Distributed Queuing or Distributed Priority Queuing? On the Design of Cache-Coherence Protocols 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

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. In this paper, we formalize two classes of cache-coherence protocols: distributed queuing cache-coherence (DQC) protocols and distributed priority queuing cache-coherence (DPQC) protocols, both of which can be implemented based on a given distributed queuing protocol. We analyze the two classes of protocols for a set of dynamically generated transactions and compare their time complexities against that of an optimal offline clairvoyant algorithm. We show that a DQC protocol is $O(N \log D_\delta)$-competitive and a DPQC protocol is $O(\log D_\delta)$-competitive for a set of $N$ transactions requesting the same object, where $D_\delta$ is the normalized diameter provided by the underlying distributed queuing protocol.
1. Introduction

Lock-based synchronization is often non-scalable, non-composable and inherently error-prone. 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. Two transactions conflict if they access the same object and one access is a write. The transactional approach to contention management [8] guarantees atomicity by ensuring that whenever a conflict occurs, only one of the transactions involved can proceed. Transactional memory API for multiprocessors have been proposed in hardware [6], in software [7], [13], and in hardware/software combination [3].

Distributed TM is motivated by the difficulties of lock-based synchronization in distributed (control-flow) programming models such as RPCs. For example, RPC calls, while holding locks, can become remotely blocked on other (RPC) 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 motivated research on the design of distributed transactional memory as a possible solution. For example, in the data-flow distributed TM model of [9] (that we also consider), object performance bottlenecks can be reduced by exploiting locality: move the object to nodes. 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 TM in distributed systems include [2], [9], [11], [15] and [16]. None of [11] and [2] present theoretical analysis of the fundamental properties of distributed TM, such as the performance bounds of their proposed TM systems. In [9], Herlihy and Sun present a Ballistic distributed cache-coherence protocol in a metric-space network, where the communication cost 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 [16], which focuses on optimizing the worst-case queue length of the distributed queue, i.e., it reduces the total number of transaction abortions. The Relay protocol is motivated by the Arrow distributed queuing protocol [4], due to the similarities between the distributed queuing problem and the problem of synchronization of write/read access requests to mobile objects in distributed TM systems. 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.

Past works in [9], [15] and [16] rely on an existing distributed queuing protocol. However, is distributed queuing the only way to solve the cache-coherence problem for distributed TM? If so, what performance bound can it achieve? If not, can we apply a more appropriate abstraction to solve the cache-coherence problem for distributed TM? Since a distributed queuing protocol does not consider the contention between two transactions, can we develop another class of protocols that take transactional contention into account? Finally, if such a new class of protocols can be developed, how do they compare against distributed queuing based cache-coherence protocols? Can they guarantee a better performance? In this paper, we answer these
We first formalize a class of cache-coherence protocols based on distributed queuing, called distributed queuing cache-coherence (DQC) protocols, which encompass all cache-coherence protocols based on distributed queuing. A DQC protocol works like a distributed queuing protocol: a distributed queue is formed by a set of transactions which request the same object. A transaction joins the queue by sending a request to the tail of the queue, and it becomes the new tail of the queue. We implement a DQC protocol based on a distributed cache-coherence protocol $C$, which provides an ordering cost $\delta_C(r_i, r_j)$ to order a request $r_j$ after request $r_i$.

Next, we formalize a novel class of cache-coherence protocols, which are based on distributed priority queuing, called distributed priority queuing cache-coherence (DPQC) protocols. For a DPQC protocol, a distributed queue is also formed by a set of transactions. However, a DPQC protocol guarantees that at any given time, only the transaction with the highest priority in the queue can commit. Moreover, we can implement a DPQC protocol based on the same distributed cache-coherence protocol $C$ as a DQC protocol. Hence, we can compare DQC and DPQC on the same ground.

We evaluate a cache-coherence protocol by measuring its competitive ratio, which is the ratio of the time complexity to commit a set of dynamically generated transactions $T$ under a certain cache-coherence protocol to the time complexity to commit the same set of transactions under an optimal clairvoyant algorithm OPT. We show that, for a set of $N$ transactions requesting the same object, the competitive ratio of a DQC protocol is $O(N \log D_\delta)$ and the competitive ratio of a DPQC protocol is $O(\log D_\delta)$, if the maximum local execution time of the transactions in $T$ is $O(\log D_\delta)$, where $D_\delta$ and $D_\delta$ are the normalized diameter and the diameter provided by the underlying distributed queuing protocol $C$, respectively. This result illustrates the advantage of DPQC protocols over DQC protocols.

Hence, we make the following contributions in this paper:

1) We formalize the class of DQC protocols, which encompass all cache-coherence protocols based on distributed queuing, e.g., the Ballistic protocol. We give the implementation of a DQC protocol based on a distributed queuing protocol $C$.

