A Quorum-Based Replication Framework for Distributed Software Transactional Memory

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

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

Abstract

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

1. Introduction

Lock-based synchronization is non-scalable, non-composable, and inherently error-prone. Transactional memory (TM) is an alternative synchronization model for shared memory objects that promises to alleviate these difficulties. In addition to a simple programming model, TM provides performance comparable to highly concurrent fine-grained locking and is composable. TM for multiprocessors has been proposed in hardware, called HTM, in software, called STM, and in hardware/software combination [1].

Similar to multiprocessor TM, distributed STM (or D-STM) is motivated by the difficulties of lock-based distributed synchronization (e.g., distributed race conditions, composability). D-STM can be supported in any of the classical distributed execution models, including a) dataflow [2], where transactions are immobile, and objects are migrated to invoking transactions; b) control flow [3], where objects are immobile and transactions invoke object operations through RPCs; and c) hybrid models (e.g., [4]), where transactions or objects are migrated, based on access profiles, object size, or locality. The different models have their concomitant tradeoffs.

D-STM can be classified based on the system architecture: cache-coherent D-STM (cc D-STM) [2], where a number of nodes are interconnected using message-passing links, and a cluster model (cluster D-STM), where a group of linked computers works closely together to form a single computer ([4], [5], [6], [7]). The most important difference between the two is communication cost. cc D-STM assumes a

metric-space network, whereas cluster D-STM differentiates between local cluster memory and remote memory at other clusters.

In this paper, we focus on cc D-STM. The data-flow cc D-STM model is proposed by Herlihy and Sun [2]. In this model, only a single (writable) copy is kept in the system. Transactions run locally and objects move in the network to meet transactions' requests. When a node v_A initiates a transaction A that requests a read/write operation on object o, its TM proxy first checks whether o is in the local cache; if not, the TM proxy invokes a *cache-coherence* (CC) protocol to locate o in the network by sending a request $CC.locate(o)$. Assume that o is in use by a transaction B initiated by node v_B. When v_B receives the request $CC.locate(o)$ from v_A, its TM proxy checks whether o is in use by an active local transaction; if so, the TM proxy invokes a contention manager to handle the conflict between A and B. Based on the result of contention management, v_B's TM proxy decides whether to abort B immediately, or postpone A's request and let B proceed to commit. Eventually, CC moves o to v_A.

In the aforementioned single copy data-flow model (or SC model), the main responsibility of CC protocol is to locate and move objects in the network. A directory-based CC protocol is often adopted such that the latest location of the object is saved in the distributed directory and the cost to locate and move an object is bounded. Such CC protocols include Ballistic [2], Relay [8] and Combine [9].

Since SC model only keeps a single writable copy of each object, it is inherently vulnerable in the presence of node and link failures. If a node failure occurs, the objects held by the failed node will be simply lost and all following transactions requesting such objects would never commit. Hence, SC model cannot afford any node failures. Ballistic and Relay also assumes a reliable and fifo logical link between nodes, since they may not perform well when the message is reordered [10]. On the other hand, Combine can tolerate partial link failures and support non-fifo message delivery, as long as a logical link exists between any pair of nodes. However, similar to other directory-based CC protocols, Combine does not permit network partitioning incurred by link failures, which may make some objects inaccessible from outer transactions. In general, SC model is not suitable in a network environment with aforementioned node/link failures.

To achieve high availability in the presence of network failures, keeping only one copy of each object in the system is not sufficient. Inherited from database systems, replication is a promising approach to build fault-tolerant D-STM systems, where each object has multiple (writable) copies. However, only a few replicated D-STM solutions have been proposed for cluster-based D-STM ([4], [5], [6], [7]). These solutions require some form of broadcasting to maintain consistency among replicas and assume a uniform communication cost across all pairs of nodes. As the result, we cannot directly apply these solutions for cc D-STM.

This paper presents QR model, a quorum-based replication cc D-STM model which provides provable fault-tolerance property in a failure-prone metric-space network, where communication cost between nodes forms a metric. To the best of our knowledge, this is the first replication cc D-STM proposal which provides provable fault-tolerant properties. In distributed systems, a quorum is a set of nodes such that the intersection of any two quorums is non-empty if one of them is a write quorum. By storing replicated copies of each object in an overlay tree quorum system motivated by the one in [11], QR model supports concurrent reads of transactions, and ensures the consistency among replicated copies at commit-time. Meanwhile, QR model exhibits a bounded communication cost of its operations, which is proportional to the communication cost from v to its closest read/write quorum, for any operation starting from node v. Compared with directory-based CC protocols, the communication cost of operations in QR model does not rely on the stretch of the underlying overlay tree (i.e., the worst-case ratio between the cost of direct communication between two nodes v and w and the cost of communication along the shortest tree path between v and w). Therefore QR model provides a more promising solution to support D-STM in the

presence of network failures with communication cost comparable with SC model.

