On the Design of Contention Managers and Cache-Coherence Protocols for Distributed Transactional Memory

Bo Zhang

Preliminary Examination Proposal 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

May 6, 2009
Blacksburg, Virginia

Keywords: Distributed Transactional Memory, Cache-Coherence, Contention Manager,
Quorum System
Copyright 2009, Bo Zhang

On the Design of Contention Managers and Cache-Coherence Protocols for Distributed Transactional Memory

Bo Zhang

(ABSTRACT)

Conventional synchronization methods based on locks and condition variables are inherently non-scalable, non-composable, and error-prone. Transactional synchronization is an alternative programming model for managing contention in accessing shared data objects, which exhibits excellent scalability and composability properties, besides programming ease. Transactional API for multiprocessor synchronization, called *Transactional Memory*, utilizes contention managers to guarantee that whenever two transactions have a conflict on a shared data object, one of them is aborted. While transactional memory has been well studied in the context of multiprocessors, few results are known for them for distributed systems where nodes communicate via message-passing links. Compared with multiprocessor transactional memory systems, the design of distributed transactional memory systems is more challenging because of the need for distributed cache-coherence protocols and the underlying (higher) network latencies involved. The choice of the combination of the contention manager and the cache-coherence protocol is critical for the performance of distributed transactional memory systems.

In this dissertation proposal, we study the design of contention managers and cache-coherence protocols for distributed transactional memory systems. We approach this design problem by first establishing the relationship and combinative behavior of contention managers and cache-coherence protocols on the performance of distributed transactional memory systems. We consider the Greedy contention manager — a contention manager with excellent properties for multiprocessors — for distributed transactional memory systems. We establish upper and lower bounds for the *competitive ratio* of the Greedy manager's makespan—i.e., the ratio of the makespan (the last completion time for a given set of transactions) of the combination of the Greedy manager and an arbitrary cache-coherence protocol to the makespan of an optimal off-line clairvoyant scheduler without considering a cache-coherence protocol. We show that, in the worst case, the competitive ratio is $O(N^2 \cdot s)$, where N is the maximum number of transactions that request the same object and s is the number of objects. On the other hand, the best-case competitive ratio is $\Omega(s)$, which matches the performance of the Greedy manager for multiprocessors. This result motivates the need to design cache-coherence protocols to improve the worst-case performance with the Greedy manager.

We propose a class of distributed cache-coherence protocols with *location-aware* property, called LAC protocols. In LAC protocols, the duration of a transaction requesting node to locate an object is determined by the communication delay between the requesting node and the node that holds the object. We prove the worst-case competitive ratio of the Greedy manager/LAC protocol combination, and show that LAC is an efficient choice for the Greedy manager to improve system performance.

We also present a novel DHT (distributed hash table)-based cache-coherence protocol, called the DHTC protocol. DHTs provide good load balancing and scalability properties. A common DHT also has a fully decentralized structure, which is essential for loosely-organized applications. We show that the DHTC protocol guarantees an $O(N \cdot s)$ competitive ratio, which is a significant improvement over an arbitrary cache-coherence protocol/Greedy manager combination.

To improve the performance of distributed transactional memory in the presence of arbitrary node failures and node joins and departures, we develop a quorum-based cache-coherence protocol. We construct a novel quorum system called, the dynamic high available B-Grid quorum system or DHB-Grid. We show that the performance of DHB-Grid is asymptotically optimal when the failure probability p, i.e, the maximum probability that a node fails, approaches 0, and degrades gracefully when p increases. We present efficient adjustment algorithms for DHB-Grid to accommodate network changes, and show an $O(\log n)$ message complexity for each adjustment, where n is the number of nodes. Based on the DHB-Grid system, we develop DHBC, a quorum-based cache-coherence protocol, which exhibits high availability and low message complexity properties.

We have focused on the Greedy manager and studied its performance for distributed transactional memory because it provides a provable worst-case performance. Fundamental and open problems that we propose to solve (post-prelim) include designing novel contention managers for distributed transactional memory and establishing their performance bounds. In addition, the design of cache-coherence protocols with such contention managers is also proposed.

Contents

1 Introduction **1**

1.1 Distributed Transactional Memory Model 2

1.2 Measures of Quality 4

1.3 Summary of Current Research and Contributions 5

1.4 Proposed Post Preliminary-Exam Work 8

1.5 Proposal Outline 9

2 Past and Related Work **10**

2.1 Transactional Memory: Overview 10

2.2 Contention Management: Application and Theory 11

2.3 Distributed Transactional Memory 11

2.4 Quorum Systems 12

3 Performance Bounds of the Greedy Manager for Distributed Transactional Memory **13**

3.1 System Model and Problem Statement 13

3.1.1 Metric-Space Network Model. 13

3.1.2 Transaction Model. 14

3.1.3 Distributed Transactional Memory Model. 14

3.1.4 Problem Statement. 14

3.2 Competitive Ratio of the Greedy Manager 15

3.2.1 Motivation and Challenge. 15

3.2.2 Competitive Ratio Analysis 17
3.3 Conclusion 22

4 Location-Aware Cache-Coherence Protocols for Distributed Transactional Memory 23
4.1 Cache Responsiveness 23
4.2 Location-Aware Cache-Coherence Protocols 24
4.3 $makespan(Greedy, LAC)$ for Multiple Objects 26
4.4 Conclusions 28

5 A DHT-Based Cache-Coherence Protocol for Distributed Transactional Memory 30
5.1 Protocol Description. 30
5.2 Protocol Analysis 33
5.3 Concluding Remarks 36

6 A Quorum-Based Cache-Coherence Protocol for Distributed Transactional Memory 37
6.1 Definitions and Preliminaries 37
6.1.1 Quorum Systems 37
6.1.2 Metrics of Quality 37
6.2 System Model 39
6.2.1 Dynamic Network Model 39
6.2.2 Probabilistic Failure Model 39
6.3 HB-Grid: High Available B-Grid Quorum Systems 40
6.4 DHB-Grid: Dynamic HB-Grid Quorum Systems 42
6.4.1 The Quorum Implementation: 3-Phase Linking 43
6.4.2 Basic Join/Leave Operation 44
6.4.3 Join/Leave Complexity 46
6.5 DHBC: The DHB-Grid-Based Cache Coherence Protocol 47
6.5.1 Protocol Description 47

6.5.2 Probe Complexity 48
6.6 Conclusion 49

7 Conclusions, Contributions, and Proposed Post Preliminary-Exam Work 51
7.1 Contributions 53
7.2 Post Preliminary-Exam Work 53

List of Figures

1.1 The data flow model of distributed transactional memory 3

3.1 Example 1: A 3-node network 19

3.2 Example 1: Link path evolution of the directory hierarchy built by Ballistic protocol 20

6.1 A B-Grid System: $n = 240$, $d = 16$, $h = 5$, $r = 3$. One quorum is shaded. . . 40

Chapter 1

Introduction

The design of *non-blocking synchronization algorithms* for accessing objects shared by multiple processes or threads has been of tremendous interest in the last two decades [47]. Conventional synchronization methods for single and multiprocessors based on locks, semaphores, and condition variables suffer from many 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. In non-blocking synchronization algorithms [47, 52], processes/threads do not need to wait when competing to access shared objects. Instead, a concurrent process/thread may either abort its own atomic operation (retrying later, optionally), or abort the atomic operation of the conflicting process/thread. Compared with lock-based synchronization algorithms that use mutually exclusive critical sections to serialize access to shared objects, non-blocking synchronization algorithms are designed to avoid requiring critical sections all together. These algorithms allow multiple processes/threads to make progress on accessing a set of shared objects without ever blocking each other.

Transactional synchronization [27, 64] is a non-blocking synchronization model for managing contention in accessing shared resources. A transaction, like a critical section, is an explicitly delimited sequence of steps (e.g., a piece of code) that is executed atomically by a single thread. To guarantee atomicity, a transaction ends by either committing (i.e., all of its updates take effect), or by aborting (i.e., updates do not effect). The concurrency control mechanism for transactions is optimistic: if a transaction aborts, it is typically retried until it commits. Transactional API for multiprocessors, called *Transactional Memories*, have been proposed in hardware [22, 27], in software [23, 28, 64], and in hardware/software combination [6]. [1]

[1] Transactional processing, the semantic ancestor of transactional memory, has been a highly successful abstraction for handling concurrency in database systems. Transactional memory is an attempt at applying the transaction concept to non-database systems, in particular, for in-memory operations. Database transactions often require an expensive commit protocol (e.g., to obtain the classical ACID properties). In

Transactions read and write shared objects. Two transactions *conflict* if they access the same object and one access is a write. A non-blocking synchronization abstraction is said to be *obstruction-free* if it guarantees that any thread, if run by itself (i.e., without any contention), will make progress in a finite number of steps. Apparently, transaction synchronization is obstruction-free. Obstruction-freedom introduces livelocks, which can be effectively minimized by a *contention manager* module. Contention managers guarantee atomicity by making sure that whenever a conflict occurs, only one of the transactions involved can proceed.

Despite the large body of work on transactional memory for multiprocessors, few results are known for them for distributed systems, exceptions being [4, 30, 45]. In this proposal, we consider the design of transactional memory in a distributed system consisting of a network of nodes that communicate by message-passing links. The system is subject to arbitrary crash failures of nodes. In addition, nodes can potentially join or leave unpredictably. [2]

1.1 Distributed Transactional Memory Model

Different models of transactional memory for distributed systems have been recently proposed. Herlihy and Sun [30] propose three competing models:

- *Control flow model.* This model has long been used to provide fault-tolerance in databases and distributed systems. In this model, data objects are typically immobile, and computations move from node to node via remote procedure calls (or RPCs). A deadlock detection mechanism is required to detect and resolve deadlocks. Atomicity is guaranteed by a commit protocol which ensures that a transaction's tentative changes either take effect at all nodes or are all discarded.

- *Data flow model.* In this model, transactions are immobile (running on a single node) and data objects move from node to node. Here, a contention manager is responsible for mediating between conflicting accesses to the same object and avoid deadlocks and livelocks. A transaction that finishes without being interrupted by a synchronization conflict simply commits.

- *Hybrid model.* In this model, data objects are migrated depending on a number of heuristics such as size and locality.

These transactional models make different trade-offs. In the control flow model, an object's home node must mediate all accesses to that object. Hence, it is likely to become a bottleneck

contrast, transactional memory can be viewed as a "lighter-weight" version of transactional processing, with no commit protocol required in most cases, thereby seeking to gain the benefits of transactional processing (e.g., semantic simplicity, fault tolerance), without incurring all its associated overhead.

[2]The definitions of node crashes and leaves are different: the node which crashes does not inform the system; on the other hand, the node which intends to leave the system informs its neighbors before leaving.

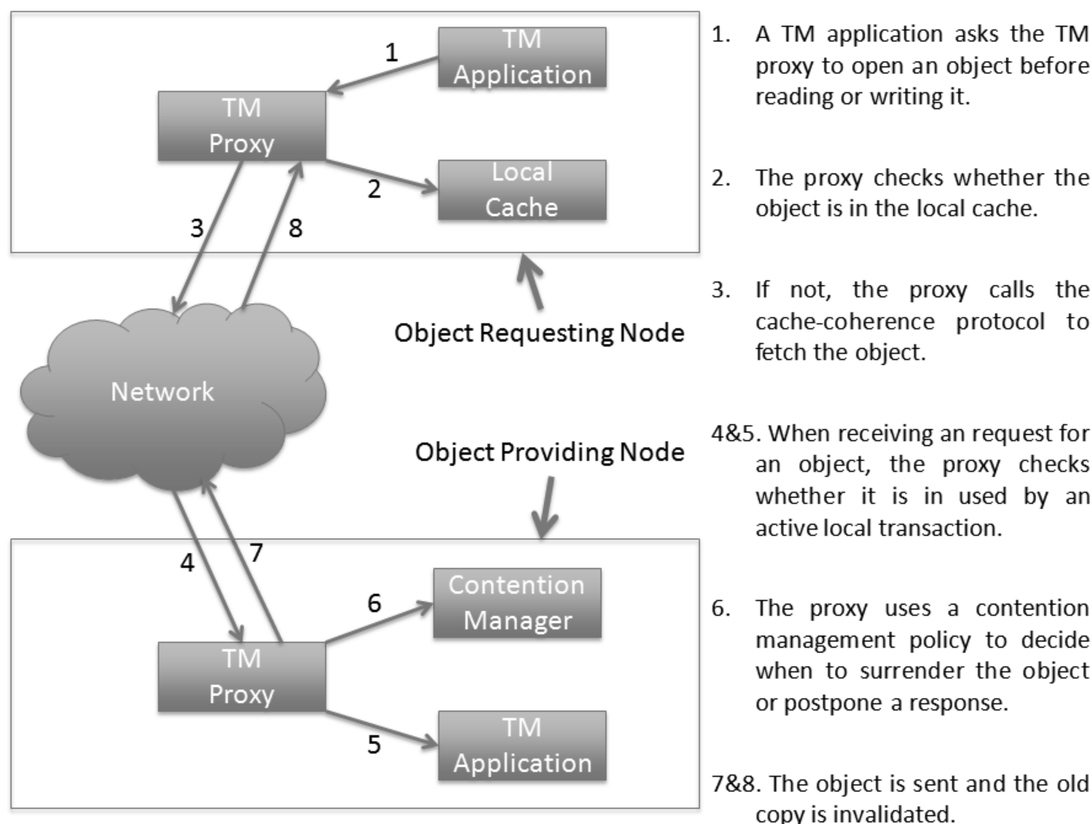

Figure 1.1: The data flow model of distributed transactional memory

if that object is a "hot spot". On the other hand, in the data flow model, there may be some applications that prefer to store objects at dedicated repositories instead of letting them migrate among nodes. Past work on multiprocessors [29] suggests 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.

We consider the data-flow model in [30] to support the transactional memory API in a distributed system. As shown in Figure 1.1, transactions are immobile (running at a single node), but objects move from node to node. Transaction synchronization is optimistic: a transaction commits only if no other transaction has executed a conflicting access. A *contention manager* module is responsible for mediating between conflicting accesses to avoid deadlocks and livelocks. The core of this design is an efficient *distributed cache-coherence* protocol. A distributed transactional memory system uses a distributed cache-coherence

protocol to support object operations. For example, when a transaction attempts to access an object, the cache-coherence protocol must locate the current cached copy of the object, move it to the requesting node's cache, and invalidate the old copy.

Similar to [30], each node is assumed to have a *transactional memory proxy* module that provides interfaces to the application and to proxies at other nodes. This module performs the following functions:

- *Data Object Management*: An application informs the proxy to open an object when it starts a transaction. The proxy is responsible for fetching a copy of the object requested by the transaction, either from its local cache or from other nodes. When the transaction asks to commit, the proxy checks whether any object opened by the transaction has been modified by other transactions. If not, the proxy makes the transaction's tentative changes to the object permanent; otherwise discards them.

- *Cache-Coherence Protocol Invocation*: The proxy is responsible for invoking a cache-coherence protocol when needed. When a new data object is created in the local cache, the proxy calls the cache-coherence protocol to *publish* it in the network. When an object is requested by a read access and is not in the local cache, the proxy calls the cache-coherence protocol to *look-up* the object and fetch a read-only copy. If it is a write request, the proxy calls the cache-coherence protocol to *move* the object to its local cache.

- *Contention Management*: When a transaction requests for an object that is currently used by an active local transaction, the proxy can either abort the local transaction and make the object available, or it can postpone a response to give the local transaction a chance to commit. This decision is made by a globally consistent contention management policy that avoids deadlocks and livelocks. An efficient contention management policy should guarantee progress: at any time, there exists at least one transaction that proceeds to commit without interruption. For example, the Greedy contention manager in [14] guarantees that the transaction with the highest priority can be executed without interruption, using a globally consistent priority policy that issues priorities to transactions.

