \texttt{InsertVersion} operation does not introduce new outgoing edge from \( T_i \) to \( PG \). The lemma follows.

We now introduce the following lemma with the help of Lemma 4 from [22]:
Lemma 14. If $PG$ of the execution of a set of transactions is acyclic, then the non-local history $H$ of the execution satisfies opacity.

We have the following theorem.

Theorem 35. In the DDA model, the non-local history $H$ of the execution of any set of transactions satisfies opacity.

Proof. The theorem follows from Lemmas 12, 13 and 14.

Properties.

Lemma 15. A transaction is aborted at most $O(C \log n)$ times before it commits with probability $1 - \frac{1}{n^2}$, where $n$ is the number of transactions in $T$ and $C$ is the maximum number of transactions concurrently conflicting with a single transaction.

Proof. Let the set of conflicting transaction of transaction $T$ denoted by $N_T$. $T$ can only be aborted when it chooses a larger $x_T$ than $x_{T'}$, the integer chosen by a conflicting transaction $T'$. The probability that for transaction $T$, no transaction $T' \in N_T$ selects the same random number $x_T = x_{T'}$ is

$$
\Pr(\#T' \in N_T | x_T = x_T) = \prod_{T' \in N_T} (1 - \frac{1}{m}) \geq (1 - \frac{1}{m})^{|N_T|} \geq (1 - \frac{1}{m})^m \geq \frac{1}{e}.
$$

Note that $|N_T| \leq C \leq m$. On the other hand, the probability that $x_T$ is at least as small as $x_{T'}$ for any conflicting transaction $T'$ is at least $\frac{1}{e(C+1)}$. Thus, the probability that $x_T$ is the smallest among all its neighbors is at least $\frac{1}{e(C+1)}$. We use the following Chernoff bound:

Lemma 16. Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $\Pr(X_i = 1) = p_i$, where $0 \leq p_i \leq 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = E[X] = \sum_{i=1}^n p_i$, and any $\delta \in (0, 1]$, $\Pr(X < (1 - \delta)\mu) < e^{-\delta^2 \mu/2}$.

By Lemma 22, if we conduct $16e(C + 1) \ln n$ trials, each having success probability $\frac{1}{e(C+1)}$, then the probability that the number of successes $X$ is less than $8 \ln n$ becomes:

$$
\Pr(X < 8 \ln n) < e^{-2\ln n} = \frac{1}{n^2}.
$$

The theorem follows.

Theorem 36. The DDA model satisfies SF-MV-permissiveness with probability $1 - \frac{1}{n^2}$.

Proof. The theorem follows from Lemma 15.

Theorem 37. The DDA model satisfies RT-UP-GC.
Proof. The theorem directly follows from the algorithm description. For any object, the earliest version it needs to keep is the latest version that precedes all live read-only transactions in real-time order. All versions earlier than this version can be GCed. The theorem follows.

Theorem 38. The DDA model supports invisible reads.

Proof. We prove the corollary by contradiction. Suppose the DDA model does not support invisible reads. Then for any history $H$, we can find a read-only transaction $T_i$ which causes the abort of a read-only transaction or a write-only transaction if $T_i$ is invisible. Note that if $T_i$ is invisible, then the edges added to $PG$ by its read operations are not observed by the STM. From the proof of Lemma 12, we know that $T_i$ only adds outgoing edges from $T_i$ to $PG$. On the other hand, an update transaction only adds incoming edges to $PG$. Hence, the only possibility of the cycle formed must be of the form $T_{i_1} \rightarrow T_i \rightarrow \ldots \rightarrow T_{i_2} \rightarrow T_{i_1}$ where: 1) $T_{i_1} \prec_H T_i$; 2) $T_{i_2}$ is an update transaction; 3) $T_{i_1}$ reads a committed version written by $T_{i_2}$. Then contradiction forms since $T_i$ and $T_{i_2}$ must be concurrent transactions. The theorem follows.

7.5 Conclusion

DDA model takes a step towards enhancing concurrency in D-STM. We have shown the tradeoff of directly adopting past conflict resolution strategies: the CM model is easy to implement and involves low communication cost in resolving conflicts. However, it may introduce a large number of unnecessary aborts. On the other hand, resolving conflicts by completely relying on establishing precedence relations can effectively reduce aborts. However, it requires frequent message exchanges, which may introduce high communication costs in D-STM. The DDA model, in some sense, plays a role between these two extremes. It allows maximum concurrency for some transactions (i.e., read-only transactions), and uses randomized priority assignment to treat “dangerous” transactions (i.e., update transactions), which will likely participate in a cycle in the underlying precedence graph. Moreover, the randomized algorithm ensures the starvation-freedom property with high probability.
Chapter 8

D-STM Contention Management Problem

In this chapter, we study the contention management problem for D-STM. We first construct a dynamic ordering conflict graph $G^*_c(\phi(\kappa))$ for an offline algorithm $(\kappa, \phi)$, which computes a $k$-coloring instance $\kappa$ of the dynamic conflict graph $G_c$ and processes the set of transactions in the order of $\phi$. We show that finding an optimal schedule is equivalent to finding the offline algorithm for which the weight of the longest weighted path in $G^*_c(\phi(\kappa))$ is minimized. We further illustrate that when the set of transactions are dynamically generated, processing transactions according to a $\chi(G_c)$-coloring of $G_c$ does not lead to an optimal schedule, where $\chi(G_c)$ is the chromatic number of $G_c$. We prove that for D-STM, any online, work conserving deterministic contention manager provides an $\Omega(\max[s, \sqrt{s}D])$ competitive ratio in a network with normalized diameter $\overline{D}$. Compared with the $\Omega(s)$ competitive ratio for multiprocessor STM, the performance guarantee for D-STM degrades by a factor proportional to $\frac{s}{\overline{D}}$. We present two randomized algorithms for D-STM. The first algorithm RANDOMIZED is motivated by existing randomized algorithms for multiprocessor STM without considering optimizing the cost of moving objects in the network. We show that the competitive ratio of RANDOMIZED is $O(s \cdot (C \log n + \log^2 n))$ for $s$ object shared by $n$ transactions and the maximum conflicting degree is $C$. To break this lower bound, we present a randomized algorithm CUTTING, which needs partial information of transactions and an approximate algorithm $A$ for the traveling salesman problem (TSP) with approximation ratio $\phi_A$. We show that the average case competitive ratio of CUTTING is $O(s \cdot \phi_A \cdot \log^2 m \log^2 n)$, which is close to $O(s)$. 

92
8.1 The D-STM Contention Management Problem

8.1.1 Conflict Graph

We build the conflict graph $G = (T, E)$ for the subset transactions $T \subseteq T$ which runs concurrently. An edge $(T_i, T_j) \in E$ exists if and only if $T_i$ and $T_j$ conflict. Let $N_T$ denote the set of neighbors of $T$ in $G$. The degree $\delta(T) := |N_T|$ of a transaction $T$ corresponds to the number of neighbors of $T$ in $G$. We denote $C = \max_i \delta(T_i)$, i.e., the maximum degree of a transaction.

The graph $G$ is highly dynamic and only consists of live transactions. A transaction joins $T$ after it (re)starts, and leaves $T$ after it commits/aborts. Therefore, $N_T, \delta(T)$ and $C$ only capture a “snapshot” of $G$ at a certain time. More precisely, they should be represented as functions of time. When there is no ambiguity, we use the simplified notations. We have $|T| \leq \min\{m, n\}$, since there are at most $n$ transactions and at most $m$ transactions can run in parallel. Then we have $\delta(T) \leq C \leq \min\{m, n\}$.

Recall that $\vec{o}(T_i) = \{o_1(T_i), o_2(T_i), \ldots\}$ denote the sequence of objects requested by transaction $T_i$. Let $\gamma(o_j)$ denote the number of transactions in $T$ that concurrently write $o_j$ and $\gamma_{\max} = \max_j \gamma(o_j)$. Let $\lambda(T_i) = \{o : o \in \vec{o}(T_i) \land (\gamma(o) \geq 1)\}$ denote the number of transactions in $T$ that can be the cause of conflicts of transaction $T_i$ and $\lambda_{\max} = \max_{T_i \subseteq T} \lambda(T_i)$. We have $C \leq \lambda_{\max} \cdot \gamma_{\max}$ and $C \geq \gamma_{\max}$.

8.1.2 Problem Measure and Complexity

Intuitively, a contention manager is a distributed scheduler installed on each node in the system. Generally speaking, a contention manager determines when a particular transaction executes in case of a conflict. Recall that, to evaluate the performance of a contention manager quantitatively, we measure the makespan, which is the total time needed to complete the given set of transactions $T$. Formally, given a contention manager $A$, makespan$_A$ denotes the time to complete all transactions in $T$ under $A$.

We measure the quality of a contention manager, by assuming Opt, the optimal, centralized, clairvoaynt scheduler which has the complete knowledge of each transaction (requested set of objects, location, released time and local execution time duration). The quality of a contention manager $A$ is measured by the ratio $\frac{\text{makespan}_A}{\text{makespan}_{\text{Opt}}}$, called the competitive ratio of $A$ on $T$. The competitive ratio of $A$ is $\max_T \frac{\text{makespan}_A}{\text{makespan}_{\text{Opt}}}$, i.e., the maximum competitive ratio of $A$ over all possible workloads.

An ideal contention manager aims to provide an optimal schedule for any given set of transactions. However, it is shown in [34] that if there exists an adversary to change the set of shared objects requested by any transaction arbitrarily in multiprocessor STM, no algo-
gorithms can do better than a simple sequential execution. Furthermore, even if the adversary can only choose the initial conflict graph and does not influence it afterwards, it is NP-hard to get a reasonable approximation of an optimal schedule [42].

We can consider the transaction scheduling problem for multiprocessor STM as a subset of the whole problem space of the transaction scheduling problem for D-STM. The two problems are equivalent as long as the communication cost \((d_{ij})\) can be ignored compared with the local execution time duration \((\tau_i)\). Therefore, extending the problem space into distributed systems only increases the problem complexity.

We depict an example of conflict graph \(G_c\) in Figure 8.1, which consists of 9 write-only transactions. Each transaction is represented as a numbered node in \(G_c\). Each edge \((T_i, T_j)\) is marked with the object with causes \(T_i\) and \(T_j\) conflict (e.g., \(T_1\) and \(T_4\) conflict on \(o_1\)). We can construct a coloring of the conflict graph \(G_c = (T_c, E)\). A 3-coloring scenario is illustrated in Figure 8.1. Transactions are partitioned into 3 sets: \(C_1 = \{T_1, T_2, T_3\}, C_2 = \{T_4, T_5, T_6\}, C_3 = \{T_7, T_8, T_9\}\). Since transactions with the same color are not connected, every set \(C_i \subset T_c\) forms an independent set and can be executed in parallel without facing any conflicts. By adopting the same argument from [34], we have the following lemma.

**Lemma 17.** An optimal offline schedule \(\text{Opt}\) determines a \(k\)-coloring \(\kappa\) of the conflict graph \(G_c\) and an execution order \(\phi_\kappa\) such that for any two sets \(C_{\phi_\kappa(i)}\) and \(C_{\phi_\kappa(j)}\) where \(i < j\), if (1) \(T_1 \in C_{\phi_\kappa(i)}\), \(T_2 \in C_{\phi_\kappa(j)}\), and (2) \(T_1\) and \(T_2\) conflicts, then \(T_2\) is postponed until \(T_1\) commits.

In other words, \(\text{Opt}\) determines the order in which an independent set \(C_i\) is executed. Generally, for a \(k\)-coloring of \(G_c\), there are \(k!\) different choices to order the independent sets. Assume that for the 3-coloring example in Figure 8.1, an execution order \(\phi_\kappa = \{C_1, C_2, C_3\}\) is selected. We can construct an ordering conflict graph \(G_c(\phi_\kappa)\), as shown in Figure 8.2.

**Definition 34** (Ordering conflict graph). For the conflict graph \(G_c\), given a \(k\)-coloring instance \(\kappa\) and an execution order \(\{C_{\phi_\kappa(1)}, C_{\phi_\kappa(2)}, \ldots, C_{\phi_\kappa(k)}\}\), the ordering conflict graph \(G_c(\phi_\kappa) = (T_c, E(\phi_\kappa), w)\) is constructed. \(G_c(\phi_\kappa)\) has following properties:
1. $G_c(\phi_\kappa)$ is a weighted directed graph.
2. For two transactions $T_1 \in C_{\phi_\kappa(i)}$ and $T_2 \in C_{\phi_\kappa(j)}$, a directed edge (or an arc) $(T_1, T_2) \in E(\phi_\kappa)$ (from $T_1$ to $T_2$) exists if: (i) $T_1$ and $T_2$ conflict over object $o$; (ii) $i < j$; and (iii) $\not\exists T_3 \in C_{\phi_\kappa(j')}$, where $i < j' < j$, such that $T_1$ and $T_3$ also conflict over $o$.
3. The weight $w(T_i)$ of a transaction $T_i$ is $\tau_i$; the weight $w(T_i, T_j)$ of an arc $(T_i, T_j)$ is $d_{ij}$.

![Figure 8.2: Ordering conflict graph $G_c(\phi_\kappa)$](image)

For example, the edge $(T_1, T_4)$ in Figure 8.1 is also an arc in Figure 8.2. However, the edge $(T_1, T_7)$ in Figure 8.1 no longer exists in Figure 8.2 because $C_2$ is ordered between $C_1$ and $C_3$ and, $T_1$ and $T_4$ also conflict on $o_1$.

Hence, any offline algorithm can be described by the pair $(\kappa, \phi_\kappa)$, and the ordering conflict graph $G_c(\phi_\kappa)$ can be constructed. Given $G_c(\phi_\kappa)$, the execution time of each transaction can be determined.

**Theorem 39.** For the ordering conflict graph $G_c(\phi_\kappa)$, given a directed path $P = \{T_{P(1)}, T_{P(2)}, \ldots, T_{P(L)}\}$ of $L$ hops, the weight of $P$ is defined as

$$w(P) = \sum_{1 \leq i \leq L} w(T_{P(i)}) + \sum_{1 \leq j \leq L-1} w(T_{P(j)}, T_{P(j+1)}).$$

Then transaction $T_0 \in T_c$ commits at time

$$\max_{P=\{T_{P(1)}, \ldots, T_0\}} t_{P(1)} + w(P),$$

where $T_{P(1)}$ starts at time $t_{P(1)}$.

**Proof.** We prove the theorem by induction. Assume $T_0 \in C_{\phi_\kappa(j)}$. When $j = 1$, $T_0$ executes immediately after it starts. At time $t_0 + \tau_0$, $T_0$ commits. There is only one path ends at $T_0$ in $G_c(\phi_\kappa)$ (which only contains $T_0$). The theorem holds.
Assume that when \( j = 2, 3, \ldots, q - 1 \), the theorem holds. Let \( j = q \). For each object \( o_i \in \partial(T_0) \), find the transaction \( T_{0(i)} \) such that \( T_{0(i)} \) and \( T_0 \) conflict over \( o_i \), and \( (T_{0(i)}, T_0) \in E(\phi_\kappa) \). If no such transaction exists for all objects, the analysis falls into the case when \( j = 1 \). Otherwise, for each transaction \( T_{0(i)} \), from Definition 34, no transaction which requests access to \( o_i \) is scheduled between \( T_{0(i)} \) and \( T_0 \). The offline algorithm \((\kappa, \phi_\kappa)\) moves \( o_i \) from \( T_{0(i)} \) to \( T_0 \) immediately after \( T_{0(i)} \) commits. Assume that \( T_{0(i)} \) commits at \( t^c_{0(i)} \). Then \( T_0 \) commits at time

\[
\max_{o_i \in \partial(T_0)} t^c_{0(i)} + w(T_{0(i)}, T_0) + w(T_0),
\]

Since \((T_{0(i)}, T_0) \in E(\phi_\kappa)\), then from the induction step we know that

\[
t^c_{0(i)} = \max_{P=(T_{P(1)}, \ldots, T_{0(i)})} t_P + w(P).
\]

Hence, by replacing \( t^c_{0(i)} \) with \( \max_{P=(T_{P(1)}, \ldots, T_{0(i)})} t_P + w(P) \), the theorem follows.