The rest of the paper is organized as follows. We introduce the system model and identify the limitations of SC model in Section 2. We present QR model and analyze its properties in Section 3. The paper concludes in Section 4.

2. Preliminaries

2.1. System Model

We consider a distributed system which consists of a set of distinct nodes that communicate with each other by message-passing links over a communication network. Similar to [2], we assume that the network contains n physical nodes scattered in a metric space of diameter D. The metric $d(u, v)$ is the distance between nodes u and v, which determines the communication cost of sending a message from u to v. Scale the metric so that 1 is the smallest distance between any two nodes.

We assume that nodes are *fail-stop* [12] and communication links may also fail to deliver messages. Further, node and link failures may occur concurrently and lead to network partitioning failures, where nodes in a partition may communicate with each other, but no communication can occur between nodes in different partitions. A node may become inaccessible due to node or partitioning failures.

We consider a set of *distributed transactions* $\mathcal{T} := \{T_1, T_2, \ldots\}$ sharing a set of objects $\mathcal{O} := \{o_1, o_2, \ldots\}$ distributed on the network. A transaction contains a sequence of requests, each of which is a read or write operation request to an individual object.

An execution of a transaction is a sequence of timed operations. An execution ends by either a commit (success) or an abort (failure). A transaction's status is one of the following three: *live*, *aborted*, or *committed*. A transaction is live after its first operation, and completes either by a commit or an abort operation. When a transaction aborts, it is restarted from its beginning immediately and may access a different set of shared objects. Two transactions are *concurrent* if they are both live at the same time. Suppose there are two live transactions T_j and T_k which request to access o_i and at least one of the access is a write. Then T_j and T_k are said to *conflict* at o_i, i.e., two live transactions conflict if they both access the same object and at least one of the accesses is a write. There are three types of conflicts: (1) Read-After-Write ($W \rightarrow R$); (2) Write-After-Read ($R \rightarrow W$); and (3) Write-After-Write ($W \rightarrow W$). A *contention manager* is responsible for resolving the conflict, and does so by aborting or delaying (i.e., postponing) one of the conflicting transactions. Most contention managers do not allow two transactions to proceed (i.e., make progress) simultaneously. In other words, two operations from different transactions over the same object cannot be overlapped if one of them is a write. In this paper, we assume an underlying contention manager which has consistent policies to assign priorities to transactions. For example, the *Greedy* contention manager [13] always assigns higher priority to the transaction earlier timestamp.

Each node has a *TM proxy* that provides interfaces to the TM application and to proxies of other nodes. A transaction performs a read/write operation by first sending a read/write access request to its TM proxy. The TM proxy invokes a CC protocol to acquire a valid object copy in the network. For a read operation, the protocol returns a read-only copy of the object. For a write operation, the CC protocol returns a writable copy of the object. When there are multiple copies (or replicas) of an object existing in the network, the CC protocol is responsible to ensure the consistency over replicas such that multiple copies of an object must appear as a single logical object to the transactions, which is termed as one-copy equivalence [14].

2.2. Motivation: limitations of SC D-STM model

As mentioned in Section 1, SC model lacks the fault-tolerant property in the presence of network failures. SC model also suffers from some other limitations.

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

Due to the inherent difficulties in supporting concurrent read operations, practical implementations of directory-based CC protocols often do not differentiate between a read and write operation of a read/write transaction (i.e., if a transaction contains both read and write operations, all its operations are treated as write operations and all its requested objects have to be moved). Such over-generalization obviously limits the possible concurrency of transactions. For example, in the scenario where the workload is composed of late-write transactions [10], a directory-based CC protocol cannot perform better than a simple serialization schedule, while the optimal schedule maybe much shorter when concurrent reads are supported for read/write operations.

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

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

more locality: as long as t_o is less than $2D-1$ (which is a quite loose bound), the object is moved more quickly.

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

3. Quorum-Based Replication Data-Flow D-STM Model

3.1. Overview

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

Definition 1 (Quorum Intersection Property): A quorum is a collection of nodes. For any two quorums q_1 and q_2, where at least one of them is a write quorum, the two quorums must have a non-empty intersection: $q_1 \cap q_2 \neq \emptyset$.

