Dynamic Analysis of the Relay Cache-Coherence Protocol for Distributed Transactional Memory

Bo Zhang
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
alexzbzb@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
binoy@vt.edu

Abstract

Transactional memory is an alternative programming model for managing contention in accessing shared in-memory data objects. Distributed transactional memory (TM) promises to alleviate difficulties with lock-based (distributed) synchronization and object performance bottlenecks in distributed systems. In distributed TM systems, both the management and consistency of a distributed transactional object are ensured by a cache-coherence protocol. The Relay protocol is a cache-coherence protocol that operates on a fixed spanning tree. The protocol efficiently reduces the total number of abortions for a given set of transactions. We analyze the Relay protocol for a set of transactions which are dynamically generated in a given time period, and compare the protocol's time complexity against that of an optimal offline clairvoyant algorithm. We show that Relay is $O(\log D)$-competitive, where D is the diameter of the spanning tree, for a set of transactions that request the same object, given the condition that the maximum local execution time of transactions is sufficiently small.

1. Introduction

Lock-based synchronization is inherently error-prone. For example, coarse-grained locking, in which a large data structure is protected using a single lock is simple and easy to use, but permits little concurrency. In contrast, with fine-grained locking, in which each component of a data structure (e.g., a bucket of a hash table) is protected by a lock, programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Both these situations are highly prone to programmer errors. In addition, lock-based code is non-composable. For example, atomically moving an element from one hash table to another using those tables' lock-based atomic methods (e.g., `insert`, `delete`) is not possible in a straightforward way: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the methods (of the two tables); if the methods were to export their locks, that will compromise safety. For these and other reasons, lock-based concurrent code is difficult to reason about, program, and maintain [9].

Transactional memory (TM) is an alternative synchronization model (for shared in-memory data objects) that promises to alleviate these difficulties. A transaction is an explicitly delimited

sequence of steps that is executed atomically by a single thread. Transactions read and write shared objects. A transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). If a transaction aborts, it is typically retried until it commits. Transactional memory API for multiprocessors have been proposed in hardware [7], in software [8], [15], and in hardware/software combination [3]. Two transactions *conflict* if they access the same object and one access is a write. The transactional approach to *contention management* [10] guarantees atomicity by ensuring that whenever a conflict occurs, only one of the transactions involved can proceed.

The difficulties of lock-based synchronization also appear in distributed (control-flow) programming models such as RPCs. For example, RPC calls, while holding locks, can become remotely blocked on other calls for locks, causing distributed deadlocks. Livelocks and lock convoying similarly occur. In addition, in the RPC model, an object can become a "hot spot," and thus a performance bottleneck. These difficulties have similarly motivated research on the design of distributed TM systems as a possible solution. For example, in the data-flow distributed TM model of [11] (that we also consider), object performance bottlenecks can be reduced by exploiting locality: move the object to nodes. Moreover, if an object is shared by a group of geographically-close clients that are far from the object's home, moving the object to the clients can reduce communication costs. Distributed (data-flow) TM can therefore alleviate these difficulties, in which distributed transactional conflicts are resolved and object consistencies are ensured through distributed contention managers and cache-coherence protocols, respectively.

Past works on transactional memory in distributed systems include [2], [11], [13], and [18]. In [13], the authors present a page-level distributed concurrency control algorithm, which maintains several distributed versions of the same data item. In [2], the authors decompose a set of existing cache-coherent TM designs into a set of design choices, and select a combination of such choices to support TM for commodity clusters. None of these works present theoretical analysis of the fundamental properties of TM for distributed systems, such as performance upper bounds of cache-coherence protocols, which is our focus.

In [11], Herlihy and Sun present a Ballistic distributed cache-coherence protocol in a metric-space network, where the communication costs between nodes form a metric. The protocol's performance is evaluated by measuring its *stretch*, which is the ratio of the protocol's communication cost for obtaining a cached copy of an object to that of the optimal communication cost. The Ballistic protocol mainly suffers from two drawbacks. First, it employs an existing distributed queuing protocol, which does not consider the contention between two transactions, and the worst-case queue length, which is $O(N_i^2)$ for N_i transactions requesting the same object. Second, its hierarchical structure degrades its scalability—e.g., whenever a node joins or departs, the whole structure has to be rebuilt.

The Relay cache-coherence protocol is proposed in [18], which focuses on optimizing the worst-case queue length of the distributed queue, i.e., it reduces the total number of transaction abortions. Motivated by the Arrow distributed queuing protocol [4] due to the similarities of the distributed queuing problem and the problem of the synchronization of write/read access requests to mobile objects for distributed TM, the Relay protocol is proposed. Operating on a network spanning tree. The Relay protocol efficiently reduces the worst-case number of total abortions to $O(N_i)$, for N_i transactions requesting the same object.

In this paper, we conduct the first *dynamic analysis* of the Relay protocol, which allows nodes

to initiate transactions at arbitrary times. In other words, we give the first "online" analysis of the Relay protocol, given a set of dynamically generated transactions in a time period. Such an analysis is outside the scope of [18] and is therefore not discussed in [18] (or for any cache-coherence protocol for that matter). We adopt the method of [12], which gives a dynamic analysis of the Arrow protocol for a set of ordering requests. As our analysis shows, for distributed TM, transactions involve two more variables than simple ordering requests: the local execution time and the worst-case number of abortions, which makes the dynamic analysis complex.

We compare the time complexity of the Relay protocol against an optimal clairvoyant offline algorithm. We show that, for a set of transactions T^i requesting access to the same object, the competitive ratio of the Relay protocol is $O(\log D)$, if the maximum local execution time of the transactions in T^i is $O(\log D)$, where D is the diameter of the spanning tree. This is the first ever dynamic analysis of a distributed TM cache-coherence protocol. Since the Ballistic protocol which adopts the same mechanism as that of the Arrow protocol is not capable of reducing the worst-case number of abortions [18], the competitive ratio of the Relay protocol is a significant improvement over past distributed TM cache-coherence protocols and distributed queuing protocols. These two results constitute the papers contributions.