Theorem 39 illustrates that the commit time of transaction \( T_0 \) is determined by one of the weighted paths in \( G^c_\kappa(\phi_\kappa) \) which ends at \( T_0 \). Specifically, if every node issues its first transaction at the same time, the commit time of \( T_0 \) is solely determined by the longest weighted path in \( G^c_\kappa(\phi_\kappa) \) which ends at \( T_0 \). However, when transactions are dynamically generated over time, the commit time of a transaction also relies on the starting time of other transactions. To accommodate the dynamic features of transactions, we construct the dynamic ordering conflict graph \( G^*_\kappa(\phi_\kappa) \) based on \( G^c_\kappa(\phi_\kappa) \).

**Definition 35** (Dynamic ordering conflict graph). Given an ordering conflict graph \( G^c_\kappa(\phi_\kappa) \), the dynamic ordering conflict graph \( G^*_\kappa(\phi_\kappa) \) is constructed by making following modifications on \( G^c_\kappa(\phi_\kappa) \):

1. For the sequence of transactions \( \{T^1_i, T^2_i, \ldots, T^L_i\} \) issued by each node \( v_i \), an arc \((T^j_{i-1}, T^j_i)\) is added to \( G^*_\kappa(\phi_\kappa) \) for \( 2 \leq j \leq L \) and \( w(T^j_{i-1}, T^j_i) = 0 \).

2. If transaction \( T_j \) which starts at \( t_j \) does not have any incoming arcs in \( G^*_\kappa(\phi_\kappa) \), then \( w(T_j) = t_j + t_j \).

**Theorem 40.** The makespan of algorithm \((\kappa, \phi_\kappa)\) is the weight of the longest weighted path in \( G^*_\kappa(\phi_\kappa) \).

\[
\text{makespan}_{(\kappa, \phi_\kappa)} = \max_{P \in G^*_\kappa(\phi_\kappa)} w(P)
\]

**Proof.** We start the proof with special cases, then extend the analysis to the general case. Assume that (i) each node issues only one transaction, and (ii) all transactions start at the same time. Then the makespan of \((\kappa, \phi_\kappa)\) is equivalent to the execution time of the last committed transaction:

\[
\text{makespan}_{(\kappa, \phi_\kappa)} = \max_{T_0 \in \mathcal{T}_c, P \in G^c_\kappa(\phi_\kappa), P = \ldots, T_0} w(P) = \max_{P \in G^c_\kappa(\phi_\kappa)} w(P) = \max_{P \in G^*_\kappa(\phi_\kappa)} w(P).
\]
Now we relax the second assumption: each node issues a single transaction at arbitrary time points. Similar to the first case, we have

$$\text{makespan}(\kappa, \phi_n) = \max_{P \in G_c(\phi_n), P = \{T_{P_1}, ..., T_0\}} (t(P_1) + w(P)).$$

Let \( P \) be the path which maximizes \( \text{makespan}(\kappa, \phi_n) \). Therefore, \( T_{P_1} \) (the head of \( P \)) has no incoming arcs in \( G^*_c(\phi_n) \) (since each node only issues a single transaction). From the construction of \( G^*_c(\phi_n) \), \( w(T_{P_1}) = t(P_1) + \tau_{P_1} \). We can find a path \( P^* \) in \( G^*_c(\phi_n) \) which contains the same elements as \( P \) with weight \( w(P^*) = t(P_1) + w(P) \), which is the longest path in \( G^*_c(\phi_n) \).

Now we relax the first assumption: each node issues a sequence of transactions, and all nodes start their first transactions at the same time. Similar to the first case, we have

$$\text{makespan}(\kappa, \phi_n) = \max_{P \in G_c(\phi_n), P = \{T_{P_1}, ..., T_0\}} (t(P_1) + w(P)).$$

Let \( P \) be the path which maximizes \( \text{makespan}(\kappa, \phi_n) \). If \( T_{P_i} \) (the head of \( P \)) is the first transaction issued by a node, the theorem follows. Otherwise, \( \forall o_i \in \bar{D}(T_{P_i}) \), \( T_{P_i} \) is the first transaction scheduled to access \( o_i \) by \( (\kappa, \phi_n) \) because there is no incoming arcs to \( T_{P_i} \) in \( G_c(\phi_n) \). If \( T_{P_i} \) is the \( l \)th transaction issued by node \( v_j \), when we convert from \( G_c(\phi_n) \) to \( G^*_c(\phi_n) \), the longest path \( P^* \) ends at \( T_0 \) is a path starting from \( T^l_j \) to \( T^l_{l-1} \), followed by an arc \( (T^l_{l-1}, T_{P_l}) \), and then followed by \( P \). Note that \( T^l_{l-1} \) commits at \( t_{P_l} \) (the starting time of \( T_{P_l} \)). Hence, we have \( w(P^*) = t(P_1) + w(T^l_{l-1}, T_{P_l}) + w(P) \). Since \( w(T^l_{l-1}, T_{P_l}) = 0 \) (from the construction of \( G^*_c(\phi_n) \)), we have \( t(P_1) + w(P) = w(P^*) \). We conclude that the path in \( G_c(\phi_n) \) corresponding to the commit time of transaction \( T_0 \) is equivalent to the longest path which ends at \( T_0 \) in \( G^*_c(\phi_n) \). The theorem follows.

Theorem 40 shows that given an offline algorithm \((\kappa, \phi_n)\), finding its makespan is equivalent to finding the longest weighted path in the dynamic ordering conflict graph \( G^*_c(\phi_n) \). Therefore, the optimal schedule \( \text{Opt} \) is the offline algorithm which minimizes the makespan.

**Corollary 5.**

$$\text{makespan}_{\text{Opt}} = \min_{\kappa, \phi_n} \max_{P \in G^*_c(\phi_n)} w(P)$$

It is easy to show that finding the optimal schedule is NP-hard. For the **one-shot scheduling problem**, where each node issues a single transaction, if \( \tau_0 = \tau \) for all transactions \( T_0 \in \mathcal{T} \) and \( D \ll \tau \), the problem becomes the classical node coloring problem. Finding the optimal schedule is equivalent to finding the chromatic number \( \chi(G_c) \). As shown in [95], computing an optimal coloring given complete knowledge of the graph is NP-hard and even computing an approximation with in the factor of \( \chi(G_c) \frac{\log \chi(G_c)}{25} \) is NP-hard as well.
If \( s = 1 \), i.e., there is only one object shared by all transactions in the network, finding the optimal schedule is equivalent to finding the traveling salesman problem (TSP) path in \( G_d \), i.e., the shortest hamiltonian path in \( G_d \). When the cost metric \( d_{ij} \) satisfies the triangle inequality, the resulted TSP is called the metric TSP and was shown to be NP-complete by Karp [96]. If the cost metric is symmetric, Christofides [97] has constructed an elegant algorithm approximating the metric TSP within approximation ratio \( 3/2 \). If the cost metric is asymmetric, the best known algorithm approximates the solution within approximation ratio \( O(\log m) \) [98], and whether a constant factor approximation algorithm exists is still an open question [99].

Figure 8.3: Conflict graph \( G_c \)

Figure 8.4: \( G_c(\phi_{\kappa_1}) \) and \( G_c(\phi_{\kappa_2}) \)

When each node generates a sequence of transactions dynamically, it is not always optimal to schedule transactions according to a \( \chi(G_c) \)-coloring. Since the conflict graph evolves over time, an optimal schedule based on a static conflict graph may lose the potential parallelism in future. In the dynamic ordering conflict graph, an temporary optimal scheduling does not imply the resulting longest weighted path is optimal. We use the following example to illustrate this claim.

**Example:** We assume a 8-node network, where each node issues two transactions sequentially. The conflict graph of transactions is depicted in Figure 8.3. For the set of transactions \( \{T_1^1, T_1^2, \ldots, T_8^1\} \) (the first transactions issued by each node), each transaction requests accesses to 3 objects and each object is requested by 2 transactions. In total 12 objects (\( o_1 \) – \( o_{12} \)) are shared among 8 transactions. For the set of transactions \( \{T_2^1, T_2^3, T_2^6, T_2^8\} \), object \( o_{13} \) is requested by all 4 transactions; for the set of transactions \( \{T_2^2, T_2^4, T_2^5, T_2^7\} \), object \( o_{14} \)
is requested by all 4 transactions. Note that for each node \( v_i \), transaction \( T^i_2 \) can only start after \( T^i_1 \) commits.

The dynamic ordering conflict graphs of two different offline algorithms \( \phi_{\kappa_1} \) and \( \phi_{\kappa_2} \) are depicted in Figures 8.4(a) and (b), respectively. Note that initially \( G_c \) only contains \( \{T^1_1, T^2_1, \ldots, T^8_1\} \). Algorithm \( \phi_{\kappa_1} \) selects a 2-coloring of \( G_c \), which is temporarily optimal: \( \{T^1_1, T^2_1, T^4_1, T^8_1\} \) and \( \{T^3_1, T^4_1, T^5_1, T^7_1\} \) are selected as two coloring sets and execute one after another. After \( \{T^1_1, T^3_1, T^6_1, T^8_1\} \) commits, \( \{T^1_2, T^2_2, T^6_2, T^8_2\} \) start to execute. However, since transactions in \( \{T^1_2, T^3_2, T^6_2, T^8_2\} \) mutually conflict, they can only be scheduled to execute sequentially. The similar schedule applies for \( \{T^2_2, T^3_2, T^7_2, T^8_2\} \). The resulted \( G^*_c(\phi_{\kappa_1}) \) contains 6 independent sets. The dashed arrows represent zero-weighted arcs (i.e., \( (T^i_1, T^j_2) \) for each node \( v_i \)).

Given the initial input of \( G_c \), algorithm \( \phi_{\kappa_2} \) selects a 4-coloring of \( G_c \), which is not temporarily optimal: \( \{T^1_1, T^2_1, T^8_1\} \), \( \{T^3_1, T^4_1\} \), \( \{T^5_1, T^7_1\} \) and \( \{T^3_1, T^7_1\} \) are selected to execute sequentially. When \( \{T^1_1, T^2_1\} \) commits, the second transactions issued by \( v_1 \) and \( v_7 \) can execute immediately in parallel with all other transactions. The similar case applies for \( \{T^2_1, T^3_1\} \), \( \{T^3_1, T^5_1\} \) and \( \{T^3_1, T^7_1\} \). Therefore, \( G^*_c(\phi_{\kappa_2}) \) only contains 5 independent sets.

To compute the makespan, we need to find the longest weighted paths in \( G^*_c(\phi_{\kappa_1}) \) and \( G^*_c(\phi_{\kappa_2}) \). Since at most the longest path in \( G^*_c(\phi_{\kappa_1}) \) may contain 6 transactions, and the longest path in \( G^*_c(\phi_{\kappa_2}) \) may contain 5 transactions, it is likely that makespan\(_{(\kappa_1, \phi_{\kappa_1})}\) > makespan\(_{(\kappa_2, \phi_{\kappa_2})}\), despite the fact that \( \phi_{\kappa_1} \) always selects a \( \chi(G_c) \)-coloring of \( G_c \).

The inherent reason that \( \chi(G_c) \)-coloring of \( G_c \) does not always lead to an optimal schedule is due to the dynamic nature of \( G_c \). If an algorithm selects a set of transactions to execute based on the \( \chi(G_c) \)-coloring of \( G_c \) at time \( t \), the chromatic number of the updated \( G_c \) after \( t \) (when committed transactions leaves \( G_c \) and new transactions joins \( G_c \)) may not always be optimal. For Example 1, the initial \( G_c \) is 2-colorable. However, after the execution of \( \{T^1_2, T^3_2, T^6_2, T^8_2\} \) by \( \phi_{\kappa_1} \), the chromatic number of the updated \( G_c \) is 5. On the other hand, \( \phi_{\kappa_2} \) always keeps the chromatic number of the updated \( G_c \) lower than 3. Therefore, for the dynamic conflict graph, local optimal selection does not imply the global optimal solution.

### 8.1.3 Lower Bound

Our analysis shows that it is NP-hard to compute an optimal schedule, even we know all information of transactions in advance. In practice, we are aiming to design an efficient online algorithm which guarantees non-trivial performance (i.e., better performance than simple serialization of all transactions). Before starting to design the contention manager, we need to know in the best case what performance bound an online contention manager could provide.

In [34] Attiya et al. showed that, for multiprocessor STM, a deterministic work conserving contention manager is \( \Omega(s) \)-competitive if the set of objects requested by a transaction
changes when the transaction restarts. We prove that for D-STM, the performance guarantee may be even worse.

**Theorem 41.** For D-STM, any online work conserving deterministic contention manager is $\Omega(\max[s, s^2/D])$-competitive, where $D := \frac{D}{\min_{d_{ij}}}$ is the normalized diameter of the cost graph $G_d$.

**Proof.** The proof uses $s^2$ transactions with the same local execution duration $\tau$. A transaction is denoted by $T_{ij}$, where $1 \leq i, j \leq s$. Each transaction $T_{ij}$ contains a sequence of 2 operations $\{R_i, W_i\}$ which first reads from object $o_i$ and then writes to $o_i$. Each transaction $T_{ij}$ is issued by node $v_{ij}$ at the same time, and object $o_i$ is held by node $v_i$ when the system starts. For each row $i$, we select a set of nodes $V_i := \{v_i, v_{i2}, \ldots, v_{is}\}$ within range of the diameter $D_i \leq D$.

Consider the optimal schedule $\text{Opt}$. Note that all transactions form an $s \times s$ matrix and transactions from the same row ($\{T_{i1}, T_{i2}, \ldots, T_{is}\}$ for $1 \leq i \leq s$) have the same operations. Therefore, at the start of the execution, $\text{Opt}$ selects one transaction from each row and in total $s$ transactions start to execute. Whenever $T_{ij}$ commits, $\text{Opt}$ selects one transaction from the rest of transactions in row $i$ to execute. Hence, at any time there are $s$ transactions than run in parallel.

The order that $\text{Opt}$ selects transactions from each row is crucial: $\text{Opt}$ should select transactions in the order such that the weight of the longest weighted path in $G^*_c(\text{Opt})$ is optimal. Since transactions from different rows run in parallel, we have

$$\text{makespan}_{\text{Opt}} = s \cdot \tau + \max_{1 \leq i \leq s} \text{Tsp}(G_d(o_i)),$$

where $G_d(o_i)$ denotes the subgraph of $G_d$ induced by $s$ transactions requesting $o_i$, and $\text{Tsp}(G_d(o_i))$ denotes the length of the traveling salesman path (TSP) of $G_d(o_i)$, i.e., the shortest path that visit each node exactly once in $\text{Tsp}(G_d(o_i))$.

Now consider an online work conserving deterministic contention manager $A$. Being work conserving, it must select to execute a maximal independent set of non-conflicting transactions. Since the first access of all transaction is a read, the contention manager starts to execute all $s^2$ transactions.