2) We formalize the class of DPQC protocols, a novel class of cache-coherence protocols based on distributed priority queuing. We show that a DPQC protocol can also be implemented based on the same distributed queuing protocol $C$.

3) We evaluate DQC and DPQC protocols by measuring their competitive ratios. Given the same underlying distributed queuing protocol, we show that DPQC protocols guarantee a worst-case competitive ratio which is a factor proportional to $N$ better than that of DQC protocols.

This is the first ever formalization of DQC protocols, and the development of DPQC protocols as an alternative, which yield a better competitive ratio. As far as we know, DPQC protocol guarantees a better competitive ratio than existing cache-coherence protocols based on distributed queuing.

The rest of the paper is organized as follows. We present our system model in Section 2. We formalize and implement the classes of DQC and DPQC protocols in Section 3, respectively. We analyze DQC and DPQC protocols and compare their performance in Section 4. The paper concludes in Section 5. Due to space constraints, we skip the proof Theorem 10 and present it in the Appendix.

2. System Model

2.1. Network Model.

We consider Herlihy and Sun’s metric-space network model [9], where the communication cost between nodes form a metric. 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 $e(u, v) \in E$.

2.2. Distributed Transactional Memory Model

A transaction is a sequence of requests, each of which is a read or write operation request to an individual object. Given a set of $s \geq 1$ objects, $\{R_1, \ldots, R_s\}$, we can use the tuple $T_j = (v_j, t_j, R(j), \tau_j)$ to identify a transaction $T_j$. We explain each field of $T_j$ as follows:
- $v_j$: the node that initiates the transaction.
- $t_i$: the time when the transaction is initiated.
- $R(j)$: the vector that describes the sequence of requests of $T_j$. Let $R(j) = \{R_1(j), \ldots, R_s(j)\}$, 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 $R_i$.
- $\tau_j$: 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.

To understand the design of the support for the transactional memory API in a distributed system, we consider Herlihy and Sun's data-flow model [9]. In this model, transactions are immobile (running at a single node), but objects move from node to node. Transactional synchronization is optimistic: a transaction commits only if no other transaction has executed a conflicting access. A contention manager assigns priorities to transactions. A running transaction could only be aborted by another transaction with a higher priority. We use $T_A \prec T_B$ to represent that transaction $T_A$ is issued a higher priority than $T_B$. Hence, if $T_A \prec T_B$, the TM proxy aborts $T_B$ and sends the object to node $A$. If $T_B \prec T_A$, the TM proxy can postpone the response to node $A$ to give the local transaction $(T_B)$ a chance to commit. The object will be sent to node $A$ at some time after $T_B$’s commit under a certain protocol.

Thus, the design of a distributed TM system under the data-flow model is composed of two parts: a contention manager to mediate conflicts and a certain 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 requester’s cache, and invalidate the old copy. Specifically, a distributed cache-coherence protocol must be able to perform the following functions:
1) When a transaction at a node $A$ attempts to access an object in the network, a distributed cache-coherence protocol is invoked to carry node $A$’s request to the node which holds the object in a finite time period.
2) When a node $B$, which receives a read/write access request for an object it holds, has made the decision whether to abort the local transaction or to postpone the response, a distributed cache-coherence protocol is invoked to move the object either immediately or after some time. In either case, the distributed cache-coherence protocol must guarantee that the object is moved to the requester in a finite time period.
3) At any given time, a distributed cache-coherence protocol must guarantee that there exists only one writable copy of each object in the network. In other words, each object can only be written by one transaction at any given time.
Different contention managers have been studied in the past [14]. 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 [5] properties:

**Definition 1:** A contention manager is **work conserving** if it always lets a maximal set of non-conflicting transactions to run. 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 [5], which uses a globally consistent priority policy that issues priorities to transactions is shown in [1] to satisfy both properties. With a given contention manager, the core of a distributed TM system lies in the design of an efficient distributed cache-coherence protocol.

3. DQC and DPQC Protocols

3.1. Distributed Queuing Protocol

One way to design a distributed cache-coherence protocol is to consider a distributed cache-coherence problem as a 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.

In this paper, we assume a given distributed queuing protocol $C$. We define the *ordering cost* of $C$ as follows:

**Definition 2 (Ordering Cost):** The ordering cost $\delta_C(r_i, r_j)$ is the communication cost to order request $r_j$ after request $r_i$ under $C$, i.e., if $v_j$ invokes the request $r_j$ at time $t_j$, then $v_i$ which invokes $r_i$ receives $r_j$ at $t_j + \delta_C(r_i, r_j)$.

In practice, $\delta_C(r_i, r_j)$ is the communication cost for $r_j$ to realize $r_i$ so that $r_j$ can be ordered after $r_i$ under $C$. For example, for the Arrow protocol [12], $\delta_C(r_i, r_j)$ is the distance of the path from $r_j$ to $r_i$ of the underlying spanning tree in the metric space. In this paper, we assume that $\delta_C(r_i, r_j)$ forms a metric, i.e., (1) $\delta_C(r_i, r_j) = 0$ if and only if $r_i = r_j$; (2) $\delta_C(r_i, r_j)$ satisfies the triangle inequality; (3)$\delta_C(r_i, r_j)$ is non-negative and symmetric.