1.2 Measures of Quality

Distributed transactional memory differs from multiprocessor transactional memory in two key aspects. First, multiprocessor transactional memory designs extend built-in cache-coherence protocols that are already supported in modern multiprocessor architectures. For example, directory-based cache-coherence protocols are used in many large multiprocessor systems [40, 54]. Distributed systems with nodes linked by communication networks typically do not come with such built-in protocols. A distributed cache-coherence protocol has

to be designed. As mentioned before, when a transaction attempts to access an object, the cache-coherence protocol must locate the current cached copy of the object, and move it to the requesting node's cache, invalidating the old copy. For example, in [30], Herlihy and Sun present a *Ballistic* cache-coherence protocol based on hierarchical clustering for tracking and moving up-to-date copies of cached objects, and suggest a finite response time for each transaction.

Secondly, the communication costs for distributed cache-coherence protocols to locate the copy of an object in distributed systems are orders of magnitude larger than that in multiprocessors and are often non-negligible. Such costs are often determined by the different physical locations of nodes that invoke transactions, as well as that of the performance of the cache-coherence protocol used. These costs directly affect the system performance.

For multiprocessor transactional memory systems, the performance of a contention manager A is evaluated by its *competitive ratio*, which is the ratio of the makespan (the last completion time of a given set of transactions) of A to the makespan of an optimal off-line clairvoyant scheduler OPT [3, 14]. For distributed transactional memory systems, the cache-coherence protocol used also affects the makespan. Hence, we have to evaluate the performance of a distributed transactional memory system by taking into account its contention management algorithm as well as the underlying cache-coherence protocol.

We use $makespan(\textsc{Opt})$ to denote the makespan of the optimal clairvoyant off-line scheduling algorithm. We evaluate the performance of the combination of a contention manager A and a cache-coherence protocol C by measuring its *competitive ratio*:

Definition 1 (Competitive Ratio)**.** *The competitive ratio of the combination (A, C) of a contention manager A and a cache-coherence protocol C is:*

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

Hence, the design of a distributed transactional memory system involves the design of the contention manager and the cache-coherence protocol. Our first goal is to design an efficient contention manager A and a cache-coherence protocol C that will minimize $CR(A, C)$. In addition, there are many other measures of quality that we need to consider depending on the underlying network environment. For example, for dynamic environments, we are concerned about load balancing, scalability and availability, etc. We describe these metrics specifically in Chapter 6, where we apply quorum systems to quantitatively define these metrics.

1.3 Summary of Current Research and Contributions

Toward minimizing $CR(A, C)$, we adopt a "lexicographic-like" design strategy: we first select a contention manager, which can guarantee a provable performance under the cache-

coherence protocol with the worst performance. In other words, we first determine the worst-case performance provided by a given contention manager. We then design a cache-coherence protocol to improve performance. Our rationale for this contention manager-first approach (as opposed to cache-coherence protocol-first) is that the performance of cache-coherence protocols is related to contention managers. By first selecting a contention manager, we can determine the performance range that a cache-coherence protocol can achieve.

Now, what contention manager should we select? Motivated by the past work on transactional memory for multiprocessors, we consider the Greedy contention manager in [14] for distributed transactional memory systems. The *pending commit* property of the Greedy manager also applies to distributed systems: at any time, the transaction with the highest priority will be executed and will never be aborted by other transactions. This property is crucial for contention managers to guarantee progress.

We establish the upper and lower bounds of the competitive ratio of the Greedy manager's makespan with an arbitrary cache-coherence protocol. We show that, in the worst case, the competitive ratio is $O(N^2 \cdot s)$, where N is the maximum number of transactions that request the same object and s is the number of objects. On the other hand, the best-case competitive ratio is $\Omega(s)$, which matches the performance of the Greedy manager for multiprocessors. Hence, we need to design cache-coherence protocols to improve the worst-case performance with the Greedy manager.

We propose a class of distributed cache-coherence protocols with *location-aware* property, called LAC protocols. In LAC protocols, the duration of a transaction requesting node to locate an object is determined by the communication cost between the requesting node and the node that holds the object. A 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 "farther" 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.

We also present a novel DHT (distributed hash table)-based cache-coherence protocol, called the DHTC protocol. DHTs provide good load balancing and scalability properties. A common DHT also has a fully decentralized structure, which is essential for loosely-organized applications. Traditional DHT applications often focus on immobile objects. In contrast, the DHTC protocol is able to efficiently track moving objects. We show that the DHTC protocol guarantees an $O(N \cdot s)$ competitive ratio, which is a significant improvement over an arbitrary cache-coherence protocol/Greedy manager combination.

To improve the performance of distributed transactional memory systems in the presence of node failures and arbitrary node joins and departures, we propose a quorum-based cache-coherence protocol. Given a finite universe $U = \{1, ..., n\}$, a *set system* $\mathcal{S} = \{S_1, ..., S_m\}$ is a collection of subsets $S_i \subseteq U$. A *quorum system* is a set system $\mathcal{S}$ that satisfies the *intersection property*: $S_i \cap S_j \neq \emptyset$ for every $S_i, S_j \in \mathcal{S}$. We present the construction of a novel quorum

system called, the dynamic high available B-Grid quorum system or DHB-Grid. We use a probabilistic failure model, where we assume that each node fails independently, and that the failure probability of each node does not exceed p, i.e., the maximum probability that a node fails. We show that the performance of the DHB-Grid system is asymptotically optimal when p approaches 0, and degrades gracefully when p increases. We propose efficient adjustment algorithms for the DHB-Grid system to adjust to network changes (e.g., node joins, node departures), and show an $O(\log n)$ message complexity for each adjustment, where n is the number of nodes. Based on the DHB-Grid system, we propose DHBC, a quorum-based cache-coherence protocol, which exhibits asymptotically optimal load, availability and probe complexity.

To summarize, our research contributions include:

1. We identify that the performance of distributed transactional memory systems is determined by two factors: the contention manager and the cache-coherence protocol used. We show that, for a single object, the optimal off-line clairvoyant scheduler for a set of transactions with the ideal cache-coherence protocol visits all nodes along the shortest Hamiltonian path. This is the first such result;

2. We present a proof of the worst-case competitive ratio of the Greedy contention manager with an arbitrary cache-coherence protocol. We show that this ratio can sometimes lead to the worst choice of transaction execution. In addition, we establish the upper and lower bound of the competitive ratio of the Greedy manager;

3. We present location-aware cache-coherence 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 the Greedy manager/LAC protocol combination, where N is the maximum number of nodes that request the same object, and s is the number of objects;

4. We design a DHT-based cache-coherence protocol, called the DHTC protocol, to improve the performance when combined with the Greedy manager. We show that the DHTC protocol guarantees an $O(N \cdot s)$ competitive ratio, which is a significant improvement over an arbitrary cache-coherence protocol/Greedy manager combination;

5. We design a quorum-based cache-coherence protocol. We show that the performance of a distributed transactional memory system is determined by the performance of the utilized quorum system. We construct a novel quorum system called, the dynamic high available B-Grid quorum system or DHB-Grid, and develop DHBC, a cache-coherence protocol based on DHB-Grid with high availability and low message complexity properties.

1.4 Proposed Post Preliminary-Exam Work

Based on our current research results, we proposed the following work:

- **Contention Manager Design for Distributed Transactional Memory.** Our current results on the combinative behavior of contention managers and cache-coherence protocols for distributed transactional memory is based on a fixed selection of contention managers. Specifically, we select the Greedy manager which can guarantee a worst-case performance without considering the design of cache-coherence protocols. This raises two fundamental and open questions in the design of contention managers:

    1. Can we select other contention managers and then design a compatible cache-coherence protocol, with a combined worst-case performance which is better than that of existing solutions? Various contention managers have been developed in the past [67]. The excellent properties of the Greedy contention manager for multiprocessors motivate our selection of that contention manager for distributed systems. However, this does not preclude the possibility of other contention managers with better performance. For example, it may be possible to establish the worst-case performance of a randomized contention manager, and then design a cache-coherence protocol to improve that worst-case performance. Note that for the Greedy manager, the worst-case competitive ratio is at least $\Omega(s)$ (in our current proposed design, we only guarantee a $O(N \cdot s)$ worst-case competitive ratio), where s is the number of objects and N is the maximum number of transactions requiring accesses to the same object. Thus, establishing improved worst-case performance of other contention managers may result in significant improvement in system performance.
    2. Can we construct a contention manager which is explicitly designed for distributed transactional memory (as opposed to adapting those which have been designed for multiprocessors) and improve the worst-case system performance? Our current work only considers contention managers that have been designed for multiprocessors. There are few results on the behavior of contention managers in a distributed systems context. Hence, it is attractive to focus on the design of contention managers which are dedicated to distributed systems. For the existing (multiprocessor) contention managers, the performance degrades for distributed transactional memory, due to the latency and the dynamic nature of the underlying network, which are not considered for multiprocessors. Hence, the design of such contention manager must take into account these factors to improve the worst-case performance.

- **Contention Manager and Cache-Coherence Protocol Design in the Probabilistic Model.** Our current results propose a probabilistic model of network node failures, where each node may fail independently with a probability less than p, i.e., the

maximum probability that a node fails. In our proposed DHBC protocol, we mainly focus on the availability, scalability, and message complexity properties during system changes. In the future, we propose to focus on the design of contention managers and cache-coherence protocols and analyze their combinative performance based on probabilistic models. Hence, instead of evaluating the performance of the combination of contention managers and cache-coherence protocols by its worst-case performance, we propose to evaluate the expected performance of such combination based on failure probabilistic distributions. This will yield a fundamentally different direction — with consequently different models and protocols – for the design of cache-coherence protocols for distributed transactional memory in dynamic environments.

1.5 Proposal Outline

The rest of this dissertation proposal is organized as follows. In Chapter 2, we overview the transactional memory literature and discuss past and related efforts. We investigate the worst-case and best-case performance of the Greedy manager for distributed transactional memory in Chapter 3. By doing so, we determine the performance range that a cache-coherence protocol can achieve with the Greedy manager. In Chapter 4, we present location-aware cache-coherence protocols, which can improve the performance when combined with the Greedy manager. We present a DHT-based cache-coherence protocol design in Chapter 5. Chapter 6 presents the DHB-Grid quorum system and the DHBC protocol for dynamic environments. We conclude the proposal in Chapter 7.

Chapter 2

Past and Related Work

2.1 Transactional Memory: Overview

The idea of transactional memory is motivated by database transactions, which is a unit of work performed within a database management systems. A database transaction must be ACID [20]: atomicity, consistent, isolated and durable to guarantee reliability. In transactional memory, concurrent threads synchronize via transactions when they access to shared memory. A transaction, in the transactional memory semantics, is a delimited sequence of steps to be executed atomically by a single thread [39]. Atomicity implies an all-or-nothing execution: the sequence of steps (i.e., reads and writes) logically occur at a single instant in time; intermediate states are not visible to other transactions.

The first idea of providing hardware support of transactions was originated in a 1986 paper by Knight [36]. The term "transactional memory" was proposed by Herlihy and Moss in [27], where they first proposed transactional memory as an architectural support for lock-free data structures. The idea was soon popularized and has been the focus of research efforts since then. Despite the early attempts in providing hardware support of transactional memory (HTM) [27, 70], Shavit and Touitou proposed software transactional memory (STM) [64] in 1995, which provides transactional memory semantics in a software runtime library. Since then transactional memory APIs for multiprocessors have been proposed both for hardware [22, 28, 50, 57, 60] and software [7, 8, 23, 24, 29, 33, 46, 48, 52, 63, 65]. Hybrid transactional memories, which allows the implemented STMs make use of advantages of HTMs such that they can be used in conjunction to improve the performance, have also been proposed in the past [6, 37, 51, 66].

2.2 Contention Management: Application and Theory

Contention managers were first proposed in [29], and were widely applied in recent software transactional memory proposals for multiprocessors [8, 15, 16, 28, 68]. For an STM with the obstruction-free property, a contention manager is responsible to ensure that the system as a whole makes progress. A comprehensive survey on contention managers is due to Scherer and Scott [67], where contention managers are empirically evaluated.

The main advantage of obstruction-free synchronization algorithm is because it supports a clean separation of concerns of correctness and progress. The core of an obstruction-free algorithm must maintain data invariants (guaranteeing correctness), and need only 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 [17, 18, 49]. The first theoretical analysis of contention management was presented in [14], where a $O(s^2)$ upper bound is given for the Greedy manager on multiprocessors with s being the number of shared objects. Attiya *et al.* [3] formulated the contention management problem as the *non-clairvoyant* job scheduling paradigm [9, 53] and improved the bound of the Greedy manager to $O(s)$. No contention manager reviewed in the literature guarantee the same worst-case performance as the Greedy manager in the semantics of multiprocessors.

2.3 Distributed Transactional Memory

Despite the large body of work of transactional memory in multiprocessor semantics, few papers in the past [4, 30, 45] investigate transactional memory for distributed systems consisting of a network of nodes. Providing transactional memory support for a cluster of memories in a distributed system is studied both in [4] and [45]. However, the behavior of transactional memory in networks is not considered in both papers. Among these efforts, Herlihy and Sun's work [30] calls our attention mostly. They present a *Ballistic* cache-coherence protocol based on hierarchical clustering for tracking and moving up-to-date copies of cached objects, and suggest a finite response time for each transaction request. They evaluate the performance of the Ballistic protocol by measuring its stretch, which describes the optimality of the duration that moving objects in the network are fetched. Our work, in contrast, studies distributed transactional memory in a more comprehensive way. We argue that the performance of the distributed transactional memory depends on the combination of the employed contention manager and cache-coherence protocol. Hence, our effort focus on the design of efficient contention managers and cache-coherenc protocols and make a judicious choice of their combination.

2.4 Quorum Systems

We employ dynamic quorum systems to support distributed transactional memory in dynamic enviroments. Quorum system is a basic tool for reliable agreement in distributed systems [12, 13, 55, 44, 58, 59, 71]. Researchers have also designed quorum systems in a dynamic environment, e.g. [2, 19, 21, 42, 56, 73]. To apply quorum systems in partitioned networks, Herlihy [26] presented dynamic quorum adjustment method for partitioned data. This method permitted an object's quorum to be adjusted dynamically in response to failures and recoveries. A transaction that is unable to progress using one set of quorums may switch to another, more favorable set, and transactions in different partitions may progress using different sets. Karumanchi et al. [34] proposed strategies that use local knowledge about the reachablility to judiciously select quorums in partitionable mobile ad hoc networks. They designed an update/query protocol to let nodes update their locations when needed. Epidemic quorums [32] have also been applied for managing replicated data, which enables highly available agreement even when a quorum is not simultaneously connected.

Quorum system is widely used in loosely-organized networks, such as ad hoc networks. One of the most popular applications is implementing location service. In the work of Haas and Liang [21], a uniform random quorum system is used for mobility management. Nodes form a virtual backbone. When a node moves, it updates its location with one quorum containing the nearest backbone node. Each source node then queries the quorum containing its nearest backbone for the location of the destination. Probabilistic quorum systems are proposed for dynamic systems [1, 11, 43], where quorums intersect with a probability. Luo et al. [41] present a Probabilistic quorum system for ad hoc networks (Pan), a collection of protocols for the reliable storage of data in mobile ad hoc networks. A gossip-based protocol is designed for quorum access and an asymmetric quorum construction is applied. These work has noticed the highly dynamic and unpredictable topology changes in ad hoc networks. Motivated by the B-Grid quorum system in [55], we design a novel dynamic high availability B-Grid quorum system (DHB-Grid) for dynamic environments, which is dedicated to support distributed transactional memory for dynamic systems.

