MUREX: A Mutable Replica Control Scheme for Structured Peer-to-Peer Storage Systems

Jehn-Ruey Jiang[1], Chung-Ta King[2] and Chi-Hsiang Liao[2]

[1]Department of Computer Science and Information Engineering
National Central University
Jhongli, 320 Taiwan

[2]Department of Computer Science
National Tsing Hua University
Hsinchu, 300 Taiwan

Abstract

This paper proposes MUREX, a mutable replica control scheme, to keep one-copy equivalence for synchronous replication in structured P2P storage systems. For synchronous replication in P2P networks, it is proper to adopt crash-recovery as the fault model; that is, nodes are fail-stop and can recover and rejoin the system after synchronizing their state with other active nodes. In addition to the state synchronization problem, we identify other two problems to solve for synchronous replication in P2P storage systems. They are the replica acquisition and the replica migration problems. On the basis of multi-column read/write quorums, MUREX conquers the problems by the replica pointer, the on-demand replica regeneration, and the leased lock techniques. We prove the correctness of MUREX, analyze and also simulate it in terms of communication cost and operation success rate.

1. Introduction

Peer-to-peer (P2P) systems are based on self-organizing, decentralized overlay networks, in which participating peer nodes play symmetric roles — both servers and clients. P2P systems are usually designed to accommodate a large number of nodes and to adapt to dynamic node joining and leaving. The most well-known application of P2P systems is the storage data sharing over Internet, as typified in P2P music file sharing [13, 17]. However, file sharing is only one of the

functions that a storage system can support. Following the great success of P2P file sharing, a natural next step is to develop wide-area, P2P storage systems to aggregate idle storage across the Internet to be a huge storage space.

P2P storage system has been an active research topic and many systems have been proposed [2, 6-8, 10, 16, 19, 21, 24]. Some systems adopt the *unstructured P2P* approach [2, 6], in which there is no restriction on the interconnection of the nodes. Unstructured P2P storage systems are easy to build and maintain, but it is difficult to guarantee the quality in accessing the stored data [18]. Many P2P storage systems [7-8, 10, 16, 19, 21, 24] are thus built on top of *structured P2P networks* [18, 20, 22, 25].

Structured P2P storage systems rely on a hashing scheme (viz., a hash function mapping) to name the peer nodes. With the hashed ID, the peer nodes can be linked with a certain interconnection structure, such as a ring, a torus, or a hypercube. The data objects to be stored in the storage system are also named with the same hash function. A data object with the hashed key k is published to and managed by the peer node whose hashed key (or ID) is "closest" to k. To fetch a data object with hashed key k, a request is routed according to the interconnection structure until the node with the closest hashed ID is reached. In this way, any data object in the storage system can be located and accessed within a certain bound of message relays, no matter where the request is initiated. This in essence builds a *distributed hash table* (DHT) across the participating nodes.

With the distributed hash table, structured P2P systems can handle dynamic node joining and leaving. Note that according to the data allocation scheme described above, any given key in the hash table will have a node taking charge of the corresponding entry. Even after that node leaves, the underlying routing scheme will always send requests for that key to the node currently having the closest ID to the key. In this way, the leaving node is substituted and the keys managed by it are taken over by the substituting node. Please refer to Fig. 1 for such a scenario (node p substitutes leaving node q). Similarly, when a node newly joins the network, it will partially substitute a certain node to manage the keys that are now closest to its ID. Please refer to Fig. 1 for such a scenario (newly joining node u partially substitutes node v).

Although the underlying P2P routing can adapt to dynamic node joining and leaving, there

is a problem that the data object stored in nodes will be lost when nodes fail or leave. A common solution to this problem is to replicate the data objects among nodes to provide high data availability. If the data objects are read-only (or non-mutable), then the P2P storage system will only need to consider where to replicate the data objects [7-8, 10]. The system becomes much complicated if the data objects are mutable [16, 19, 21, 24]. In this paper, we concentrate on mutable P2P storage systems because they are desirable by most practical applications.