3.2. Distributed Queuing Cache-Coherence Protocols

We can design a distributed cache-coherence protocol based on distributed queuing, called a *distributed queuing cache-coherence* (DQC) protocol. For example, the Ballistic protocol [9] is a DQC protocol. Each transaction is considered as some ordering requests for a set of objects. Therefore, for a set of $s$ objects, there are $s$ distributed queues established. However, for a distributed cache-coherence protocol, a distributed queue is no longer fixed — an aborted transaction has to join the queue again and therefore the length of the queue is dynamically increased.

We illustrate the DQC protocol in Figure 1. A distributed queue is formed by a set of transactions which request for a single object. A transaction joins the queue by sending a request to the tail of the
queue, and becomes the new tail of the queue, i.e., the enqueue operation. Like a distributed queuing protocol, the enqueue operation simply “orders” a transaction after its predecessor and becomes the new tail of the queue. There are two possible cases corresponding to the dequeue operation. If the head of the queue \( T_H \) commits, it will be removed from the queue after sending the object to its successor in the queue. If it is aborted by its successor, transaction \( T_H \) is restarted immediately and requests to join the queue again. We assume that, at the start, the queue contains a dummy transaction at the object’s initial location. Hence, at any given time, the queue contains at least one transaction.

Assume that there is a DQC protocol \( P \). It should provide two functions:

1. \( P.\text{findpre}(T_i) \): Let transaction \( T_i \) find its predecessor in the queue. This operation corresponds to the enqueue operation to append \( T_i \) to the tail of the queue.
2. \( P.\text{movesuc}(T_i) \): Move the object from \( T_i \) to its successor in the queue. This operation corresponds to the dequeue operation to let \( T_i \) leave the queue (it will join the queue again if it is aborted by its successor).

We can implement \( P.\text{findpre}(T_i) \) directly using the given distributed queuing protocol \( C \). For the cost of \( P.\text{movesuc}(T_i) \), in practice, the object is only moved when its destination is known. The simplest way is just moving it along the shortest path between two nodes. We define their costs as follows.

**Definition 3 (Locating Cost \( \delta^P(T_i, T_j) \)):** The cost for \( T_i \) to find its predecessor \( T_j \) under \( P \).

**Definition 4 (Moving Cost \( \zeta^P(T_i, T_j) \)):** The cost to move an object from \( T_i \) to its successor \( T_j \) under \( P \).

Then we have \( \delta^P(T_i, T_j) = \delta^C(T_i, T_j) \) and \( \zeta^P(T_i, T_j) = O(\delta^P(T_i, T_j)) \).

![Figure 1. The DQC protocol](image1.png)

![Figure 2. The DPQC protocol](image2.png)

### 3.3. Distributed Priority Queuing Cache-Coherence Protocols

Now we present the class of distributed priority queuing cache-coherence (DPQC) protocols based on distributed priority queuing. For each object, a distributed priority queue is established. Unlike the enqueue operation of a DQC protocol, it is not required to append a transaction to the tail of the queue for a DPQC protocol. Instead, a transaction is inserted somewhere in the queue as long as its priority is “learned” by the queue. For the dequeue operation, the transaction that leaves the queue is the one with the highest priority.

We illustrate the DPQC protocol in Figure 2. At any given time, the dequeued transaction is the highest priority element \( T_H \) of the queue. Hence, \( T_H \) only leaves the queue after it commits and sends the object
to another element $T_{H'}$ of the queue. Since only the highest priority element can be dequeued, it is obvious that at any given time, the object is held by the transaction with the highest priority of the queue. Hence, $T_{H'}$ should be the transaction with the highest priority of the queue after $T_H$ leaves.

A distributed cache-coherence protocol $P'$ based on the distributed priority queuing model should provide the following functions:

1. $P'.insert(T_i)$: Insert transaction $T_i$ into the queue. This operation corresponds to the enqueue operation.
2. $P'.movesuc(T_i, T_j)$: Move the object from $T_i$ to the second highest priority transaction $T_j$. This operation corresponds to the dequeue operation to let $T_i$ leave the queue after it commits.
3. $P'.movepre(T_i, T_j)$: Move the object from $T_j$ to a newly joined transaction $T_i$ with a higher priority, i.e., $T_j$ is aborted by a newly joined transaction.

We can define the corresponding costs of $P'.insert(T_i)$, $P'.movesuc(T_i, T_j)$, and $P'.movepre(T_i, T_j)$ as $\Delta P'(T_i)$, $Z_{suc}^P(T_i, T_j)$, and $Z_{pre}^P(T_i, T_j)$ for a DPQC protocol $P'$, respectively.