Chapter 3

Performance Bounds of the Greedy Manager for Distributed Transactional Memory

In this chapter, we establish the upper and lower bounds of the competitive ratio of the Greedy manager's makespan with an arbitrary cache-coherence protocol. We show that, in the worst case, the competitive ratio is $O(N^2 \cdot s)$, where N is the maximum number of transactions that request the same object and s is the number of objects. On the other hand, the best-case competitive ratio is $\Omega(s)$, which matches the performance of the Greedy manager for multiprocessors.

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 [30]. We use a *complete undirected* graph $G = (V, E)$, where $|V| = n$, to model the cost of the underlying network. The *cost* of an edge connecting any two nodes v_i and v_j, denoted $d(i, j)$, is measured by the communication cost between v_i and v_j provided by the underlying routing and other network protocols. We scale the metric so that 1 is the smallest cost between any two nodes. All n nodes are assumed to be contained in a metric space of diameter $Diam$.

3.1.2 Transaction Model.

We are given a set of $m \geq 1$ transactions $T_1, ..., T_m$ and a set of $s \geq 1$ objects $R_1, ..., R_s$. Since each transaction is invoked on an individual node, we use v_{T_i} to denote the node that invokes the transaction T_i, and $V_T = \{v_{T_1}, ..., v_{T_m}\}$. We use $T_i \prec T_j$ to represent that transaction T_i is issued a higher priority than T_j by the contention manager (see distributed transactional memory model).

Each transaction is a sequence of actions, each of which is an access to a single object. Each transaction T_j requires the use of $R_i(T_j)$ units of object R_i for one of its actions. If T_j updates R_i, i.e., a write operation, then $R_i(T_j) = 1$. If it reads R_i without updating, then $R_i(T_j) = \frac{1}{n}$, i.e., the object can be read by all nodes in the network simultaneously. When $R_i(T_j) + R_i(T_k) > 1$, T_j and T_k conflict at R_i. We use $v^0_{R_i}$ to denote the node that holds R_i at the start of the system, and $v^j_{R_i}$ to denote the jth node that fetches R_i. We denote the set of nodes that requires the use of the same object R_i as $V^{R_i}_T := \{v_{T_j} | R_i(T_j) \geq 0, j = 1, ..., m\}$.

An execution of a transaction T_j is a sequence of *timed actions*. Generally, there are four action types that may be taken by a single transaction: *write*, *read*, *commit*, and *abort*. When a transaction is started on a node, a cache-coherence protocol is invoked to locate the current copy of the object and fetch it. The transaction then starts its action sequence and may perform local computations (not involving access to objects) between consecutive actions. A transaction completes either with a commit or an abort. The duration of transaction T_j running locally (without taking into account the time for fetching objects) is denoted as τ_i.

3.1.3 Distributed Transactional Memory Model.

We apply the same data-flow model proposed in Chapter 1.

3.1.4 Problem Statement.

We evaluate the performance of a distributed transactional memory system by measuring its *makespan*. Given a set of transactions accessing a set of objects under a contention manager A and a cache-coherence protocol C, $makespan(A, C)$ denotes the duration that the given set of transactions are successfully executed under the contention manager A and cache-coherence protocol C.

It is well-known that optimal off-line scheduling of tasks with shared resources is NP-complete [10]. 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 OPT, knows the sequence of object accesses of the transaction in each execution.

We use $makespan(\textsc{Opt})$ to denote the makespan of the optimal clairvoyant off-line scheduling algorithm. From Definition 1, we evaluate the performance of the combination of a contention manager A and a cache-coherence protocol C by measuring its *competitive ratio*:

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

Thus, our goal is to solve the following problem: Given a contention manager, how to design a cache-coherence protocol to minimize the competitive ratio?

3.2 Competitive Ratio of the Greedy Manager

3.2.1 Motivation and Challenge.

The past works on transactional memory systems for multiprocessors motivate our selection of the contention manager. 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 contention manager proposed in [14] satisfies this property. Two non-trivial properties are established for the Greedy manager in [14] and [3]:

- 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.

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 cache-coherence protocols, for distributed systems, this cost — which depends on the cache-coherence 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 cache-coherence 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:

(1) 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}.

(1) 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 cache-coherence 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.

3.2.2 Competitive Ratio Analysis

Let the makespan of a set of transactions which require accesses to an object R_i, be denoted as $makespan_i$. It is composed of three parts:

(1) Traveling Makespan ($makespan_i^d$): the total time that R_i travels in the network.

(1) Execution Makespan ($makespan_i^\tau$): the duration of transactions' executions involving R_i, including all successful and aborted executions; and

(1) Idle Time (I_i): the time that R_i waits for a transaction request.

Let the makespan for all move requests for R_i by an optimal off-line algorithm Opt, be denoted as $makespan_i(\text{Opt})$. For the set of nodes $V_T^{R_i}$ that invoke transactions with requests for object R_i, we build a *complete* subgraph $G_i = (V_i, E_i)$, where $V_i = \{V_T^{R_i} \bigcup v_{R_i}^0\}$ and the cost of $e_i(j,k)$ is $d(j,k)$. We use $H(G_i, v_{R_i}^0, v_{T_j})$ to denote the cost of the *minimum-cost Hamiltonian path* that visits each node from $v_{R_i}^0$ to v_{T_j} exactly once. Now, we have:

Theorem 1.

$$makespan_i^d(\text{Opt}) \geq \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j})$$

$$makespan_i^\tau(\text{Opt}) = \sum_{v_{T_j} \in V_T^{R_i}} \tau_j$$

Proof. The execution of the given set of transactions with the minimum makespan schedules each transaction exactly once, which implies that R_i only has to visit each node in $V_T^{R_i}$ once. In this case, the node travels along a Hamiltonian path in G_i starting from $v_{R_i}^0$. Hence, we can lower-bound the traveling makespan by the cost of the minimum-cost Hamiltonian path. On the other hand, object R_i is kept by node v_{T_j} for a duration τ_j for a successful commit. The execution makespan is lower-bounded by the sum of τ_j. The theorem follows. □

The problem of finding a minimum cost Hamiltonian path in a graph is generalized as the *traveling salesman path problem* (or TSPP) [38, 5], a problem closely related to the Traveling Salesman Problem (TSP) that replaces the constraint of a *cycle* by a *path*. It is well-known that finding such an algorithm is NP-hard. For an undirected metric-space network, where edge lengths satisfy the triangle inequality, the best known approximation ratio is 5/3 due to Hoogeveen [31].

Generally, the shortest way to visit all the vertices of an arbitrary graph may not be a simple path, i.e., it may visit some vertices or edges multiple times. However, G_i is a *metric completion graph* where the cost between any pair of nodes is the cost of the shortest path connecting the nodes. Under this condition, the shortest way to visit all the vertices of G_i is a simple path that visits each vertex exactly once.

Now we focus on the makespan of the Greedy manager with a given cache-coherence protocol C. As mentioned before, to implement distributed transactional memory, a distributed cache-coherence protocol is needed, and the cost for locating and moving objects must be taken into account. We define these costs:

Definition 2 (Locating Cost)**.** *In a given metric-space network, the locating cost $\delta^C(i,j)$ is the cost for a transaction running on a node i to successfully locate an object held by node j under a cache-coherence protocol C.*

The locating cost does not include the cost for moving the object. When the object is located, we assume that the cost for moving the object from node j to node i is $d(i,j)$. We use the metric *stretch* to evaluate the responsiveness of a cache-coherence protocol.

Definition 3 (Stretch)**.** *The stretch of a cache-coherence protocol C for a given metric-space network $G=(V,E)$ is the maximum ratio of the locating cost to the cost between two nodes:*

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

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 2.

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

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

Definition 4 (Priority Hamiltonian Path)**.** *The priority Hamiltonian path for a subgraph G_i is a path which starts from $v^0_{R_i}$ and visits each node from the lowest priority to the highest priority.*

Formally, the priority Hamiltonian path is $v^0_{R_i} \rightarrow v_{T_{N_i}} \rightarrow v_{T_{N_i-1}} \cdots \rightarrow v_{T_1}$, where $N_i = |V_T^{R_i}|$ and $T_1 \prec T_2 \prec \ldots \prec T_{N_i}$. We use $H^p(G_i, v^0_{R_i})$ 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 $B(t) = \{b_i(t)|b_1(t) \prec b_2(t) \prec \ldots\}$. 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 overtook by other transactions if they are closer to $r(t)$. In the worst case, the transaction requested by $b_1(t)$ is overtook 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 $r(i) \to b_{|B(t)|}(t) \to b_{|B(t)|-1}(t) \to \ldots \to b_1(t)$. The cost of this path is composed of two parts: the cost of $r(i) \to b_{|B(t)|}(t)$ and the cost of $b_{|B(t)|}(t) \to b_{|B(t)|-1}(t) \to \ldots \to b_1(t)$. 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 $makespan^d_i(Greedy, C)$:

$$makespan^d_i(Greedy, C) \le 2N_i \cdot H^p(G_i, v^0_{R_i}),$$

The upper bound of the execution makespan can be proved directly. For any transaction T_j, it can be overtook 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 $\sum_{j \le k \le N_i} \tau_k$. By summing them over all transactions, we have:

$$makespan^\tau_i(Greedy, C) \le \sum_{1 \le j \le N_i} j \cdot \tau_j$$

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 cache-coherence protocol. The system will not stop after the first object has been located.

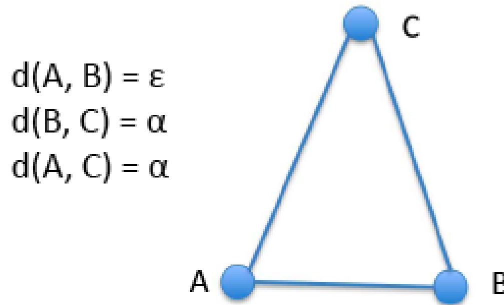

Figure 3.1: Example 1: A 3-node network

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

$$I_i(Greedy, C) \le N_i \cdot Stretch(C) \cdot I_i(\text{Opt})$$

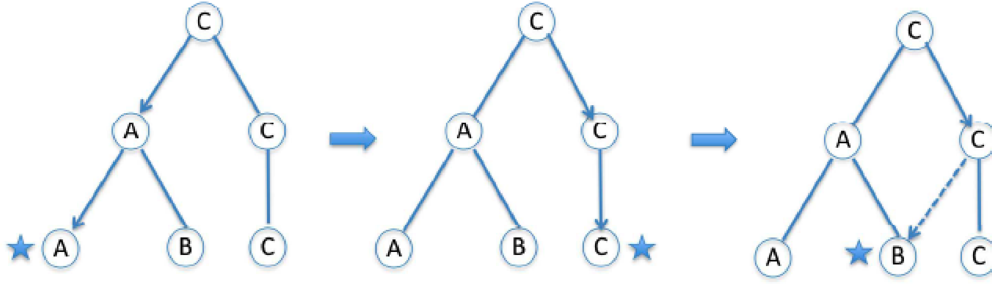

Figure 3.2: Example 1: Link path evolution of the directory hierarchy built by Ballistic protocol

The theorem follows. □

Example 1: in the following example, we show that for the *Ballistic protocol* [30], the upper bound in Theorem 2 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 3.1. Based on the Ballistic protocol, a 3-level directory hierarchy is built, shown in Figure 3.2. 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 the Ballistic protocol, 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 no-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 2 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 $(Greedy, C)$ for s objects. Let $N = \max_{1 \leq i \leq s} N_i$, i.e., N is the maximum number of nodes that request the same object.

Theorem 3.

$$CR(Greedy, C) = O(\max[N^2 \cdot s, N \cdot Stretch(C)])$$

Proof. We first derive the bounds of $makespan^d$ and $makespan^\tau$ 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, $makespan^\tau(\text{OPT}))$ satisfies:

$$makespan^\tau(\text{OPT}) \geq \frac{\sum_{i=1}^m \tau_i}{s}.$$

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

$$makespan^d(\text{OPT}) \geq \max_{1 \leq i \leq s} \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}).$$

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

$$makespan(\text{OPT}) \geq I(\text{OPT}) + \frac{\sum_{i=1}^m \tau_i}{s} + \max_{1 \leq i \leq s} \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}).$$

Note that, whenever the Greedy manager is not idle, at least one of the transactions that is processed will be completed. However, from Theorem 2, we know that it may be overtook by all transactions with lower priorities, and therefore the penalty time cannot be ignored. Using the same argument of Theorem 2, we have:

$$makespan^\tau(Greedy, C) \leq \sum_{i=1}^{s} \sum_{k=1}^{N_i} k \cdot \tau_k.$$

The traveling makespan of transaction T_j is the sum of the traveling makespan of each object that T_j involves. We have:

$$makespan^d(Greedy, C) \leq \sum_{i=1}^{s} 2N_i \cdot H^p(G_i, v_{R_i}^0) \leq s \cdot 2N^2 \cdot \max_{1 \leq i \leq s} \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}).$$

Hence, the makespan of the Greedy manager satisfies:

$$makespan(Greedy, C) \leq I(Greedy, C) + \sum_{i=1}^{s} \sum_{k=1}^{N_i} k \cdot \tau_k + s \cdot 2N^2 \cdot \max_{1 \leq i \leq s} \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}).$$

The theorem follows. □

Theorem 3 provides an upper bound on the competitive ratio of the Greedy manager. In other words, it describes the worst-case performance of the Greedy manager. On the other hand, what is the best performance that we can expect with the Greedy manager? Can we prove a lower bound on the competitive ratio of the Greedy manager? We have:

Theorem 4.

$$makespan_i^d(Greedy, C) \geq H^p(G_i, v_{R_i}^0)$$

$$makespan_i^\tau(Greedy, C) \geq \sum_{v_{T_j} \in V_T^{R_i}} \tau_j$$

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. □

In the best case, the cache-coherence protocol C provides a constant stretch. Hence, we can establish a lower bound for the Greedy manager:

Theorem 5.

$$CR(Greedy, C) = \Omega(s)$$

Proof. The theorem can be proved by using the same argument of Theorem 3 combined with Theorem 4. □

3.3 Conclusion

Theorem 3 and 5 give the performance range of the Greedy manager for distributed transactional memory. Hence, the design goal is to minimize the cost of the dominating part among the three parts of the total makespan. From Theorem 3, we observe that the dominating part is often either the traveling makespan or the idle time. In the following chapter, we propose a class of cache-coherence protocols which provide an $O(N \log N \cdot s)$ competitive ratio, which significantly improves the bound of Theorem 3.

Chapter 4

Location-Aware Cache-Coherence Protocols for Distributed Transactional Memory

In this chapter, we propose a class of distributed cache-coherence 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 Cache Responsiveness

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

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

Let

$$V_T^{R_i}(T_j) = \{v_{T_k} | v_{T_k} \prec v_{T_j}, v_{T_k}, v_{T_j} \in V_T^{R_i}\}$$

for any graph G. Let

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

and

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

We have the following theorem.

Theorem 6. *A transaction T_j's request for object R_i with the Greedy manager and cache-coherence protocol C is satisfied within time*

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

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 $v_{R_i} \in V_T^{R_i}(T_j)$. The overtaking failure can happen at most $|V_T^{R_i}(T_j)|$ times until all transactions that have higher priority than T_j commit. The lemma follows.

□

Theorem 6 shows that for a set of objects, the responsiveness for a cache-coherence protocol is determined by its locating cost.

4.2 Location-Aware Cache-Coherence Protocols

We now define a class of cache-coherence protocols which satisfy the following property:

Definition 5 (Location-Aware Cache-Coherence 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 cache-coherence 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 cache-coherence 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 2. The following theorem gives the upper bound of $CR_i^d(Greedy, LAC)$.

Theorem 7.

$$CR_i^d(Greedy, LAC) = O(N_i \log N_i)$$

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 $v^0_{R_i}$ that visits all nodes with lower priorities than T_j.