The rest of the paper is organized as follows. We present our system model and describe the distributed TM cache-coherence problem in Section 2. In Section 3, for the paper's completeness, we summarize the Relay protocol. The section also describes the protocol's operation for a set of dynamically generated transactions (this is outside the scope of [18]). Section 4 presents the dynamic analysis and compares the Relay protocol against an optimal clairvoyant offline algorithm. The paper concludes in Section 5.

2. System Model and Problem Description

2.1. Network Model.

We consider a distributed system with n nodes. Let $G = (V, E, d)$ be a weighted connected graph, where $|V| = n$ and d is a function that maps E to the set of positive real numbers. Specifically, we use $d(u, v)$ to denote the communication cost of the edge $e(u, v) \in E$.

We assume a *fixed*-rooted spanning tree $\mathcal{T}$ of G. Given the spanning tree $\mathcal{T}$, we define the distance in $\mathcal{T}$ between a pair of two nodes, u and v, to be the sum of the lengths of the edges on the unique path in $\mathcal{T}$ between u and v, denoted by $d_{\mathcal{T}}(u, v)$. We define the *diameter* D of $\mathcal{T}$ as: $D = \max_{u,v \in V} d_{\mathcal{T}}(u, v)$; and the *normalized diameter* D_0 of $\mathcal{T}$ as: $D_0 = \max_{u,v,x,y \in V} \{\frac{d_{\mathcal{T}}(u,v)}{d_{\mathcal{T}}(x,y)}\}$.

2.2. Distributed Queuing Problem

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

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

Definition 1 (Ordering Algorithm): An ordering algorithm is a distributed algorithm which defines a total order on $\mathcal{R}$ such that in the end each node that initiates requests knows the predecessors of all its requests.

Note that such an algorithm must be distributed. For example, the Arrow protocol [14] is a simple distributed queuing protocol based on path reversal.

2.3. Distributed Transactional Memory Model

We consider the data-flow model of Herlihy and Sun in [11], which converts the distributed TM cache-coherence problem to a special distributed queuing problem where the length of the distributed queue is dynamically changed based on contentions. Compared with the distributed queuing problem, for distributed TM, we deal with transactions instead of ordering requests. A transaction is a sequence of requests, each of which is a read or write operation request to an individual object. Given a set of $s \geq 1$ objects, $\{R_1, \ldots, R_s\}$, we can use the tuple $T_j = (v_j, t_j, \vec{R(j)}, \tau_j)$ to identify a transaction T_j. We explain each field of T_j as follows:

- v_i: the node that initiates the transaction.
- t_i: the time when the transaction is initiated.
- $\vec{R(j)}$: the vector that describes the sequence of requests of T_j. Let $\vec{R(j)} = \{R_1(j), \ldots, R_s(j)\}$, where $R_i(j) \in \{0, 1, \frac{1}{n}\}$ represents the units of R_i required by T_j. If T_j does not require access to R_i, then $R_i(j) = 0$. If T_j updates R_i, i.e., a write operation, then $R_i(j) = 1$. If it reads R_i without updating, then $R_i(j) = \frac{1}{n}$, i.e., at most the object can be read by n nodes simultaneously. Suppose there are two transactions T_j and T_k, and $R_i(j) + R_i(k) > 1$. Then T_j and T_k *conflict* at R_i.
- τ_j: the duration of T_j's *successful local execution*. An execution of a transaction is a sequence of *timed actions*. Generally, there are four action types that may be taken by a single transaction: *write*, *read*, *commit*, and *abort*. An execution ends by either a commit (success) or an abort (failure). A successful local execution of T_j is a successful execution when all objects requested by T_j already reside in v_j, i.e., there is no need to fetch those objects from the network.

Distributed queuing protocols that consider ordering requests cannot be directly used for distributed TM since they usually do not provide efficient mechanisms to mediate conflicts among multiple transactions over a set of objects: when a node holding the object receives a new request, it simply sends the object to the requesting node. Since a transaction is atomic, i.e., a transaction can only commit as long as all its read/write operations have executed, such a simple mechanism could cause deadlocks and livelocks. For example, suppose there are two transactions T_j and T_k, and both of them request write accesses for two objects R_1 and R_2. Suppose, initially R_1 is held by T_j and R_2 is held by T_k. Hence, v_j sends a request for R_2 and v_k sends a request for R_1. If we use a simple distributed queuing protocol to order these requests, it is very likely that both transactions are aborted and R_1 and R_2 are moved to T_k and T_j, respectively. As a result, both transactions cannot proceed in this case.

To understand the elements of the design to support the transactional memory API in a distributed system, we consider Herlihy and Sun's data-flow model [11]. In this model, trans-

actions are immobile (running at a single node), but objects move from node to node, just like mobile objects in the distributed queuing problem. Transactional 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. A contention manager assigns priorities to transactions. A running transaction could only be aborted by another transaction with a higher priority. We use $T_j \prec T_k$ to represent that transaction T_j is issued a higher priority than T_k. Revisiting the previous example, suppose $T_j \prec T_k$. In this case, T_j first commits after R_2 is moved to v_j, and then R_1 is moved to v_k to let T_k commit.

Thus, the design of a distributed TM system is composed of two parts: a contention manager to mediate conflicts and a protocol (equivalent to the distributed queuing protocol) to locate and move objects in the network. Such a protocol is called a *distributed cache-coherence* protocol. 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.

Different contention managers have been studied in the past [16]. An efficient contention management policy should guarantee progress—i.e., at any given time, there exists at least one transaction that proceeds to commit without interruption. In this paper, we assume a fixed contention manager, which satisfies the *work conserving* [1] and *pending commit* [6] properties:

Definition 2: A contention manager is *work conserving* if it always lets a maximal set of non-conflicting transactions to run.