The question now is how to implement $P'$. Note that we present the implementation of a DQC protocol $P$ based on a given distributed queuing protocol $C$. So if we can implement $P'$ based on the same distributed queuing protocol, then we can compare these two distributed cache-coherence protocols on the same ground. Therefore, in this section, we present a distributed priority queuing protocol implementation based on the given distributed queuing protocol $C$.

The main idea is to maintain a distributed queue in the priority order. Assuming a queue $Q = \{Q_H, Q_{H+1}, \ldots, Q_{H+l}\}$, where $Q_i$ is the predecessor of $Q_{i+1}$ and $Q_H$ is the head of the queue, we have $Q_H \prec Q_{H+1} \prec \ldots \prec Q_{H+l}$. To insert a transaction $T_i$ into the queue, we have to find the correct position to insert it such that the priority order of the queue is not violated. As long as the queue is correctly maintained, the $movesuc(T_i, T_j)$ operation works in the same way as that of the distributed queuing protocol: the object is moved to its successor after a transaction commits. The $P'.movepre(T_i, T_j)$ function is only performed when the highest priority transaction of the queue is aborted by a newly joined transaction. Then the new transaction becomes the new head of the queue (the predecessor of the old head) and the object is moved to it.

![Figure 3. $P'.findpre(T_i)$ and $P'.movesuc(T_i)$](image)

![Figure 4. $P'.movesuc(T_i, T_j)$](image)

In order to implement these functions, we need to revisit the two basic functions of $P$: $P'.findpre(T_i)$ and $P'.movesuc(T_i)$. We describe them in Figure 3. For a DQC protocol, a global directory is maintained to always point to the tail of the queue ($Q_T$ in Figure 3(a)). Each element of the queue keeps a local pointer to its successor. If a transaction $T_T$ wants to join the queue, the $P'.findpre(T_i)$ function performs the following two operations:

1. $P'.send(T_i)$: Send a request to the tail of the queue $Q_T$ following the global directory, and thus a local pointer to $T_i$ is created at $Q_T$;
2 $P'.redirect(T_i)$: Redirect the global directory to let it point to $T_i$, and $T_i$ becomes the new tail of the queue.

Such operations of $P'.findpre(T_i)$ is shown in Figure 3. We now focus on the cost of these two operations. Note that we define the total cost of $P'.findpre(T_i)$ as the locating cost $\delta_s^T(T_i, T_j)$, where $T_j$ is $T_i$'s predecessor. Let the cost of $P'.send(T_i)$ and $P'.redirect(T_i)$ be denoted as $\delta_s^P(T_i, T_j)$ and $\delta_r^P(T_i, T_j)$, respectively. Then we have: $\{\delta_s^T(T_i, T_j), \delta_s^P(T_i, T_j)\} \leq \delta_r^P(T_i, T_j)$.

Now we focus on the $P'.movesuc(T_i)$ function. Since at any given time, only the head of the queue can be dequeued, $P'.movesuc(T_i)$ is equivalent to $P'.movesuc(Q_H)$ when $T_i$ is the head of the queue. As shown in Figure 3, when $Q_H$ wants to leave the queue, the $P'.movesuc(T_i)$ function performs the following two operations:

1 $P'.mvobj(T_i)$: Move the object to $T_i$'s successor following the local pointer of $T_i$;
2 $P'.rmvptr(T_i)$: Remove the local pointer of $T_i$, and $T_i$’s successor now becomes the new head of the queue.

Similarly, we let the cost of $P'.mvobj(T_i)$ and $P'.rmvptr(T_i)$ be denoted as $\zeta_m^P(T_i, T_j)$ and $\zeta_f^P(T_i, T_j)$, respectively, where $T_j$ is $T_i$’s successor. Then we have: $\{\zeta_m^P(T_i, T_j), \zeta_f^P(T_i, T_j)\} \leq \zeta_r^P(T_i, T_j)$.

The function $P'.movesuc(T_i, T_j)$ is illustrated in Figure 4. Note that $T_i$ is the head of the queue, and $T_j$ is $T_i$’s successor. The only difference between $P'.movesuc(T_i, T_j)$ and $P'.movesuc(T_i)$ of the distributed queuing model is that $P'.movesuc(T_i, T_j)$ needs a $P'.redirect(T_j)$ operation to redirect the global directory to $T_j$ ($P'.redirect(T_j)$ in Figure 4(b). In this way, the global directory is maintained to point to the head of the queue.

![Figure 5. $P'.insert(T_i)$](image)

![Figure 6. $P'.movepre(T_i, T_j)$](image)

Now we can implement the $P'.insert(T_i)$ function with the operations provided by $P'.findpre(T_i)$ and $P'.movesuc(T_i)$, as shown in Figure 5. Assume a queue in the priority order. Let the global directory point to the head of the queue (the one with the highest priority) (Figure 5).