Definition 6. *Nearest Neighbor Path: In a graph G, the nearest neighbor path is constructed as follows [62]:*

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 $\{v^1_{R_i} \prec, ..., \prec v^{N_i}_{R_i}\}$ to denote these nodes in the priority-order, then in the worst case, the object may travel in the reverse order before arriving at $v^1_{R_i}$. 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 $v^0_{R_i} \rightarrow v^{N_i}_{R_i} \rightarrow ... \rightarrow v^2_{R_i} \rightarrow v^1_{R_i}$. In this path, transaction invoked by by $v_{R^j_i}$ is overtaken by all transactions with priorities lower than j, implying

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

and

$$d(v^j_{R_i}, v^{j-1}_{R_i}) < d(v^j_{R_i}, v^k_{R_i}), 1 \leq k \leq j - 2.$$

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

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 [62].

$$\frac{NN(G^j_i, v^{j-1}_{R_i})}{\min_{v^k_{R_i} \in G^j_i} H(G_i, v^{j-1}_{R_i}, v^k_{R_i})} \leq \lceil \log(N_i - j + 1) \rceil + 1 \tag{4.1}$$

Theorem 1 from [62] 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 $makespan^d_i(Greedy, LAC)$:

$$makespan^d_i(Greedy, LAC) \leq \sum_{1 \leq j \leq N_i} NN(G^j_i, v^{j-1}_{R_i})$$

$$\leq \sum_{1\leq j\leq N_i} \min_{v^k_{R_i}\in G^j_i} H(G^j_i, v^{j-1}_{R_i}, v^k_{R_i}) \cdot (\lceil \log(N_i - j + 1)\rceil + 1).$$

Note that

$$\min_{v_{T_j}\in V^{R_i}_T} H(G_i, v^0_{R_i}, v_{T_j}) \geq \min_{v^k_{R_i}\in G^j_i} H(G^j_i, v^{j-1}_{R_i}, v^k_{R_i}).$$

Combined with Theorem 2, we derive the competitive ratio for traveling makespan of the Greedy manager with a LAC protocol as:

$$CR^d_i(Greedy, LAC) = \frac{makespan^d_i(Greedy, LAC)}{makespan^d_i(\textsc{Opt})}$$

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

The theorem follows. □

Example 2: We now revisit the scenario in Example 1 by applying LAC protocols. Note that $T_B \prec T_C$ 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 $T_B \prec T_C$ to $T_C \prec T_B$. In this scenario, the upper bound of Theorem 7 is asymptotically tight. T_C may be overtook 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 2 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.

4.3 $makespan(Greedy, LAC)$ for Multiple Objects

Theorem 7 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 the same object. Now,

Theorem 8. *The competitive ratio CR(Greedy, LAC) is*

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

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

$$I(Greedy, LAC) \leq N \cdot Stretch(LAC) \cdot Diam \cdot I(\textsc{Opt}) \tag{4.2}$$

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 $N_i \cdot \delta^{LAC}(i, j)$ 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 cache-coherence 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 $makespan^d$ and $makespan^\tau$ 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, $makespan^\tau(\textsc{Opt}))$ satisfies:

$$makespan^\tau(\textsc{Opt}) \geq \frac{\sum_{i=1}^m \tau_i}{s}.$$

In the optimal schedule, s writing transactions run concurrently, implying each object R_i travels independently. From Theorem 2, $makespan^d(\textsc{Opt})$ satisfies:

$$makespan^d(\textsc{Opt}) \geq \max_{1 \leq i \leq s} \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}).$$

Hence, we bound the makespan of the optimal schedule by

$$makespan(\textsc{Opt}) \geq I(\textsc{Opt}) + \frac{\sum_{i=1}^m \tau_i}{s} + \max_{1 \leq i \leq s} \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}).$$

Note that, whenever the Greedy manager is not idle, at least one of the transactions that are processed will be completed. However, from Theorem 3 we know that it may be overtook

by all transactions with lower priorities, and therefore the penalty time cannot be ignored. Use the same argument of Theorem 3, we have

$$makespan^{\tau}(Greedy, LAC) \leq \sum_{i=1}^{s} \sum_{k=1}^{N_i} k \cdot \tau_k.$$

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 7, we have

$$makespan^{d}(Greedy, LAC) \leq \sum_{i=1}^{s} \sum_{j=1}^{N_i} NN(G_i^j, v_{R_i}^{j-1})$$

$$\leq s \cdot \sum_{j=1}^{N} \min_{v_{R_i}^k \in G_i^j} H(G_i^j, v_{R_i}^{j-1}, v_{R_i}^k) \cdot (\lceil \log(N-j+1) \rceil + 1)$$
$$\leq s \cdot (\lceil \log(N!) \rceil + N_i) \cdot \max_{1 \leq i \leq s} \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}).$$

Hence, the makespan of the Greedy manager with a LAC protocol satisfies:

$$makespan(Greedy, LAC) \leq I(Greedy, LAC) + \sum_{i=1}^{s} \sum_{k=1}^{N_i} k \cdot \tau_k$$

$$+ s \cdot (\lceil \log(N!) \rceil + N) \cdot \max_{1 \leq i \leq s} \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v_{R_i}^0, v_{T_j}).$$

The theorem follows. □

Remarks: Theorem 8 generalizes the performance of makespan of (Greedy,LAC). In order to lower this upper bound as much as possible, we need to design a LAC protocol with $Stretch(LAC) \leq s \log N$. In this case, the competitive ratio for (Greedy, 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 the Greedy manager degraded for distributed systems. As we stated in Section 3.2, the penalty of overtaking failures is the main reason for the performance degradation.

4.4 Conclusions

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

cache-coherence protocol in Chapter 3. Hence, we propose a location-aware property for cache-coherence 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 cache-coherence protocols; then find an appropriate cache-coherence protocol to improve performance.

In this chapter we propose a class of cache-coherence protocols with location-aware property. This is the first attempt to investigate the combinative behavior of contention managers and cache-coherence protocols. For all LAC protocols, the competitive ratio upper bound is $O(N \log N \cdot s)$ when combined with the Greedy manager. Is this the best we can do? In the next chapter, we propose a DHT-based cache-coherence protocol which exhibits a better performance.

Chapter 5

A DHT-Based Cache-Coherence Protocol for Distributed Transactional Memory

In this chapter, we present a novel DHT (distributed hash table)-based cache-coherence protocol, called the DHTC protocol. DHTs provide good load balancing and scalability properties. A common DHT also has a fully decentralized structure, which is essential for loosely-organized applications. Traditional DHT applications often focus on immobile objects. In contrast, the DHTC protocol is able to efficiently track moving objects. We show that the DHTC protocol guarantees an $O(N \cdot s)$ competitive ratio, which is a significant improvement over an arbitrary cache-coherence protocol/Greedy manager combination.

5.1 Protocol Description.

We build a DHT in the network which is used by the proposed cache-coherence protocol. The DHT performs the following operation [69]: given a key, it maps the key on to a node. The key is the filename of the data object, which is used as a keyword to identify the object. The DHT-based lookup protocol can efficiently locate the node which holds the key. It uses *consistent hashing*, which assigns an *identifier* (ID) to each node and key using a base hash function such as SHA-1. A node's ID is chosen by hashing its IP address, while a key's ID is produced by hashing the key itself. Consistent hashing assigns keys to nodes in an efficient way such that the load is balanced with high probability, i.e., all nodes receive approximately the same number of keys.

In a DHT, each node maintains a set of links (i.e., a routing table). To lookup a key, a node selects a neighbor in the routing table. The most essential property of the DHT is that, for any key ID k, the node either owns k or has a link to a node that is closer to k in terms of

the *keyspace distance* defined by the specific DHT to represent an abstract distance between two key IDs. The routing algorithm to the owner of k can then be implemented in a greedy way: at each step, forward the routing message to the neighbor in the routing table whose ID is closest to k. The message arrives at the owner of k when there is no such neighbor. Two important metrics are used to evaluate a DHT: (1) the maximum number of forwarding hops in any route (or route length) and (2) the maximum number of neighbors in the routing table (or maximum node degree). We assume that our applied DHT has an $O(\log n)$ bound for both route length and maximum node degree, for an n-node network, which is common for most DHTs [61, 69, 72].

While DHTs provide an effective way to locate objects, those objects are typically considered to be immobile [61, 69, 72]. In such contexts, an object just has to be published once. However, in our data-flow model, objects are moving in the network. How do we track the moving object in the network efficiently? Intuitively, when the object is moved to a new node, it has to be republished so that the node owning the key has the latest information of the location of the object. However, this strategy is not efficient since it induces large cost for republishing. In our proposed cache-coherence protocol, called DHTC, an object is published when it is created, and no republishing is needed.

The following operation is performed to publish an object:

<u>`Publish(`R_i`)`</u>: When an object R_i is created at node $v^0_{R_i}$, it produces the key ID of R_i, denoted by $key(R_i)$. The DHT lookup protocol is invoked to locate the node that is mapped to $key(R_i)$, denoted by $home(R_i)$. Node $home(R_i)$ keeps a link $home(R_i).link(R_i)$ to node $v^0_{R_i}$ when it receives the publish message.

Now transactions can be invoked to request accesses to R_i. All requests are forwarded to $home(R_i)$. The link state of $home(R_i)$ is updated whenever a transaction request arrives at $home(R_i)$. A two-phase operation is performed for a transaction T_j to lookup the latest location of R_i:

<u>`Lookup(`R_i`)` (Up Phase)</u>: When a transaction T_j invoked at v_{T_j} requests a read or write access to R_i, it calls the DHT lookup protocol to locate $home(R_i)$. Node $home(R_i)$ forwards the message to $home(R_i).link$ after receiving the request message. Link $home(R_i).link$ is updated by v_{T_j}, i.e., $home(R_i).link$ keeps track of the latest transaction request of R_i.

In the DHTC protocol, a transaction request for an object R_i is only forwarded to $home(R_i)$ when the request is made for the first time. When the transaction is aborted, it is restarted immediately. When restarted, the transaction does not need to send the previous requests again. Instead, the cache-coherence protocol "saves" those requests by updating link states of each node involved in the transaction requests for the same object. By doing so, the DHTC protocol just needs to locate the object once for each transaction even if the transaction is aborted many times.

Each node sets a datum $flag(T_k)$ after it invokes a transaction T_k. If T_k has successfully committed, $flag(T_k) = 1$; otherwise $flag(T_k) = 0$. For any transaction $T_k \in V_T^{R_i}$, it may

encounter multiple access conflicts on the same object from other transactions. We arrange those transactions in the chronological order, denoted as $\{T_k^1(R_i), T_k^2(R_i), ..., T_k^l(R_i)\}$. The following links are maintained by node v_{T_k}:

- Next transaction link $next(R_i)$: Assume that transaction T_k does not encounter any access conflict on R_i until it commits. Object R_i is kept by v_{T_k} until a new transaction request $T_{k'}$ of R_i is received after T_k's commit. Link $v_{T_k}.next(R_i)$ is updated by $v_{T_{k'}}$.
- Successor transaction link $suc(R_i)$: Assume that transaction T_k encounters access conflicts on R_i during its execution. At any time, $v_{T_k}.suc(R_i) = v_{T_k^j(R_i)}$, where $T_k^j(R_i)$ is the highest priority transaction among all transactions in $\{T_k^1(R_i), T_k^2(R_i), ..., T_k^l(R_i)\}$ whose priorities are lower than T_k.
- Predecessor transaction link $pre(R_i)$: Assume that transaction T_k is aborted by transaction $T_k^{l'}(R_i)$. In this case, $T_k^{l'}(R_i) \prec T_k$. Link $v_{T_k}.pre(R_i)$ is updated by $v_{T_k^{l'}(R_i)}$.

<u>`Lookup`(R_i) (Down Phase)</u>: When a transaction T_j's request for accessing object R_i arrives at node v_{T_k}, $flag(T_k)$ is checked:

- If $v_{T_k}.flag(T_k) = 1$, then T_k has successfully committed. Two possible cases exist:
  - Object R_i is held by v_{T_k}. R_i has been located and the lookup process is completed.
  - Object R_i is not held by v_{T_k}. The transaction request is now forwarded to $v_{T_k}.next(R_i)$ or $v_{T_k}.suc(R_i)$ by examining those two links. Depending on our updating policy, only one of them is a non-null link.
- If $v_{T_k}.flag(T_k) = 0$, then T_k has not successfully committed. Two possible cases exist:
  - Object R_i is held by v_{T_k}. R_i has been located and the lookup process is completed.
  - Object R_i is not held by v_{T_k}. The transaction request is forwarded to $v_{T_k}.pre(R_i)$.

`Lookup`(R_i) performs one operation: forwarding a transaction's request to the latest location of the object R_i. The following operation is performed after the object is located:

<u>`Execute`(R_i)</u>: When a transaction T_j that requests access of R_i locates R_i at node v_{T_k}, $flag(T_k)$ is checked:

- If $v_{T_k}.flag(T_k) = 1$, then T_k has successfully committed. The object R_i is idle, and is moved to v_{T_j}. v_{T_k}'s next transaction link is updated as $v_{T_k}.next(R_i) = v_{T_j}$.
- If $v_{T_k}.flag(T_k) = 0$, then R_i is currently in use by transaction T_k. The local contention manager of v_{T_k} compares priorities of T_k and T_j. Two possible cases exist:

- If $T_j \prec T_k$, then T_k is aborted. $v_{T_k}.pre(R_i)$ is set to v_{T_j} (i.e., $v_{T_k}.pre(R_i) = v_{T_j}$). Object R_i is moved to v_{T_k} and $v_{T_j}.suc(R_i)$ is set to v_{T_k} (i.e., $v_{T_j}.suc(R_i) = v_{T_k}$).
- If $T_k \prec T_j$, then T_j is postponed to let T_k commit. Link $v_{T_k}.suc(R_i)$ is then examined. If it is *null*, then it is set to v_{T_j}. A *link update message* $lnkmsg(v_{T_k}, R_i)$ is sent to v_{T_j}. Otherwise, the contention manager compares the priorities of T_j and $T_{k'}$, where $T_{k'}$ is the transaction request on R_i from $v_{T_k}.suc(R_i)$. There are two possible cases:
  * If $T_j \prec T_{k'}$, $v_{T_k}.suc(R_i)$ is set to v_{T_j} (i.e., $v_{T_k}.suc(R_i) = v_{T_j}$). Link update messages $lnkmsg(v_{T_{k'}}, R_i)$ and $lnkmsg(v_{T_j}, R_i)$ are sent to v_{T_j} and $v_{T_{k'}}$, respectively.
  * If $T_{k'} \prec T_j$, a link update message $lnkmsg(v_{T_j}, R_i)$ is sent to $v_{T_{k'}}$ and a link update message $lnkmsg(v_{T_{k'}}, R_i)$ is sent to v_{T_j}.

For any transaction T_j, $v_{T_j}.suc(R_i)$ is checked for each requested object R_i after it commits. Object R_i is moved to $v_{T_j}.suc(R_i)$ if it is a non-null link.

A link update message may be sent after `Execute(`R_i`)`. The following operation is performed for nodes receiving a link update message:

`Update(`$lnkmsg(v_{T_j}, R_i)$`)`: When a link update message $lnkmsg(v_{T_j}, R_i)$ arrives at node v_{T_k}, the local contention manager compares the priorities of T_j and T_k:

- If $T_j \prec T_k$, then $v_{T_k}.pre(R_i)$ is set to v_{T_j} (i.e., $v_{T_k}.pre(R_i) = v_{T_j}$).
- If $T_k \prec T_j$, then link $v_{T_k}.suc(R_i)$ is checked:
  - If $v_{T_k}.suc(R_i) = null$, then $v_{T_k}.suc(R_i)$ is set to v_{T_j} (i.e., $v_{T_k}.suc(R_i) = v_{T_j}$).
  - If $v_{T_k}.suc(R_i) \neq null$, the local contention manager compares the priorities of T_j and $T_{k'}$, where $T_{k'}$ is the transaction request on R_i from $v_{T_k}.suc(R_i)$. Two cases exist:
    * If $T_j \prec T_{k'}$, $v_{T_k}.suc(R_i)$ is set to v_{T_j} (i.e., $v_{T_k}.suc(R_i) = v_{T_j}$). Link update messages $lnkmsg(v_{T_{k'}}, R_i)$ and $lnkmsg(v_{T_j}, R_i)$ are sent to v_{T_j} and $v_{T_{k'}}$, respectively.
    * If $T_{k'} \prec T_j$, a link update message $lnkmsg(v_{T_j}, R_i)$ is sent to $v_{T_{k'}}$ and a link update message $lnkmsg(v_{T_{k'}}, R_i)$ is sent to v_{T_j}.