After the first read operation, for each row $i$, all transactions in row $i$ attempt to write $o_i$ but only one of them can commit and others abort. Otherwise, atomicity is violated since inconsistent states of some transactions may be accessed. When a transaction restarts, the adversary determines that all transactions change to write to the same object, e.g., $\{R_i, W_i\}$. Therefore the rest $s^2 - s$ transaction can only be executed sequentially after the first $s$ transaction execute in parallel and commit. Then we have

$$\text{makespan}_A \geq (s^2 - s + 1) \cdot \tau + \min_{G_d} \text{Tsp}(G_d(s^2 - s + 1)),$$
where $G_d(s^2 - s + 1)$ denotes the subgraph of $G_d$ induced by a subset of $s^2 - s + 1$ transactions.

Now we can compute the competitive ratio of $A$. We have

\[
\frac{\text{makespan}_A}{\text{makespan}_{\text{opt}}} \geq \max\left[\frac{s^2 - s + 1}{s \cdot \tau}, \frac{\min_{G_d} \text{Tsp}(G_d(s^2 - s + 1))}{\max_{1 \leq i \leq s} \text{Tsp}(G_d(o_i))}\right]
\]

\[
\geq \max\left[\frac{s^2 - s + 1}{s}, \frac{(s^2 - s + 1) \cdot \min_{G_d} d_{ij}}{(s - 1) \cdot \frac{D}{s}}\right] = \Omega(\max[s, \frac{s^2}{D}]). \text{ The theorem follows.}
\]

Theorem 41 shows that for D-STM, an online work conserving deterministic contention manager cannot provide a similar performance guarantee compared with multiprocessor STM. When the normalized network diameter is bounded (i.e., $\bar{D}$ is a constant, where new nodes join the system without expanding the diameter of the network), it can only provide an $\Omega(s^2)$-competitive ratio. In the next section, we present an online randomized contention manager, which needs partial information of a transaction in advance in order to provide a better performance guarantee.

## 8.2 Algorithms

### 8.2.1 Algorithm RANDOMIZED

We first present Algorithm RANDOMIZED (Algorithm 6), a randomized algorithm motivated by techniques adopted in designing randomized scheduling algorithms for multiprocessor STM, e.g., [42] and [43]. This algorithm is similar to Algorithm ONLINE-GREEDY proposed by Sharma et al [43], which uses a variation of Algorithm RANDOMIZEDROUND proposed by Schneider and Wattenhofer as a subroutine to resolve conflicts. We make the algorithm adaptive for D-STM and show that without considering the nodes’ locations in conflict resolution, the makespan of randomized algorithm in D-STM is worse than its performance in multiprocessor STM systems with a $O(\mathcal{C})$ factor from a worst-case perspective.

Compared with ONLINE-GREEDY, the difference of RANDOMIZED is that it measures time in discrete time steps where each time step represents the duration $\tau + D$, i.e., the local execution time duration plus the diameter of the cost graph $G_d$. This modification is due to the characteristics of D-STM: without any conflict, the duration for a transaction to commit is at most $\tau + D$: the local execution duration plus the maximum time to move an object in the network. Note that we assume that when a transaction starts, it knows the set of required objects so that the all requests can be sent at the same time.

Time is divided into frames, each of which contains $\Phi = \Theta(\ln(n))$ time steps. Each transaction $T_i$ is assigned an initial random time period consisting of $q_i$ frames, where $q_i$ is chosen
randomly, independently and uniformly from the range \([0, \alpha - 1]\), where \(\alpha = \mathcal{C}/\ln n\). The priorities are assigned in two phases. In the first phase, each transaction has two priorities: \(\text{low}\) or \(\text{high}\) \((\pi^1)\). Transaction \(T_i\) is initially in low priority and switches to high priority in the first time step of frame \(F_i = q_i\) and remains in high priority thereafter until it commits. In the second phase, each transaction selects an integer \(\pi^2 \in [1, m]\) randomly when starts or restarts. The conflicts are resolved in lexicographic order based on priority vectors. If a low priority transaction \(\pi^1 = 1\) conflicts with a high priority transaction \(\pi^1 = 0\), the low priority transaction is always aborted. If two transactions with same priority of \(\pi^1\) conflict, the transaction with lower \(\pi^2\) proceeds and the other transaction aborts. Whenever a transaction is aborted by a remote transaction, the requested object is moved to the remote transaction immediately.

### Algorithm 6: Algorithm RANDOMIZED

**Input:** A set of transactions \(T\); the conflict graph \(G_c\); the cost graph \(G_d\) with diameter \(D\); each transaction has the same local execution time duration \(\tau\); each node knows \(\mathcal{C}\), the maximum degree of a transaction in \(G_c\).

**Output:** An execution schedule of \(T\).

Divide time into frames of \(\Phi' = 16e \cdot \Phi \ln n\) time steps, where \(\Phi = \mathcal{C} + 1 + e^2 + 2 \ln n\);

Each transaction \(T_i\) chooses a random number \(q_i \in [0, \alpha - 1]\) for \(\alpha = \mathcal{C} \ln n\);

Each transaction is assigned to frame \(F_i = q_i\);

Associate pair of priorities \(\langle \pi^1_i, \pi^2_i \rangle\) to each transaction \(T_i\);

**foreach** time step \(t = 0, 1(\tau + D), 2(\tau + D), 3(\tau + D), \ldots\) **do**

**PHASE 1:** Priority Assignment

**foreach** transaction \(T_i\) **do**

- if \(t < F_i \cdot (\tau + D)\Phi'\) then
  - Priority \(\pi^1_i \leftarrow 1(\text{low})\);
- else
  - Priority \(\pi^1_i \leftarrow 0(\text{high})\);

**PHASE 2:** Conflict Resolution

**On** (re)start of transaction \(T_i\):

- \(\pi^2_i \leftarrow \text{random integer in } [1, m]\);

**On** conflict of transaction \(T_i\) with transaction \(T_j\):

- if \(\pi^1_i \neq \pi^1_j\) then **Abort** the transaction with low Priority;
- else
  - if \(\pi^2_i < \pi^2_j\) then **Abort** \((T_i, T_j)\); else **Abort** \((T_j, T_i)\);

**Analysis.** The algorithm efficiently reduces the number of conflicting transactions for each transaction \(T_i\) in its assigned frame \(F_i\). Because \(q_i\) is chosen at random among \(\mathcal{C} / \ln n\) positions it is expected that \(T_i\) will conflict with at most \(O(\ln n)\) transactions in its assigned frame \(F_i\) which become simultaneously high priority in \(F_i\). Let \(N_{T_i}\) denote the subset of conflicting transactions with \(T_i\) which become high priority during frame \(F_i\). We have the following lemmas from [43] and [42], resp.

**Lemma 18.** \(|N_{T_i}| > |\Phi - 1|\) with probability at most \(\frac{1}{n^2}\).

**Lemma 19.** A transaction is aborted at most \(16e(\mathcal{C} + 1) \ln n\) times before commits with probability at least \(1 - \frac{1}{n^2}\).
Then we have the following lemma by using the same argument of Lemma 7 in [43].

**Lemma 20.** In algorithm **Randomized** all transactions commit by the end of their assigned frames with probability at least \( 1 - \frac{2}{n} \).

**Theorem 42.** Algorithm **Randomized** produces a schedule of makespan \( O((C \log n + \log^2 n)(\tau + D)) \) with probability at least \( 1 - \frac{2}{n} \). The competitive ratio of **Randomized** is \( O(s \cdot (C \log n + \log^2 n)) \).

**Proof.** The makespan follows from Lemma 20 immediately. To prove the competitive ratio, we know that

\[
\text{makespan}_{\text{Opt}} \geq \max_{1 \leq i \leq s} \left( \tau \cdot \gamma_i + \text{Tsp}(G_d(a_i)) \right) \geq \frac{\tau \cdot \sum_{1 \leq i \leq s} \gamma_i + \sum_{1 \leq i \leq s} \text{Tsp}(G_d(a_i))}{s}.
\]

Note that \( C \leq \sum_{1 \leq i \leq s} \gamma_i \) and \( \sum_{1 \leq i \leq s} \text{Tsp}(G_d(a_i)) \geq \text{Tsp}(G_d) \geq D \). The theorem follows.

**Randomized** directly adopts an existing randomized scheduling algorithm from multiprocessor STM to D-STM. When \( D \ll \tau \), the algorithm provides the same competitive ratio as **Online-Greedy** [43]. However, when \( D \) increases to \( \Theta(\tau) \), the competitive ratio is worsened by a \( O(C) \) factor. This is because past randomized scheduling algorithms do not consider the cost of moving objects between transactions. When a transaction is aborted, it immediately moves the object to the winning transaction without considering the moving cost. As the result, in D-STM where the moving cost dominates the local execution cost, **Randomized** is far from optimal.

### 8.2.2 Algorithm CUTTING

We present Algorithm **Cutting** (Algorithm 7), a randomized scheduling algorithm based on a partitioning constructed on the cost graph \( G_d \). To partition the cost graph, we first construct an approximation TSP path (ATSP path) in \( G_d \) \( \text{ATSP}_A(G_d) \) by selecting an approximation TSP algorithm \( A \). Specifically, \( A \) provides the approximation ratio \( \phi_A \), such that for any graph \( G \), \( \frac{\text{ATSP}_A(G)}{\text{Tsp}(G)} = O(\phi_A) \). Note that if \( d_{ij} \) satisfies the triangle inequality, the best known algorithm provides an \( O(\log m) \) approximation [98]; if \( d_{ij} \) is symmetric as well, a constant \( \phi_A \) is achievable [97]. We assume that a transaction knows partial knowledge in advance: a transaction \( T_i \) knows its required set of objects \( \vec{o}_i \) after starts. Therefore, a transaction can send all its requests of objects immediately after starts.
Algorithm 7: Algorithm CUTTING

Input: A set of transactions $T$; the conflict graph $G_c$; the cost graph $G_d$ with diameter $D$; each transaction has the same local execution time duration $\tau$; the approximation TSP algorithm $A$; a $(C, A)$ partitioning $\mathcal{P}(C, A, v)$ on $G_d$; each transaction $T_i$ knows $\delta_i$ after starts;

Output: An execution schedule of $T$.

procedure Abort $(T_1, T_2)$
Abort transaction $T_2$;
$T_2$ waits until $T_1$ commits or aborts;
end procedure

On (re)start of transaction $T_1$:
\[ \pi_1 \leftarrow \text{random integer in } [1, m]; \]
On conflict of transaction $T_1$ invoked at node $v^{j_1} \in P_t$, with transaction $T_2$ invoked at node $v^{j_2} \in P_t$:
\[ \text{if } t_1 = t_2 \text{ or } \exists \text{ integer } \nu \geq 1 \text{ s.t. } \left\lfloor \frac{\max(\psi(v^{j_1}), \psi(v^{j_2}))}{2^\nu} \right\rfloor = \min\{\psi(v^{j_1}), \psi(v^{j_2})\} \text{ then} \]
// one transaction is an ancestor of the other in $B_t(P_t)$
if $j_1 < j_2$ then Abort $(T_1, T_2)$; else Abort $(T_2, T_1)$;
else
\[ \text{if } \pi_1 < \pi_2 \text{ then Abort } (T_1, T_2); \text{ else Abort } (T_2, T_1); \]
end procedure

Based on the constructed ATSP path $\text{ATSP}_A$, we define the $(C, A)$ partitioning on $G_d$, which divides $G_d$ into $O(C)$ partitions. A constructed partition $P$ is a subset of nodes which satisfies either: 1) $|P| = 1$; or 2) for any pair of nodes $(v_i, v_j) \in P$, $d_{ij} \leq \frac{\text{ATSP}_A}{c}$.

Definition 36 $(C, A)$ partitioning. In the cost graph $G_d$, the $(C, A)$ partitioning $\mathcal{P}(C, A, v)$ divides $m$ nodes into $O(C)$ partitions in two phases.

Phase I Randomly select a node $v$ and let node $v^j$ be the $j^{th}$ node (excluding $v$) on the ATSP path $\text{ATSP}_A(G_d)$ starting from $v$. Hence $\text{ATSP}_A(G_d)$ can be represented by a sequence of nodes $\{v^0, v^1, \ldots, v^{m-1}\}$.

1. Starting from $v^0$, add $v^0$ to $P_1$ and set $P_1$ as the current partition.
2. Check $v^1$. If $\text{ATSP}_A(G_d)[v^0, v^1] \leq \frac{\text{ATSP}_A G_d}{c}$, where $\text{ATSP}_A(G_d)[v^1, v^2]$ is the length of the part of $\text{ATSP}_A(G_d)$ from $v^0$ to $v^1$, add $v^1$ to $P_1$. Else add $v^1$ to $P_2$ and set $P_2$ as the current partition.
3. Repeat Step 2 until all nodes are partitioned. For each node $v^k$ and the current partition $P_t$, it checks the length of $\text{ATSP}_A(G_d)[v^j, v^k]$, where $v^j$ is the first element added to $P_t$.\[ \text{If } \text{ATSP}_A(G_d)[v^j, v^k] \leq \frac{\text{ATSP}_A G_d}{c}, \text{ then } v^k \text{ is added to } P_t. \text{ Else } v^k \text{ is added to } P_{t+1} \text{ and } P_{t+1} \text{ is set as the current partition.} \]

Phase II Inside each partition $P_t = \{v^k, v^{k+1}, \ldots\}$, each node $v^k$ is assigned a partition index $\psi(v^j) = (j \mod k)$, i.e., its index inside the partition.

The conflict resolution is also two-phase. In the first phase, CUTTING assigns each transaction a partition index. When two transactions $T_1$ (invoked by node $v^{j_1}$) and $T_2$ (invoked by $v^{j_2}$) conflict, the algorithm first checks: 1) whether they are from the same partition $P_t$; 2) if so, whether $\exists$ integer $\nu \geq 1$ s.t. $\left\lfloor \frac{\max(\psi(v^{j_1}), \psi(v^{j_2}))}{2^\nu} \right\rfloor = \min\{\psi(v^{j_1}), \psi(v^{j_2})\}$. Note that
by checking these two conditions, an underlying binary tree $B_t(P_t)$ is constructed in $P_t$ as follows:

1. Set $v^{j_0}$ as the root of $B_t(P_t)$ (level 1), where $\psi(v^{j_0} = 0)$, i.e., the first node added to $P_t$.
2. Node $v^{j_0}$'s left pointer points to $v^{j_0+1}$ and right pointer points to $v^{j_0+2}$. Nodes $v^{j_0+1}$ and $v^{j_0+2}$ belongs to level 2.
3. Repeating Step 2 by adding nodes sequentially to each level from left to right. In the end $O(\log_2 m)$ levels are constructed.

Note that by satisfying these two conditions, the transaction with the smaller partition index must be an ancestor of the other transaction in $B_t(P_t)$. Therefore, a transaction may conflict with at most $O(\log_2 m)$ ancestors in this case. Cutting resolves the conflict greedily so that the transaction with smaller partition index always aborts the other transaction.

In the second phase, each transaction selects a integer $\pi \in [1, m]$ randomly when starts or restarts. If one transaction is not an ancestor of another transaction, the transaction with lower $\pi$ proceeds and the other transaction aborts. Whenever a transaction is aborted by a remote transaction, the requested object is moved to the remote transaction immediately.

### 8.2.3 Analysis

In the analysis below, we study two efficiency measures of Algorithm Cutting from the average-case perspective: the response time (how long it takes for an individual transaction to commit) and the makespan in average.

**Lemma 21.** A transaction $T$ needs $O(C \log^2 m \log n)$ trials from the moment it is invoked until commits in average case.