Definition 3: A contention manager obeys the *pending commit* property if, at any given time, some running transaction will execute uninterrupted until it commits.

For example, the Greedy contention manager in [6] which uses a globally consistent priority policy that issues priorities to transactions is shown in [1] to satisfy both properties.

A simple design for the cache-coherence protocol is to directly use an existing distributed queuing protocol, as suggested in [11]. In [11], Herlihy and Sun present the *Ballistic* protocol, which is based on the Arrow distributed queuing protocol built on a hierarchical clustering network structure which provides a better stretch than the simple spanning tree structure. However, current distributed queuing protocols do not consider the contention between two transactions. Thus, an aborted transaction has to restart and join the distributed queue again. As a result, the length of the distributed queue increases and the worst-case queue length is $O(N_i^2)$ for N_i transactions requesting the same object. In Section 3, we summarize the Relay protocol for completeness. Relay provides an optimal $O(N_i)$ worst-case queue length.

3. The Relay Protocol

The Relay protocol is motivated by the Arrow protocol which is based on path reversal on a network spanning tree. It is a distributed cache-coherence protocol designed for the synchronized management of transactional accesses to mobile objects (i.e., the data flow TM model) in a network. When multiple nodes in the network transactionally request an object concurrently, the transactional requests must be queued in some order, and the object must travel from one node to another down the queue. To manage such a distributed queue, an efficient distributed cache-coherence protocol must solve three problems: a) how to order the requests from different

nodes into a single queue; b) how to provide the necessary information to nodes such that each node knows the location of its successor in the queue and the object can be forwarded down the queue; and c) how to efficiently reduce the length of the queue. Note that the protocol is "distributed" in the sense that no single node needs to have the global knowledge of the queue. Each node only needs to know its successor in the queue and will forward the object to it.

The Relay protocol is initialized in the same way as the Arrow protocol. The protocol runs on a fixed spanning tree $\mathcal{T}$ of G. Each node v keeps an "arrow" or a pointer $p(v)$ to itself or to one of its neighbors in T. If $p(v) = v$, then v is the tail of the queue, i.e., the next request should be forwarded to v. In this case, the node v is defined as a *"sink"*. Clearly, at any time, there exists only one sink for each object. If $p(v) = u$, then $p(v)$ only knows the "direction" of the tail of the queue and the request is forwarded following that direction. At the start, the node v_{tail}, where the object resides, is selected to be the tail of the queue. Each node $v \in V$ maintains a pointer $p(v)$ and is initialized so that following the pointers from any node leads to the tail, as shown in Figure 1(a).

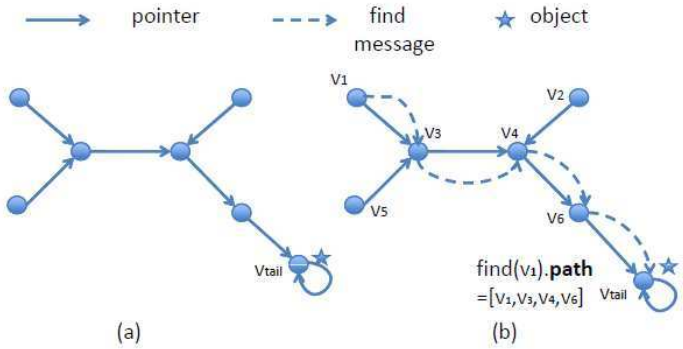

Figure 1. (a) Initialization. (b) Node v_1 sends a find message.

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

The find message $find(v_1)$ keeps a *path vector* $\vec{path}$ to record the path it travels. Each node receiving the find message from v_1 appends its ID to $find(v_1).\vec{path}$. When the find message arrives at the tail of the queue, the vector $find(v_1).\vec{path}$ records the path from v_1 to the tail v_{tail}. Such an operation is shown in Figure 1(b).

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

- Case 1: If $T_1 \prec T_{tail}$, then T_{tail} is aborted and the object is moved to v_1. The pointers are updated when the object is moved. To let the pointers update correctly, node v_{tail} sends a *move message* $move(v_{tail})$ with a *route vector* $\vec{route}$ which records the route that $move(v_{tail})$ will travel. In this case, $move(v_{tail}).\vec{route} = find(v_1).\vec{path}$. Hence, node v_{tail} sends the object with $move(v_{tail})$ to $move(v_{tail}).\vec{route}[max]$ (the last element of $move(v_{tail}).\vec{route}$). Meanwhile, node v_{tail} sets $p(v_{tail})$ to $move(v_{tail}).\vec{route}[max]$. Then T_{tail} restarts and immediately sends a $find(v_{tail})$ message to $p(v_{tail})$. Suppose a node u receives a move message from one of its neighbors. It updates $move(v_{tail}).\vec{route}$ by removing $move(v_{tail}).\vec{route}[max]$ and sends the

object to the new $move(x).\vec{route}[max]$, setting $p(u) = move(v_{tail}).\vec{route}[max]$. Finally, when the object arrives at v_1, $p(v_1)$ is set to v_1 and all pointers are updated. Such operations guarantee that at any time, there exists only one sink in the network, and, from any node, following the direction of its pointer leads to the sink. Such an operation is shown in Figure 2(a).

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

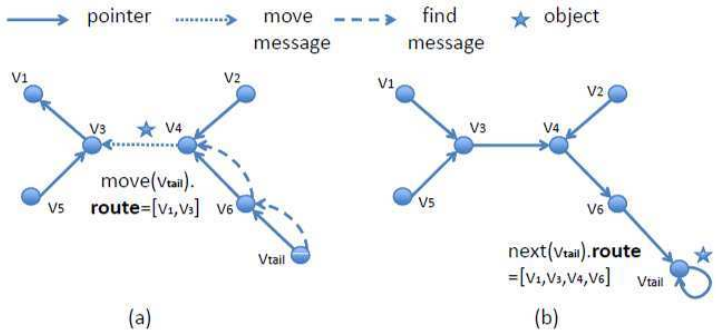

Figure 2. (a) Case 1: $T_1 \prec T_{tail}$. (b) Case 2: $T_{tail} \prec T_1$.