We will now show that for all functions, the global directory is maintained to point to the head of the queue. At first, a $P'.send(T_i)$ operation is performed. Then $Q_H$ establishes a local pointer to $T_i$ (Figure 5(b)). Then the contention manager at $Q_H$ compares the priorities of its successor $Q_{H+1}$ and $T_i$. If $Q_{H+1} < T_i$, then $T_i$ should be queued after $Q_{H+1}$; if not then $T_i$ should be queued just after $Q_H$. Assume that $Q_{H+k} < T_i < Q_{H+k+1}, k \geq 0$. We need $k$ $P'.mvobj(Q_{H+j})$ ($j = 0, 1, \ldots, k-1$) operations
to move \( T_i \)'s request to \( Q_{H+k} \), as shown in Figure 5(c). After these operations, \( Q_{H+k} \) keeps a local pointer to \( T_i \), and it knows that \( T_i < Q_{H+k+1} \). Hence, another \( P.mvobj(Q_H) \) operation is performed to move \( Q_{H+k+1} \)'s request to \( T_i \), and \( Q_{H+k} \) removes its old local pointer (to \( Q_{H+k+1} \)) by \( P.rmvrptr(Q_{H+k}) \) (Figure 5(d)). Therefore, \( T_i \) is inserted between \( Q_{H+k} \) and \( Q_{H+k+1} \).

Note that the \( P'.insert(T_i) \) function implies that \( Q_H < T_i \). In practice, it is possible that \( T_i < Q_H \), and \( P'.movepre(T_i, T_j) \) function is performed, as shown in Figure 6. The first step is the same as \( P'.insert(T_i) \): \( T_i \)'s request is sent to \( Q_H \) following the global directory, and \( Q_H \) establishes a local pointer to \( T_i \) (Figure 6(b)). Since \( T_i < Q_H \), \( Q_H \) is aborted and the object is moved to \( T_i \). Hence a \( P.movesuc(Q_H) \) operation is needed to move the object to \( T_i \) and remove \( Q_H \)'s local pointer to \( T_i \) (Figure 6(c)). \( T_i \) can immediately establish a local pointer to \( Q_H \) when it receives the object from \( Q_H \). Finally, a \( P.redirect(T_i) \) operation is needed to maintain the global directory to point to \( T_i \), the new head of the queue (Figure 6(d)).

### 3.4. Cost Measures

**Cost measure for DQC protocols.** We now assume that there is a set of \( N \) transactions \( T := \{T_1, T_2, \ldots, T_N\} \) that require access to the same object. Therefore, we can construct a distributed queue \( Q := \{Q_0, Q_1, Q_2, \ldots, Q_L\} \) under DQC protocol \( P \) where \( L \geq N \). Specifically, we arrange each element \( Q_i \) in the order such that \( Q_i \) is the predecessor of \( Q_{i+1} \). Note that \( Q_0 \) is the dummy transaction at the object’s initial location. Hence, each transaction in \( T \) is mapped to at least one element in \( Q \), since each transaction may join the queue multiple times. Arrange the set of transactions \( T \) in the priority-order. Specifically, let \( T = \{T_1, T_2, \ldots, T_N\} \), where \( T_i < T_j \) if \( i < j \). We have the following theorem:

**Theorem 1:** \( \sigma_j \leq j \) for \( j \leq N \) and \( \sigma_i \) is the number of rounds that transaction \( T_i \) joins the queue.

**Proof:** The theorem can be directly proved by work conserving and pending commit properties of the contention manager.

From Theorem 1, we have the following corollary:

**Corollary 2:** \( L \leq \frac{N(N-1)}{2} \)

Theorem 1 and Corollary 2 give the upper bounds of the enqueue rounds for a single transaction and the total enqueue rounds for a set of transactions, respectively. It tells us that the enqueue rounds of a transaction depends on its priority. Intuitively, a transaction with a higher priority implies fewer enqueue rounds, and less cost to commit.

Let the cost to commit a transaction \( T_i \) denoted as \( M_i \). Hence, a transaction \( T_i \) invoked at time \( t_i \) will commit at \( t_i + M_i \). We can further define the total cost for a set of transactions \( T \) as: \( M = \max_{i=1}^{N} (t_i - t^1 + M_i) \), where \( t^1 = \min_{i=1}^{N} t_i \). The total cost \( M \) describes the time complexity to commit all transactions in \( T \). The total cost is composed of three parts: the cost for the object to travel in the network, the cost for executing operations on the object by transactions, and the cost for the object to wait for requests. Hence, a more useful cost measure is the amortized cost of a single transaction \( T_i \), i.e., the contribution made by transaction \( T_i \) to the total cost \( M \).

**Theorem 3:** Let the amortized cost of \( T_i \) under \( P \) be defined as:

\[
M(i) = \sum_{k=1}^{\sigma_i} \left[ \delta^C(Q_i(k), Q_{i(k)-1}) + \zeta^P(Q_i(k-1), Q_i(k)) + \tau_i \right].
\]

Then we have \( M \leq \sum_{i=1}^{N} M(i) \).