**Proof.** We start from a transaction $T$ invoked by the root node $v^\psi \in B_t(P_t)$. Since $v^\psi$ is the root node, transaction $T$ cannot be aborted by another ancestor in $B_t(P_t)$. Hence, $T$ can only be aborted when it chooses a larger $\pi$ than $\pi'$, the integer chosen by a conflicting transaction $T'$ invoked by node $v^{\psi'} \in P_t'$. The probability that for transaction $T$, no transaction $T' \in N_T$ selects the same random number $\pi' = \pi$ is

\[
\Pr(\exists T' \in N_T | \pi' = \pi) = \prod_{T' \in N_T} (1 - \frac{1}{m}) \geq (1 - \frac{1}{m})^{\delta(T)} \geq (1 - \frac{1}{m})^m \geq \frac{1}{e}.
\]

Note that $\delta(T) \leq C \leq m$. On the other hand, the probability that $\pi$ is at least as small as $\pi'$ for any conflicting transaction $T'$ is at least $\frac{1}{(C+1)}$. Thus, the probability that $\pi$ is the smallest among all its neighbors is at least $\frac{1}{e(C+1)}$. We use the following Chernoff bound:

**Lemma 22.** Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $\Pr(X_i = 1) = p_i$, where $0 \leq p_i \leq 1$. Then, for $X = \sum_{i=1}^{n} X_i$, $\mu = E[X] = \sum_{i=1}^{n} p_i$, and any $\delta \in (0, 1]$, $\Pr(X < (1 - \delta)\mu) < e^{-\delta^2 \mu/2}$. 

By Lemma 22, if we conduct $16e(C + 1)\ln n$ trials, each having success probability $\frac{1}{e(C+1)}$, then the probability that the number of successes $X$ is less than $8\ln n$ becomes: $\Pr(X < 8\ln n) < e^{-2\ln n} = \frac{1}{n^2}$.

Now we examine the transaction $T'$ invoked by node $v^{\psi'} \in P$, where $v^{\psi'}$ is the left child of the root node $v^\psi$ in $BT(P)$. When $T'$ conflicts with $T$, it aborts and holds off until $T$ commits or aborts. Hence, $T'$ can be aborted by $T$ at most $16e(C + 1)\ln n$ times with probability $1 - \frac{1}{n^2}$. On the other hand, $T'$ needs at most $16e(C + 1)\ln n$ to choose the smallest integer among all conflicting transactions with probability $1 - \frac{1}{n^2}$. Hence, in total $T'$ needs at most $32e(C + 1)\ln n$ trials with probability $(1 - \frac{1}{n^2})^2 > (1 - \frac{1}{n^2})$.

Therefore, by induction, the transaction $T^L$ invoked by a level-$L$ node $v^{\psi_L}$ of $BT(P)$ needs at most $(1 + \log_2 L)\log_2 L \cdot 8e(C + 1)\ln n$ trials with probability at least $1 - \frac{(1 + \log_2 L)\log_2 L}{2n^2}$.

Now we can calculate the average number of trials:

$$E[\# \text{ of trials a transaction needs to commit}] = O(C \log^2 m \log n),$$

since when the starting point of the ATSP path is randomly selected, the probability that an transaction is located at level $L$ is $1/2L_{\max} - L + 1$. The lemma follows.

\[\text{Lemma 23.} \quad \text{The average response time of a transaction is } O(C \log^2 m \log n) \cdot \left(\tau + \frac{\text{ATSP}_A}{C}\right).\]

\[\text{Proof.} \quad \text{From Lemma 21, each transaction needs } O(C \log^2 m \log n) \text{ trials in average. We now study the duration of a trial, i.e., the time until a transaction can pick a new random number. Note that a transaction can only pick a new random number after it is aborted (locally or remotely). Hence, if a transaction conflicts with a transaction in the same partition, the duration is at most } \tau + \frac{\text{ATSP}_A}{C}; \text{ if it conflicts with a transaction in another partition, the duration is at most } \tau + D. \text{ Note that a transaction sends its requests of objects simultaneously once after (re)starts. If a transaction conflicts with multiple transactions, the first conflicting transaction it knows is the transaction closed to it. From Lemma 21, a transaction can be aborted by a transaction out of its partition by at most } 16e(C + 1)\ln n \text{ times. Hence, the expected commit time of a transaction is } O(C \log^2 m \log n) \cdot \left(\tau + \frac{\text{ATSP}_A}{C}\right). \text{ The lemma follows.} \]

\[\text{Theorem 43.} \quad \text{Algorithm Cutting produces a schedule of average-case makespan } O(C \log^2 m \log n + \log^2 n) \cdot (\tau + \frac{\text{ATSP}_A}{C}). \text{ The average-case competitive ratio of Cutting is } O(s \cdot \phi_A \cdot (\log^2 m \log n + \log^2 n)).\]

\[\text{Proof.} \quad \text{We first find that } \text{makespan}_{opt} \geq \max_{1 \leq i \leq s} \left(\tau \cdot \gamma_i + \text{Tsp}(G_d(o_i))\right), \text{ since } \gamma_i \text{ transactions concurrently conflict on } o_i. \text{ Hence at one time only one of them can commit and object moves along a certain path to visit } \gamma_i \text{ transactions one after another. Then we have}

$$\text{makespan}_{opt} \geq \max_{1 \leq i \leq s} \left(\tau \cdot \gamma_i + \text{Tsp}(G_d(o_i))\right) \geq \frac{\tau \cdot \sum_{1 \leq i \leq s} \gamma_i}{s} + \frac{\sum_{1 \leq i \leq s} \text{Tsp}(G_d(o_i))}{s}. $$
Therefore, the competitive ratio of Cutting is

\[
\frac{\text{makespan}_{\text{opt}}}{\text{makespan}_{\text{cutting}}} = s \cdot (\log^2 m \log n + \log^2 n) \cdot \frac{\tau \cdot C + \text{ATSP}_A}{\tau \cdot \sum_{1 \leq i \leq s} \gamma_i + \sum_{1 \leq i \leq s} \text{Tsp}(G_d(o_i))}.
\]

Note that \( C \leq \sum_{1 \leq i \leq s} \gamma_i \) and \( \sum_{1 \leq i \leq s} \text{Tsp}(G_d(o_i)) \geq \text{Tsp}(G_d) \). The theorem follows.

\[\square\]

### 8.3 Conclusion

We present two algorithms for contention management in D-STM. While Randomized directly adopts existing randomized algorithms from multiprocessor STM, it exhibits a \( O(C) \) factor worse competitive ratio than multiprocessor STM randomized algorithms. Cutting provides an efficient competitive ratio from the average-case perspective. This is the first such results for the design of contention management algorithms for D-STM. It requires each transaction to be aware of its requested set of object when starts. This is essential in our algorithms since each transaction can send requests to objects simultaneously after starts. If we remove this restriction, the original results do not hold since a transaction can only send the request of an object once after the previous operation done. This increases the resulted makespan by at least \( O(s) \) factor.
Chapter 9

A Quorum-Based Replication D-STM Framework

Single copy (SC) D-STM proposals keep only one writable copy of each object in the system and are not fault-tolerant in the presence of network node failures in large-scale distributed systems. In this chapter, we propose a quorum-based replication (QR) D-STM model, which provides provable fault-tolerant property without incurring high communication overhead compared with SC model. QR model operates on an overlay tree constructed on a metric-space failure-prone network where communication cost between nodes forms a metric. QR model stores object replicas in a tree quorum system, where two quorums intersect if one of them is a write quorum, and ensures the consistency among replicas at commit-time. The communication cost of an operation in QR model is proportional to the communication cost from the requesting node to its closest read or write quorum. In the presence of node failures, QR model exhibits high availability and degrades gracefully when the number of failed nodes increases, with reasonable higher communication cost.

9.1 Motivation

In the aforementioned single copy D-STM model (or SC model), the main responsibility of CC protocols is to locate and move objects in the network. A directory-based protocol is often adopted such that the latest location of the object is saved in the distributed directory and the cost to locate and move an object is bounded. We now consider a distributed system where nodes are fail-stop [100] and communication links may also fail to deliver messages. Further, node failures may occur concurrently and lead to network partitioning failures, where nodes in a partition may communicate with each other, but no communication can occur between nodes in different partitions. A node may become inaccessible due to node or partitioning failures.
Since SC model only keeps a single writable copy of each object, it is inherently vulnerable in the presence of node failures. If a node failure occurs, the objects held by the failed node will be simply lost and all following transactions requesting such objects would never commit. Hence, SC model cannot afford any node failures. For example, BALLISTIC [26] assumes a reliable and fifo logical link between nodes, since they may not perform well when the message is reordered [18]. COMBINE [30] can tolerate partial link failures and support non-fifo message delivery, as long as a logical link exists between any pair of nodes. However, similar to other directory-based protocols, COMBINE does not permit network partitioning incurred by link failures, which may make some objects inaccessible from outer transactions. In general, SC model is not suitable in a network environment with aforementioned node failures.

To achieve high availability in the presence of node failures, keeping only one copy of each object in the system is not sufficient. Apart from lack of fault-tolerance property, SC model also suffers from some other limitations.

### 9.1.1 Limited Support of Concurrent Reads

Although directory-based CC protocols for SC model allows multiple read-only copies of an object existing in the system, these protocols lacks the explanation on how they maintain the consistency over read-only and writable copies of objects. Consider two transactions \(A\) and \(B\), where \(A\) contains operations \{\text{read}(o_1), \text{write}(o_2)\} and \(B\) contains operations \{\text{read}(o_2), \text{write}(o_1)\}. In SC model, the operations of \(A\) and \(B\) could be interleaved, e.g., transaction \(A\) reads \(o_1\) before \(B\) writes to \(o_1\), and transaction \(B\) reads \(o_2\) before \(A\) writes to \(o_2\). Obviously, transactions \(A\) and \(B\) conflict on both objects. In order to detect the conflict, each object needs to keep a record for any of its readers. When transaction \(A\) (or \(B\)) detects a conflict on object \(o_2\) (or \(o_1\)), it does not know: i) the type of the conflicting transaction (read-only or read/write); and ii) the status of the conflicting transaction (live/aborted/committed). It is not possible for a contention manager to make distributed agreement without these knowledge (e.g., it is not necessary to resolve the conflict between a live transaction and an aborted/committed transaction). To keep each object updated with the knowledge of its readers, a transaction has to send messages to all objects in its readset once after its termination (commit or abort). Unfortunately, in SC model such mechanism incurs high communication and message overhead, and it is still possible that a contention manager may make a wrong decision if it detects a conflict between the time the conflicting transaction terminated and the time the conflicting object receives the updated information, due to the relatively high communication latency.

Due to the inherent difficulties in supporting concurrent read operations, practical implementations of directory-based CC protocols often do not differentiate between a read and write operation of a read/write transaction (i.e., if a transaction contains both read and write operations, all its operations are treated as write operations and all its requested ob-
objects have to be moved). Such over-generalization obviously limits the possible concurrency of transactions. For example, in the scenario where the workload is composed of late-write transactions [18], a directory-based CC protocol cannot perform better than a simple serialization schedule, while the optimal schedule maybe much shorter when concurrent reads are supported for read/write operations.

### 9.1.2 Limited Locality

One major concern of directory-based CC protocols is to exploit locality in large-scale distributed systems, where remote access is often several orders of magnitude slower than local ones. Reducing communication cost and remote accesses is the key to achieving good performance for D-STM implementations. Existing CC protocols claim that the locality is preserved by their location-aware property: the cost to locate and move the objects between two nodes \( u \) and \( v \) is often proportional to the shortest path between \( u \) and \( v \) in the directory. In such a way directory-based CC protocols route transactions’ requests efficiently: if two transactions requests an object at the same time, the transaction “closer” to the object in the directory will get the object first. The object will be first sent to the closer transaction, then to the further transaction.

Nevertheless, it is unrealistic to assume that all transactions start at the same time. Even if two transactions start at the same time, since a non-clairvoyant transaction may access a sequence of objects, it is possible that a closer transaction may request to access an object much later than a further transaction. In such cases, transactions’ requests may not be routed efficiently by directory-based CC protocols. Consider two transactions \( A \) and \( B \), where \( A = \{ \langle \text{some work} \rangle, \text{write}(o) \} \) and \( B = \{ \text{write}(o), \langle \text{some work} \rangle \} \). Object \( o \) is located at node \( v \). Let \( d(v, v_A) = 1 \) and \( d(v, v_B) = d(v_A, v_B) = D \), it is possible that \( o \) first receives \( B’s \) request of \( o \). Assume that \( o \) is sent to \( B \) from \( v \), then the directory of \( o \) points to \( v_B \). Transaction \( A’s \) request of \( o \) is forwarded to \( v_B \) and a conflict may occur at \( v_B \). If \( B \) is aborted, the object \( o \) is moved to \( v_A \) from \( v_B \). In this scenario, object \( o \) has to travel at least \( 3D \) distance to let two transactions \( A \) commit. On the other hand, when object \( o \) receives \( B’s \) request at \( v \), if we let \( o \) waits for time \( t_o \) to let \( A’s \) request reach \( v \), then \( o \) could be first moved to \( v_A \) and then to \( v_B \). In this case, object \( o \) travels \( t_o + D + 1 \) distance to let two transactions commit. Obviously the second schedule may exploit more locality: as long as \( t_o \) is less than \( 2D - 1 \) (which is a quite loose bound), the object is moved more quickly.

In practice, it is often impractical to predict \( t_o \). As the result, directory-based CC protocols often overlook possible locality by simply keeping track of the single writable copy of each object. Such locality can be more exploited to reduce communication cost and improve performance.
9.2 Quorum-Based Replication D-STM Model

9.2.1 Overview

We present QR model, a quorum-based replication data-flow D-STM model, where multiple (writable) copies of each object are distributed at several nodes in the network. To perform a read or write operation, a transaction reads an object by reading object copies from a read quorum, and writes an object by writing copies to a write quorum. A quorum is assigned with the following restriction:

**Definition 37** (Quorum Intersection Property). A quorum is a collection of nodes. For any two quorums $q_1$ and $q_2$, where at least one of them is a write quorum, the two quorums must have a non-empty intersection: $q_1 \cap q_2 \neq \emptyset$.

Generally, by constructing a quorum system over the network, QR model is able to keep multiple copies of each object. QR model provides 5 operations for a transaction: read, write, request-commit, commit and abort. Particularly, QR model provides a request-commit operation to validate the consistency of its readset and writset before it commits. A transaction may request to commit if it is not aborted by other transactions before its last read/write operation. Concurrency control solely occurs during the request-commit operation: if a conflict is detected, the transaction may get aborted or abort the conflicting transaction. After collecting the response of the request-commit operation, a transaction may commit or abort.

We first present read and write operations of QR model in Algorithm 8. In following algorithms, notation “$msg \triangleright v$” is interpreted as “receiving $msg$ from node $v$”, and notation “$msg \triangleright v$” is interpreted as “sending $msg$ to node $v$”.

**Read.** When transaction $T$ at node $v$ starts a read operation, it sends a request message $req(T, \text{read}(o))$ to a selected read quorum $q_r$. The algorithm to find and select a read or write quorum will be elaborated in the next section. Node $v'$, upon receiving $req(T, \text{read}(o))$, checks whether it has a copy of $o$. If not, it sends a null response to $v$.

In QR model, each object copy contain three fields: the value field, which is the value of the object; the version number field, starting from 0, and the protected field, a boolean value which records the status of the copy. The protected field is maintained and updated by request-commit, commit and abort operations. Each object copy $o$ keeps a potential readers list $PR(o)$, which records the identities of the potential readers of $o$. Therefore, if $v'$ has a copy of $o$, it adds $T$ to $PR(o)$ and sends a response message $rsp(T, o) \triangleright v$, which contains a copy of $o$.

