Brief Announcement: Queuing or 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

In distributed transactional memory (TM) systems, both the management and consistency of a distributed transactional object are ensured by a cache-coherence protocol. 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, where \( D_\delta \) is the normalized maximum communication latency provided by the underlying distributed queuing protocol.

1. Distributed TM

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. Distributed 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 TM as a possible solution.

We consider Herlihy and Sun’s data-flow distributed TM model [1]. In this model, transactions are immobile (running at nodes which invoke them), but objects move from node to node. Transactional synchronization is optimistic: a transaction commits only if no other transaction has executed a conflicting access. In the data-flow model, each node has a TM proxy that provides interfaces to the TM application and to proxies of other nodes. When a transaction at node \( A \) requests a read/write access to an object \( R_i \), its TM proxy first checks whether \( R_i \) is in its local cache; if not, the TM proxy invokes a distributed cache-coherence protocol to fetch \( R_i \) in the network. The node \( B \), which holds \( R_i \), checks whether it is in use by an active local transaction when it receives the request for \( R_i \) from node \( A \). If not, the TM proxy of node \( B \) sends \( R_i \) to node \( A \) and invalidates its own copy. If so, the proxy invokes the contention manager module to mediate conflicting access requests for \( R_i \).

Distributed transactional contention on an object, which is now distributed in the data-flow TM model, can be managed by ordering transactional requests on the object using a distributed queue. Thus, distributed cache coherence protocols can be designed based on distributed queuing. Since a transaction accesses a set of (distributed) objects, transactional contention on the corresponding set of distributed
queues must be managed to ensure transactional atomicity. We consider the design of two classes of cache-coherence protocols: distributed queuing cache-coherence (DQC) protocols and distributed priority queuing cache-coherence (DPQC) protocols. We assume a fixed contention manager, which satisfies the work conserving [2] and pending commit [3] properties.

2. DQC and DPQC Protocols

A DQC protocol uses a distributed queue to manage transactional contention on a distributed object. For example, the Ballistic protocol [1] and the class of LAC protocols [4] are DQC protocols. We formalize the behavior of DQC protocols as follows. Each transaction is considered as a sequence of ordering requests. 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 may join the queue again and therefore the length of the queue is dynamically increased. DQC protocols work in the following way:

1. For each object, a distributed queue is formed by transactions which request read/write accesses to that object. Formally, for each object $R_i$, a distributed queue $Q_i$ is established. A transaction $T_j$ requests to join $Q_i$ if and only if it requires read/write accesses to $R_i$.

2. Enqueue operation: a transaction joins a queue by sending a request to the current tail of the queue, and becomes the new tail of the queue. To implement this operation, for each object, the system has to maintain and update a directory which always points to the latest tail of the queue.

3. Dequeue operation: a transaction can only leave a distributed queue after it becomes the head of that queue. The object is always held by the head of the queue. If the head $T_H$ of a queue commits, it leaves that queue after sending the object to its successor in the queue. If it is aborted (by its successor in the queue), transaction $T_H$ is restarted immediately. In this case, the transaction may request to join the queue again.

4. A transaction joins a sequence of distributed queues according to the same order of the sequence of objects that it requests. For example, assume a transaction $T_j$ with a sequence of operations \{write($R_1$), write($R_2$), read($R_3$)\}. The transaction joins the queue $Q_1$ first. After its write operation to object $R_1$, the transaction joins the queue $Q_2$ to request object $R_2$. In the same way, the transaction joins the queue $Q_3$ after its write operation to object $R_2$. Hence, a transaction may participate in multiple queues at the same time. If, at any time during the aforementioned steps, a transaction is aborted by its successor in a queue, the transaction is dequeued from all participating distributed queues and passes each object to its successor (if any) of the corresponding queues.

DPQC protocols use distributed priority queuing for managing transactional contention. DPQC protocols work in the following way:

1. Similar to DQC protocols, for each object, a distributed priority queue is formed by transactions which request read/write accesses to that object.

2. Enqueue operation: for DPQC protocols, each transaction is only enqueued once in each distributed queue it requests. A transaction is inserted into a queue such that the priority order of the queue is not violated, i.e., each element’s priority is always higher than its successor’s. Unlike the enqueue operation of a DQC protocol, it is not required to append a transaction at the tail of a queue. Instead, a transaction is inserted in a queue after the queue learns of its priority.

3. Dequeue operation: a transaction can only leave a distributed queue after it commits. If the head transaction $T_H'$ of a queue commits, it leaves that queue after sending the object to its successor in the queue. If it is aborted by a higher priority transaction $T_H''$, $T_H$ is restarted immediately and $T_H''$ becomes the new head of the queue. In this case, $T_H'$ becomes the successor of $T_H''$. 
3. Comparison

We evaluate a cache-coherence protocol $C$ by measuring its competitive ratio $\rho$, which is the ratio of the time complexity to commit a set of $N$ dynamically generated transactions under $C$ to the time complexity to commit the same set of transactions under an optimal clairvoyant algorithm $OPT$. Motivated by the techniques in [5], which conducts the dynamic analysis of the Arrow distributed protocol, we prove the following theorem in [6]:

**Theorem 1:**

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

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

where $H$ is the total cost of the traveling salesman path with respect to the network cost metric $d(v_j, v_k)$, $\tau_j$ is the local execution duration of $T_j$, $\sigma_j$ and $\mu_j$ are the number of aborts of $T_j$ under DQC and DPQC protocols, respectively, and $D_\delta$ is the maximum communication latency provided by the underlying distributed queuing protocol. We have the following corollary for a range of the value of $\max_{j=1}^N \tau_j$.

**Corollary 2:**

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

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

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 a cache-coherence protocol determines the overall performance, since it determines the total cost for the object to travel 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, a distributed TM system is more similar to a multiprocessor TM system, where the underlying contention manager determines the maximum abort times of each transaction.

**References**