**Cost Measure for DPQC protocols.** Now we can measure the costs of \( P'.movesuc(T_i, T_j) \), and \( P'.movepre(T_i, T_j) \) since they are implemented by a set of functions of \( P \).

For \( P'.movesuc(T_i, T_j) \), we have: \( Z_{\text{suc}}^{P}(T_i, T_j) = \zeta^P(T_i, T_j) + \delta^P(T_j, T_i) \). Note that \( \delta^P(T_j, T_i) \) is the cost to redirect the global directory from \( T_i \) to \( T_j \).
For \( P'.movepre(T_i, T_j) \), we have: \( Z^P_{pre}(T_i, T_j) = \delta^p_i(T_i, T_j) + \zeta^P(T_j, T_i) + \delta^p_i(T_i, T_j) \).

For a DPQC protocol, a transaction can only be aborted when it becomes the head of the queue and a higher priority transaction joins the queue. Let the number of abortions of transaction \( T_i \) be denoted as \( \mu_i \). We have the following theorem.

**Theorem 4:** \( \mu_i \leq i - \lambda_i - 1; \sum_{i=1}^{N} \mu_i \leq N - 1 \)

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

A transaction can only be aborted when a new transaction joins the queue. Therefore the second part of the theorem can be proved directly. The theorem follows. \( \square \)

Now we focus on the total cost \( M' \) for a set of transactions \( T \) under \( P' \). Note that \( M' \) describes the time complexity to commit all transactions in \( T \) under \( P' \). Similar to a DQC protocol, we have the following theorem:

**Theorem 5:** Let the amortized of \( T_i \) under \( P' \) be defined as:

\[
M'(i) = \begin{cases} 
Z^P_{suc}(T_i, Q_{i.suc}) + \tau_i, & Q_{i.H} < T_i \\
Z^P_{suc}(T_i, Q_{i.suc}) + \tau_i + Z^P_{pre}(T_i, Q_{i.H}) + \tau^{i.H}, & T_i < Q_{i.H}
\end{cases}
\]

(2)

where \( Q_{i.suc} \) is \( T_i \)'s successor when \( T_i \) leaves the queue. Then we have \( M' \leq \sum_{i=1}^{N} M'(i) \).

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

4. Analysis

4.1. Cost Metrics

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

\[
c_q(Q_j, Q_i) := \delta^C(Q_j, Q_i) + \zeta^P(Q_i, Q_j).
\]

From Equation 1, we have:

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

(3)

**Cost Metric of** \( P' \). We decompose each transaction \( T_i \) into a set of sub-transactions \( \{T_i(1), T_i(2), \ldots, T_i(\mu_i + 1)\} \). Specifically, we have \( T_i = (v_i, t_i, \tau_i) \) and \( T_i(k) = (v_i, t_i(k), \tau_i) \), where \( t_i(1) = t_i \) and \( t_i(k) \) is the time when \( T_i \)'s \( (k-1) \)th abortion occurs. We arrange all these sub-transactions in the order of obtaining the object: \( T' = \{T(0), T(1), T(2), \ldots, T(L')\} \), where \( N \leq L' \leq 2N - 1 \), since each transaction may obtain the object multiple times. The second inequality stems from Theorem 4. Each
sub-transaction $T_i(k)$ is mapped to one element in $T'$. $T(i) = (v(i), t(i), \tau(i))$ denotes the $i^{th}$ transaction that receives the object in $P'$'s order. We define the cost metric to order $T(j)$ after $T(i)$ under $P'$ as
\[
  c_t(T(i), T(j)) := 2\delta^c(T(j), T(i)) + \zeta^p(T(i), T(j)).
\]

Note that $P'$ is based on the distributed queuing protocol $C$. From Equation 2 we have:
\[
  \sum_{i=1}^{N} M'(i) \leq \sum_{k=1}^{L'} c_t(T(k-1), T(k)) + \sum_{j=1}^{N} (\mu_j + 1) \tau_j.
\]

**Cost Metric of Opt.** To evaluate the cost of $P$ and $P'$, we now consider the cost of an optimal clairvoyant offline ordering algorithm, denoted $\text{Opt}$, that has 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 $\phi_O$ 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 $T_j = ((v_j, t_j, \tilde{R}(j), \tau_j))$, the algorithm $\text{Opt}$ already knows the succeeding transaction $T_k = ((v_k, t_k, \tilde{R}(k), \tau_k))$. When the object is available at $v_j$, the algorithm can immediately send the object to $v_k$. Hence, we define the transaction $T_j$'s completion time in the order $\phi_O$ as $t^O_{j\phi}$. We therefore define the moving cost $c_O(T_j, T_k)$ of ordering $T_k$ after $T_j$ in the $\phi_O$ order as:
\[
  c_O(T_j, T_k) := d(v_j, v_k) + \max\{0, t^O_{j\phi} - t_k + d(v_j, v_k)\} + \tau_k
\]
\[
  \geq d(v_j, v_k) + \max\{0, t_j - t_k + d(v_j, v_k)\} + \tau_k.
\]