Transaction $T$ waits to collect responses until it receives all responses from a read quorum. Among all copies it receives, it selects the copy with the highest version number as the valid copy of $o$. The read operation finishes.
Algorithm 8: QR model: read and write

```plaintext
procedure Read (v, T, o)
    Local Phase:
    ReadQuorum (v, req(T, read(o)));
    wait until find(v) = true;
    foreach d ▹ v do
        if d.version > data(o).version then
            data(o) ← d;
        end if
    end foreach
    add o to T.readset;
    Remote Phase:
    Upon receiving req(T, read(o)) ▹ v;
    if data(o) exists then
        add T to PR(o);
        rsp(T, o) ▹ v;
    end if
```

```plaintext
procedure Write (v, T, o, value)
    Local Phase:
    ReadQuorum (v, req(T, write(o)));
    wait until find(v) = true;
    foreach d ▹ v do
        if d.version > data(o).version then
            data(o) ← d;
        end if
        dataCopy(o) ← data(o);
        dataCopy(o).value ← value;
        dataCopy(o).version ← data(o).version + 1;
    end foreach
    add o to T.writeset;
    Remote Phase:
    Upon receiving req(T, write(o)) ▹ v;
    if data(o) exists then
        add T to PW(o);
        rsp(T, o) ▹ v;
    end if
```

Write. The write operation is similar to the read operation. Transaction T sends a request message `req(T, write(o))` to a selected read quorum. Note that T does not need to send request to a write quorum because in this step it only needs to collect the latest copy of `o`. If a remote node `v'` has a copy of `o`, it adds `T` to `o`’s potential writers list `PW(o)` and sends a response message to `T` with a copy of `o`.

Transaction `T` selects the copy with the highest version number among the responses from a read quorum. Then it creates a temporary local copy (`dataCopy(o)`) and updates it with the value it intends to write, and increases its version number by 1 compared with the selected copy.

Remarks: The read and write operations of QR model are simple: a transaction just has to fetch all latest copies of required objects and perform all computations locally. Unlike a directory-based CC protocol, there is no need to construct and update a directory for each shared object. In QR model a transaction can always query its “closest” read quorum to locate the latest copy of each object required. Therefore the locality is preserved.
Algorithm 9: QR model: request-commit

```
procedure Request-Commit (v, T)
Local Phase:
WriteQuorum (v, req_cmt(T));
AT(T) ← ∅;
wait until find(v) = true;
if ∃rsp_cmt(T, abort) received then
    Abort (v, T);
else
    foreach rsp_cmt(T, cmt, CT(T)) do
        AT(T) ← AT(T) ∪ CT(T);
    Commit (v, T);
Remote Phase:
Upon receiving req_cmt(T) < v;
if CT(T) = ∅ then
    rsvp_cmt(T, cmt, CT(T)) ⊳ v;
else
    CM (T, CT(T));
    if CT(T) ≠ ∅ then
        rsvp_cmt(T, cmt, CT(T)) ⊳ v;
    if abort(T) = false then
        foreach o_T ∈ T. writerset do
            o_T.protected ← true;
        yo, remove T from PR(o) and PW(o);
procedure Conflict-Detect (v, T)
foreach o_T ∈ T. readset ∪ T. writerset of object o do
    if data(o). protected = true or
        data(o). version > o_T. version
        then
            abort(T) ← true;
            rsvp_cmt(T, abort) ⊳ v;
        break;
    if data(o). version = o_T. version then
        if data(o). value ≠ o_T. value then
            abort(T) ← true;
            rsvp_cmt(T, abort) ⊳ v;
            break;
        else
            add PW(o) to CT(T);
            if o_T ∈ T. writerset then
                add PR(o) to CT(T);
procedure CM (T, CT(T))
foreach T' ∈ CT(T) do
    if T' ≠ T then
        abort(T) ← true;
        rsvp_cmt(T, abort) ⊳ v;
        CT(T) ← ∅;
        break;
```

If a transaction is not aborted (by any other transaction) during all its read and operations, the transaction can request to commit by requesting to propagate its changes to objects into the system. The concurrency control mechanism is needed when any non-consistent status of an object is detected. The request-commit operation is presented in Algorithm 9.

**Request-Commit.** When transaction T requests to commit, it sends a message req_cmt(T) (which contains all information of its readset and writerset) to a write quorum q_w. Note that it is required that for each transaction T, and ∀q_r, q_w selected by T, q_r ⊆ q_w.

In the remote phase, when node v' receives the message req_cmt(T), it immediately removes T from its potential read and write lists of all objects and creates an empty conflicting transactions list CT(T) which records the transactions conflicting with T. Node v' determines the conflicting transactions of T in the following manner:
1) if o_T. protected = true, then T must be aborted since o_T is waiting for a possible update;
2) if o_T is a copy read or written by T of object o, and the local copy of o at v' (data(o)) has the higher version than o_T, then T reads a stale version of o. In this case, T must be aborted.
3) if o_T is a copy read by T of object o, and the local copy of o at v' (data(o)) has the same
version with \( o_T \), then \( T \) conflicts with all transactions in \( PW(o) \) (potential writers of object copy \( data(o) \)).

4) if \( o_T \) is a copy written by \( T \) of object \( o \), and the local copy of \( o \) at \( v' \) (\( data(o) \) has the same version with \( o_T \), then \( T \) conflicts with all transactions in \( PW(o) \cup PR(o) \) (potential readers and writers of \( data(o) \)).

The contention manager at \( v' \) compares priorities between \( T \) and its conflicting transactions (line 21). If \( \forall T'' \in CT(T), T < T'' \) (\( T \) has the higher priority than any of its conflicting transactions), \( T \) is allowed to commit by \( v' \). Node \( v' \) sends a message \( rs_p.cmt(T, cmt, CT(T)) \) with \( CT(T) \) to \( v \) and sets the status of \( data(o) \) as \( protected \), for any \( o \in T.writeset \). If \( \exists T'' \in CT(T) \) such that \( T'' < T \), then \( T \) is aborted. Node \( v' \) sends \( rs_p.cmt(T, abort) \) to \( v \) and resets \( CT(T) \).

In the local phase, transaction \( T \) collects responses from all nodes in the write quorum. If any \( rs_p.cmt(T, abort) \) message is received, \( T \) is aborted. If not, \( T \) can proceed to the commit operation. In this case, transaction \( T \) saves conflicting transactions from all responses into an aborted transactions list \( AT(T) \).

**Algorithm 10:** QR model: commit and abort

```plaintext
1 procedure Commit (v, T)
2 Local Phase:
3 foreach object o ∈ T.writeset do
4     data(o) ← dataCopy(o);
5 foreach T' ∈ AT(T) do
6     req_abt(T') > T'?
7 WriteQuorum (v, commit(T));
8     wait until find(v) = true;
9 Remote Phase:
10 Upon receiving commit(T):
11 foreach o_T ∈ T.writeset of object o do
12     data(o) ← o_T;
13     data(o).protected ← false;
14 Upon receiving req_abt(T'):
15 Abort (v', T')
```

**Remarks:** For each transaction \( T \), its concurrency control mechanism is carried by the request-commit operation. Therefore, the request-commit operation must guarantee that all existing conflicts with \( T \) are detected. Note that a remote node makes this decision based on its potential read and write lists. Therefore, these lists must be efficiently updated: a terminated transaction must be removed from these lists to avoid an unnecessary conflict detected. By letting \( q_r \subseteq q_w \) for all \( q_r \) and \( q_w \) selected by the same transaction \( T \), QR model guarantees that all \( T \)’s records in potential read and write lists are removed during \( T \)’s request-commit operation.

On the other hand, if \( v' \) allows \( T \) proceed to commit, then \( v' \) needs to protect local object copies written by \( T \) from other accesses until \( T \)’s changes to these objects propagate to \( v' \).
These objects copies become valid only after receiving $T$’s commit or abort information. We describe $T$’s commit and abort operations in Algorithm 10.

**Commit.** When $T$ commits, it sends a message $commit(T)$ to each node in the same write quorum $q_w$ as the one selected by the request-commit operation. Meanwhile, it sends a request-abort message $req_{abt}(T')$ for any $T' \in AT(T)$. In the remote phase, when a node $v'$ receives $commit(T)$, for any $o \in T.writeset$, it updates $data(o)$ with the new value and version number, and sets $data(o).protected = false$. If a transaction $T'$ receives $req_{abt}(T')$, it aborts immediately.

**Abort.** A transaction may abort in two cases: after the request-commit operation, or receives a request-abort message. When $T$ aborts, it rolls back all its operations of local objects. Meanwhile, it sends a message $abort(T)$ to each node in the write quorum $q_w$ (which is the same as the write quorum selected by the request-commit operation). Then transaction $T$ restarts from the beginning. In the remote phase, when a node $v'$ receives $abort(T)$, it removes $T$ from any of its potential read and write list (if it has not done so), and sets $data(o).protected = false$ for any $o \in T.writeset$.

### 9.2.2 Quorum Construction: Flooding Protocol

One crucial part of QR model is the construction of a quorum system over the network. We adopt the hierarchical clustering structure similar to the one described in [26]. An overlay tree with depth $L$ is constructed. Initially, all physical nodes are leaves of the tree. Starting from the leaf nodes at level $l = 0$, parent nodes at the immediate higher level $l + 1$ is elected recursively so that their children are all nodes at most at distance $2^l$ from them.

Our quorum system is motivated by the classic tree quorum system [46]. On the overlay tree, a quorum system is constructed by FLOODING protocol such that each constructed quorum is a valid tree quorum.

We present FLOODING protocol in Algorithm 11. For each node $v$, when the system starts, a basic read quorum $Q_r(v)$ and a basic write quorum $Q_w(v)$ are constructed by BASICQUORUMS method. The protocol tries to construct $Q_r(v)$ and $Q_w(v)$ by first putting root into these quorums and setting a distance variable $\delta$ to $d(v, root)$. Starting from level $= L - 1$, the protocol recursively selects the majority of descendants $levelHead = closestMajority(v, parent, level)$ for each parent selected in the previous level ($l + 1$), so that the distance from $v$ to $closestMajority(v, parent, level)$ is the minimum over all possible choices. Note that $closestMajority(v, parent, level)$ only contains parent’s descendants at level $level$. We define the distance from $v$ to a quorum $Q$ as: $d(v, Q) := \max_{v' \in Q} d(v, v')$. The basic write quorum $Q_w(v)$ is constructed by including all selected nodes.
### Algorithm 11: Flooding Protocol

1. **procedure** `BasicQuorums` \((v, \text{root})\)
   
   ```
   \delta \leftarrow d(v, \text{root});
   \text{Q}_w(v) \leftarrow \{\text{root}\};
   \text{Q}_r(v).\text{level} \leftarrow L;
   \text{currentHead} \leftarrow \{\text{root}\};
   
   \textbf{for} \ level = L - 1, L - 2, \ldots, 0 \ \textbf{do}
   
   \text{levelHead} \leftarrow \emptyset;
   
   \textbf{foreach} \ parent \in \text{currentHead} \ \textbf{do}
   
   \text{new} \leftarrow \text{closestMajority} (v, \text{parent}, \text{level});
   \text{add new to Q}_w(v);\n   \text{add new to levelHead};
   
   \textbf{if} \ d(v, \text{levelHead}) < \delta \ \textbf{then}
   
   \text{Q}_r(v).\text{level} \leftarrow \text{level};
   \delta \leftarrow d(v, \text{levelHead});
   
   \text{currentHead} \leftarrow \text{levelHead};
   ```

2. **procedure** `WriteQuorum` \((v, \text{msg})\)
   
   ```
   \text{msg} \triangleright \text{Q}_w(v);
   \textbf{if} \ v' \in \text{Q}_r(v) \ \textbf{is} \ \text{down} \ \textbf{then}
   
   \text{find} \leftarrow \text{false};
   \text{validAns} \leftarrow \emptyset;
   \text{validLevel} \leftarrow \emptyset;
   
   \textbf{for} \ level = 1, \ldots, L \ \textbf{do}
   
   \text{msg} \triangleright \text{ancestor} (v, \text{level});
   
   \textbf{if} \ \text{ancestor} (v, \text{level}) \ \text{is} \ \text{up} \ \textbf{then}
   
   \text{find} \leftarrow \text{true};
   \textbf{break};
   
   \textbf{if} \ \text{find} = \text{false} \ \textbf{then}
   
   \text{DownProbe} (v', \text{Q}_r(v).\text{level}, \text{read});
   
   \text{restart} \ \text{ReadQuorum} (v, \text{msg});
   ```

3. **procedure** `ReadQuorum` \((v, \text{msg})\)
   
   ```
   \text{msg} \triangleright \text{Q}_r(v);
   \text{find} \leftarrow \text{false};
   
   \textbf{if} \ v' \in \text{Q}_r(v) \ \textbf{is} \ \text{down} \ \textbf{then}
   
   \text{find} \leftarrow \text{false};
   
   \textbf{if} \ v' \neq \text{root} \ \textbf{then}
   
   \textbf{for} \ level = [\text{Q}_r(v).\text{level} + 1, L] \ \textbf{do}
   
   \text{msg} \triangleright \text{ancestor} (v, \text{level});
   
   \textbf{if} \ \text{ancestor} (v, \text{level}) \ \text{is} \ \text{up} \ \textbf{then}
   
   \text{find} \leftarrow \text{true};
   \textbf{break};
   
   \textbf{if} \ \text{find} = \text{false} \ \textbf{then}
   
   \text{DownProbe} (v', \text{Q}_r(v).\text{level}, \text{read});
   
   \text{restart} \ \text{ReadQuorum} (v, \text{msg});
   ```

4. **procedure** `DownProbe` \((v, \text{validLevel}, \text{type})\)
   
   ```
   \text{curRdHead} \leftarrow v;
   \text{curWtHead} \leftarrow v;
   \text{noWriteQ} \leftarrow \text{false};
   
   \textbf{for} \ level = [\text{validLevel} - 1, 0] \ \textbf{do}
   
   \text{levelRdHead} \leftarrow \emptyset;
   \text{levelWtHead} \leftarrow \emptyset;
   
   \textbf{foreach} \ parent \in \text{curRdHead} \ \textbf{do}
   
   \text{msg} \triangleright \text{descend} (\text{parent}, \text{level}) \cap \text{Q}_w(v);
   
   \textbf{if} \ w \ \text{is} \ \text{down} \ \textbf{then}
   
   \text{add} w \ \text{to} \ \text{levelRdHead};
   
   \textbf{if} \ \text{type} = \text{write} \ \textbf{then}
   
   \textbf{foreach} \ parent \in \text{curWtHead} \ \textbf{do}
   
   \textbf{if} \exists \text{newSet} = \text{closestMajority} (v, \text{parent}, \text{level}) \ \textbf{then}
   
   \text{msg} \triangleright \text{newSet};
   \text{add newSet to levelWtHead};
   
   \textbf{else}
   
   \text{noWriteQ} \leftarrow \text{true};
   \textbf{break};
   
   \textbf{if} \ \text{noWriteQ} = \text{true} \ \textbf{then}
   
   \textbf{break};
   
   \textbf{if} \ \text{levelRdHead} = \emptyset \ \text{and} \ \text{type} = \text{read} \ \textbf{then}
   
   \text{find} \leftarrow \text{true};
   \textbf{break};
   
   \textbf{else}
   
   \text{curRdHead} \leftarrow \text{levelRdHead};
   \text{curWtHead} \leftarrow \text{levelWtHead};
   
   \textbf{if} \ \text{noWriteQ} = \text{false} \ \text{and} \ \text{type} = \text{write} \ \textbf{then}
   
   \text{find} \leftarrow \text{true};
   ```
At each level, after a set of nodes levelHead has been selected, the protocol checks the distance from v to levelHead (d(v, levelHead)). If d(v, levelHead) < \delta, then the protocol replaces \( Q_r(v) \) with levelHead and sets \( \delta \) to d(v, levelHead). If d(v, levelHead) \( \geq \delta \), the protocol continues to the next level. At the end, \( Q_r(v) \) contains a set of nodes from the same level, which is the levelHead closest from v for all levels.