Generally, by constructing a quorum system over the network, QR model is able to keep multiple copies of each object. QR model provides 5 operations for a transaction: read, write, request-commit, commit and abort. Particularly, QR model provides a request-commit operation to validate the consistency of its readset and writeset before it commits. A transaction may request to commit if it is not aborted by other transactions before its last read/write operation. Concurrency control solely occurs during the request-commit operation: if a conflict is detected, the transaction may get aborted or abort the conflicting transaction. After collecting the response of the request-commit operation, a transaction may commit or abort.

We first present read and write operations of QR model in Algorithm 1.

Algorithm 1: QR model: read and write

```
1  procedure READ (v, T, o)
2  Local Phase:
3  READQUORUM (v, req(T, read(o)));
4  wait until find(v) = true;
5  foreach returned value d from node v_i do
6      if d.version > data(o).version then
7          data(o) ← d;
8  add o to T.readset;
9  Remote Phase:
10 Upon receiving req(T, read(o)) from node v;
11 if data(o) exists then
12     add T to PR(o);
13     send rsp(T, o) to v;
14 procedure WRITE (v, T, o, value)
15 Local Phase:
16 READQUORUM (v, req(T, write(o)));
17 wait until find(v) = true;
18 foreach returned value d from node v_i do
19     if d.version > data(o).version then
20         data(o) ← d;
21 dataCopy(o) ← data(o);
22 dataCopy(o).value ← value;
23 dataCopy(o).version ← data(o).version + 1;
24 add o to T.writeset;
25 Remote Phase:
26 Upon receiving req(T, write(o)) from node v;
27 if data(o) exists then
28     add T to PW(o);
29     send rsp(T, o) to v;
```

Read. When transaction T at node v starts a read operation, it sends a request message $req(T, read(o))$ to a selected read quorum q_r. The algorithm to find and select a read or write quorum will be elaborated in the next section. Node v', upon receiving $req(T, read(o))$, checks whether it has a copy of o. If not, it sends a null response to v.

In QR model, each object copy contain three fields: the value field, which is the value of the object; the version number field, starting from 0, and the $protected$ field, a boolean value which records the status of the copy. The $protected$ field is maintained and updated by request-commit, commit and abort

operations. Each object copy o keeps a *potential readers list* $PR(o)$, which records the identities of the potential readers of o. Therefore, if v' has a copy of o, it adds T to $PR(o)$ and sends a response message $rsp(T, o)$ to v, which contains a copy of o.

Transaction T waits to collect responses until it receives all responses from a read quorum. Among all copies it receives, it selects the copy with the highest version number as the valid copy of o. The read operation finishes.

Write. The write operation is similar to the read operation. Transaction T sends a request message $req(T, write(o))$ to a selected read quorum. Note that T does not need to send request to a write quorum because in this step it only needs to collect the latest copy of o. If a remote node v' has a copy of o, it adds T to o's *potential writers list* $PW(o)$ and sends a response message to T with a copy of o.

Transaction T selects the copy with with the highest version number among the responses from a read quorum. Then it creates a temporary local copy ($dataCopy(o)$) and updates it with the value it intends to write, and increases its version number by 1 compared with the selected copy.

Remarks: The read and write operations of QR model are simple: a transaction just has to fetch all latest copies of required objects and perform all computations locally. Unlike a directory-based CC protocol, there is no need to construct and update a directory for each shared object. In QR model a transaction can always query its "closest" read quorum to locate the latest copy of each object required. Therefore the locality is preserved.

Algorithm 2: QR model: request-commit

```
1  procedure REQUEST-COMMIT (v, T)
2  Local Phase:
3  WRITEQUORUM (v, req_cmt(T));
4  AT(T) ← ∅;
5  wait until find(v) = true;
6  if any rsp_cmt(T, abort) message is received then
7      ABORT (v, T);
8  else
9      foreach rsp_cmt(T, commit, CT(T)) message do
10         AT(T) ← AT(T) ∪ CT(T);
11     COMMIT (v, T);
12 Remote Phase:
13 Upon receiving req_cmt(T) from node v;
14 CT(T) ← ∅;
15 abort(T) ← false;
16 CONFLICT-DETECT (v, T);
17 if abort(T) = false then
18     if CT(T) = ∅ then
19         send rsp_cmt(T, commit, CT(T)) to v;
20     else
21         CONTENTION-MANAGEMENT (T, CT(T));
22         if CT(T) ≠ ∅ then
23             send rsp_cmt(T, commit, CT(T)) to v;
24 if abort(T) = false then
25     foreach o_T ∈ T.writeset for object o do
26         o_T.protected ← true;
27 remove T from PR(o) and PW(o) for any object o;

28 procedure CONFLICT-DETECT (v, T)
29 foreach o_T ∈ T.readset ∪ T.writeset of object o do
30     if data(o).protected = true or
31     data(o).version > o_T.version then
32         abort(T) ← true;
33         send rsp_cmt(T, abort) to v;
34         break;
35     if data(o).version = o_T.version then
36         if data(o).value ≠ o_T.value then
37             abort(T) ← true;
38             send rsp_cmt(T, abort) to v;
39             break;
40         else
41             add PW(o) to CT(T);
42             if o_T ∈ T.writeset then
43                 add PR(o) to CT(T);

44 procedure CONTENTION-MANAGEMENT (T, CT(T))
45 foreach T' ∈ CT(T) do
46     if T' ≺ T then
47         abort(T) ← true;
48         send rsp_cmt(T, abort) to v;
49         CT(T) ← ∅;
50         break;
```