The total cost of an optimal algorithm with respect to $R_t$ therefore becomes:
\[
  \text{cost}_{\text{Opt}} = \min_{\phi} \left\{ \sum_{j=1}^{N} c_O(T_{\phi_{(j-1)}}, T_{\phi_{(j)}}) \right\}
\]

Hence, $\phi_O$ is an order which minimizes the sum of Equation 4. Now, we can define the **competitive ratio** of $P$ and $P'$:

**Definition 5 (Competitive Ratio):** $\rho_P = \frac{M}{\text{cost}_{\text{Opt}}}$, $\rho_{P'} = \frac{M'}{\text{cost}_{\text{Opt}}}$

### 4.2. Order Analysis

We now focus on the orders in $Q$ and $T'$ produced by $P$ and $P'$, respectively. Motivated by the method in [10], we can prove that the orders produced by $P$ and $P'$ correspond to two nearest neighbor traveling salesman paths (TSPs) by defining two new comparable cost metrics.

**Definition 6:** $\delta^Q(Q_i, Q_j) := t_{Q_i} - t_{Q_j} + d^c(Q_i, Q_j)$, where $t_{Q_j}$ is the time that $Q_j$ requests to join the queue.

**Definition 7:**
\[
  c_T(T(i), T(j)) := \begin{cases} 
    t(j) - t(i) + Z_{\text{pre}}^p(T(i), T(j)), & T_j \prec T_i \\
    t(j) - t(i) + Z_{\text{suc}}^p(T(i), T(j)), & T_i \prec T_j 
  \end{cases}
\]

We have the following theorem.

**Theorem 6:** The orders of $Q$ and $T'$ are defined by the two nearest neighbor TSPs on metrics $\delta^Q(Q_i, Q_j)$ and $c_T(T(i), T(j))$ starting with $Q_0$ and $T(0)$, respectively. Further, $\delta^Q(Q_i, Q_j) \geq 0$ for all pairs of $Q_i$ and $Q_j$, $c_T(T(i), T(j)) \geq 0$ for all pairs of $T(i)$ and $T(j)$.

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

We now focus on the order of \( Q \). Assume that \( Q_{k'} \) is the transaction that minimizes \( c_Q(Q_{k'-1}, Q_l) \) for all \( Q_l \in Q \setminus \{Q_0, Q_1, \ldots, Q_{k-1}\} \). From the definition of \( Q \), we know that \( Q_{k'+1} \) will receive the object from \( Q_{k'} \). Note that at time \( t_{Q_{k'}} + \delta^C(Q_{k'}, Q_{k'-1}) \), the object is moved from \( Q_{k'-1} \) to \( Q_{k'} \). Hence, the transaction that minimizes \( c_Q(Q_{k'}, Q_{l'}) \) for all \( Q_{l'} \in Q \setminus \{Q_0, Q_1, \ldots, Q_k\} \) is \( Q_{k'+1} \), which is the first transaction that was ordered after \( Q_{k'} \).

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

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

For the order of \( T' \), similar induction steps hold. The theorem follows.

Lemma 7: \( C_Q \geq \frac{1}{2} \sum_{k=1}^{L' \forall} c_Q(Q_{k-1}, Q_k), C_T \geq \frac{1}{2} \sum_{k=1}^{L'} N_i(T(k-1), T(k)) \)

In the following theorem, we give the upper bounds of \( c_Q(Q_i, Q_j) \) and \( c_T(T(i), T(j)) \).

**Theorem 8:**

\[
c_Q(Q_i, Q_j) \leq D_\delta + D_\xi + \max_{i=1}^{N} \tau_i
\]

and

\[
c_T(T(i), T(j)) \leq 2D_\delta + D_\xi + \max_{i=1}^{N} \tau_i,
\]

where

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

and \( D_\xi = \max_{Q_i, Q_j \in Q} \zeta^P(Q_i, Q_j) = \max_{T(i), T(j) \in T'} \zeta^P(T(i), T(j)) \).

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

Let \( Q_k \) and \( Q_{k+1} \) be two consecutive transactions in the order of \( Q \). Let \( \epsilon := c_Q(Q_k, Q_{k+1}) - \delta^C(Q_{k-1}, Q_k) + \epsilon \). If \( \epsilon > 0 \), for all transactions \( Q_l \), where \( l \geq k+1 \), \( t_{Q_{k+1}} \) can be replaced by \( t_{Q_{k+1}} - \epsilon \) without increasing the costs of \( C \) and OPT. By applying this method as many times as possible, we have Equation 5. The same argument holds for the order of \( T' \) and we have Equation 6. The theorem follows.