5.2 Protocol Analysis

Protocol Correctness. We prove the correctness of the protocol in two steps. First, we show that `Lookup(`R_i`)` can efficiently locate R_i. In the up phase, transactions invoked by

nodes in $V_T^{R_i}$ locate $home(R_i)$ by using the DHT lookup protocol. Let the transactions invoked by nodes in $V_T^{R_i}$ be arranged in the chronological order by which they arrive at $home(R_i)$, denoted as $\{T^1(R_i), T^2(R_i), ..., T^{N_i}(R_i)\}$. Now, we have:

Theorem 9. *In the down phase of the* `Lookup`*(R_i), transaction $T^j(R_i)$ can locate R_i in at most $j-1$ hops.*

Proof. Assume that the object is located at $v_{T^k(R_i)}$, where $1 \leq k \leq j-1$. If $T^k(R_i)$ has committed, i.e., object R_i is idle, then there are two possible cases:

- *Case 1*: $k = j-1$. Now, R_i is located in hop 1.
- *Case 2*: $k < j-1$. Now, $T^{j-1}(R_i) \prec T^k(R_i)$. In this case, $T^{j-1}(R_i)$ must have committed and R_i is moved through $v_{T^k(R_i)}.suc(R_i)$. In the worst case, $T^{j-1}(R_i)$ is the highest priority transaction in $\{T^1(R_i), T^2(R_i), ..., T^{j-1}(R_i)\}$ and is overtook by $\{T^1(R_i), T^2(R_i), ..., T^{j-2}(R_i)\}$. Thus, the object is moved to the lowest transaction in $\{T^1(R_i), T^2(R_i), ..., T^{j-2}(R_i)\}$. Therefore, the transaction request will be forwarded through successor transaction links, starting from $v_{T^k(R_i)}.suc(R_i)$ to the lowest transaction in $\{T^1(R_i), T^2(R_i), ..., T^{j-2}(R_i)\}$. Hence, it takes at most $j-1$ hops for transaction $T^j(R_i)$ to locate R_i.

In the same way, we can prove that it takes at most $j-1$ hops if R_i is not idle. Note that in this case, the transaction request will be forwarded through the predecessor transaction links, starting from $T^{j-1}(R_i)$, in the worst case. The theorem follows. □

In the second step, we prove that all transactions will successfully commit under the DHTC protocol.

Theorem 10. *Object R_j moves at most N_i hops after transaction $T^{N_i}(R_i)$ locates R_i to let all transactions in $V_T^{R_i}$ commit successfully.*

Proof. Assume that $T^{N_i}(R_i)$ locates R_i at $v_{T^k(R_i)}$. If R_i is idle, then all transactions in $\{T^1(R_i), T^2(R_i), ..., T^{N_i-1}(R_i)\}$ have committed and R_i just needs to move one hop to let $T^{N_i}(R_i)$ commit. If R_i is not idle, we divide $\{T^1(R_i), T^2(R_i), ..., T^{N_i-1}(R_i)\}$ into two subsets such that $\{T^{l_1}(R_i), T^{l_2}(R_i), ..., T^{l_r}(R_i)\}$ is the set of transactions that have committed, and $\{T^{l_{r+1}}(R_i) \prec T^{l_{r+2}}(R_i) \prec ... \prec T^{l_{N_i-1}}(R_i)\}$ is the set of transactions that have not committed. We observe that $T^k(R_i) = T^{l_{r+1}}(R_i)$. There are two possible cases:

- *Case 1*: $T^{N_i}(R_i) \prec T^k(R_i)$. Now, $T^k(R_i)$ is aborted and $v_{T^{N_i}(R_i)}$ is set as its predecessor. Object R_i is moved to $v_{T^{N_i}(R_i)}$ to let $T^{N_i}(R_i)$ commit. After $T^{N_i}(R_i)$'s commit, R_i is moved through successor transaction links starting from $T^{N_i}(R_i).suc(R_i)$ to let all transactions in $\{T^{l_{r+1}}(R_i), T^{l_{r+2}}(R_i), ..., T^{l_{N_i-1}}(R_i)\}$ commit. Since there is no transaction request after $T^{N_i}(R_i)$, R_i only needs to traverse each node that invokes transaction

in $\{T^{l_{r+1}}(R_i), T^{l_{r+2}}(R_i), ..., T^{l_{N_i-1}}(R_i)\}$ to let all transactions commit. Since there is at most $N_i - 1$ elements in $\{T^{l_{r+1}}(R_i), T^{l_{r+2}}(R_i), ..., T^{l_{N_i-1}}(R_i)\}$, the maximum number of hops moved by R_i is N_i.

- *Case 2*: $T^k(R_i) \prec T^{N_i}(R_i)$. According to our cache-coherence protocol, the links of nodes invoking transactions in $\{T^{l_{r+1}}(R_i), T^{l_{r+2}}(R_i), ..., T^{l_{N_i-1}}(R_i)\}$ are rearranged, and there exists a transaction $T^{l_{r'}}(R_i)$ such that $v_{T^{l_{r'}}(R_i)}.suc(R_i) = v_{T^{N_i}(R_i)}$ and $v_{T^{N_i}(R_i)}.suc(R_i) = v_{T^{l_{r'+1}}(R_i)}$. After $T^k(R_i)$'s commit, R_i is moved through successor transaction links starting from $T^k(R_i).suc(R_i)$ to let all transactions in $\{T^{l_{r+1}}(R_i),$ $T^{l_{r+2}}(R_i), ..., T^{l_{r+1}}(R_i), T^{N_i}(R_i), T^{l_{r+2}}(R_i), ..., T^{l_{N_i-1}}(R_i)\}$ commit. Using the same argument as before, the theorem follows.

□

Performance Analysis. Having established the correctness of the protocol, we now evaluate its performance.

Theorem 11.

$$Stretch(DHTC) = O(N \cdot Diam)$$

Proof. The locating cost of DHTC for transaction $v_{T_j}(R_i)$ is composed of two parts. The first part is the cost to locate $home(R_i)$, which is $O(\log N_i \cdot Diam)$ for the DHT lookup protocol. The second part is the cost to forward the transaction request from $home(R_i)$ to the location of R_i, which is $O(N_i \cdot Diam)$ from Theorem 9. Summing the two parts, the theorem follows. □

Our stretch is higher than that of the Ballistic protocol in [30] because we track the latest location of the object. For the Ballistic protocol, each transaction request tracks its predecessor, which changes the link state of the directory. When a transaction is aborted, the Ballistic protocol is called again to locate the object.

In the DHTC protocol, the idle time to process all transactions is reduced since each transaction only needs to locate the object once, no matter how many times it is aborted. In the worst case, all transactions are executed in the priority order after the last transaction has located the object.

Theorem 12.

$$I_i(Greedy, DHTC) \le N_i \cdot Diam \cdot I_i(\textsc{Opt}) \quad (5.1)$$

$$makespan_i^d(Greedy, DHTC) \le 2N_i \cdot \min_{v_{T_j} \in V_T^{R_i}} H(G_i, v^0_{R_i}, v_{T_j}) \quad (5.2)$$

Proof. Equation (5.1) can be directly proved by Theorem 11. To prove Equation (5.2), we only have to show that R_i moves at most $2N_i$ hops to let all transactions invoked by $V_T^{R_i}$ commit. We know that there are only two possible outcomes when R_i moves to a new location: transaction commit or abort. Since there are only N_i transaction requests to locate R_i, transactions invoked by $V_T^{R_i}$ are aborted at most N_i times. Hence, the total number of hops that R_i travels is at most $2N_i$. The theorem follows. □

Finally, we establish the competitive ratio of $(Greedy, DHTC)$:

Theorem 13. $CR(Greedy, DHTC) = O(N \cdot s)$

Proof. The theorem can be proved by using the same argument of Theorem 3 combined with Theorem 12. □

Compared to Theorem 3's upper bound, the performance is significantly improved.

5.3 Concluding Remarks

Two main factors determine the performance of cache-coherence protocols for distributed transactional memory: the total locating cost and the extra traveling makespan of overtaking failures. The locating cost is measured by the stretch of the cache-coherence protocol. Since locating moving objects in the network induces large cost, we reduce the total locating cost by ensuring that each transaction request locates the object just at the first time when the request is invoked. Once the object is located, the transaction request will be saved in the network until it commits. By doing this, the total locating cost increases only as a polynomial of N, where N is the maximum number of nodes that request the same object. On the other hand, the total number of overtakings does not exceed N because an overtaking only occurs when a new transaction request locates the object. Consequently, the combination of $(Greedy, DHTC)$ guarantees an improved worst-case performance.

We have assumed a static network. DHTs also exhibit good performance when nodes can enter or leave the physical network. However, when a node $home(R_i)$ crashes, the link to the node which invokes the last transaction requesting R_i is lost. Hence, some failure detection mechanism [35] should be designed for each $home(R_i)$ and will cause large message overhead in the network. In the next chapter, we apply quorum systems to support distributed transactional memory for dynamic systems.

Chapter 6

A Quorum-Based Cache-Coherence Protocol for Distributed Transactional Memory

In this chapter, we propose DHBC, a quorum-based cache-coherence protocol for distributed transactional memory for dynamic systems. We prove that DHBC guarantees asymptotically optimal load, availability and prove complexity in the presence of node failures and system changes.

6.1 Definitions and Preliminaries

6.1.1 Quorum Systems

Definition 7. *Given a finite universe $U = \{1, ..., n\}$, a* set system *$\mathcal{S} = \{S_1, ..., S_m\}$ is a collection of subsets $S_i \subseteq U$. A* quorum system *is a set system $\mathcal{S}$ that satisfies the* intersection property*: $S_i \cap S_j \neq \emptyset$ for every $S_i, S_j \in \mathcal{S}$. The sets of the system are called* quorums.

6.1.2 Metrics of Quality

The quality of quorum systems is evaluated by a following metrics.

- *Load*: A *strategy* is a probabilistic rule that governs which quorum is chosen each time. In other words, a strategy is a probabilistic distribution over quorum sets, giving each quorum a probability by which it is accessed by the user. Formally, we have the following definition:

Definition 8. *Given a quorum system $\mathcal{S} = \{S_1, ..., S_m\}$ over a universe U, then $w \in [0,1]^m$ is a strategy for $\mathcal{S}$ if it is a probability distribution over the quorums $S_j \in \mathcal{S}$, i.e., $\sum_{j=1}^{m} w_j = 1$.*

A strategy induces a load on each element, which is the sum of the probabilities of quorums it belongs to. For a given quorum system $\mathcal{S}$, the load induced by a strategy w on a quorum system $\mathcal{S}$ is the load on the *busiest* element i, i.e., $\mathcal{L}_w(\mathcal{S}) = \max_{i \in U} l_w(i)$, where $l_w(i) = \sum_{S_j \ni i} w_j$. The system load is the minimum load over all strategies w: $\mathcal{L}(\mathcal{S}) = \min_w\{\mathcal{L}_w(\mathcal{S})\}$. Note that $\mathcal{L}(\mathcal{S})$ is achieved only if the quorums are chosen according to the optimal strategy. Due to various strategies used to access the quorum system, the actual load might be higher than $\mathcal{L}(S)$. On the other hand, $\mathcal{L}(S)$ is property to the combinatorial structure of the quorum system, and not to the protocol using the system. If the load is low, then the frequency of accessing each element is low, indicating that the system has capacity to perform paralleled tasks. Naor and Wool prove that $\mathcal{L}(S) \geq \max\{\frac{1}{c(\mathcal{S})}, \frac{c(\mathcal{S})}{n}\}$, which implies that $\mathcal{L}(\mathcal{S}) \geq \frac{1}{\sqrt{n}}$.

- *Availability*: The failure probability $F_p(\mathcal{S})$ of a quorum system is the probability that there exists at least one quorum containing no faulty servers, assuming that each element fails with probability p. This failure probability measures how resilient the system is, and we would like $F_p(\mathcal{S})$ to be as small as possible.

- *Probe Complexity*: The probe complexity is the number of probed elements before a witness of the state of the system is found, i.e. a *live* quorum or a quorum with all its elements failed. The complexity of the algorithm for finding a quorum should be low. On the other hand, the load of the quorum system induced by the algorithm must be taken into consideration. For the non adaptive algorithm, i.e., an algorithm which decides the set of elements to probe *before* it gains any knowledge as to which elements failed and which did not. To improve the probe complexity, the algorithms are allowed to be adaptive, i.e., the next element to be probed can selected according to the outcome of the previous probes. Such algorithms require collecting information of the system, which is determined by the specific network and quorum system. The corresponding communication overhead must be taken into account during the design of such algorithms.

- *Scalability*: The good scalability of a dynamic quorum system is two fold. At first, the system should maintain the good qualities in the static environment, i.e. load, availability and probe complexity. Secondly, the join and leave operation should be applied with low time and message complexity.

6.2 System Model

We apply the same system model proposed in Section 3.1. In addition, we use following models for dynamic systems.

6.2.1 Dynamic Network Model

We propose a dynamic network model, where nodes join and leave the network arbitrarily. We assume a decentralized network, where the failure of a node does not interfere other nodes' existence in the network. Each node has a set of *neighbors* and can directly communicate with them by simple heartbeat messages. A node can also communicate with other distant nodes when knows their addresses by applying some routing algorithms.

6.2.2 Probabilistic Failure Model

A quorum system is constructed upon a dynamic network. We use a probabilistic model of failures in the system. We assume that each element fails independently with a fixed probability less than or equal to p. Hence, this probability describes the dynamic nature of the system. We assume the failures are *crash* failures, i.e. "lying" failures such as Byzantine failures are not considered. We also assume all failures are *detectable*, i.e. an element's state (live or dead) can be known after probing.

In the probabilistic model, the probe complexity of a quorum system is the *expected* number of probes by given the fixed probability p. For some quorum systems, the probe complexity in the probabilistic model is significantly better than their probe complexity in the worst case model. Naor and Wieder [56] prove the tradeoff between the load of a quorum system and its probe complexity for non adaptive algorithms in the probabilistic failure model with $p < \frac{1}{2}$.

Lemma 1. *Let $\mathcal{S}$ be a quorum system over universe U with a load of $\mathcal{L}(\mathcal{S})$. Assume that each element in U fails with some fixed probability $p < \frac{1}{2}$. Let $X \subseteq U$ be a predefined set of elements such that*

$$Pr[X \text{ contains a live quorum}] \geq \frac{1}{2},$$

then

$$|X| \geq \frac{1}{2\log(1/p)+1} \cdot \frac{\log[1/4\mathcal{L}(\mathcal{S})]}{\mathcal{L}(\mathcal{S})}$$

This implies that for a non adaptive algorithm which achieves the optimal load $O(\frac{1}{\sqrt{n}})$, at least $\Omega(\sqrt{n}\log n)$ elements must be probed. This fact motivates us to design algorithms that achieve good balance between somewhat contradictory measures of quality.