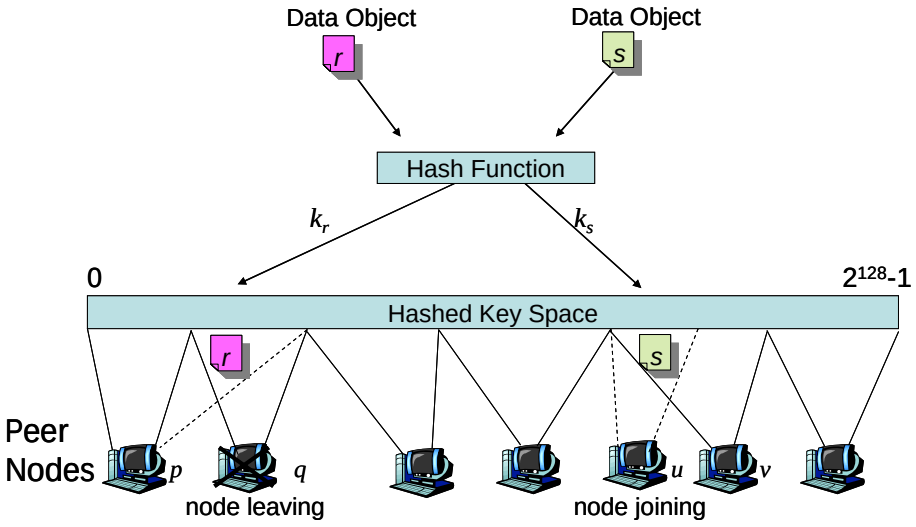

Figure 1. The scenarios of node joining and leaving.

In mutable P2P storage systems, data replication must obey the criteria of one-copy equivalence to ensure data consistency. There are two types of mechanisms to achieve such a criterion: synchronous replication and asynchronous replication. The former requires that each write operation should finish updating all replicas before the next write operation proceeds. The latter regards a local write operation as complete once data object is written to the local replica; data object is then asynchronously written to other replicas. The synchronous replication can ensure data consistency strictly, but may have long operation latency. On the other hand, the asynchronous replication may violate data consistency, but has shorter latency. However, when data inconsistency occurs, complex log-based mechanisms should be invoked to roll back the system to a consistent state. In this paper, we adopt synchronous replication since we take data consistency as the most significant factor and we regard that there may not be available storage

for storing logs for system roll-back in asynchronous replication.

For synchronous replication in P2P networks, it is proper to adopt crash-recovery as the fault model. In a crash-recovery system, nodes are fail-stop and can recover and rejoin the system after synchronizing their state with other active nodes. In addition to the *state synchronization problem*, we have two more problems to solve for synchronous replication in P2P storage systems. First, in P2P environments, an active node *p* may substitute some failing/leaving node *q* in the recovery process. Thus, node *p* must acquire the replicas hosted by node *q* somehow. Below, we call this the *replica acquisition problem*. Second, a newly joining node *u* will partially substitute an active node *v* to share *v*'s load by hosting part of *v*'s replicas. Thus, part of *v*'s replicas should be migrated to node *u*. Below, we call this the *replica migration problem*.

In this paper, we propose MUREX, a mutable replica control scheme, to keep one-copy equivalence for synchronous replication in structured P2P storage systems. On the basis of multi-column read/write quorums, MUREX conquers the problems mentioned by the *replica pointer*, the *on-demand replica regeneration*, and the *leased lock* techniques. We will prove the correctness of MUREX, analyze and also simulate it in terms of communication cost and operation success rate.

The rest of the paper is organized as follows. Preliminaries are given in Section 2. In Section 3, we discuss the problems encountered in realizing synchronous replication for P2P mutable storage systems. We then in Section 4 show how MUREX solves the problems, and analyze and simulate MUREX in Section 5. Some related works are introduced in Section 6, and concluding remarks are drawn in Section 7.