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

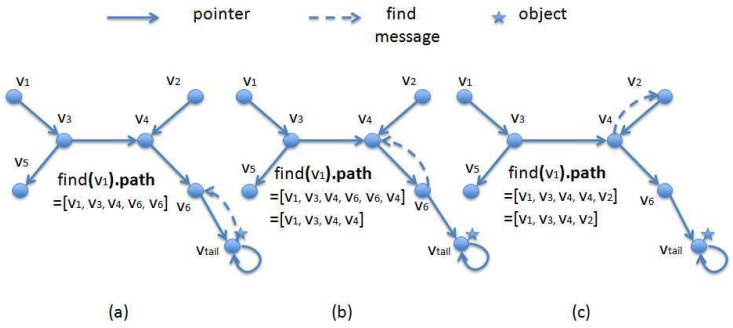

Figure 3. Example: $T_{tail} \prec T_2 \prec T_1$. Node v_{tail} receives $find(v_2)$ after $find(v_1)$ and forwards $find(v_1)$ to v_2. The path vector $find(v_1).\vec{path}$ is updated.

A problem appears when v_{tail} forwards find messages from other nodes to a new node, e.g., $find(v_1)$ to v_2. In this case, the path vector should record the path from v_1 to v_2. However, since $find(v_1)$ is forwarded along the path $v_1 \rightarrow v_{tail} \rightarrow v_2$, the path recorded in $find(v_1).\vec{path}$ is not the shortest path from v_1 to v_2 in the spanning tree $\mathcal{T}$. Hence, the path vector has to be correctly updated to record the shortest path. We illustrate this update policy with the help of an example, as shown in Figure 3.

Since there is only one path in a spanning tree between two nodes such that each node in the path is visited exactly once, the path vector is updated to detect and eliminate nodes that have been visited multiple times. In the example of Figure 3, node v_{tail} has to forward $find(v_1)$ to v_2. Initially, $find(v_1).\vec{path} = [v_1, v_3, v_4, v_6]$. When node v_6 receives $find(v_1)$, it first checks the last two

elements of $find(v_1).\vec{path}$, which are v_4 and v_6. Since they are different, v_6 simply appends its ID to the path vector, as shown in Figure 3(a). Now, $find(v_1)$ arrives at v_4 and the last two elements of $find(v_1).\vec{path}$ are the same (v_6). Node v_4 has to check the third last element of the path vector (v_4) to see whether a loop forms. Hence, a loop forms by $[v_4, v_6, v_6, v_4]$ and v_6 is deleted from the path vector since it is not on the shortest path from v_1 to v_2, as shown in Figure 3(b). When $find(v_1)$ arrives at v_2, it finds that the last two elements of the path vector are the same, but the third last element is not v_2. Hence v_4 should exist on the path vector, as shown in Figure 3 (c).

The correctness of the protocol can be proved from the protocol description. The pointers are only "flipped" when the object is moved, which guarantees that at any time there is only one sink in the network and following the pointer from any node leads to the sink. The key to proving the correctness of the protocol is that find and move messages are forwarded along the correct path on T. As explained in the protocol description, we use path vectors and route vectors to record paths. As long as they are correctly updated, a find or a move message can be forwarded along the unique path on T to its destination.

The most important advantage of the Relay protocol is that it reduces the total number of abortions. The following theorem is proved in [18]:

Theorem 1: The total number of abortions of N_i transactions requesting the same object under Relay is at most $N_i - 1$.

In other words, the Relay protocol upper bounds the length of the distributed queue to $2N_i - 1$ for N_i transactions requesting the same object.

We now focus on the dynamic analysis of the Relay protocol in the following section.

4. Analysis

4.1. Cost Measures

Cost of Relay. We first focus on the cost of an individual object R_i. Let $T^i = \{T_j \in T : R_i(j) > 0\}$, i.e., T^i is the set of transactions that require accesses to R_i. For brevity, in the rest of the paper, we refer to the node and time of a transaction T_j directly as v_j and t_k, respectively.

Generally, a cache-coherence protocol performs two functions: 1) locating the up-to-date copy of the object and 2) moving it in the network to meet transactions' requests. We define their costs as follows:

Definition 4 (Locating Cost): In a given graph G, the locating cost $\delta^C(T_j, T_k)$ is the communication cost for a transaction request invoked by node T_j to travel in the network, to successfully locate an object held by node T_k, under a cache-coherence protocol C.

Definition 5 (Moving Cost): In a given graph G, the moving cost $\zeta^C(T_j, T_k)$ is the communication cost for an object held by node T_j to travel in the network to node T_k, which invokes a transaction request of the object, under a cache-coherence protocol C.

As shown in the description of the Relay protocol, each transaction locates the object via the direct path in the spanning tree in the same way as the Arrow protocol. On the other hand, the object is moved along the direct path on the spanning tree because the path vector is correctly updated. The locating cost and moving cost of Relay are: $\delta^C(T_j, T_k) = d_{\mathcal{T}}(v_j, v_k)$ and $\zeta^C(T_j, T_k) = d_{\mathcal{T}}(v_j, v_k)$.