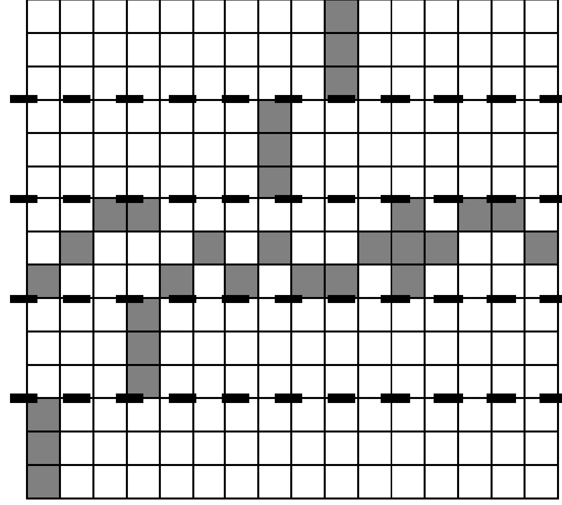

Figure 6.1: A B-Grid System: $n = 240$, $d = 16$, $h = 5$, $r = 3$. One quorum is shaded.

6.3 HB-Grid: High Available B-Grid Quorum Systems

We use a class of quorum systems: the *B-Grid* systems [55]. We start with the a precise definition of the B-Grid system:

Definition 9. B-Grid System*: Arrange the elements in rectangular grid of width d. Split the grid logically into h bands of r rows each. The total number of elements is $n = dhr$. Call r elements in a column that are al contained in a single band a* mini-column. *Then a quorum consists of one mini-column in every band, and a representative element in each mini-column of one band.*

The following lemmas are proved in [55]:

Lemma 2. $\mathcal{L}(\textit{B-Grid}) = \frac{d+hr-1}{dhr}$.

Lemma 3. $F_p(\textit{B-Grid}) \leq (dp^r)^h + h(1-q^r)^d$*, where* $q = 1 - p$.

Following these lemmas, we show some relationships of d, h, r and p for an efficient quorum construction.

Theorem 14. *If* $h \geq 4$ *and* $d \geq 2$*, then* $F_p = 1$ *when* $p \geq 1/2$.

Proof. We prove the above lemma by contradiction. Assume $F_p \leq 1$. From Lemma 3 we immediately have $dp^r \leq 1$ and $p^r \leq \frac{1}{d}$. Since $p \geq 1/2$, we know that $q^r \leq p^r \leq \frac{1}{d}$. Applying this to Lemma 3, we have

$$h(1-q^r)^d \geq h(1-\frac{1}{d})^d \geq 4 \cdot (1-\frac{1}{2})^2 = 1,$$

which forms a contradiction. □

From above theorem we find that the availability of B-Grid is not acceptable when $p \geq 1/2$. This fact motivates us to find an efficient quorum construction when $0 < p < 1/2$. In the next lemma we give a quorum construction under which F_p decays exponentially fast when $0 < p < 1/2$.

Theorem 15. *If $0 < p < 1/2$, let $c = \frac{\ln q}{\ln p} \in (0, 1)$. Then for any $k \in (1, \frac{1}{c})$, there exists a B-Grid quorum construction with $r = \lfloor k \log_{1/p} d \rfloor$, under which $F_p(\text{B-Grid}) \leq e^{-[(k-1)\ln d]\cdot h} + e^{-\frac{1}{2}d^{(1-ck)}}$ for large values of d such that $\ln h < \frac{1}{2}d^{(1-ck)}$.*

Proof. With $r = \lfloor k \log_{1/p} d \rfloor$, we have

$$F_p \leq (dp^r)^h + h(1 - q^r)^d \leq d^{(1-k)\cdot h} + h(1 - d^{-ck})^d.$$

Note that $(1 - d^{-ck})^{d^{ck}} \simeq e^{-1}$ for large values of d, then

$$F_p \leq e^{-[(k-1)\ln d]\cdot h} + h \cdot e^{d^{(1-ck)}} \leq e^{-[(k-1)\ln d]\cdot h} + e^{-\frac{1}{2}d^{(1-ck)}} \tag{6.1}$$

□

The theorem above gives us a set of choices to construct a B-Grid quorum system. Note that in Equation 6.1 F_p consists of 2 parts. By different values of d and k, one term may dominate another term and makes up the major part of F_p. The following theorem gives the quorum construction under which F_p is minimized:

Theorem 16. *For all B-Grid quorum systems constructed in Theorem 15, a HB-Grid quorum system with the minimal failure probability is constructed by letting $d = n^{\frac{1}{2-c}}$, $r = \lfloor \log_{1/p} d \rfloor + 1$ and $h = \frac{n}{rd}$. The quorum system exhibits an asymptotical optimal failure probability: for any positive constant $\epsilon < \frac{1-c}{c}$, we can find the asymptotical upper bound for F_p:*

$$F_p = O(\max[\exp(-\frac{\ln 1/p}{1 + \log_{1/p} d} \cdot n^{\frac{1-c(1+\epsilon)}{2-c(1+\epsilon)}}), \exp(-\frac{1}{2} \cdot n^{\frac{1-c(1+\epsilon)}{2-c(1+\epsilon)}})])$$

for large values of d such that $d \geq (p^{-1})^{\epsilon^{-1}}$.

Proof. We first prove that for a set of quorum systems constructed in Theorem 15 with $k = k_0$, the quorum system with $d = n^{\frac{1}{2-ck_0}}$ has the minimal failure probability.

Assume $d = n^i$, from Equation 6.1 we have

$$F_p \leq e^{-\frac{(\ln 1/p)(k_0-1)}{1+\log} n^{1-i}} + e^{-\frac{1}{2}n^{i(1-ck_0)}}$$

Hence, $F_p = O(\max[\exp(-\frac{(\ln 1/p)(k_0-1)}{k_0}n^{1-i}), \exp(-\frac{1}{2}n^{i(1-ck_0)})])$. When the two terms are the same order of n, the order of F_p is minimized. In this case, we have

$$F_p = O(\max[\exp(-\frac{(\ln 1/p)(k_0-1)}{k_0} \cdot n^{\frac{1-ck_0}{2-ck_0}}), \exp(-\frac{1}{2} \cdot n^{\frac{1-ck_0}{2-ck_0}})]) \qquad (6.2)$$

Now we prove the second part of the theorem. Since $1 < k < \frac{1}{c}$, we have $F_p = \Omega(\exp(n^{\frac{1-c}{2-c}}))$. We want to select k_0 such that F_p is asymptotical to its lower bound when d increases. Hence, k_0 decreases and gets close to 1 when d increases.

Let $r = \lfloor k \log_{1/p} d \rfloor = \lfloor \log_{1/p} d \rfloor + 1$, then we have $k \leq 1 + \frac{1}{\log_{1/p} d}$. For any small positive constant $\epsilon < \frac{1-c}{c}$, we can find a sufficient large d such that $\frac{1}{\log_{1/p} d} \leq \epsilon$ and $d \geq (p^{-1})^{\epsilon^{-1}}$. Given this relationship, we have $n^{\frac{1-ck}{2-ck}} \geq n^{\frac{1-c(1+\epsilon)}{2-c(1+\epsilon)}}$. The theorem follows.

□

For HB-Grid quorum systems, we immediately have the following corollary:

Corollary 1. *$\mathcal{L}$(B-Grid) $= O(n^{-\frac{1-c}{2-c}})$.*

The following propositions give examples for HB-Grid quorum systems construction under different ranges of p. They can be proved by plugging the parameters into Theorem 16 and Corollary 1.

Proposition 17. *If $d = n^{0.78}$, $r = \lfloor \log_{9/4} d \rfloor + 1$ and $h = \frac{n}{rd}$, then $\mathcal{L}$(B-Grid) $= O(n^{-0.22})$ and $F_p = O(\max[\exp(-\frac{\ln 9/4}{1+\log_{9/4} d} \cdot n^{-0.22}), \exp(-\frac{1}{2} \cdot n^{-0.22})])$ in the range $0 \leq p \leq 4/9$.*

Proposition 18. *If $d = n^{0.61}$, $r = \lfloor \log_3 d \rfloor + 1$ and $h = \frac{n}{rd}$, then $\mathcal{L}$(B-Grid) $= O(n^{-0.39})$ and $F_p = O(\max[\exp(-\frac{\ln 3}{1+\log_3 d} \cdot n^{-0.39}), \exp(-\frac{1}{2} \cdot n^{-0.39})])$ in the range $0 \leq p \leq 1/3$.*

Proposition 19. *If $d = n^{0.52}$, $r = \lfloor \log_8 d \rfloor + 1$ and $h = \frac{n}{rd}$, then $\mathcal{L}$(B-Grid) $= O(n^{-0.48})$ and $F_p = O(\max[\exp(-\frac{\ln 8}{1+\log_8 d} \cdot n^{-0.48}), \exp(-\frac{1}{2} \cdot n^{-0.48})])$ in the range $0 \leq p \leq 1/8$.*

Remarks: Theorem 16 shows a HB-Grid quorum construction with the minimal failure probability for all quorum systems constructed in Theorem 15 when $0 < p < 1/2$. From above propositions we find that the performance degrades elegantly when p increases. The performance is asymptotical optimal ($\mathcal{L}$(B-Grid) $= O(\sqrt{n}), F_p = O(\sqrt{n})$)when p gets close to 0.

6.4 DHB-Grid: Dynamic HB-Grid Quorum Systems

In this section we suggest a DHB-Grid quorum system that operates in a dynamic network environment, where nodes may join and leave. We focus on implementing quorum system

in a dynamic environment and designing efficient algorithms that allow nodes join and leave with low overhead and achieve high integrity. The probe complexity must be taken into consideration such that the probe algorithms designed for static systems can also enjoy low complexity in its counterpart in dynamic systems.

6.4.1 The Quorum Implementation: 3-Phase Linking

We consider a HB-Grid quorum system constructed in a dynamic environment. We can use a 3-field coordinate to represent a node's position: (α, β, γ), where α, β and γ represent the band, row in the band and mini-column in the band respectively. Specifically, $\alpha \in [1, .., h]$ and $\beta \in [1, ..., r]$ from the top down; $\gamma \in [1, ..., d]$ from left to right. A dynamic quorum system DHB-Grid is implemented in the following way:

1. *Mini-column Linking*: Nodes at (α, β', γ) have links to nodes at $(\alpha, \beta' - 1, \gamma)$. $(2 \leq \beta' \leq r)$.
2. *Hierarchical Row Linking*: Nodes at $(\alpha, \beta'', \gamma'')$ have links to nodes at $(\alpha, \beta'', 2\gamma'')$ and $(\alpha, \beta'', 2\gamma'' + 1)$. $(\beta'' \in \{1, r\}$, $1 \leq \gamma'' \leq 2^{\lfloor \log_2 d \rfloor} - 1)$. Hierarchies formed by nodes in the top row and the bottom row are called the *top row hierarchy* and the *bottom row hierarchy* of band α, respectively.
3. *Hierarchical Band Linking*: Nodes at $(\alpha', 1, 1)$ have links to nodes at $(2\alpha', 1, 1)$ and $(2\alpha' + 1, 1, 1)$. $(1 \leq \alpha' \leq 2^{\lfloor \log_2 h \rfloor} - 1)$. The hierarchy is called the *band hierarchy.*

The implementation is 3-phase.

- In the mini-column phase, a vertical linking chain is formed from the lowest to the highest element in each mini-column.
- In the row phase, for each band α, two hierarchical linking structures are implemented in the top and bottom row with *root* nodes at $(\alpha, \beta'', 1)$ and there are $\lceil \log d \rceil$ levels in each hierarchy. For any node x at level l, we call the node that links to it at level $l - 1$ as x's *top row predecessor* or *bottom row predecessor* at level $l - 1$, denoted by $trp^{l-1}(x)$ or $brp^{l-1}(x)$. We recursively define x's predecessor at each level $1 \leq l' \leq l$ in the same way and use $trp(x)$ or $brp(x)$ to denote x's *top row predecessor set* or *bottom row predecessor set.*
- A similar hierarchical structure is implemented in the band phase. Root nodes of each top row hierarchy form a band hierarchy with the root at $(1, 1, 1)$ and there are $\lceil \log h \rceil$ levels. For any node y, we define its *band predecessor* at level l' and *band predecessor set* in the same way, denoted by $bp^{l'}(y)$ and $bp(y)$ respectively.

6.4.2 Basic Join/Leave Operation

Now we assume a node n_0 at $(\alpha_0, \beta_0, \gamma_0)$ that wishes to leave the system. The following operation is performed: in its mini-column, nodes above it are moved downwards by one unit, which guarantees that the *"empty"* position is appeared at the top part of each mini-column. Meanwhile, links of each phase are adjusted to make this empty position can be probed with low complexity.

We first describe the operation in the mini-column phase:

- If there's no mini-column link from n_0, i.e., n_0 is the top element in the γth mini-column of the αth band, the mini-column link from $(\alpha_0, \beta_0 + 1, \gamma_0)$ to n_0 is removed.
- Otherwise,
  1. The mini-column link from n_0 to $(\alpha_0, \beta_0 - 1, \gamma_0)$ is removed.
  2. The mini-column link from $(\alpha_0, \beta_0 + 1, \gamma_0)$ to n_0 (for $\beta_0 \leq r - 1$) is *redirected* by a link from $(\alpha_0, \beta_0 + 1, \gamma_0)$ to $(\alpha_0, \beta_0 - 1, \gamma_0)$.
  3. Nodes at $(\alpha_0, \beta_0', \gamma_0)$ where $1 \leq \beta_0' \leq \beta_0 - 1$ are moved downwards by one unit, with new coordinate $(\alpha_0, \beta_0' + 1, \gamma_0)$.

In the row and band phases, the following operation is performed:

1. *Top Row Linking Redirection*: If n_0 is the top element in the γth mini-column of the αth band, the row links to and from n_0 are redirected to links to and from $(\alpha_0, \beta_0 + 1, \gamma_0)$. Otherwise, no redirection is needed.
2. *Hierarchical Linking Adjustment (Up Phase)*: If the coordinate of any top node n_1 of band α_1 changes from $(\alpha_1, \beta_1, \gamma_1)$ to $(\alpha_1, \beta_1 + 1, \gamma_1)$, and n_1 is in the level l_1 in the top row hierarchy, the following top row linking hierarchy adjustment is performed:
   - If $l_1 = 1$, STOP. n_1 is the root.
   - Otherwise,
     (a) Compare coordinates between n_1 and $trp^{l_1-1}(n_1)$ with coordinate $(\alpha_1^{l_1-1}, \beta_1^{l_1-1}, \gamma_1^{l_1-1})$. If $\beta_1 + 1 \leq \beta_1^{l_1-1}$, STOP. Otherwise, *swap* n_1 and $trp^{l_1-1}(n_1)$ in the row hierarchy by exchanging their top row links and band links (if any). Note nodes are not moved in the system, but their positions in the top row hierarchy are swapped.
     (b) Repeat the first step until it stops or n_1 reaches the top of the hierarchy, i.e. n_1 becomes the root.
   - If the coordinate of any node n_2 changes from $(\alpha_2, \beta_2, \gamma_2)$ to $(\alpha_2, \beta_2 + 1, \gamma_2)$ in the band hierarchy, the same adjustment as the row hierarchy is performed.

The leave operation has two key properties:

1. Nodes are restricted to only move vertically in a band. This property is designed to reserve the integrity of quorums. Since a quorum is composed of a mini-column of each band and a element of each mini-column of one band, moving elements vertically in a band doesn't change a quorum.
2. Top row links are redirected to maintain linkings between top elements of mini-columns in a band. These elements may reside in different rows in the system. The hierarchical linking adjustment guarantees that elements in the higher levels in the hierarchy are always located in the lower rows in the quorum system. This property is designed to help a new node that wishes to join the system find an empty position to "insert" efficiently.

Now there is a node n_3 that wishes to join the system. The join operation is following: if there is any empty position in the system, n_3 is inserted to this position. Otherwise, n_3 is "appended" to the lowest row of a band.