If a transaction is not aborted (by any other transaction) during all its read and operations, the transaction can request to commit by requesting to propagate its changes to objects into the system. The concurrency control mechanism is needed when any non-consistent status of an object is detected. The request-commit operation is presented in Algorithm 2.

Request-Commit. When transaction T requests to commit, it sends a message $req\_cmt(T)$ (which contains all information of its readset and writeset) to a write quorum q_w. Note that it is required that for each transaction T, and $\forall q_r, q_w$ selected by T, $q_r \subseteq q_w$.

In the remote phase, when node v' receives the message $req\_cmt(T)$, it immediately removes T from its potential read and write lists of all objects and creates an empty *conflicting transactions list* $CT(T)$ which records the transactions conflicting with T. Node v' determines the conflicting transactions of T in the following manner:
1) if $o_T.protected = true$, then T must be aborted since o_T is waiting for a possible update;
2) if o_T is a copy read or written by T of object o, and the local copy of o at v' ($data(o)$) has the higher version than o_T, then T reads a stale version of o. In this case, T must be aborted.
3) if o_T is a copy read by T of object o, and the local copy of o at v' ($data(o)$) has the same version with o_T, then T conflicts with all transactions in $PW(o)$ (potential writers of object copy $data(o)$).
4) if o_T is a copy written by T of object o, and the local copy of o at v' ($data(o)$ has the same version with o_T, then T conflicts with all transactions in $PW(o) \cup PR(o)$ (potential readers and writers of $data(o)$.

The contention manager at v' compares priorities between T and its conflicting transactions. If $\forall T' \in CT(T)$, $T \prec T'$ (T has the higher priority than any of its conflicting transactions), T is allowed to commit by v'. Node v' sends a message $rsp\_cmt(T, commit)$ with $CT(T)$ to v and sets the status of $data(o)$ as $protected$, for any $o \in T.writeset$. If $\exists T' \in CT(T)$ such that $T' \prec T$, then T is aborted. Node v' sends a message $rsp\_cmt(T, abort)$ to v and resets $CT(T)$.

In the local phase, transaction T collects responses from all nodes in the write quorum. If any $rsp\_cmt(T, abort)$ message is received, T is aborted. If not, T can proceed to the commit operation. In this case, transaction T saves conflicting transactions from all responses into an aborted transactions list $AT(T)$.

Algorithm 3: QR model: commit and abort

```
1  procedure COMMIT (v, T)
2  Local Phase:
3  foreach object o ∈ T.writeset do
4      data(o) ← dataCopy(o);
5  foreach T' ∈ AT(T) do
6      send req_abt(T') message;
7  WRITEQUORUM (v, commit(T));
8  wait until find(v) = true;
9  Remote Phase:
10 Upon receiving commit(T):
11 foreach o_T ∈ T.writeset for object o do
12     data(o) ← o_T;
13     data(o).protected ← false;
14 Upon receiving req_abt(T'):
15 ABORT (v', T')
16 procedure ABORT (v, T)
17 Local Phase:
18 foreach object o ∈ T.writeset do
19     discard dataCopy(o);
20 WRITEQUORUM (v, abort(T));
21 wait until find(v) = true;
22 Remote Phase:
23 Upon receiving abort(T):
24 foreach o_T ∈ T.writeset for object o do
25     data(o).protected ← false;
26 remove T from PR(o) and PW(o) for any object o;
```

Remarks: For each transaction T, its concurrency control mechanism is carried by the request-commit operation. Therefore, the request-commit operation must guarantee that all existing conflicts with T are detected. Note that a remote node makes this decision based on its potential read and write lists. Therefore, these lists must be efficiently updated: a terminated transaction must be removed from these lists to avoid an unnecessary conflict detected. By letting $q_r \subseteq q_w$ for all q_r and q_w selected by the same transaction T, QR model guarantees that all T's records in potential read and write lists are removed during T's request-commit operation.