Each transaction may suffer from a number of abortions before it commits. Let $\lambda_i^*(j)$ denote the number of abortions of transaction T_j under Relay for a conflict on object R_i and $\lambda_i(j) = \lambda_i^*(j) + 1$, i.e., $\lambda_i(j)$ is the total number of times that T_j receives the object R_i. We have the following theorem:

Theorem 2: Assume $v_{i,j}^{\nearrow}(m)$ (or $v_{i,j}^{\searrow}(m)$) is T_j's m^{th} destination (or source) for locating (or moving) the object R_i. The total cost of transaction T_j with respect to object R_i under Relay is:

$$\text{cost}_R^i(T_j) \leq \sum_{m=1}^{\lambda_i(j)} [d_{\mathcal{T}}(v_j, v_{i,j}^{\nearrow}(m)) + \text{dist}_{\mathcal{T}}(v_{i,j}^{\nearrow}(m), v_{i,j}^{\searrow}(m)) + d_{\mathcal{T}}(v_j, v_{i,j}^{\searrow}(m)) + \tau_j], \quad (1)$$

where $\text{dist}_{\mathcal{T}}(v_{i,j}^{\nearrow}(m), v_{i,j}^{\searrow}(m))$ is the total communication cost for the the m^{th} find message from v_j to travel along a certain path from $v_{i,j}^{\nearrow}(m)$ to $v_{i,j}^{\searrow}(m)$ in the spanning tree $\mathcal{T}$, including the idle time that the find message waits for other transactions' commit.

Proof: The complete execution of T_j with respect to R_i is shown in Figure 4. Each time T_j sends a find message, it waits until the object has arrived. The m^{th} find message first arrives at $v_{i,j}^{\nearrow}(m)$ and such locating cost is $d_{\mathcal{T}}(v_j, v_{i,j}^{\nearrow}(m))$. Since the find message may be forwarded to other nodes, we have to take into account such costs. The path from $v_{i,j}^{\nearrow}(m)$ to $v_{i,j}^{\searrow}(m)$ is not necessarily the shortest path on the spanning tree since some nodes may be visited multiple times. The idle time is the total time that the find message waits on $v_{i,j}^{\searrow}(m)$ for its transaction's commit. Finally, the object stays at T_j for at most τ_j time before T_j aborts or commits. The theorem follows. □

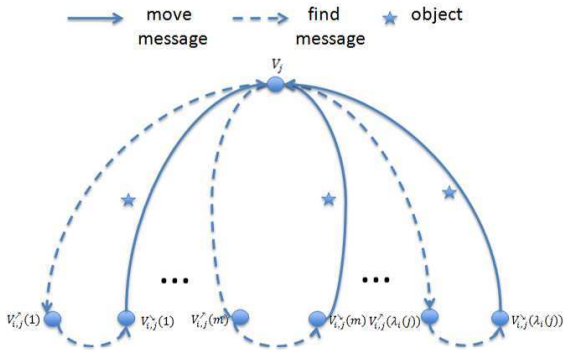

Figure 4. The complete execution of T_j with respect to R_i: T_j is aborted by the transaction on $v_{i,j}^{\nearrow}(m), m \geq 2$.

Note that Equation 1 gives the total communication cost of a single transaction T_j. From another point of view, an object started to move in the network and be get involved by transactions once it receives the first transaction request. The total time complexity is composed the time that the object travels and the time that the object is accessed by transactions. Hence, a more useful cost measure is the *amortized cost* of a single transaction, i.e., the contribution made by a single transaction to the total cost of a set of transactions. We have the following theorem.

Theorem 3: Let the amortized cost of a transaction T_j with respect to R_i under Relay be denoted as $c_R^i(T_j)$. Then,

$$c_R^i(T_j) \leq \sum_{m=1}^{\lambda_i(j)} [d_{\mathcal{T}}(v_j, v_{i,j}^{\searrow}(m)) + \tau_j]. \quad (2)$$

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

Proof: The total cost of a set of transactions with respect to R_i is the sum of the R_i's traveling distance in the network and the local execution cost of transactions which require accesses to R_i. From Figure 4, we can see that for the m^{th} find message, such traveling cost is $d_{\mathcal{T}}(v_j, v_{i,j}^{\searrow}(m))$ and the local execution cost is at most τ_j. We now prove that all locating costs and $\text{dist}_{\mathcal{T}}(v_{i,j}^{\nearrow}(m), v_{i,j}^{\searrow}(m))$ are covered by other transactions' amortized cost. When $m \geq 2$, the find message is sent immediately after the object is moved from T_j. Hence, such locating cost is covered by the moving cost from v_j to $v_{i,j}^{\nearrow}(m)$ and the execution cost for the transaction on $v_{i,j}^{\nearrow}(m)$. For $m \geq 1$, when $v_{i,j}^{\nearrow}(m)$ forwards the find message to $v_{i,j}^{\searrow}(m)$, the cost of this distance is covered by the local execution cost and the moving cost for the set of transactions on $\{v_{i,j}^{\nearrow}(m), next(v_{i,j}^{\nearrow}(m)), next(next(v_{i,j}^{\nearrow}(m))), \ldots, v_{i,j}^{\searrow}(m)\}$. Such cost also covers the idle time (if any) that the m^{th} find message waits on $v_{i,j}^{\searrow}(m)$, since the object is moved to v_j immediately when it is available on $v_{i,j}^{\searrow}(m)$. The theorem follows.

□

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

We index all sub-transactions $S^i = \{S_0 = (v_0, t_0, \vec{R(0)}, \tau_0), S_1 = (v_1, t_1, \vec{R(1)}, \tau_1), \ldots, \}$,where $S_j \in T^i$, in increasing order with respect to t_j, with ties broken arbitrarily, i.e., $j < k \Rightarrow t_j < t_k$. For the Relay protocol, let ϕ_R be the order of obtaining the object by sub-transactions S^i which is induced by Relay, i.e., $\phi_R(j)$ denotes the index of the j^{th} sub-transaction that receives the object in Relay's order. We use $S_0 = (\text{root}, 0)$ to represent the "virtual" transaction (token) at the initial location of the object R_i. Hence we have $S_{\phi_R(0)} = S_0$.

We define the cost metric to order a sub-transaction S_k after S_j as follows: $c_R(S_j, S_k) := d_{\mathcal{T}}(v_j, v_k)$. We have the following theorem:

Theorem 4:

$$\sum_{j=1}^{|T^i|} \sum_{m=1}^{\lambda_i(j)} d_{\mathcal{T}}(v_j, v_{i,j}^{\searrow}(m)) = \sum_{k=1}^{|S^i|} c_R(S_{\phi_R(k-1)}, S_{\phi_R(k)}). \tag{3}$$

In other words, the total moving cost of the set of transactions T^i is equivalent to the cost of ordering a set of sub-transactions S^i which are decomposed from T^i.