The operation is started by randomly selecting a node n_4 in the existing quorum system with with coordinate $(\alpha_4, \beta_4, \gamma_4)$.

Insert Operation: Follow vertical links starting from n_4 to the top node of its mini-column. Denote this top node as n_5 with coordinate $(\alpha_5, \beta_5, \gamma_5)$.

- If $\beta_5 \geq 2$, then some elements have left the γ_5th mini-column. In this case, n_3 is inserted to $(\alpha_5, \beta_5 - 1, \gamma_5)$. A vertical link is created from n_3 to n_2.
- Otherwise, follow top row links starting from n_5 to probe its top row predecessors $trp(n_5)$.
  - If a node $n_6 \in trp(n_5)$ is probed with coordinate $(\alpha_6, \beta_6, \gamma_6)$ such that $\beta_6 \geq 2$, stop probing. Some elements have left the γ_6th mini-column of this band. Insert n_3 to $(\alpha_6, \beta_6 - 1, \gamma_6)$.
  - Otherwise, no such node is probed until the probing reaches the root of this top row hierarchy, which implies that no element has left this band. Denote the root n_7 with coordinate $(\alpha_7, \beta_7, \gamma_7)$. Follow band links starting from n_7 to probe its band predecessors $bp(n_7)$.
    * If a node $n_8 \in bp(n_7)$ is probed with coordinate $(\alpha_8, \beta_8, \gamma_8)$ such that $\beta_8 \geq 2$, stop probing. Some elements have left the γ_8th mini-column of this band. Insert n_3 to $(\alpha_8, \beta_8 - 1, \gamma_8)$.
    * Otherwise, no such node is probed until the probing reaches the root of the band hierarchy, which implies that no element has left the quorum system.

Denote the root n_9 and follow the vertical links from n_9 to the bottom element of its mini-column. Denote the bottom element n_{10} with coordinate $(\alpha_{10}, \beta_{10}, \gamma_{10})$. Follow the bottom row links starting from n_{10} to find the rightmost element of row β_{10} in band α_{10}. Denote the rightmost element n_{11} with coordinate $(\alpha_{11}, \beta_{11}, \gamma_{11})$. Insert n_3 to $(\alpha_{11}, \beta_{11}, \gamma_{11} + 1)$.

Similarly, the linking of each phase has to be adjusted when a new node joins the system. Denote the new coordinate of n_3 $(\alpha_3, \beta_3, \gamma_3)$.

1. *Vertical Linking Redirection*: when n_3 is inserted to the quorum system, it must be the top element of the mini-column γ_3. A new vertical link is created from node at $(\alpha_3, \beta_3 + 1, \gamma_3)$ (if any) to n_3.
2. *Row Linking Redirection*: as n_3 is the top element of the mini-column γ_3, the original top row links to and from node at $(\alpha_3, \beta_3 + 1, \gamma_3)$ are redirected to links to and from n_3.
3. *Hierarchical Linking Adjustment*: n_3 joins the top row hierarchy by replacing the node at $(\alpha_3, \beta_3 + 1, \gamma_3)$. Assume n_3 is in the level l_3 in the top row hierarchy, the following adjustment is performed:
   - If l_3 is the lowest level of the top row hierarchy, STOP. No adjustment is needed.
   - Otherwise,
     (a) Compare the β-coordinates between n_3 and its children at level $l_3 + 1$. If β-coordinates of both children are less or equal to β_3, STOP. Otherwise, there must exists a child with β-coordinate greater than β_3. Swap n_3 with this child by exchanging their top row links and band links (if any). If both children are available for swapping, select the one with the greater β-coordinate. If they have the same β-coordinates, randomly select one and swap.
     (b) Repeat the first step until it stops or n_3 reaches the bottom of the hierarchy, i.e., n_3 becomes a leaf node.
   - If the β-coordinate of any node in the band hierarchy decreases, the same adjustment as the top row hierarchy is performed.

6.4.3 Join/Leave Complexity

Before analyzing the time and message complexity of the join/leave operations we show the property of the hierarchical linking adjustment.

Theorem 20. *The time and message complexity for the hierarchical linking adjustment of DHB-Grid is* $O(\log n)$

For the leave operation, the worst case is that n_0 is a bottom element of a band. In this case, there are at most $r - 1$ elements that move downwards. Hence the time and message complexity of this part is $O(r) = O(\log n)$. Note that the total time and message overhead is the sum of this part and the hierarchical linking adjustment. For the join operation, the worst case is that n_3 random selected a bottom element of a full band where no elements has left, and the root node of the top row hierarchy of this band is a leaf node of the band hierarchy. In this case, the total number of nodes that n_3 visits is $r + \log\lceil d\rceil + \log\lceil h\rceil = O(\log n)$. The total time and message overhead is the sum of this part and the hierarchical linking adjustment. Hence the time and message complexity for the join/leave operations are both $O(\log n)$.

6.5 DHBC: The DHB-Grid-Based Cache Coherence Protocol

6.5.1 Protocol Description

We propose a distributed cache-coherence protocol based on DHB-Grid quorum systems in the dynamic environment. We describe the cache-coherence protocol imposed on the DHB-Grid quorum system. Nodes use a probe algorithm $Probe(\mathcal{S})$ to probe a live quorum in the system. The protocol perform three operations: publish, lookup and move.

- *Publish*: When a new object is created in the local cache of node i, its transactional memory proxy invokes cache-coherence protocol to publish the metadata of the object in the distributed system. The format of the metadata should include 3 fields: $< ObjectID, Address, WriteCount >$. The $WriteCount$ field records the number of times that the object is accessed by a write request. Initially this field is set to 0. The node publishes the object by sending its metadata to all elements accessed by $Probe(\mathcal{S})$.

- *Lookup*: A lookup operation is performed when a transaction invokes a read request for a object. The node j looks up the object by sending the request to all elements accessed by $Probe(\mathcal{S})$. The metadata with the *highest* $WriteCount$ field is selected and the request is forwarded to its address. The requested node sends a read-only copy to j when the object becomes available.

- *Move*: A move operation is performed when a transaction invokes a write request for a object. The node k sends the request to all elements accessed by $Probe(\mathcal{S})$. The metadata with the *highest* $WriteCount$ field is selected and the request is forwarded to its address. The node which stores the metadata adds the $WriteCount$ field by 1 and sends the metadata to all elements accessed by $Probe(\mathcal{S})$. The requested node directly sends the object to k when it becomes available.

Remarks: From the description of the protocol we find that the performance of the cache-coherence protocol is determined by the performance of its underlying quorum system and probe algorithm (access strategy). Thus, we evaluate the performance by measuring the complexity of $Probe(\mathcal{S})$.

6.5.2 Probe Complexity

We are interested in the probe complexity of DHB-Grid systems. Our probe algorithm is motivated by the *coloring* method in [25]. Each element is colored with either red (indicating that the element has failed) or green (indicating a live element). A set of elements is red (or green) if all its elements are.

Algorithm 1: R_Probe (Phase I): *A Band with a live element in each mini-column*

```
1 Random select a band i;
2 Mode ← green;
3 for column k ← 1 to d do
4     Probe U_k^i;
5     if an element u_j is found s.t. c(u_j) == Mode then
6         W_1 = W_1 ∪ {u_j}
7     else
8         Mode ← c(U_k^i), W_1 ← ∅;
9         break;
```

The input to our algorithms is some coloring of the elements. Probing an element i reveals its color, denoted $c(i)$. The probe algorithm consists of two independent phases: probing a band with a live element in each mini-column of the band and probing a live mini-column in every band.

Algorithm 1 examines the columns of a randomly selected band i one by one. In stage k, the algorithm probes the r elements of column k. While doing so, the algorithm either maintains a set W_1 consisting of one live element from column 1 to k, or ends the algorithm by finding a red column and erases the current set W_1. If the algorithm does not succeed to find a live set W_1 for all columns, it has to be called again to probe another band until it succeeds or all bands have been probed.

Algorithm 2 examines each band to find a live mini-column. For each band i, a column k is random selected to probe all its elements. If a live column is probes, all its elements are added into set W_2 and the algorithm continues to probe the next band. The algorithm ends when a live mini-column is probed for all bands, or a band with no live mini-column is probed.

We proceed with an analysis of the algorithm R_Probe. Let X denote the number of probed elements, and let $X = X(1) + X(2)$, where $X(1)$, $X(2)$ represent the numbers of probed

Algorithm 2: R_Probe (Phase II): *A live column in each band*

```
1  for band i ← 1 to h do
2      while Mode != green do
3          Random select a column k;
4          Mode ← green;
5          Probe U_k^i;
6          if c(U_k^i) == Mode then
7              W_2 = W_2 ∪ {u_j};
8          else
9              Mode ← red;
10             if all columns of band i have been probed then
11                 W_2 ← ∅;
12                 break;
13     if Mode == red then
14         break;
```

elements in Phase I and Phase II of the algorithm, respectively. Observe that in a HB-Grid system, $d = n^{\frac{1}{2-c}}$, $r = \lfloor \log_{1/p} d \rfloor + 1$ and $h = \frac{n}{rd}$. The expected numbers of probed elements of both phases, $E[X(1)]$ and $E[X(2)]$, can be bounded as follows:

$$E[X(1)] = \frac{1}{1-dp^r} \cdot d \cdot \frac{1}{q} = \frac{d}{q(1-dp^{\lfloor \log_{1/p} d \rfloor + 1})} \simeq \frac{d}{q(1-dp^{(\log_{1/p} d)+1})} = \frac{d}{q^2} = O(n^{\frac{1}{2-c}})$$

$$E[X(2)] = \frac{1}{q^r} \cdot r \cdot h = \frac{1}{q^{\lfloor \log_{1/p} d \rfloor + 1}} \cdot \frac{n}{d} \simeq \frac{n}{q \cdot d^{1-c}} = O(n^{\frac{1}{2-c}})$$

Combined above results, we have $E(X) = E[X(1)] + E[(X2)] = O(n^{\frac{1}{2-c}})$.

Note that in a DHB-Grid, the quorum size is $d + hr - 1 = O(n^{\frac{1}{2-c}})$. Hence, the probe complexity of R_Probe is the best we can expect. We have the following theorem:

Theorem 21. *For any DHB-Grid where elements fails with a fixed probability $0 \leq p < 1/2$, the probabilistic probe complexity of algorithm R_Probe is $\Theta(n^{\frac{1}{2-c}})$.*

6.6 Conclusion

We propose a novel DHB-Grid system, which exhibits asymptotical optimal load and availability for the probabilistic failure model. Based on it, we propose DHBC, a DHB-Grid-based cache-coherence protocol. We show that the performance of DHBC depends on the probe

algorithm $Probe(\mathcal{S})$. For DHB-Grid system, we propose a 2-phase probe algorithm which achieves optimal probe complexity for the probabilistic failure model.

In this chapter we focus on metrics of load, availability and probe complexity, which are unique properties for dynamic environments. We show that the such performance is highly depend on the performance of underlying quorum systems. DHBC is only the first attempt to design cache-coherence protocols upon DHB-Grid systems. In other words, DHB-Grid systems provides us a high performance structure upon which we can approach the design of contention managers and cache-coherence protocols. We can propose cache-coherence protocols which provide similar properties as LAC protocols or DHTC protocols in the semantics of DHB-Grid systems.

Chapter 7

Conclusions, Contributions, and Proposed Post Preliminary-Exam Work

In this dissertation proposal, we study the design of contention managers and cache-coherence protocols for distributed transactional memory systems. We focus on the design goal of minimizing the competitive ratio of the contention manager and cache-coherence protocol combination.

Our first approach on solving this design 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 against the makespan of the optimal off-line clairvoyant scheduler for distributed transactional memory systems. We show that, for each single 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, without considering the design of cache-coherence protocols, the upper bound of the competitive ratio is $O(N^2 \cdot s)$, where N is the maximum number of transactions requesting the same 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 ratio 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 the 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. In LAC protocols, the duration of a transac-

tion requesting node to locate an object 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 Greedy manager/LAC protocol combination, and show that LAC is an efficient choice for the Greedy manager to improve performance.

Our results imply that there are two main factors which determine the performance of cache-coherence protocols for distributed transactional memory: the total locating cost and the extra traveling makespan of overtaking failures (proposed in Chapter 3). We therefore design a DHT-based cache-coherence protocol, called the DHTC protocol, to improve the performance when combined with the Greedy manager. The locating cost is measured by the stretch of the cache-coherence protocol. Since locating moving objects in the network induces large cost, we reduce the total locating cost by ensuring that each transaction request locates the object just for the first time when the request is invoked. Once the object is located, the transaction request will be saved in the network until it commits. By doing this, the total locating cost increases only as a polynomial of N, where N is the maximum number of nodes that request the same object. On the other hand, the total number of overtakings does not exceed N because an overtaking only occurs when a new transaction request locates the object. We show that the DHTC protocol guarantees an $O(N \cdot s)$ competitive ratio, which is a significant improvement over an arbitrary cache-coherence protocol/Greedy manager combination.

Dynamic environments (e.g., node failures, joins, departures) introduce additional challenges on the design and performance of distributed transactional memory systems. For example, the system must be decentralized and load balanced. A decentralized system guarantees that every node in the network has the same importance, and hence if a node crashes, it does not affect the system performance dramatically. By distributing balanced load over all nodes in the network, it is unlikely that a particular nodes is a "hot spot" and become a bottleneck. The system should also exhibit good scalability and availability properties, and provide reasonable communication overhead in the presence of network changes. To model the failure of nodes in the network, we use a probabilistic model of failures in the system, where we assume that each node fails independently and the failure probability of each node does not exceed p. We apply quorum systems in distributed cache-coherence protocol design. We construct a novel quorum system called the dynamic high available B-Grid quorum system or DHB-Grid. We show that for the DHB-Grid system, the performance is asymptotically optimal when p approaches 0, and degrades gracefully when p increases. We present efficient adjustment algorithms for DHB-Grid to accommodate network changes, and show an $O(\log n)$ message complexity for each adjustment, where n is the number of nodes. Based on the DHB-Grid system, we propose DHBC, a quorum-based cache-coherence protocol which exhibits good scalability, availability, load balancing, and low communication complexity properties.

7.1 Contributions

Our research contributions are summarized as follows:

1. We identify that the performance of distributed transactional memory systems is determined by the contention manager and the cache-coherence protocol used;
2. We establish the upper and lower bounds of the competitive ratio of the Greedy manager with an arbitrary cache-coherence protocol;
3. We show that the worst-case performance of the Greedy manager with an efficient LAC protocol is improved and predictable;
4. We show that the DHTC protocol significantly improves the performance of the Greedy manager over an arbitrary cache-coherence protocol/Greedy manager combination;
5. We show how a quorum-based cache-coherence protocol can efficiently adapt to system failures and network changes, with high availability and low message complexity properties.