When node v requests to access a read quorum, the protocol invokes ReadQuorum(v, msg) method. Initially, node v sends msg to every node in \( Q_r(v) \). If all nodes in \( Q_r(v) \) are accessible from v, then a live read quorum is found. If any node \( v' \) in \( Q_r(v) \) is down, then the protocol needs to probe \( v' \)'s substituting nodes sub(\( v' \)) such that sub(\( v' \)) \( \cup \) \( Q_r(v) \) \( \setminus \) \( v' \) still forms a read quorum.

The protocol first finds if there exists any \( v' \)'s ancestor available. If so, \( v' \)'s substituting node has been found. If not, the protocol probes downwards from \( v' \) to check if there exists \( v' \) substituting nodes such that a constructed read quorum is a subset of \( Q_w(v) \) by calling DownProbe method.

The protocol invokes WriteQuorum(v, msg) method when node v requests to access a write quorum. Similar to ReadQuorum(v, msg), node v first sends msg to every node in \( Q_w(v) \). If any node \( v' \) is down, then the protocol first finds if there is a live ancestor of \( v' \) (validAns(\( v' \))). Starting from validAns(\( v' \)), the protocol calls DownProbe to probe downwards.

DownProbe method works similarly as BasicQuorums by recursively probing an available closest majority set of descendants for each parent selected in the previous level. By adopting DownProbe method, Flooding protocol guarantees that ReadQuorum and WriteQuorum can always probe an available quorum if at least one live read (or write) quorum exists in the network.

9.2.3 Analysis

We first analyze the properties of the quorum system constructed by Flooding, then we prove the correctness and evaluate the performance of QR model.

**Lemma 24.** Any read quorum \( q_r \) or write quorum \( q_w \) constructed by Flooding is a classic tree quorum defined in [46].

**Proof.** From the description of Flooding, we know that for a tree of height \( h + 1 \),

\[
q_r = \{\text{root}\} \lor \{\text{majority of read quorums for subtrees of height } h\},
\]

\[
q_w = \{\text{root}\} \cup \{\text{majority of write quorums for subtrees of height } h\}.
\]

From Theorem 1 in [46], the lemma follows. \( \square \)
Then we immediately have the following lemma.

**Lemma 25.** For any two quorums \( q_1 \) and \( q_2 \) constructed by FLOODING, where at least one of them is a write quorum, \( q_r \cap q_w \neq \emptyset \).

**Lemma 26.** For any read quorum \( q_r(v) \) and write quorum \( q_w(v) \) constructed by FLOODING for node \( v \), \( q_r(v) \subseteq q_w(v) \).

**Proof.** The theorem follows from the description of FLOODING. If no node fails, the theorem holds directly since \( Q_r(v) \subseteq Q_w(v) \).

If a node \( v' \notin Q_r(v) \) fails, then \( q_r(v) = Q_r(v) \). If \( v' \in Q_w(v) \), FLOODING detects that \( v' \) is not accessible when it calls WRITEQUORUM method. If \( \text{level}(q) \geq Q_r(v).\text{level} \), then FLOODING adds \( Q_r(v) \) to \( q_w(v) \) and starts to probe \( v' \)'s substituting nodes; if \( \text{level}(v') < Q_r(v).\text{level} \), then the level of \( v' \)'s substituting node is at most \( Q_r(v).\text{level} \) and then the protocol starts to probe downwards. In either case, \( q_r(v) \subset q_w(v) \).

If a node \( v' \in Q_r(v) \) fails, then FLOODING detects that \( v' \) is not accessible when it calls READQUORUM or WRITEQUORUM method. Both methods start to probe \( v' \)'s substituting nodes from \( v' \). When probing upwards, \( v' \)'s ancestors are visited. If a live \( \text{ancestor}(v') \) is found, then both methods add \( \text{ancestor}(v') \) to the quorum. Then READQUORUM stops and WRITEQUORUM continues probing downwards from \( \text{ancestor}(v') \). The theorem follows.

With the help of Lemmas 25, we have the following theorem.

**Theorem 44.** QR model provides 1-copy equivalence for all objects.

**Proof.** We first prove that for any object \( o \), if at time \( t \), no transaction requesting \( o \) is propagating its change to \( o \) (i.e., in the commit operation), then all transactions accessing \( o \) at \( t \) get the same copy of \( o \).

Note that if any committed transaction writes to \( o \) before \( t \), there exists a write quorum \( q_w \) such that \( \{ \forall v \in q_w \} \land \{ \forall v' \notin q_w \} \), \( \text{data}(o,v).\text{version} > \text{data}(o,v').\text{version} \). If any transaction \( T \) accesses \( o \) at time \( t \), it collects a set of copies from a read quorum \( q_r \). From Lemma 25, \( \exists v \in \{ q_w \cap q_r \} \) such that \( \text{data}(o,v) \) is collected by \( T \). Note that read and write operations select the object copy with the highest version number. Hence, for any transaction \( T \), \( \text{data}(o,v) \) is selected as the latest copy.

We now prove that for any object \( o \), if at time \( t \): 1) a transaction \( T \) is propagating its change to \( o \); and 2) another transaction \( T' \) accesses a read quorum \( q_r \) before \( T \)'s change propagates to \( q_r \), then \( T' \) will never commit.

Note that in this case, \( T' \) reads a stale version \( o_{T'} \) of \( o \). When it requests to commit (if it is not aborted before that), it sends the request to a write quorum \( q_w \). Then \( \exists v \in q_w \), such that: 1) \( T \)'s change of \( o \) still has not propagated to \( v \) and \( \text{data}(o,v).\text{protected} = \text{true} \); or
2) T’s change has been applied to data(o, v) and data(o, v).version > o_T.version. In either case, T is aborted by CONFLICT-DETECT method.

As the result, at any time, the system exhibits that only one copy exists for any object and transactions observing an inconsistent state of object never commit. The theorem follows.

We can prove that QR model provides one-copy serializability [101].

**Theorem 45.** QR model implements one-copy serializability.

**Proof.** The theorem follows from Lemma 26 and Theorem 44,

QR model provides five operations and every operation incurs a remote communication cost. We now analyze the communication cost of each operation.

**Theorem 46.** If a live read quorum q_r(v) exists, the communication cost of a read or write operation that starts at node v is O(k · d(v, q_r(v))) for k ≥ 1, where k is the number of nodes failed in the system. Specifically, if no node fails, the communication cost is O(d(v, Q_r(v))).

**Proof.** For a read or write operation, the transaction calls READQUORUM method to collect the latest value of the object from a read quorum. If no node fails, the communication cost is 2d(v, Q_r(v)). If a node v’ ∈ Q_r(v) fails, the transaction needs to probe v’’s substituting nodes to construct a new read quorum. The time for v to restart the probing is at most 2d(v, Q_r(v)). Note that ∀q_r(v), d(v, Q_r(v)) ≤ d(v, q_r(v)).

In the worst case, if k nodes fail and v detects only one failed node at each it accesses a read quorum, at most k rounds of probing are needed for v to detect a live read quorum. On the other hand, v always starts probing from the closest possible read quorum. Therefore for each round, the time for v to restart the probing is at most 2d(v, q_r(v)). The theorem follows.

**Theorem 47.** If a live write quorum q_w(v) exists, the communication cost of a request-commit, commit or abort operation that starts at node v is O(k · d(v, q_w(v))) for k ≥ 1, where k is the number of nodes failed in the system. Specifically, if no node fails, the communication cost is O(v, Q_w(v)).

**Proof.** Similar to Theorem 46, the communication cost of other three operation can be proved in the same way. The theorem follows from the same argument of the proof of Theorem 46.

Theorems 46 and 47 illustrate the advantage of exploiting locality for QR model. For read and write operations starting from v, the communication cost is only related to the distance from v to its closest read quorum. If no node fails, the communication cost is bounded by
2d(v, Q_r(v)). Note that d(v, Q_r(v)) \leq d(v, \text{root}) from the construction of Q_r(v). On the other hand, the communication cost of other three operations is bounded by O(v, Q_w(v)). Since each transaction involves at most two operations from \{\text{request-commit, commit, abort}\}, when the number of read/write operations increases, the communication cost of a transaction only increases proportional to d(v, Q_r(v)). Compared with directory-based protocols, the communication cost of a operation in QR model is not related to the stretch provided by the underlying overlay tree.

When the number of failed nodes increases, the performance of each operation degrades linearly. In QR model, it is crucial to analyze the availability of the constructed quorum system. From the construction of the quorum system we know that if a live quorum exists, FLOODING protocol can always probe it. Let p be the probability that node lives and R_h be the availability of a read quorum, i.e., at least one live read quorum exists in a tree of height h. Then we have the following theorem.

**Theorem 48.** Assuming the degree of each node in the tree is at least 2d + 1, the availability of a read quorum is

\[
R_{h+1} \geq p + (1 - p) \cdot \left[ \left( \frac{2d}{d+1} \right) (R_h)^{d+1} (1 - R_h)^d + \left( \frac{2d}{d+2} \right) (R_h)^{d+2} (1 - R_h)^{d-1} + \ldots + (R_h)^{2d} (1 - R_h) \right]
\]

**Proof.** From [46], we have

\[
R_h = \text{Prob\{Root is up\}} + \text{Prob\{Root is down\}} \times \text{[Read Availability of Majority of Subtrees]}.
\]

Note that in our overlay tree, if a node v at level h + 1 is down, then one of its descendants at h is also down for h \geq 0, because they are mapped to the same physical node. The theorem follows.

Similarly, let W_h be the availability of a write quorum in a tree of height h, then

**Theorem 49.**

\[
W_{h+1} \geq p \cdot \left[ \left( \frac{2d}{d+1} \right) (W_h)^{d+1} (1 - W_h)^d + \left( \frac{2d}{d+2} \right) (W_h)^{d+2} (1 - W_h)^{d-1} + \ldots + (W_h)^{2d} (1 - W_h) \right]
\]

**Proof.** The theorem follows from the same argument of the proof of Theorem 48.

Initially, R_0 and W_0 is p (only the root exists). Theorems 48 and 49 provide the recurrence relations of R_h and W_h, which can be used to calculate specific tree configurations. As the result, FLOODING provides the availability similar to the classic tree quorum system in [46].
9.3 Conclusion

QR model requires that at least one read and one write quorums live in the system. If no live read (or write) quorum exists, FLOODING protocol cannot proceed after READQUORUM (or WRITEQUORUM) operation. In this case, a reconfiguration of the system is needed to rebuild a new overlay tree structure. Each node then runs FLOODING protocol to find their new basic read and write quorums.

QR model exhibits graceful degradation in a failure-prone network. In a failure-free network, the communication cost imposed by QR model is comparable with SC model. When failures occur, QR model continues executing operations with high probability and reasonable higher communication cost. Such property is especially desirable for large-scale distributed systems in the presence of failures.
Chapter 10

Prototype Implementation and Experimental Results

We implement a set of CC protocols and conflict resolution strategies for a comprehensive comparison based on HyFlow framework [47]. HyFlow is a Java D-STM framework that provides pluggable support for different combinations of design options. HyFlow exports a simple distributed programming model that excludes locks: atomic sections are defined as transactions using (Java 5) annotations. Inside an atomic section, reads and writes to shared, local and remote objects appear to take effect instantaneously. No changes are needed to the underlying virtual machine or compiler.

10.1 HyFlow D-STM framework

We show the architecture of HyFlow in Figure 10.1. HyFlow provides pluggable support for cache coherence protocols, transactional synchronization and recovery mechanisms, conflict resolution strategies, and network communication protocols. A HyFlow runtime handler and five modules form the basis of the architecture, including Transaction Manager, Instrumentation Engine, Locator, Transaction Validation Module, and Communication Manager.

The HyFlow runtime handler represents a standalone entity that delegates application-level requests to the framework. HyFlow uses run-time instrumentation to generate transactional code, like other (multiprocessor) STM such as Deuce [102], yielding almost two orders of magnitude superior performance than reflection-based STM (e.g., DSTM2 [2]).

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

The Instrumentation Engine modifies class code at runtime, adds new fields, and modifies
annotated methods to support transactional behavior. Further, it generates callback functions that work as “hooks” for Transaction Manager events such as onWrite, beforeWrite, beforeRead, etc.

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

The Locator has three main tasks: 1) providing access to the object owned by the current node, 2) locating and sending access requests to remote objects, and 3) retrieving any required object meta-data (e.g., latest version number). Objects are located with their IDs using the Directory Manager, which encapsulates a CC protocol. Upon object creation, the Directory Manager is notified and publishes the object in the network. The CC protocol is called when to locate a remote object, or to move an object to a remote node.

The Transaction Validation Module ensures data consistency by validating transactions upon their completion. An encapsulated conflict resolution module is consulted when conflicts occur.

10.2 Benchmarks

We developed a suite of macro and microbenchmarks to evaluate the performance of D-STM, including distributed version of Bank, Loan and Vacation benchmarks similar to the STAMP benchmark suite [36], which was rewritten with HyFlow’s distributed STM APIs. We also developed a suite of microbenchmarks for comparison.
Bank Benchmark. We built a distributed version of a banking application, which maintains a set of accounts distributed over bank branches. Two kinds of atomic transactions are implemented: transfer transaction, which transfers a given amount between two accounts, and total balance transaction, which computes the total balance for given accounts. Figure 10.5 shows an example transactional code of transfer transaction in HyFlow, in which two bank accounts are accessed and an amount is atomically transferred between them.

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

The code is self-maintained and no lock is used at the programming level. Atomicity, consistency, and isolation are guaranteed (for the transfer operation). Composability is also achieved: the atomic withdraw and deposit methods have been composed into the higher-level atomic transfer operation. A conflicting transaction is transparently retried. Note that the location of the bank account is hidden from the program. It may be cached locally, migrate to the current node, or accessed remotely using remote calls, which is transparently accomplished by HyFlow.

Loan Benchmark. Loan is a simple money transfer application, in which a set of asset (i.e., with monetary value) holders is distributed over nodes. A loan transaction is configured by two parameters: 1) branching, and 2) nesting. In a loan transaction, an account issues a loan request to one or more other accounts, determined by the branching parameter. The request recipient forwards this request to other accounts, and this propagation continues till the level determined by the nesting parameter is reached. An entire loan request takes effect atomically, by virtue of the request being implemented as an atomic transaction. The number of inter-node remote object calls grow exponentially with the number of participating objects (i.e., the nesting level). For example, for a single transaction, 20 inter-node calls occur for a nesting level of 6, and 376 inter-node calls occur for a nesting level of 12. Loan involves significant number of remote calls that grows exponentially with nesting.
Vacation Benchmark. This is a distributed version of the STAMP [36] STM vacation benchmark application. This application implements an online transaction processing system that tracks customer reservations for air travel. The distributed object here is the customer and trip records. The system supports three types of transactions: reservation, cancelation, and update. These transactions are read/write, as they change the object. However, another administrative transaction, which simply prints-out all current reservations, is provided to assess read-only transactions.