Proof: The cost $d_{\mathcal{T}}(v_j, v_{i,j}^{\searrow}(m))$ is the moving cost from $v_{i,j}^{\searrow}(m)$ to v_j. Since the object is moved along this path, we know that $v_{i,j}^{\searrow}(m)$ receives the object just before v_j. From the definition of the transaction decomposition, the theorem follows. □

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

Thus, the total cost of the Relay protocol with respect to R_i is given by:

$$\text{cost}_{Relay}^i = \sum_{j=1}^{|T^i|} c_R^i(T_j) = \sum_{j=1}^{|T^i|} [d_{\mathcal{T}}(v_j, v_{i,j}^{\nearrow}(1)) + \lambda_i(j)\tau_j] + \sum_{k=1}^{|S^i|} c_R(S_{\phi_R(k-1)}, S_{\phi_R(k)}) \tag{4}$$

Cost of OPT. We now consider the cost of an optimal clairvoyant offline ordering algorithm, denoted OPT, that has a complete knowledge of all the transactions T. Clearly, an optimal offline

algorithm just has to order each transaction to receive the object once to commit. Let ϕ_O be the order of OPT. For the cost of OPT, we have to take into account its complete knowledge of all transactions. For a transaction $T_j = ((v_j, t_j, \vec{R}(j), \tau_j))$, the algorithm OPT already knows the succeeding transaction $T_k = ((v_k, t_k, \vec{R}(k), \tau_k))$. When the object is available at v_j, the algorithm can immediately send the object to v_k. Hence, we define the transaction T_j's completion time in the order ϕ_O as t_j^O. We therefore define the moving cost $c_O^i(T_j, T_k)$ of ordering T_k after T_j in the ϕ_O order as: $c_O^i(T_j, T_k) := d_{\mathcal{T}}(v_j, v_k) + \max\{0, t_j^O - t_k + d_{\mathcal{T}}(v_j, v_k)\} + \tau_k \geq d_{\mathcal{T}}(v_j, v_k) + \max\{0, t_j - t_k + d_{\mathcal{T}}(v_j, v_k)\} + \tau_k$. The total cost of an optimal algorithm with respect to R_i then becomes:

$$\text{cost}^i_{\text{OPT}} = \min_{\phi}\{\sum_{j=1}^{|T^i|} c_O^i(T_{\phi_O(j-1)}, T_{\phi_O(j)})\} \tag{5}$$

Hence, ϕ_O is an order which minimizes the sum of Equation 5.

The competitive ratio ρ_i achieved by the Relay protocol is the ratio between the cost of Relay and the cost of an optimal offline ordering algorithm:

$$\rho^i := \frac{\text{cost}^i_{Relay}}{\text{cost}^i_{\text{OPT}}} \tag{6}$$

4.2. Dynamic Analysis of the Relay Protocol

We now focus on the analysis of the order ϕ_R produced by the Relay protocol. As suggested in [12], the order produced by the Arrow protocol corresponds to a nearest neighbor traveling salesman path (TSP) on the set of requests by defining a new comparable cost metric. Motivated by this method, we first define a new cost metric c_T. Then, we show that the cost of ordering all sub-transactions in ϕ_R with respect to c_T is comparable to the cost^i_{Relay}.

Definition 6: Let S_j and S_k be two sub-transactions such that Relay orders S_j before S_k, i.e., $\phi_R(S_j) < \phi_R(S_k)$. Then the cost metric $c_T^i(S_j, S_k)$ is defined as:

$$c_T^i(S_j, S_k) := t_k + \overnearrow{d_i}(k) + \text{dist}_i(k) - t_j - \overnearrow{d_i}(j) - \text{dist}_i(j)$$

We have the following theorem.

Theorem 5: The order of ϕ_R is defined by a nearest neighbor TSP path on the metric $c_T^i(S_j, S_k)$, starting with the sub-transaction S_0. Further, $c_T(S_j, S_k) \geq 0$ for all pairs of request r_j and r_k.

Proof: We prove Theorem 5 by induction. The object is initialized at S_0. For this dummy token, $t_0 = \overnearrow{d_i}(0) = \text{dist}_i(0) = 0$. The sub-transaction S_j which minimizes $t_j + d_{\mathcal{T}}(v_j, v_0)$ arrives at v_0 first. By the definition of ϕ_R, this is the sub-transaction $S_{\phi_R(1)}$. In this case, $\overnearrow{d_i}(j) = d_{\mathcal{T}}(v_j, v_0)$ and $\text{dist}_i(j) = 0$. The sub-transaction T_j is the one that minimizes $c_T^i(S_0, S_k)$ for all $S_k \in S^i \backslash \{S_0\}$. Clearly, $c_T^i(S_0, S_{\phi_R(1)}) \geq 0$.

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

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

$$\begin{aligned}
0 &\leq c_T^i(S_{\phi_R(k'-1)}, S_{\phi_R(k+1)}) - c_T^i(S_{\phi_R(k'-1)}, S_{\phi_R(k')}) \\
&= t_{k'+1} + d_i^{\nearrow}(k'+1) + \text{dist}_i(k'+1) - t_{k'-1} - d_i^{\nearrow}(k'-1) - \text{dist}_i(k'-1) \\
&\quad - (t_{k'} + d_i^{\nearrow}(k') + \text{dist}_i(k') - t_{k'-1} - d_i^{\nearrow}(k'-1) - \text{dist}_i(k'-1)) \\
&= c_T^i(S_{\phi_R(k')}, S_{\phi_R(k+1)}).
\end{aligned}$$

The theorem follows. □

Let C_T^i be the cost of ordering all sub-transactions in ϕ_R with respect to c_T^i. We have the following theorem.