7.2 Post Preliminary-Exam Work

We propose the following post preliminary-exam work:

- **Contention Manager Design for Distributed Transactional Memory.** As mentioned before, an obstruction-free algorithm only guarantees progress in the absence of contention. Hence, a contention manager is essential to mediate contentions on the same object. For the design of distributed transactional memory systems, the selection of the contention manager is obviously important since it determines the range of the performance that a cache-coherence protocol (when combined with that contention manager) can achieve. Our current results depend on the Greedy contention manager, the selection of which is motivated by its provable properties in the semantics of multiprocessors. However, this raises the question of whether the selection of other contention managers can improve performance. Generally, there are two approaches for such a selection:

  1. Select an existing contention manager which is proposed for multiprocessors, and prove its worst-case and best-case performance for distributed systems. Design a compatible cache-coherence protocol to improve the worst-case performance when combined with that contention manager. There have been a number of different contention management policies proposed for multiprocessors. The following is a brief list of some of the most common ones in the literature:

  - *Aggressive.* This contention manager always chooses to abort a conflicting transaction at conict time. This is the most basic contention manager.
  - *Polite.* This contention manager uses exponential back-off techniques, similar to that used in network protocols, to determine which transaction to abort.
  - *Randomized.* This contention manager chooses which transaction to abort by making a random choice.
  - *Karma.* The Karma contention manager aborts the transaction, which has performed the least amount of work when a conflict occurs.
  - *Eruption.* This contention manager increases the priority of a transaction that others are waiting on in order to make it complete faster (priority is used in arbitrating which transaction is aborted). The idea of this contention manager is similar to the popular priority inheritance technique used in lock-based real-time concurrency control.
  - *Kindergarten.* This policy encourages transactions to take turns accessing shared memory (like children accessing toys or other shared objects).
  - *Timestamp.* This policy attempts to be as fair as possible, and uses timestamps to arbitrate among the transactions trying to access a shared object.
  - *Queue-on-Block.* This policy causes conflicting transactions to wait in a queue for the transaction currently accessing the shared object, and spin on a "finished" flag that is set by the object-accessing transaction at its commit time. If a transaction has waited too long, which is determined in an application-specific manner, it aborts the object-accessing transaction and accesses the shared object. Thus, this policy has similar behavior as that of spinlocks.

  The properties of these contention managers are worth further studying. We do not know whether these contention managers can provide similar or improved, provable properties as that of the Greedy manager. For example, it is not known whether it is possible to establish a better performance range for the randomized contention manager with high probability? Note that for the Greedy manager, the worst-case competitive ratio is at least $\Omega(s)$. Our current results only guarantee an $O(N \cdot s)$ worst-case competitive ratio, where s is the number of objects and N is the maximum number of transactions requiring accesses to the same object. Hence, it would be a significant improvement if better worst-case performance of other contention managers can be established.

2. Construct a contention manager which is explicitly designed for distributed transactional memory, with improved worst-case performance. Another direction is to design a contention manager targeting the behavior of distributed transactional memory. Since the design of contention managers directly affect the performance of the system, it is appealing to focus on the design of contention managers which are dedicated to distributed systems. Such design should take into account factors which are unique to distributed systems, such as the communication cost between nodes, higher network latencies, network topologies, and the dynamic

nature of the underlying network. We have shown that for existing contention managers, the performance degrade dramatically because of the lack of consideration of such factors. The burden of alleviating such performance degradation is currently exclusively that of the cache-coherence protocols. Hence, the design of such contention managers give us greater flexibility in optimizing system performance.

- **Contention Manager and Cache-Coherence Protocol Design in the Probabilistic Model** We propose a probabilistic model to model network failures. For example, each node may fail independently and the failure probability may not exceed p. (Of course, other models are possibe.) Under such a model, our proposed DHB-Grid cache-coherence protocol focuses on achieving good load balancing, decentralization, scalability, and availability properties. This design approach is orthogonal to that of the LAC and DHTC cache coherence protocol design approach, which focus on the optimization of the competitive ratio for a set of transactions. For the probabilistic model, it is appealing to investigate its expected competitive ratio depending on a failure probability p. Such an approach typically involves using randomized methods in the design of cache-coherence protocols. It is possible that some combinations of contention managers and cache-coherence protocols which cannot guarantee a desirable worst-case performance may behave well in the probabilistic model. Research in this direction will allow us to establish the worst-case and average-case behaviors of different combinations of contention managers and cache-coherence protocols, understand the concomitant trade-offs, and design more optimized contention managers and cache-coherence protocols for distributed transactional memory.

Bibliography

[1] Abraham, I., Malkhi, D.: Probabilistic quorums for dynamic systems. Distrib. Comput. 18(2): 113–124 (2005)

[2] Y. Amir, A. Wool: Optimal Availability quorum systems: theory and practice. Inform. Process. Lett., 65 (5): 223–228 (1998)

[3] Hagit Attiya, Leah Epstein, Hadas Shachnai, Tami Tamir: Transactional contention management as a non-clairvoyant scheduling problem. In PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing, 308–315 (2006)

[4] R. L. Boccino, V. S. Adve, B. L. Chamberlain: Software Transactional Memory for Large Scale Clusters. In PPoPP'08, 247–258 (2008)

[5] Chandra Chekuri, Martin Pál: An O(log n) Approximation Ratio for the Asymmetric Traveling Salesman Path Problem. Theory of Computing, 3 (1): 197–209 (2007)

[6] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems, 336-346 (2006)

[7] Dice, D., Shavit, N.: Understanding Tradeoffs in Software Transactional Memory. In Proceedings of the international Symposium on Code Generation and Optimization, 21–33 (2007)

[8] Dolev, S., Hendler, D., Suissa, A.: CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In PODC'08: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, 125-134 (2008)

[9] Edmonds, J., Chinn, D. D., Brecht, T., Deng, X.: Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. J. of Scheduling 6 (3): 231–250 (2003)

[10] M. R. Garey, R. L. Graham: Bounds for Multiprocessor Scheduling with Resource Constraints: SIAM Journal on Computing, 4(2): 287–200 (1975)

[11] Friedman, R., Kliot, G., Avin, C.: Probabilistic quorum systems in wireless ad hoc networks In Proceeding of the IEEE International Conference on Dependable Systems and Networks (DSN), 277–286 (2008)

[12] A.W. Fu: Delay-optimal Quorum Consensus for Distributed Systems. IEEE Transactions on Parallel and Distributed Systems, 8 (1): 59–69 (1997)

[13] Seth Gilbert, Grzegorz Malewicz: The Quorum Deployment Problem. In Proceedings of 8th International Conference on Principles of Distributed Systems (OPODIS), 316–330 (2004)

[14] Rachid Guerraoui, Maurice Herlihy, Bastian Pochon: Toward a theory of transactional contention managers. In PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, 258–264 (2005)

[15] R. Guerraoui, M. Herlihy, M. Kapalka, B. Pochon: Robust Contention Management in software transactional memory. In Proceedings of the OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)

[16] R. Guerraoui, M. Herlihy, B. Pochon: Polymorphic Contention Management. In Proceedings of the 19th International Symposium on Distributed Computing (DISC'05), 303–323 (2005)

[17] Guerraoui, R., Kapalka, M.: 2008. On the correctness of transactional memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08), 175–184 (2008)

[18] Guerraoui, R., Henzinger, T. A., Singh, V.: 2008. Permissiveness in Transactional Memories. In Proceedings of the 22nd international Symposium on Distributed Computing (DISC '08), 305–319 (2008)

[19] Anupam Gupta, Bruce M. Maggs, Florian Oprea, Michael K. Reiter: Quorum Placement in Networks to Minimize Access Delays. In PODC '05: Proceedings of 24th Annual ACM Symposium on Principles of Distributed Computing, 87–96 (2005)

[20] Gray, J.: The transaction concept: virtues and limitations. In Readings in Database Systems, Morgan Kaufmann Publishers, 140-150 (1988)

[21] Z.J. Haas, B. Liang: Ad hoc mobility management with uniform quorum systems. IEEE/ACM Transactions on Networking, 7 (2): 409–418 (1999)

[22] Lance Hammond, Vicky Wong, Mike Chen, Ben Hertzberg, Brian D. Carlstrom, John D. Davis, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, Kunle Olukotun: Transactional Memory Coherence and Consistency. In Proceedings of the 31st Annual International Symposium on Computer Architecture, 102 (2004)

[23] Tim Harris, Keir Fraser: Language Support for Lightweighted Transactions. In Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 388–402 (2003)

[24] Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), 48–60 (2005)

[25] Hassin, Y., Peleg, D.: Average probe complexity in quorum systems. J. Comput. Syst. Sci. 72 (4) 592–616 (2006)

[26] Maurice Herlihy: Dynamic Quorum Adjustment for Partitioned Data. ACM Transactions on Database Systems, 12 (2): 170–194 (1987)

[27] Maurice Herlihy, J. Eliot B. Moss, J. Eliot, B. Moss: Transactional Memory: Architectural Support for Lock-Free Data Structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture, 289–300 (1993)

[28] Maurice Herlihy, Victor Luchangco, Mark Moir: Obstruction-free Synchronization: Double-ended Queues as an Example. In Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS), 522–529 (2003)

[29] Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, III: Software transactional memory for dynamic-sized data structures. In PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing, 92–101 (2003)

[30] Maurice Herlihy, Ye Sun: Distributed Transactional Memory for Metric-space Networks. Distributed Computing, 20(3): 195–208 (2007)

[31] J. A. Hoogeveen, Analysis of Christofides' heuristic: some paths are more difficult than cycles: Operations Research Letters, 10: 291 (1991)

[32] JoAnne Holliday, Rober Steinke, Divyakant Agrawal, Amr El Abbadi: Epidemic Quorums for Managing Replicated Databases. IEEE Transactions on Knowledge and Data Engineering, 15 (5): 1218–1238 (2003)

[33] Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In PODC '94: Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing 151–160 (1994)

[34] Goutham Karumanchi, Srinivasan Muralidharan, Ravi Prakash: Information Dissemination in Partitionable Mobile Ad Hoc Networks. In Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS), 4 (1999)

[35] J. Kleinberg, M. Sandler, A. Slivkins: Network failure detection and graph connectivity. In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 176–185 (2004)

[36] T. F. Knight: An architecture for most functional languages. In Proceedings of ACM Lisp and Functional Programming Conference, 500–519 (1986)

[37] Kumar, S., Chu, M., Hughes, C. J., Kundu, P., Nguyen, A. Hybrid transactional memory. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06), 209–220 (2006)

[38] Fumei Lam, Alantha Newman: Traveling salesman path problems. Math. Program., 113 (1): 39–59 (2008)

[39] James R. Larus, Ravi Rajwar: Transactional Memory. Morgan & Claypool (2006)

[40] Laudon, J. and Lenoski, D.: The SGI Origin: a ccNUMA highly scalable server. SIGARCH Comput. Archit. News, 25(2): 241–251(1997)

[41] Jun Luo, Jean-Pierre Hubaux, Patrick Th. Eugster: PAN: providing reliable storage in mobile ad hoc networks with probabilistic quorum systems. In Proceedings of MobiHoc, 1–12 (2003)

[42] N. Lynch, A. Shvartsman: Rambo: A reconfigurable atomic memory service for dynamic networks. In Proceedings of the 16th International Symposium on Distributed Computing, 173–190 (2002)

[43] Malkhi, D., Reiter, M., Wright, R.: Probabilistic quorum systems. In PODC'97: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, 267-273 (1997)

[44] Dahlia Makhi, Michael Reiter: Byzantine Quorum System. Distributed Computing, 11 (4): 203–213 (1998)

[45] K. Manassiev, M. Mihailescu, C. Amza: Exploiting Distributed Version Concurrency in a Transactional Memory Cluster. In PPoPP'06, 198–208 (2006)

[46] Marathe, V. J., Scherer, W. N., Scott, M. L.: Design tradeoffs in modern software transactional memory systems. In Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-Time Support For Scalable Systems (LCR '04), 81 1–7 (2004)

[47] V. J. Marathe, M. L. Scott: A Qualitative Survey of Modern Software Transactional Memory Systems. Technical Report TR 839, Department of Computer Science, University of Rochester, (2004).

[48] Virendra J. Marathe, William N. Scherer, III, Michael L. Scott: Adaptive Software Transactional Memory. In Proceedings of the 19th International Symposium on Distributed Computing, 354–368 (2005)

[49] Martin, M., Blundell, C., Lewis, E.: Subtleties of Transactional Memory Atomicity Semantics. IEEE Comput. Archit. Lett. 5 (2) 17 (2006)

[50] Martnez, J. F., Torrellas, J.: Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In Proceedings of the 10th international Conference on Architectural Support For Programming Languages and Operating Systems (ASPLOS-X), 18–29 (2002)

[51] Minh, C. C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In ISCA '07: Proceedings of the 34th Annual international Symposium on Computer Architecture, 69–80 (2007)

[52] Moir, M.: Practical implementations of non-blocking synchronization primitives. In PODC '97: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing 219–228 (1997)

[53] Motwani, R., Phillips, S., Torng, E.: Non-clairvoyant scheduling. In Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, 422–431 (1993)

[54] Mukherjee, S. S., Bannon, P., Lang, S., Spink, A., Webb, D.: The Alpha 21364 Network Architecture. In Proceedings of the the Ninth Symposium on High Performance interconnects, 113 (2001)

[55] Moni Naor, Avishai Wool: The Load, Capacity, and Availability of Quorum Systems. SIAM Journal on Computing, 27 (2): 423–447 (1998)

[56] Moni Naor, Udi Wieder: Scalable and Dynamic Quorum Systems. Distributed Computing, 17 (4): 311–322 (2005)

[57] Oplinger, J., Lam, M. S.: Enhancing software reliability with speculative threads. In Proceedings of the 10th international Conference on Architectural Support For Programming Languages and Operating Systems (ASPLOS-X), 184–196 (2002)

[58] F. Oprea, M. Reiter: Minimizing Response Time for Quorum-System Protocols over Wide-Area Networks. In Proceeding of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 409–418 (2007)

[59] Peleg, D., Wool, A.: Crumbling walls: a class of practical and efficient quorum systems. In PODC '95: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, 120–129 (1995)

[60] Ravi Rajwar, James R Goodman: Transactional Lock-Free Execution of Lock-Based Programs. In Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), 5–17 (2002)

[61] Rowstron, A. I., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In Proceedings of the IFIP/ACM international Conference on Distributed Systems Platforms Heidelberg, 329-350 (2001)

[62] Daniel J. Rosenkrantz, Richard E. Stearns, Philip M. Lewis, II: An Analysis of Several Heuristics for the Traveling Salesman Problem. SIAM Journal on Computing, 6 (3): 563–581 (1977)

[63] Saha, B., Adl-Tabatabai, A., Jacobson, Q.: Architectural Support for Software Transactional Memory. In Proceedings of the 39th Annual IEEE/ACM international Symposium on Microarchitecture, 185–196 (2006)

[64] N. Shavit, D. Touitou: Software Transactional Memory. In PODC '95: Proceedings of the 22nd Annual Symposium on Principles of Ditributed Computing, 204-213 (1995)

[65] Shpeisman, T., Menon, V., Adl-Tabatabai, A., Balensiefer, S., Grossman, D., Hudson, R. L., Moore, K. F., Saha, B.: Enforcing isolation and ordering in STM. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation 78–88 (2007)

[66] Shriraman, A., Spear, M. F., Hossain, H., Marathe, V. J., Dwarkadas, S., Scott, M. L.: An integrated hardware-software approach to flexible transactional memory. SIGARCH Comput. Archit. News 35 (2) 104–115 (2007)

[67] William N. Scherer, III, Michael L. Scott: Advanced contention management for dynamic software transactional memory. In PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, 240–248 (2005)

[68] William N. Scherer, III and Michael L. Scott: Randomization in STM Contention Management (POSTER). In Proceedings of the 24th ACM Symposium on Principles of Distributed Computing, (2005)

[69] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. 11(1): 17-32 (2003)

[70] Stone, J. M., Stone, H. S., Heidelberger, P., Turek, J.: Multiple Reservations and the Oklahoma Update. IEEE Parallel and Distributed Technology, 1(4) 58–71 (1993)

[71] T. Tsuchiya, M. Yamaguchi, T. Kikuno: Minimizing the maximum delay for reaching consensus in quorum-based mutual exclusion schemes. IEEE Transactions on Parallel and Distributed Systems, 10 (4): 337–345 (1999)

[72] Zhao, B. Y., Kubiatowicz, J. D., Joseph, A. D.: Tapestry: an Infrastructure for Fault-Tolerant Wide-Area Location and Routing. University of California at Berkeley Technical Report, CSD-01-1141 (2001)

[73] Bo Zhang, Binoy Ravindran: SOQ: A Service-Oriented Quorum-Based Protocol for Resilient Real-Time Communication in Partitionable Networks. In Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), 192–199 (2008)