Microbenchmarks. Microbenchmarks include implementations of a distributed linked list, a distributed binary search tree and a distributed red-black tree. Nodes are located at different nodes, while links between nodes are replaced by keys of neighbor nodes. Element manipulation operations are defined as transactions — e.g., add, remove, and contains.

10.3 Candidate CC Protocols and Conflict Resolution Strategies

10.3.1 CC Protocols

We mainly implement four CC protocols in HyFlow: Arrow [49], Ballistic [26], Relay and Home. We introduced Arrow and Relay in Chapter 6.

Ballistic operates on a hierarchical clustering of a metric-space network, where the communication cost between nodes forms the metric. The hierarchical architecture is constructed based on a cost metric between nodes: two nodes $x$ and $y$ are connected at level $l$ if and only if $d(x, y)$ (the cost between $x$ and $y$) is less than $2^{l+1}$, where the smallest cost between two nodes in the network is normalized to 1. Figure 10.3 shows an example hierarchical clustering, where the connectivity of a node at three levels is illustrated. By adopting this procedure for every node, the final hierarchical clustering graph can be calculated.

A directory hierarchy is constructed based on the hierarchical clustering: at each level $l$, a maximal independent set of the graph is selected as Leader$^l$, and at level $l + 1$, only nodes from Leader$^l$ join the connectivity graph. For a node $x$ at level $l$, its home parent home$(x)$ is the node closest to $x$ at level $l + 1$. Further, home$^l(x)$ is the level-$l$ home directory of $x$, where home$^0(x) = x$ and home$^i(x)$ is the home parent of home$^{i-1}(x)$. As shown in Figure 10.4, the solid lines correspond to home links. A move parent set of $x$ of level $l$ is defined as the subset of nodes at level $l + 1$ within distance $4 \cdot 2^{l+1}$ of $x$ (thick lines). A directory formed by home links always point to the object (solid arrows).

Assume that node $A$ sends a request of object $o_i$ to the directory. At each level $l$, home$^l(A)$ sends requests to its move parent set (send$(i)(r_A)$), and sets home$^l(A)$’s local link pointing to home$^{l-1}(A)$ (redirect$(i)(r_A)$, dashed arrows), until it discovers a downward link. When a downward link is discovered, the protocol follows the chain of downward links and sets each
downward link to null ($rmvptr(i)(T_A)$, dashed lines). Then the request from $A$ is queued at the object’s location (node $B$), and the object will be sent to $A$ immediately or after some time ($mvobj(i)(T_A)$). Hence, based on BALLISTIC, we can implement its DQCC and DPQCC versions, following the description of Chapter 5. In the following section, we use B-DQCC and B-DPQCC to denote the DQCC and DPQCC versions of BALLISTIC, respectively.

**HOME** is centralized directory protocol similar to Jackal [48]. **HOME** uses a client-server scheme to locate and move objects: each object’s location is updated in a server (or a set of servers). A node sends a request of an object by first consulting the server. Whenever an object is moved, its corresponding record in the server is updated.

### 10.3.2 Conflict Resolution Strategies

We implemented the DDA model introduced in Chapter 7. We also implemented **Greedy** (introduced in Chapter 4), **KARMA** [50] and **KINDERGARTEN** [50] contention managers for a comprehensive evaluation.

**KARMA.** KARMA contention manager aborts the transaction, which has performed the least amount of work when a conflict occurs.

**KINDERGARTEN.** KINDERGARTEN contention manager encourages transactions to take turns
accessing an object by maintaining a hit list (initially empty) of enemy transactions in favor of which a thread has previously aborted.

10.4 Testbed

We conducted our experiments on a network comprising of 24 nodes, each of which is an Intel Xeon 1.9GHz processor, running Ubuntu Linux, and interconnected by message passing links with bounded link delay. In our experiments, the communication cost corresponds to the network latency between nodes. We set the smallest cost to 1 ms and randomly generate a metric-space network within the diameter (the largest latency between nodes) configured to 20 ms. We ran the experiments using a one processor per node configuration, to eliminate the contention between two processors of the same node, which is not the focus of D-STM.

10.5 Experimental Results

10.5.1 Evaluation of DQCC and DPQCC Protocols

![Graphs showing performance metrics](image-url)

(a) Throughput of Bank

(b) Average number of aborts per transaction of Bank

Figure 10.5: Performance of Bank benchmark under HOME, B-DQCC, and B-DPQCC protocols.

To evaluate DQCC and DPQCC protocols, we used BALLISTIC [26] as our underlying distributed queuing protocol and implemented its DQCC and DPQCC versions: B-DQCC and BPQCC, respectively. We implemented GREEDY and KARMA as the underlying contention managers. To evaluate the performance of the CC protocols under the Bank Benchmark, we distributed 10 shared bank accounts (i.e., objects) uniformly across 24 nodes and executed
100 transactions at each node. Under this setting, we measured the transactional throughput for each CC protocol.

Figure 10.5(a) shows the throughput of the Bank benchmark under different CC protocols, under increasing number of nodes. The performance of HOME and B-DQCC degrades rapidly when the number of nodes increases, resulting in significantly lower throughput than B-DPQCC. B-DPQCC maintains a comparable performance of throughput (with slight degradation) when the number of shared objects decreases. Generally, B-DPQCC outperforms other protocols by more than 50%, and the performance of B-DQCC is even worse than HOME.

![Graph showing throughput](image1)

(a) Throughput of Loan

Figure 10.6: Performance of Loan benchmark under HOME, B-DQCC, and B-DPQCC protocols.

Figure 10.5(b) illustrates the average number of aborts per transaction of the Bank benchmark. The figure reveals the inherent advantage of B-DPQCC. When the number of nodes increases, the average number of aborts increases linearly (or even faster) under HOME and B-DQCC, while B-DPQCC keeps this number at a very low level (less than 1). Hence, when the network size increases (i.e., number of nodes increases), the performance of B-DPQCC is more attractive.

For the Loan benchmark, 10 accounts (i.e., shared objects) were distributed uniformly on 24 nodes, and 100 transactions were executed at each node. Each node requests access to 5 shared objects. The performance of the Loan benchmark is shown in Figure 10.6. Similar to the Bank benchmark, B-DPQCC outperforms other protocols by more than 100% due to its ability to efficiently reduce the average number of aborts when the network size increases.

Under the Loan benchmark, B-DQCC outperforms HOME by more than 50% when the number of nodes increases.

Figure 10.7 shows the execution time for a single node to commit 100 transactions of differ-
Figure 10.7: Execution time for a single node to commit 100 transactions under Bank and Loan benchmarks.

ent protocols under the Bank and Loan benchmarks. We divide the execution time into two parts: local execution time, which is the total time duration for the node’s local processing, including local computation, local synchronization, language-level processing, etc.; and remote execution time, which is the total time duration for the node’s remote communication to acquire objects.

When the network size increases, the percentage of remote execution time also increases and gradually becomes the major part of the total execution time. In particular, for HOME and B-DQCC protocols, the remote execution time increases much faster than the local execution time. For B-DPQCC, the remote execution time increases relatively slowly. These results imply that B-DPQCC outperforms other protocols, as it better optimizes remote communication cost.
The throughput of the Vacation benchmark is shown in Figure 10.8, with the same setting as Bank and Loan. When the network size increases (and the contention increases), the performance of B-DQCC and Home protocols degrades linearly, and they exhibit poor performance when the number of nodes is large. On the other hand, B-DQCC protocol maintains a relatively stable performance and outperforms B-DQCC and Home protocols by more than 100% when the number of nodes increases.

![Figure 10.10: Throughput of distributed binary search tree and red-black tree microbenchmarks.](image)

(a) Distributed binary search tree  
(b) Distributed red-black tree

Figures 10.9 and 10.10 illustrate the throughput of the different CC protocols under the distributed linked list, distributed binary search tree, and distributed red-black tree microbenchmarks, where 10 objects were distributed uniformly on 24 nodes and 1000 transactions were executed at each node. We used more number of transactions in the microbenchmark experiments, due to the smaller transaction execution times involved, allowing for faster experimental rounds. Again, B-DPQCC outperforms other competing protocols by more than 100% when network size increases. Moreover, note that B-DQCC also performs much better than Home for the microbenchmarks. This is due to the relatively smaller local execution time duration of the microbenchmarks, and the domination of the remote communication cost in the total transaction execution makespan.

### 10.5.2 Evaluation of RELAY

We evaluate the performance of Home, Arrow, Ballistic and Relay under Bank benchmark, with Greedy implemented as the underlying contention manager. We distributed a set of shared bank accounts (i.e., objects) uniformly across up to 24 nodes and executed 100
Figure 10.11: Bank Throughput under HOME, ARROW, BALLISTIC and RELAY protocols.
transactions at each node. Under this setting, we measured the transactional throughput for each CC protocol.

Figures 10.11(a) to 10.11(c) show the throughput when 10 accounts are distributed, and each node executes 100 transactions. We observe that when the link delay is relatively small (1\,ms), RELAY provides a comparable performance against HOME and BALLISTIC. When the link delay increases to a certain level (larger than 10\,ms), RELAY outperforms all other protocols by more than 200%.

When we decrease the number of shared accounts to 5, the contention increases and the resulting throughput is shown in Figures 10.11(d) to 10.11(f). The performance of HOME, ARROW and BALLISTIC degrades rapidly when contention is high, resulting in significantly lower throughput than RELAY. RELAY maintains a comparable performance (with slight degradation) when the number of shared objects decreases.
These results show that RELAY provides a competitive throughput when network size increases (which increases the cost of each abort) and the number of shared objects decreases (which increases the contention over objects) by efficiently reducing the number of aborts.

Figure 10.12 further shows the execution time for a single node to commit 100 transactions of different protocols when 10 accounts are distributed. We divide the execution time into two parts: local execution time, which is the total time duration for the node’s local processing, including local computation, local synchronization, language-level processing, etc.; and remote execution time, which is the total time duration for the node’s remote communication to acquire objects.

When the link delay increases, the percentage of remote execution time also increases and gradually becomes the major part of the total execution time. In particular, for HOME, ARROW and BALLISTIC protocols, the remote execution time increases much faster than the local execution time. For RELAY protocol, the remote execution time increases relatively slow. These results implies that RELAY outperforms other protocols due to its ability to restrict the increase of remote communication cost.

Figures 10.13(a) to 10.13(f) illustrate the average number of aborts per transaction under the same scenario corresponding to Figures 10.11(a) to 10.11(f), respectively. The figures reveal the inherent advantage of RELAY. When the number of nodes increases, the average number of aborts increases linearly (or even faster) under HOME, ARROW, and BALLISTIC. Hence, when the network size increases, RELAY’s performance is more attractive.

The figures also explain the degradation of HOME, ARROW, and BALLISTIC when the number of shared objects decreases: a transaction suffers approximately 50% more aborts, on average, when the number of shared objects decreases by 50%. On the other hand, the average number of aborts under RELAY is low (less than 0.5), which implies that more than 50% of transactions commit during their first execution.

Our results also show that the average number of aborts per transaction, under RELAY, is much lower than the worst-case, predicted by the worst-case analysis (from Theorem 23, in the worst case, the average number of aborts per transaction is $\frac{N-1}{N}$ for a set of $N$ transactions).

Figures 10.14 and 10.15 show the effects of two separate factors (with all else being unchanged): the link delay and the number of shared objects, respectively. Figure 10.14 shows that the throughput of RELAY degrades slowly when the link delay increases linearly. Since a larger link delay implies a longer execution time for a single transaction, the degradation in performance is unavoidable. Despite this, RELAY’s performance under large link delay is still acceptable.

In Figure 10.15, the number of shared bank accounts decreases linearly, implying increasing contention over all transactions. RELAY’s degradation is slow, similar to Figure 10.14. These results illustrate the advantages of RELAY: when the network condition worsens and the contention increases, RELAY’s performance degrades relatively slow.
Figure 10.13: Average number of aborts per transaction of Bank benchmark under HOME, ARROW, BALLISTIC and RELAY protocols.
10.5.3 Evaluation of the DDA Model

To evaluate the DDA model, we first used HOME and RELAY as two underlying CC protocols, and compared the DDA model with GREEDY and KARMA contention managers under the Bank benchmark by distributing a set of shared bank accounts (i.e., objects) uniformly across 24 nodes and executed 100 transactions at each node. We control the level of contention by change the percentage of shared objects accessed by a single transaction. Figures 10.16(a) and 10.16(b) shows that in high contention environments (where each transaction accesses 80% of shared objects), the DDA model always outperforms selected contention management policies by upto 30%-40%. When the workload is write dominated, the DDA model reduces the number of aborts by guaranteeing that a write-only transaction is never aborted by another write-only transaction. The randomized priority assignment also assures the starvation-freedom of a single transaction with high probability. When the workload is read dominated, the number of aborts is significantly reduced since the DDA model never aborts a read-only transaction.

The DDA model does not always outperforms selected contention management policies, as illustrated in Figures 10.16(c) and 10.16(d). In low contention environments (where each transaction accesses 20% of shared objects), the DDA model does not perform as well as in high contention environments. The penalty is that in the DDA model, a transaction has to insert object versions for each object it requested, which incurs high communication overhead. Yet the DDA model does show an approximately same performance compared with selected contention management policies. This is due to its RT-UP-GC mechanism which discards the useless object versions and reduces the overhead to insert versions efficiently.

We then used BALLISTIC as the underlying CC protocol, and compared the DDA model with GREEDY, KARMA and KINDERGARTEN contention managers under the Bank bench-
Figure 10.16: Bank throughput of the DDA model

mark. We distributed a set of shared bank accounts (i.e., objects) uniformly across 24 nodes and executed 100 transactions at each node. Under this setting, we measured the average number of aborts per transaction under difference conflict resolution strategies, as shown in Figures 10.17(a) to 10.17(f). When the workload is read/write-balanced (Figures 10.17(a) to 10.17(c)), the DDA model outperforms the other three contention managers by about 30% – 60%. When the workload is read-dominated (Figures 10.17(d) to 10.17(f)), the DDA model performs much better: its average number of aborts per transaction is over 200% lower than that of the competing contention managers. Specifically, when we reduce the number of shared accounts (objects), which increases contention, the average number of aborts for the contention managers increases faster than linearly, while for the DDA model, it increases much slower (than a linear increase).

Figure 10.18 shows the execution time for a single node to commit 100 transactions of
Figure 10.17: Average number of aborts per transaction of Bank benchmark under Greedy, Karma, Kindergarten and the DDA model.
the Bank Benchmark under different conflict resolution strategies when 5 accounts are distributed. We divide the execution time into two parts: local execution time, which is the total time duration for the node’s local processing, including local computation, local synchronization, language-level processing, etc.; and remote execution time, which is the total time duration for the node’s remote communication to acquire objects.

As shown in the figure, the execution time of the DDA model is much smaller: the remote execution time of the DDA model is 20% to 240% smaller than that of GREEDY, KARMA, and KINDERGARTEN contention managers. These results illustrate that the DDA model outperforms other contention managers due to its ability to minimize the remote communication cost.

Figure 10.18: The execution time for a single node to commit 100 transactions of the Bank Benchmark with 90% reads, and 5 accounts are distributed.
Chapter 11

Conclusions and Future Work

In this dissertation, we study the design of cache-coherence protocols, conflict resolution strategies, and replication protocols for D-STM. Our focus is on understanding the performance — both worst-case performance bounds and experimental performance — of cache-coherence protocols and conflict resolution strategies. We are also interested in understanding this performance in both isolation (i.e., each problem considered independently), and in conjunction. Additionally, we are interested in designing D-STM protocols with provable fault-tolerance properties. We now draw conclusions from each of the dissertation’s results.