Theorem 6:

$$C_T^i \geq \sum_{k=1}^{|S^i|} c_R(S_{\phi_R(k-1)}, S_{\phi_R(k)}) - D,$$

where D is the diameter of the spanning tree $\mathcal{T}$.

Proof: We first show that:

$$c_T^i(S_{\phi_R(k-1)}, S_{\phi_R(k)}) \geq c_R(S_{\phi_R(k-2)}, S_{\phi_R(k-1)}) \tag{7}$$

where $k \geq 2$. Note that $c_R(S_{\phi_R(k-2)}, S_{\phi_R(1)}) = d_{\mathcal{T}}(v_{\phi_R(k-2)}, v_{\phi_R(k-1)})$ by definition. Since $c_T^i(S_{\phi_R(k-1)}, S_{\phi_R(k)}) = t_k + d_i^{\nearrow}(k) + \text{dist}_i(k) - t_{k-1} - d_i^{\nearrow}(k-1) - \text{dist}_i(k-1)$, note that the object arrives at S_{k-1} at time $t_{k-1} + d_i^{\nearrow}(k-1) + \text{dist}_i(k-1) + d_{\mathcal{T}}(v_{\phi_R(k-2)}, v_{\phi_R(k-1)})$. Hence, the fastest way for $S_{\phi_R(k)}$ to get the object is that the object is moved to $S_{\phi_R(k)}$ once it arrives at $S_{\phi_R(k-1)}$, i.e., $S_{\phi_R(k)}$ aborts $S_{\phi_R(k-1)}$. In this case, $t_k + d_i^{\nearrow}(k) + \text{dist}_i(k) = t_{k-1} + d_i^{\nearrow}(k-1) + \text{dist}_i(k-1) + d_{\mathcal{T}}(v_{\phi_R(k-2)}, v_{\phi_R(k-1)})$, which is minimum. Equation 7 follows. By summing up over k, we have:

$$C_T^i \geq \sum_{k=1}^{|S^i|} c_R(S_{\phi_R(k-1)}, S_{\phi_R(k)}) + t_{\phi_R(1)} + d_{\mathcal{T}}(v_{\phi_R(1)}, v_0) - d_{\mathcal{T}}(v_{\phi_R(|S^i|-1)}, v_{\phi_R(|S^i|)}),$$

which completes the proof. □

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

Lemma 1: Let $S_{\phi_R(k)}$ and $S_{\phi_R(k+1)}$ be two consecutive sub-transactions in order ϕ_R. Let $\epsilon := c_T(S_{\phi_R(k)}, S_{\phi_R(k+1)}) - d_{\mathcal{T}}(v_{\phi_R(k-1)}, v_{\phi_R(k)}) - \tau_k$. If $\epsilon > 0$, for all sub-transactions $S_{\phi_R(l)}$ where $l \geq k+1$, $t_{\phi_R(l)}$ can be replaced by $t_{\phi_R(l)} - \epsilon$ without increasing the cost of Relay and OPT.

Proof: The proof follows the same argument of the proof of Lemma 2.6 in [12]. □

By applying Lemma 1 as many times as possible, we have the following theorem:

Theorem 7: The upper bound of the cost $c_T^i(S_j, S_k)$ of the longest edge on Relay's path is: $c_T^i(S_j, S_k) \leq D + \max_{l=1}^{|S^i|} \tau_l$.

4.3. Competitive Ratio of the Relay Protocol

We first define the Manhattan metric c_M which is comparable to c_O^i.

Definition 7 (Manhattan Metric): The Manhattan metric $c_M(T_j, T_k)$ is defined as:

$$c_M(t_j, t_k) := d_{\mathcal{T}}(v_j, v_k) + |t_j - t_k| + \tau_j + \tau_k.$$

Lemma 2: Let ϕ be an ordering and C_O^i and C_M be the costs for ordering all transactions in order ϕ with respect to c_O and c_M. The Manhattan cost is bounded by: $C_M \leq 2C_O + t_{\phi_{(|T^i)|}}$.

Proof: We can lower bound the optimal cost of c_O^i by:

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

Let $D_{\mathcal{T}} = \sum_{j=1}^{|T^i|}\{d_{\mathcal{T}}(v_{\phi(j-1),\phi(j)}) + \tau_j + \tau_{j-1}\}$. Then we have: $2C_O^i \geq D_{\mathcal{T}} + 2\sum_{j=1}^{|T^i|} \max\{0, t_{\phi(j-1)} - t_j\} = D_{\mathcal{T}} + \sum_{j=1}^{|T^i|} |0, t_{\phi(j-1)} - t_j| - t_{\phi(|T^i|)} = C_M - t_{\phi(|T^i|)}$ □

The lemma follows.

We use the following lemma from [12]:

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

Hence, we have the following theorem to make C_M comparable to C_O^i:

Theorem 8:

$$C_M \leq 6C_O^i$$

Proof: The theorem can be proved by Lemmas 2 and 3. Note that we have $c_M \geq c_{M'}$ and $t_{|T^i|} \geq t_{\phi(T^i)}$. Then the theorem follows. □

We now compare C_M and C_T^i with the help of the following theorem from [12]:

Theorem 9: Let V be a set of $N := |V|$ and let $d_n : V \times V \to \Re$ and $d_o : V \times V \to \Re$ be the distance functions between nodes of V. For d_n and d_o, the following conditions hold:

$$d_o(u, v) = d_o(v, u), \quad d_n(u, v) = d_n(v, u)$$

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

$$d_o(u, w) \leq d_o(u, v) + d_o(v, w)$$

Let C_N be the length of a nearest neighbor TSP tour with respect to the distance function d_n and let C_O be the length of an optimal TSP tour with respect to the distance function d_o. Then, $C_N \leq \frac{3}{2}\lceil \log_2(D_{NN}/d_{NN})\rceil \cdot C_O$ holds, where D_{NN} and d_{NN} are the lengths of the longest and the shortest non-zero edge on the nearest neighbor tour with respect to d_n.