On the other hand, if v' allows T proceed to commit, then v' needs to protect local object copies written by T from other accesses until T's changes to these objects propagate to v'. These objects copies

become valid only after receiving T's commit or abort information. We describe T's commit and abort operations in Algorithm 3.

Commit. When T commits, it sends a message $commit(T)$ to each node in the same write quorum q_w as the one selected by the request-commit operation. Meanwhile, it sends a request-abort message $req\_abt(T')$ for any $T' \in AT(T)$. In the remote phase, when a node v' receives $commit(T)$, for any $o \in T.writeset$, it updates $data(o)$ with the new value and version number, and sets $data(o).protected = false$. If a transaction T' receives $req\_abt(T')$, it aborts immediately.

Abort. A transaction may abort in two cases: after the request-commit operation, or receives a request-abort message. When T aborts, it rolls back all its operations of local objects. Meanwhile, it sends a message $abort(T)$ to each node in the write quorum q_w (which is the same as the write quorum selected by the request-commit operation). Then transaction T restarts from the beginning. In the remote phase, when a node v' receives $abort(T)$, it removes T from any of its potential read and write list (if it has not done so), and sets $data(o).protected = false$ for any $o \in T.writeset$.

3.2. Quorum construction: FLOODING protocol

One crucial part of QR model is the construction of a quorum system over the network. We adopt the hierarchical clustering structure similar to the one described in [2]. An overlay tree with depth L is constructed. Initially, all physical nodes are leaves of the tree. Starting from the leaf nodes at level $l = 0$, parent nodes at the immediate higher level $l + 1$ is elected recursively so that their children are all nodes at most at distance 2^l from them.

Our quorum system is motivated by the classic *tree quorum system* [11]. On the overlay tree, a quorum system is constructed by FLOODING protocol such that each constructed quorum is a valid tree quorum.

We present FLOODING protocol in Algorithm 4. For each node v, when the system starts, a *basic read quorum* $Q_r(v)$ and a *basic write quorum* $Q_w(v)$ are constructed by BASICQUORUMS method. The protocol tries to construct $Q_r(v)$ and $Q_w(v)$ by first putting $root$ into these quorums and setting a distance variable δ to $d(v, root)$. Starting from $level = L-1$, the protocol recursively selects the majority of descendants $levelHead = closestMajority(v, parent, level)$ for each $parent$ selected in the previous level ($level + 1$), so that the distance from v to $closestMajority(v, parent, level)$ is the minimum over all possible choices. Note that $closestMajority(v, parent, level)$ only contains $parent$'s descendants at level $level$. We define the distance from v to a quorum Q as: $d(v, Q) := \max_{v' \in Q} d(v, v')$. The basic write quorum $Q_w(v)$ is constructed by including all selected nodes.

At each $level$, after a set of nodes $levelHead$ has been selected, the protocol checks the distance from v to $levelHead$ ($d(v, levelHead)$). If $d(v, levelHead) < \delta$, then the protocol replaces $Q_r(v)$ with $levelHead$ and sets δ to $d(v, levelHead)$. If $d(v, levelHead) \geq \delta$, the protocol continues to the next level. At the end, $Q_r(v)$ contains a set of nodes from the same level, which is the $levelHead$ closest from v for all levels.

When node v requests to access a read quorum, the protocol invokes READQUORUM(v, msg) method. Initially, node v sends msg to every node in $Q_r(v)$. If all nodes in $Q_r(v)$ are accessible from v, then a live read quorum is found. If any node v' in $Q_r(v)$ is down, then the protocol needs to probe v''s substituting nodes $sub(v')$ such that $sub(v') \cup Q_r(v) \backslash v'$ still forms a read quorum.

The protocol first finds if there exists any v''s ancestor available. If so, v''s substituting node has been found. If not, the protocol probes downwards from v' to check if there exists v' substituting nodes such that a constructed read quorum is a subset of $Q_w(v)$ by calling DOWNPROBE method.

The protocol invokes WRITEQUORUM(v, msg) method when node v requests to access a write quorum. Similar to READQUORUM(v, msg), node v first sends msg to every node in $Q_w(v)$. If any node v' is down, then the protocol first finds if there is a live ancestor of v' ($validAns(v)$). Starting from $validAns(v)$, the protocol calls DOWNPROBE to probe downwards.