### 4.3. Comparison

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

**Definition 8 (Manhattan Metric):** The Manhattan metric \( c_M(T_j, T_k) \) is defined as:

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

**Lemma 9:** Let \( \phi \) be an ordering, and \( C_O \) and \( C_M \) be the costs for ordering all transactions in order \( \phi \) with respect to \( c_O \) and \( c_M \). The Manhattan cost is bounded by: \( C_M \leq 2C_O + t_{\phi(N)} \).

**Proof:** We can lower bound the optimal cost of \( c_O \) by:

\[
c_O(T_j, T_k) \geq d(v_j, v_k) + \max\{0, t_j - t_k\} + \tau_k
\]
Let $D_O = \sum_{j=1}^{N} \{d(v_{\phi(j-1)}, v_{\phi(j)}) + \tau_j + \tau_{j-1}\}$. Then we have:

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

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

**Theorem 10:**

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

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

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

From Theorem 10, we know that the competitive ratio is determined by the maximum $\tau_j$. We have the following corollary for a possible range of the value of the maximum $\tau_j$.

**Corollary 11:**

$$\rho_P = O(N \log D_\delta), \quad \rho_{P'} = O(\log D_\delta)$$

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

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

**5. Conclusion**

In this paper, we formalize two classes of cache-coherence protocols for distributed TM systems, and compare their performance in terms of competitive ratio. We compare DQC and DPQC protocols with the optimal offline algorithm OPT. In practice, it is often hard to describe the algorithm OPT explicitly. Hence, we adopt an analytical method similar to the methods used in [1], [5] and [10], where the optimal algorithm is implicitly described by its cost. We believe that this is an efficient and sufficient way to evaluate the worst-case performance of a cache-coherence protocol.

We conclude that based on the same distributed queuing protocol, a DPQC protocol guarantees a much better performance bound than a DQC protocol. Further, if the maximum local execution time is sufficiently small (up to the logarithmic order of $D_\delta$), a DQC protocol is $O(N \log D_\delta)$-competitive and a DPQC protocol is $O(\log D_\delta)$-competitive. This result can be explained in the following way. For a system in which the network latency is the significant part of the communication cost, the selection of cache-coherence protocols determines the overall performance, since it determines the total cost that the object travels in the network. On the other hand, for a system in which the local execution time is relatively large, the total execution cost of transactions will be the dominating part of the total time complexity. In this case, the distributed TM system is more similar to a TM system on multiprocessors, where the underlying contention manager determines the maximum abortion times of each transaction.

There are several directions for future work. One possible direction is to design a cache-coherence protocol with desired fault-tolerance properties. We assume a bounded communication cost between nodes and evaluate the worst-case performance in this paper. Studying the average-case performance of cache-coherence protocols in a network with stochastic behavior of message loss and delay will be an interesting future direction.
References


Appendix: Proof of Theorem 10

We use the following lemma from [10]:

**Lemma 12:** Let \( c_M'(T_j, T_k) := d(v_j, v_k) + |t_j - t_k| \) and \( C_M' \) be the cost of ordering all requests in order \( \phi \) with respect to \( c_M' \). Then, \( C_M' \geq \frac{3}{2} t_N \) where \( t_N = \max_{i=1}^{N} t_i \).

Hence, we have the following theorem to make \( C_M \) comparable to \( C_O \):

**Theorem 13:** \( C_M \leq 6C_O \)

**Proof:** The theorem can be proved by Lemmas 9 and 12. Note that we have \( c_M \geq c_M' \) and \( t_N \geq t_{\phi(N)} \). Then the theorem follows.

We now compare \( C_M, C_Q, \) and \( C_T \) with the help of the following lemma from [10]:

**Lemma 14:** Let \( V \) be a set of \( N := |V| \). Let \( d_n : V \times V \rightarrow \mathbb{R} \) and \( d_o : V \times V \rightarrow \mathbb{R} \) be the distance functions between nodes of \( V \). For \( d_n \) and \( d_o \), the following conditions hold:

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

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

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

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

Now we can compare \( C_Q, C_T, \) and \( C_M \) based on Lemma 14.

**Theorem 15:**

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

**Proof:** This theorem follows from Theorem 8 and Lemma 14. Note that \( c_Q \) and \( c_T \) comply with the condition for \( d_o(u, v) \), and \( c_M \) complies with the condition for \( d_n(u, v) \). And the triangle inequality holds for \( c_M \). Finally, we can bound the shortest value of \( c_T \) by \( \min_{v_j, v_k \in V} d(v_j, v_k) \). The theorem follows.

Now we can prove Theorem 10. We have

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

where the first inequality follows from Theorem 3, the second equality follows from Equation 3, and the third inequality follows from Lemma 7. On the other hand, \( \text{cost}_{\text{opt}} \geq H + \sum_{j=1}^{N} \tau_j \). Then Equation 7 holds. We can prove Equation 8 in the same way. The theorem follows.