Now we have the following theorem:

Theorem 10:

$$C_T^i \leq 2\left\lceil \log_2(D_0 + \max_{j=1}^{|T^i|} \tau_j)\right\rceil (C_M - 2\sum_{k=1}^{|T^i|} \tau_k)$$

This theorem follows from Theorems 1 and 9. Note that c_T^i and c_M comply with the conditions for $d_n(u, v)$ and $d_o(u, v)$, respectively. By Lemma 1, we have $c_T^i S_j, S_k \leq c_M T_j, T_k$. And the triangle

inequality holds for c_M. Finally, we can bound the shortest value of c_T^i by $\min_{v_j,v_k \in V} d(v_j, v_k)$. The theorem follows.

Theorem 11:

$$\rho^i = \frac{\text{cost}^i_{Relay}}{\text{cost}^i_{\text{OPT}}} = O\Big(\max[\log(D_0 + \max_{j=1}^{|T^i|} \tau_j), \frac{|T^i| \max_{j=1}^{|T^i|} \tau_j}{H^i_{\mathcal{T}}}]\Big)$$

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

Proof: From Equation 4, Theorems 6 and 8, we have:

$$\text{cost}^i_{Relay} \leq 12 \left\lceil \log_2(D_0 + \max_{j=1}^{|T^i|} \tau_j) \right\rceil \text{cost}^i_{\text{OPT}} + D + \sum_{j=1}^{|T^i|} \lambda_i(j)\tau_j.$$

Since $\text{cost}^i_{\text{OPT}} = \sum_{j=1}^{|T^i|} \Big(d_{\mathcal{T}}(v_{\phi_O(j-1)}, v_{\phi_O(j)}) + \max\{0, t^O_{\phi_O(j)} - t_{\phi_O(j)} + d_{\mathcal{T}}(v_{\phi_O(j-1)}, v_{\phi_O(j)})\} + \tau_{\phi_O(j)} \Big) \leq H^i_{\mathcal{T}} + \sum_{j=1}^{|T^i|} \tau_j$, the theorem follows. □

From Theorem 11, we know that ρ^i is determined by the value of the maximum τ_j. We have the following theorem for a possible range of the value of the maximum τ_j.

Theorem 12:

$$\rho^i = O(\log D)$$

if

$$\max_{j=1}^{|T^i|} \tau_j = O\Big(\log D \cdot \min_{v_k, v_l \in V} d_{\mathcal{T}}(v_k, v_l) \Big)$$

In other words, if the maximum local execution time of a set of transactions T^i is sufficiently small (up to the logarithmic order of the diameter of the spanning tree), the competitive ratio ρ^i is $O(\log D)$.

5. Conclusions

We conclude that the Relay protocol is $O(\log D)$-competitive for a set of transactions with sufficiently small maximum local execution time. Hence, the Relay protocol is appropriate for distributed systems, in which the network latency plays the major role in the total time complexity. For the transactions with maximum local execution time, we can use Theorem 11 to analyze the competitive ratio. When the maximum local execution time of transactions is sufficiently large, i.e., $\Omega(D)$, the execution time will be the dominating part of the total time complexity. In this case, the performance of a distributed TM system is not determined by the cache-coherence protocol, but by the underlying contention manager, which determines the maximum number of abortion times of a single transaction, just like the case for multiprocessors.

The Relay protocol works on a fixed spanning tree. Hence, finding a good spanning tree is an important problem. The most recent breakthrough on this is due to Emek and Peleg [5], who present an $O(\log n)$-approximation algorithm for finding the spanning tree with the maximum stretch in a graph. The Relay protocol is designed to support multiple objects. Since the protocol is totally distributed (all nodes are of the same importance in the protocol), it avoids significantly overloading some nodes in the network.

There are several directions for future work. Fault-tolerance is an important issue. Similar to [17], a self-stabilizing algorithm can also be designed for the Relay protocol.

References

[1] Hagit Attiya, Leah Epstein, Hadas Shachnai, Tami Tamir: Transactional contention management as a non-clairvoyant scheduling problem. In PODC '06, 308–315

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

[3] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In ASPLOS'06, 336–346

[4] Demmer, Michael J., Herlihy, Maurice: The Arrow Distributed Directory Protocol. In DISC '98, 119–133

[5] Emek, Yuval, Peleg, David: Approximating Minimum Max-Stretch spanning Trees on unweighted graphs. In SODA '04, 261–270

[6] Rachid Guerraoui, Maurice Herlihy, Bastian Pochon: Toward a theory of transactional contention managers. In PODC '05, 258–264

[7] 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 ISCA'04, 102–113

[8] Maurice Herlihy, Victor Luchangco, Mark Moir: Obstruction-free Synchronization: Double-ended Queues as an Example. In ICDCS'03, 522–529

[9] Herlihy, Maurice, Luchangco, Victor, Moir, Mark: A flexible framework for implementing software transactional memory. In OOPSLA '06, 253–262

[10] Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, III: Software transactional memory for dynamic-sized data structures. In PODC '03, 92–101

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

[12] Kuhn, Fabian, Wattenhofer, Roger: Dynamic analysis of the arrow distributed protocol. In SPAA '04, 294–301

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

[14] Raymond, Kerry: A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst., 7(1): 61–77 (1989)

[15] N. Shavit, D. Touitou: Software Transactional Memory. In PODC '95, 204–213

[16] William N. Scherer, III, Michael L. Scott: Advanced contention management for dynamic software transactional memory. In PODC '05, 240–248 (2005)

[17] Srikanta Tirthapura, Maurice Herlihy: Self-Stabilizing Distributed Queuing. IEEE Transactions on Parallel and Distributed Systems, 17(7): 646–655 (2006)

[18] Bo Zhang, Binoy Ravindran: Relay: A Cache-Coherence Protocol for Distributed Transactional Memory. In OPODIS'09, To Appear.