Algorithm 4: FLOODING protocol

```
1  procedure BASICQUORUMS (v, root)
2  δ ← d(v, root);
3  Q_r(v) ← {root};
4  Q_w(v) ← {root};
5  Q_r(v).level ← L;
6  currentHead ← {root};
7  for level = L − 1, L − 2, ..., 0 do
8      levelHead ← ∅;
9      foreach parent ∈ currentHead do
10         new ← closestMajority(v, parent, level);
11         add new to Q_w(v);
12         add new to levelHead;
13     if d(v, levelHead) < δ then
14         Q_r(v) ← levelHead;
15         Q_r(v).level ← level;
16         δ ← d(v, levelHead);
17     currentHead ← levelHead;

18 procedure WRITEQUORUM (v, msg)
19 send msg to every node in Q_w(v);
20 if v' ∈ Q_r(v) is down then
21     find(v) ← false;
22     validAns(v) ← null;
23     validLevel(v) ← null;
24     for level = 1, ..., L do
25         send msg to ancestor(v, level);
26         if ancestor(v, level) is up then
27             validAns(v) ← ancestor(v, level);
28             validLevel(v) ← level;
29             break;
30     if validAns(v) = null then
31         restart WRITEQUORUM (v, msg);
32     if validLevel(v) > Q_r(v).level then
33         send msg to every node in Q_r(v);
34     DOWNPROBE (validAns(v), validLevel(v), write);
35     if find(v) = false then
36         restart WRITEQUORUM (v, msg);

37 procedure READQUORUM (v, msg)
38 send msg to every node in Q_r(v);
39 find(v) ← false;
40 if v' ∈ Q_r(v) is down then
41     find(v) ← false;
42     if v' ≠ root then
43         for level = Q_r(v).level + 1, ..., L do
44             send msg to ancestor(v, level);
45             if ancestor(v, level) is up then
46                 find(v) ← true;
47                 break;
48     if find(v) = false then
49         DOWNPROBE (v', Q_r(v).level, read);
50     if find(v) = false then
51         restart READQUORUM (v, msg);

52 procedure DOWNPROBE (v, validLevel, type)
53 curReadHead ← v;
54 curWriteHead ← v;
55 noWriteQuorum ← false;
56 for level = validLevel − 1, ..., 0 do
57     levelReadHead ← ∅;
58     levelWriteHead ← ∅;
59     foreach parent ∈ curReadHead do
60         send msg to every node in descendants(parent, level) ∩ Q_w(v);
61         if w is down then
62             add w to levelReadHead;
63     if type = write then
64         foreach parent ∈ currentWriteHead do
65             if ∃newSet = closestMajority(v, parent, level) then
66                 send msg to every node in newSet;
67                 add newSet to levelWriteHead;
68             else
69                 noWriteQuorum ← true;
70                 break;
71     if noWriteQuorum = true then
72         break;
73     if levelReadHead = ∅ and type = read then
74         find(v) ← true;
75         break;
76     else
77         curReadHead ← levelReadHead;
78         curWriteHead ← levelWriteHead;
79 if noWriteQuorum = false and type = write then
80     find(v) ← true;
```

DOWNPROBE method works similarly as BASICQUORUMS by recursively probing an available closest majority set of descendants for each parent selected in the previous level. By adopting DOWNPROBE method, FLOODING protocol guarantees that READQUORUM and WRITEQUORUM can always probe an available quorum if at least one live read (or write) quorum exists in the network.

3.3. Analysis

We first analyze the properties of the quorum system constructed by FLOODING, then we prove the correctness and evaluate the performance of QR model.

Lemma 1: Any read quorum q_r or write quorum q_w constructed by FLOODING is a classis tree quorum defined in [11].

Proof: From the description of FLOODING, we know that for a tree of height $h+1$,

$$q_r = \{root\} \vee \{\text{majority of read quorums for subtrees of height h}\},$$

$$q_w = \{root\} \cup \{\text{majority of write quorums for subtrees of height h}\}.$$

From Theorem 1 in [11], the lemma follows. □

Then we immediately have the following lemma.

Lemma 2: For any two quorums q_1 and q_2 constructed by FLOODING, where at least one of them is a write quorum, $q_r \cap q_w \neq \emptyset$.

Lemma 3: For any read quorum $q_r(v)$ and write quorum $q_w(v)$ constructed by FLOODING for node v, $q_r(v) \subseteq q_w(v)$.

Proof: The theorem follows from the description of FLOODING. If no node fails, the theorem holds directly since $Q_r(v) \subseteq Q_w(v)$.

If a node $v' \notin Q_r(v)$ fails, then $q_r(v) = Q_r(v)$. If $v' \in Q_w(v)$, FLOODING detects that v' is not accessible when it calls WRITEQUORUM method. If $level(q) \geq Q_r(v).level$, then FLOODING adds $Q_r(v)$ to $q_w(v)$ and starts to probe v''s substituting nodes; if $level(v') < Q_r(v).level$, then the level of v''s substituting node is at most $Q_r(v).level$ and then the protocol starts to probe downwards. In either case, $q_r(v) \subset q_w(v)$.