LaC protocols: location-aware CC protocols working with the Greedy manager

Our first approach for solving this problem is to select a fixed contention manager, which guarantees a provable worst-case performance even when it is combined with the worst-possible cache-coherence protocols. Motivated by the excellent properties of the Greedy contention manager for multiprocessors, we determine its worst-case makespan and compare that with the makespan of the optimal off-line clairvoyant scheduler for D-STM systems. We show that, for each object, the optimal scheduler visits all nodes requesting the object via the shortest Hamiltonian path. We then establish the worst-case competitive ratio of the Greedy manager with an arbitrary cache-coherence protocol. We show that, with an arbitrary cache-coherence protocol, the upper bound of the competitive ratio is $O(N^2 \cdot s)$, where $N$ is the maximum number of transactions requesting an object and $s$ is the number of objects. Moreover, we derive an $\Omega(s)$ lower bound for the competitive ratio of the Greedy manager, which depicts its best-case performance. By doing so, we establish the range of the competitive ratios that the Greedy manager can achieve. Since its worst-case is far from optimal — ideally, we desire a matching upper bound with the lower bound — we need to design cache-coherence protocols to improve performance.

Thus, we design cache-coherence protocols that improve the worst-case competitive ratio of the Greedy manager. We propose a class of distributed cache-coherence protocols with location-aware property, called LaC protocols. For LaC protocols, the time that a transaction’s node takes to locate an object requested by the transaction is determined by the
communication cost between the requesting node and the node that holds the object. We prove an \( O(N \log N \cdot s) \) competitive ratio for the combination (GREEDY, LAC), and show that LAC is an efficient choice for the GREEDY manager to improve performance.

Distributed queueing-based CC protocols. Past directory-based cache-coherence protocols model the cache-coherence problem as a distributed queuing problem, due to the fundamental similarities between the two, and use distributed queuing protocols. Transactions requesting read/write access to an object are queued in a distributed queue for that object. However, what performance bound can be achieved with distributed queuing was previously open. Since a distributed queuing protocol does not consider transactional contention, contention-aware protocols may outperform distributed queuing-based cache-coherence protocols, both in terms of (worst-case) performance bounds and in the performance exhibited in practical implementation.

We formalize the class of all cache-coherence protocols based on distributed queuing, called distributed queuing cache-coherence (DQCC) protocols. We implement a DQCC protocol using a distributed queuing protocol \( C \), which provides an ordering cost \( \delta_C(r_i, r_j) \) to order a request \( r_j \) after request \( r_i \).

Next, we formalize a novel class of cache-coherence protocols, which are based on distributed priority queuing, called distributed priority queuing-based cache-coherence (or DPQCC) protocols. Similar to DQCC, a DPQCC protocol also enqueues transactions contending for an object in a distributed queue. However, it guarantees that, at any given time, only the transaction with the highest priority in the queue can commit. Moreover, we can implement a DPQCC protocol based on the same distributed queuing protocol \( C \) as a DQCC protocol, allowing the two to be compared on the same ground.

We show that, for a set of \( N \) transactions requesting an object, the competitive ratios of a DQCC and a DPQCC protocol are \( O(N \log D_\delta) \) and \( O(\log D_\delta) \), respectively, if the maximum local execution time of the transactions in \( T \) is \( O(\log D_\delta) \), where \( D_\delta \) and \( \overline{D}_\delta \) are the normalized diameter and the diameter of the underlying distributed queuing protocol \( C \), respectively. This result illustrates the advantage of DPQCC over DQCC.

Our evaluation shows that DPQCC yields better transactional throughput than DQCC, by a factor of 50% – 100%, on macrobenchmarks (distributed version of Bank, Loan and Vacation of STAMP benchmark [36]) and microbenchmarks (distributed linked list, binary search tree and red-black tree). DPQCC outperforms DQCC and HOME as it better optimizes remote communication cost.

Relay: a high-performance CC protocol On the other hand, motivated by the distributed queuing problem, the principles of management of distributed queues also apply to D-STM: transaction requests have to be ordered in a queue and each transaction needs to know the location of its successor in the queue so that it knows where to forward the object to. Distributed queuing protocols that consider ordering requests cannot be directly used for D-STM. This is because, such protocols usually do not provide efficient mechanisms to mediate...
conflicts among multiple transactions over a set of objects — e.g., when a node holding an object receives a new request, it simply sends the object to the requesting node. Past distributed queuing protocols do not consider the contention between transactions: an abort of a transaction increases the length of the queue since the transaction has to be restarted. We show that for the Arrow protocol, which is a distributed queuing protocol (and also used in the design of the Ballistic cache-coherence protocol), the worst-case number of total aborts is $O(N^2)$ for $N$ transactions requesting an object.

Therefore, we propose Relay, a cache-coherence protocol which aims to reduce the worst-case number of total aborts. Similar to Arrow, Relay works on a network spanning tree. The idea of Relay is as follows: when a transaction $T_i$ is aborted by another transaction $T_j$, $T_i$ is queued after $T_j$ once $T_i$ has been restarted. In other words, a transaction can only be aborted by the same transaction once for each object. As a result, Relay efficiently reduces the worst-case number of total aborts to $O(N)$.

Our experimental studies show that Relay outperforms Arrow [49], Ballistic [26] and Home protocols by more than 200% when the network link delay increases under the Bank benchmark. Our experimental results also illustrate the inherent advantage of Relay: Relay keeps the average number of aborts per transaction at a very low level (less than 0.4), which guarantees that more than 60% of transactions commit during their first execution.

Dependence-aware non-conservative conflict resolution. Past D-STM cache-coherence protocols assume a contention management-based conflict resolution strategy. While easy to implement, such a contention management approach may lead to significant number of unnecessary aborts, especially for read-dominated workloads [18]. This raises a fundamental question: can conflict resolution strategies be designed that increases concurrency under a general cache-coherence protocol?

Our solution to this problem is inspired by past multiprocessor STM works on enhancing concurrency by establishing precedence relations among transactions. A transaction can commit as long as the correctness criterion is not violated by its established precedence relations with other transactions. Generally, the precedence relations among all transactions form a global precedence graph. By maintaining the precedence graph in time and keeping it acyclic, a TM system can efficiently avoid unnecessary aborts.

We leverage this idea for distributed systems and develop the DDA model. We identify the two inherent limitations of establishing precedence relations in D-STM. First, there is no centralized unit to monitor precedence relations among transactions in distributed systems, which are “scattered” in the network. For a transaction to observe the status of the precedence graph before the next operation, significant communication (between transactions) is necessary. In the DDA model, we design a set of algorithms to avoid frequent intertransaction communications. In the model, read-only and write-only transactions never abort by keeping proper versions of each object. Each transaction only keeps precedence relations based on its local knowledge. Our algorithms guarantee that, when a transaction reads/writes an object based on its local knowledge, the underlying precedence graph
keeps acyclic. On the other hand, we adopt a contention management policy to handle non-
write-only update transactions, which involve both read and write operations. This strategy
ensures that an update transaction is efficiently processed when it potentially conflicts with
another transaction, and ensures system progress.

Second, when a transaction commits, it should insert a new object version for each object
it writes to. Since objects are shared by all transactions, the transaction may not hold
all objects it requires to insert versions. Hence, a transaction cannot insert versions in
a centralized way. As a result, different objects may observe different commit times for
the same transaction. Such phenomenon can cause a transaction to erroneously decide its
precedence relations, and introduce unnecessary aborts or violate correctness. We design
a set of algorithms to efficiently detect and update precedence relations and ensure that
even when a wrong detection occurs, the operations of related transactions are adjusted to
accommodate such errors without violating correctness.

Our experimental studies with the DDA model’s implementation show that, DDA outper-
forms competitor contention management policies including GREEDY [35], KARMA [50] and
KINDERGARTEN [50] contention managers by upto 30%-40% during high contention under
the Bank benchmark. For read/write-balanced workloads, the DDA model outperforms these
contention management policies by 30% – 60% on average. For read-dominated workloads,
the model outperforms by over 200%

**Complexity of distributed contention management.** We also study the complexity of con-
tention management in D-STM. We establish the relationship of this problem to a combina-
tion of the graph coloring problem and the TSP problem. Finding an optimal schedule of
contention management in D-STM consists of a set of sub-problems which are all NP-hard.
In general, an optimal schedule needs to examine all $k$-coloring instances of the underlying
conflict graph for all possible $k$. For each $k$-coloring instance, an optimal schedule needs to
find the order of transactions’ execution such that the cost of moving objects is minimized —
equivalent to find a TSP path in the network for each object. Compared with the problem
complexity of finding an optimal schedule for transactions in multiprocessor STM systems,
which is directly related to finding the chromatic number of the conflict graph (which is also
NP-hard), the D-STM contention management problem is more complex.

We prove that for D-STM, any online, work conserving, deterministic contention manager
provides an $\Omega(\max\{s, \frac{s^2}{D}\})$ competitive ratio in a network with normalized diameter $D$ and
$s$ shared objects. Compared with the $\Omega(s)$ competitive ratio for multiprocessor STM, the
performance guarantee for D-STM degrades by a factor proportional to $\frac{s}{D}$. We present a
randomized algorithm, called RANDOMIZED, with a competitive ratio $O(s \cdot (C \log n + \log^2 n))$
for $s$ objects shared by $n$ transactions, with a maximum conflicting degree $C$. To break this
lower bound, we present a randomized algorithm CUTTING, which needs partial information
of transactions and an approximate TSP algorithm $A$ with approximation ratio $\phi_A$. We
show that the average case competitive ratio of CUTTING is $O(s \cdot \phi_A \cdot \log^2 m \log^2 n)$, which
is close to $O(s)$. 
**Quorum-based replicated D-STM: Provable fault-tolerance.** It is desirable for a D-STM system to provide provable fault-tolerance properties without introducing significant performance degradation in the presence of node failures. Our proposed QR model preserves the competitive performance compared with the SC model, which implies that if no node fails, the QR model exhibits a performance similar to the SC model. When node failure occurs, the performance of the QR model degrades gracefully. Meanwhile, the QR model guarantees provable availability: as long as a write quorum exists, at least one available (writable) copy of each object lives in the system.

The QR model has bounded communication cost for its operations, which is proportional to the communication cost from node $v$ to its closest read/write quorum, for any operation starting from $v$. Compared with directory-based CC protocols, the communication cost of operations in the QR model does not rely on the stretch of the underlying overlay tree (i.e., the worst-case ratio between the cost of direct communication between two nodes $v$ and $w$ and the cost of communication along the shortest tree path between $v$ and $w$). Thus, the QR model allows D-STM to tolerate node failures with communication cost comparable with that of the single copy D-STM model.

**11.1 Summary of Contributions**

To summarize, the dissertation’s contributions include:

1. We prove the worst-case performance bound of the Greedy manager for D-STM. We propose Lac CC protocols, and show that the combination of the Greedy manager and Lac provides an improved worst-case performance bound.

2. We formalize the class of DQCC and DPQCC protocols, both of which can be implemented using distributed queuing protocols. We show that DPQCC outperforms DQCC by providing an improved worst-case performance bound.

3. We propose Relay, a cache-coherence protocol which efficiently reduces the worst-case number of total aborts and exhibits a provable worst-case performance bound for a set of dynamically generated transactions.

4. We propose the DDA model, which enhances concurrency by adopting different conflict resolution strategies based on the types of transactions.

5. We study the distributed contention management problem to understand the problem complexity and establish performance bounds. We establish its relationship to the TSP problem, propose two randomized algorithms: Randomized and Cutting, which yield a non-trivial average-case performance bound.
6. We propose the QR model, a quorum-based replication D-STM model. Compared with the single copy model, in the presence of node failures, the QR model exhibits high availability and degrades gracefully when the number of failed nodes increases, with reasonable higher communication cost.

11.2 Future Work

Based on the dissertation’s results, we propose the following problems as promising directions for future research.

An interesting direction is to support non-transactional instructions in D-STM. Although D-STM provides a safer and more scalable alternative to lock-based synchronization, it may not be appropriate for all instructions. D-STM systems typically execute optimistically, using rollback to recover from conflicts between transactions. However, there are some operations whose side effects cannot, in general, be rolled-back. Such operations, e.g., I/O, system calls, actuator commands, etc, are often referred to as irreversible or irrevocable operations. To support such non-transactional instructions in D-STM systems, several candidate solutions can be considered:

1. Deferring non-transactional operations until commit. Such a mechanism must be compatible with the desired D-STM correctness criterion.
2. Executing system calls during the transaction and reversing the side effects on abort through compensating actions. Similarly, the desired correctness criterion for transactions must be satisfied.
3. Ensuring that transactions with non-transactional operations always commit.
4. Ensuring that data written by a transaction with non-transactional operations is not visible until the transaction commits.

Each of these candidate solutions may not by itself be sufficient and has concomitant trade-offs. Optimistic approaches may violate correctness and may lead to an inconsistent system state, while pessimistic approaches may keep non-transactional operations from executing concurrently, limiting performance. Nevertheless, it is far from clear how these solutions perform and interact with each other, especially to achieve a desired correctness criterion (e.g., opacity). Empirical evaluation and theoretical analysis are essential to develop a proper suite of candidate solutions for handling non-transactional instructions.

Another direction is to support nesting in D-STM. In D-STM, if a nested transaction is aborted, the behavior of its parent transaction depends on the nesting model. D-STM implementations typically maintain a transaction’s readset and writerset, i.e., a list of memory locations that a transaction has read from or written to, respectively. The nesting model determines whether and when a nested transaction’s readset and writerset are merged into its parent transaction’s readset and writerset. Three nesting models have been studied in past multiprocessor STM efforts [103, 104, 105]: flat nesting, closed nesting, and open nesting.
In D-STM, a problem arises when a parent transaction aborts after its nested transaction commits, especially for the open nesting model. Note that when a transaction aborts, all its operations need to be rolled-back and all its changes should be discarded. In this sense, the changes made by the committed nested transaction should also be “recovered.” Simply sending an abort message to the aborted transaction may not work, since the aborted transaction has already committed at the system-level. Thus, the open nesting model may require higher-level constructs for rollbacks of aborted transactions, or for concurrency control between transactions. For example, the programmer may need to use “abstract locks” in the code to propagate the object access information of the nested transaction to its parent transaction such that certain transactional interleavings are prevented in advance. Such mechanisms, based on locks, may suffer from the same drawbacks of other distributed lock-based synchronization algorithms.

The D-STM model can use a “compensating” transaction that undoes the effect of a committed open-nested transaction if its parent transaction aborts. The compensation step can make sure that a running transaction commits without accessing any inconsistent state of an object. However, compensation may not produce an optimal schedule for open nesting, since a compensation step significantly increases the local execution time of a single transaction. Thus, it is desirable to design highly efficient D-STM algorithms to support (open) nested transactions without incurring significant penalty.

Fault-tolerant D-STM is another interesting direction. In the dissertation’s QR model, we do not differentiate between shared objects: every object is of equal importance when its replicas are created. This approach is feasible, but may not be the best strategy to replicate objects. One possible strategy is to create more replicas for “popular” objects, i.e., objects requested by most transactions, and create less replicas for objects requested by few transactions. The benefit of such a partial replication approach is obvious: the number of redundant replicas is reduced, which further reduces the potential cost to manage the replicas. In other words, instead of assigning priorities to transactions, we can also assign priorities to objects based on each object’s usage. Various policies can be designed to assign priorities to objects, and their fault-tolerance properties can be established.
Bibliography