If a node $v' \in Q_r(v)$ fails, then FLOODING detects that v' is not accessible when it calls READQUORUM or WRITEQUORUM method. Both methods starts to probe v''s substituting nodes from v'. When probing upwards, v''s ancestors are visited. If a live $ancestor(v')$ is found, then both methods add $ancestor(v')$ to the quorum. Then READQUORUM stops and WRITEQUORUM continues probing downwards from $ancestor(v')$. The theorem follows. □

With the help of Lemmas 2, we have the following theorem.

Theorem 4: QR model provides 1-copy equivalence for all objects.

Proof: We first prove that for any object o, if at time t, no transaction requesting o is propagating its change to o (i.e., in the commit operation), then all transactions accessing o at t get the same copy of o.

Note that if any committed transaction writes to o before t, there exists a write quorum q_w such that $\{\forall v \in q_w\} \wedge \{\forall v' \notin q_w\}$, $data(o,v).version > data(o,v').version$. If any transaction T accesses o at time t, it collects a set of copies from a read quorum q_r. From Lemma 2, $\exists v \in \{q_w \cap q_r\}$ such that $data(o,v)$ is collected by T. Note that read and write operations select the object copy with the highest version number. Hence, for any transaction T, $data(o,v)$ is selected as the latest copy.

We now prove that for any object o, if at time t: 1) a transaction T is propagating its change to o; and 2) another transaction T' accesses a read quorum q_r before T's change propagates to q_r, then T' will never commit.

Note that in this case, T' reads a stale version $o_{T'}$ of o. When it requests to commit (if it is not aborted before that), it sends the request to a write quorum q_w. Then $\exists v \in q_w$, such that: 1) T's change of o still has not propagated to v and $data(o,v).protected = true$; or 2) T's change has been applied to $data(o,v)$ and $data(o,v).version > o_T.version$. In either case, T is aborted by CONFLICT-DETECT method.

As the result, at any time, the system exhibits that only one copy exists for any object and transactions observing an inconsistent state of object never commit. The theorem follows. □

With the help of Lemma 3 and Theorem 4, we can prove that QR model provides one-copy serializability [14].

Theorem 5: QR model implements one-copy serializability.

QR model provides five operations and every operation incurs a remote communication cost. We now analyze the communication cost of each operation.

Theorem 6: If a live read quorum $q_r(v)$ exists, the communication cost of a read or write operation that starts at node v is $O(k \cdot d(v, q_r(v)))$ for $k \geq 1$, where k is the number of nodes failed in the system. Specifically, if no node fails, the communication cost is $O(d(v, Q_r(v)))$.

Proof: For a read or write operation, the transaction calls READQUORUM method to collect the latest value of the object from a read quorum. If no node fails, the communication cost is $2d(v, Q_r(v))$. If a node $v' \in Q_r(v)$ fails, the transaction needs to probe v''s substituting nodes to construct a new read quorum. The time for v to restart the probing is at most $2d(v, Q_r(v))$. Note that $\forall q_r(v), d(v, Q_r(v)) \leq d(v, q_r(v))$.

In the worst case, if k nodes fail and v detects only one failed node at each it accesses a read quorum, at most k rounds of probing are needed for v to detect a live read quorum. On the other hand, v always starts probing from the closest possible read quorum. Therefore for each round, the time for v to restart the probing is at most $2d(v, q_r(v))$. The theorem follows. □

Similar to Theorem 6, the communication cost of other three operation can be proved in the same way.

Theorem 7: If a live write quorum $q_r(v)$ exists, the communication cost of a request-commit, commit or abort operation that starts at node v is $O(k \cdot d(v, q_w(v)))$ for $k \geq 1$, where k is the number of nodes failed in the system. Specifically, if no node fails, the communication cost is $O(v, Q_w(v))$.

Theorems 6 and 7 illustrate the advantage of exploiting locality for QR model. For read and write operations starting from v, the communication cost is only related to the distance from v to its closest read quorum. If no node fails, the communication cost is bounded by $2d(v, Q_r(v))$. Note that $d(v, Q_r(v)) \leq d(v, root)$ from the construction of $Q_r(v)$. On the other hand, the communication cost of other three operations is bounded by $O(v, Q_w(v))$. Since each transaction involves at most two operations from $\{\text{request-commit}, \text{commit}, \text{abort}\}$, when the number of read/write operations increases, the communication cost of a transaction only increases proportional to $d(v, Q_r(v))$. Compared with directory-based protocols, the communication cost of a operation in QR model is not related to the stretch provided by the underlying overlay tree.

When the number of failed nodes increases, the performance of each operation degrades linearly. In QR model, it is crucial to analyze the availability of the constructed quorum system. From the construction of the quorum system we know that if a live quorum exists, FLOODING protocol can always probe it. Let p be the probability that node lives and R_h be the availability of a read quorum, i.e., at least one live read quorum exists in a tree of height h. Then we have the following theorem.

Theorem 8: Assuming the degree of each node in the tree is at least $2d + 1$, the availability of a read quorum is

$$R_{h+1} \geq p + (1-p) \cdot \Big[\binom{2d}{d+1} (R_h)^{d+1} (1-R_h)^d + \binom{2d}{d+2} (R_h)^{d+2} (1-R_h)^{d-1} + \ldots + (R_h)^{2d} (1-R_h) \Big]$$

Proof: From [11], we have

$$R_h = Prob\{\text{Root is up}\} + Prob\{\text{Root is down}\} \times [\text{Read Availability of Majority of Subtrees}].$$

Note that in our overlay tree, if a node v at level $h+1$ is down, then one of its descendants at h is also down for $h \geq 0$, because they are mapped to the same physical node. The theorem follows. □

Similarly, let W_h be the availability of a write quorum in a tree of height h, then

Theorem 9:

$$W_{h+1} \geq p \cdot \Big[\binom{2d}{d+1} (W_h)^{d+1} (1-W_h)^d + \binom{2d}{d+2} (W_h)^{d+2} (1-W_h)^{d-1} + \ldots + (W_h)^{2d} (1-W_h) \Big]$$

Initially, R_0 and W_0 is p (only the root exists). Theorems 8 and 9 provide the recurrence relations of R_h and W_h, which can be used to calculate specific tree configurations. As the result, FLOODING provides the availability similar to the classic tree quorum system in [11].

4. Conclusion

QR model requires that at least one read and one write quorums live in the system. If no live read (or write) quorum exists, FLOODING protocol cannot proceed after READQUORUM (or WRITEQUORUM) operation. In this case, a reconfiguration of the system is needed to rebuild a new overlay tree structure. Each node then runs FLOODING protocol to find their new basic read and write quorums.

QR model exhibits graceful degradation in a failure-prone network. In a failure-free network, the communication cost imposed by QR model is comparable with SC model. When failures occur, QR model continues executing operations with high probability and reasonable higher communication cost. Such property is especially desirable for large-scale distributed systems in the presence of failures.

References

[1] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory, Second Edition*, Morgan and Claypool, 2010.

[2] Maurice Herlihy and Ye Sun, "Distributed transactional memory for metric-space networks," *Distributed Computing*, vol. 20, no. 3, pp. 195–208, 2007.

[3] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath, *Jini Specification*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain, "Software transactional memory for large scale clusters," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2008, pp. 247–258, ACM.

[5] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza, "Exploiting distributed version concurrency in a transactional memory cluster," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2006, pp. 198–208, ACM.

[6] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson, "Distm: A software transactional memory framework for clusters," in *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, Washington, DC, USA, 2008, pp. 51–58, IEEE Computer Society.

[7] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues, "D2stm: Dependable distributed software transactional memory," in *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, Washington, DC, USA, 2009, PRDC '09, pp. 307–313, IEEE Computer Society.

[8] Bo Zhang and Binoy Ravindran, "Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory," in *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*, Berlin, Heidelberg, 2009, pp. 48–53, Springer-Verlag.

[9] Hagit Attiya, Vincent Gramoli, and Alessia Milani, "A provably starvation-free distributed directory protocol," in *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, Berlin, Heidelberg, 2010, SSS'10, pp. 405–419, Springer-Verlag.

[10] Hagit Attiya and Alessia Milani, "Transactional scheduling for read-dominated workloads," in *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*, Berlin, Heidelberg, 2009, pp. 3–17, Springer-Verlag.

[11] D. Agrawal and A. El Abbadi, "The tree quorum protocol: an efficient approach for managing replicated data," in *Proceedings of the sixteenth international conference on Very large databases*, San Francisco, CA, USA, 1990, pp. 243–254, Morgan Kaufmann Publishers Inc.

[12] Richard D. Schlichting and Fred B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, pp. 222–238, August 1983.

[13] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon, "Toward a theory of transactional contention managers," in *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, 2005, pp. 258–264, ACM.

[14] Philip A. Bernstein and Nathan Goodman, "Multiversion concurrency control - theory and algorithms," *ACM Trans. Database Syst.*, vol. 8, pp. 465–483, December 1983.