2. Preliminaries

As a replica control scheme, MUREX needs to ensure data consistency. In this paper, we adopt the *one-copy equivalence* consistency criteria, which states that the set of replicas must behave as if there were only a single copy. Conditions to ensure one-copy equivalence are

(1) no pair of write operations can proceed at the same time,

(2) no pair of a read operation and a write operation can proceed at the same time,

(3) a read operation always returns the replica that the last write operation writes.

Quorum-based schemes are popular mechanisms to enforce one-copy equivalence for synchronous replication since they render relatively high data availability and low communication cost. The basic concept of such schemes is described as follows. Each data object has n replicas, each of which is associated with a version number. A read operation should read-lock and access a read quorum of replicas and return the replica owning the largest version number. On the other hand, a write operation should write-lock and access a write quorum of replicas and then updates them with the new version number, which is one more than the largest version number just encountered. If we restrict the write-write exclusion and the write-read lock exclusion, and restrict that any pair of a read quorum and a write quorum, and any two write quorums have a non-empty intersection, then one-copy equivalence is guaranteed.

There are several mechanisms proposed in the literature for forming read and write quorums, such as tree quorums [1], majority quorums [9, 23], grid quorums [4], and multi-column quorums [12], etc. MUREX adopts the multi-column quorums, which have the smallest quorums (constant-sized quorums in the best case) among the mechanisms. It is noted that smaller quorums imply few accesses of replicas, which in turn imply lower communication cost. Furthermore, as shown in [12], multi-column quorums are candidates to achieve the highest availability, which is the probability for a quorum to be formed in an error-prone environment.

Multi-column quorums are constructed with the aid of the *multi-column structure* $MC(m)\equiv(C_1,...,C_m)$, which is a list of pairwise disjoint sets of replicas. Each set C_i is called a *column* and must satisfy $|C_i|>1$ for $1\le i\le m$. For example, $(\{r_1, r_2\}, \{r_3, r_4, r_5\}, \{r_6, r_7, r_8, r_9\})$ and $(\{r_1, r_2, r_3, r_4, r_5\}, \{r_6, r_7\}, \{r_8, r_9\})$ are multi-column structures, where $r_1,...,r_9$ are replicas of a data object.

By organizing data replicas as multi-column structure $MC(m)\equiv(C_1,...,C_m)$, the write and the read quorums are defined as follows:

A ***write quorum*** under $MC(m)$ is a set that contains all replicas of some column C_i, $\mathbf{1\le i\le m}$ (note that i=1 is included), and one replica of each of the columns $C_{i+1},...,C_m$.

A ***read quorum*** under $MC(m)$ is either

Type-1: a set that contains one replica of each of the columns $C_1,...,C_m$.

or

Type-2: a set that contains all replicas of some column C_i, $\mathbf{1<i\leq m}$ (note that i=1 is excluded), and one replica of each of the columns $C_{i+1},...,C_m$.

For example, under $MC(2)=(\{r_1, r_2, r_3\}, \{r_4, r_5\})$, the possible write quorums are $\{r_4, r_5\}$, $\{r_1, r_2, r_3, r_4\}$, $\{r_1, r_2, r_3, r_5\}$, and the possible read quorums are $\{r_1, r_4\}$, $\{r_1, r_5\}$, $\{r_2, r_4\}$, $\{r_2, r_5\}$, $\{r_3, r_4\}$, $\{r_3, r_5\}$ (of type-1) and $\{r_4, r_5\}$ (of type-2). Note that the write quorum definition and the type-2 read quorum definition are identical except that the latter does not include the sets composed of all replicas in C_1 and one replica from each of $C_2,...,C_m$.

As we have mentioned, multi-column quorums have constant size in the best case. Below, we discuss the sizes for quorums under the multi-column structure MC(m) whose columns all have the size s, where $m>s$. It is noted that we will denote such a multi-column structure as MC(m, s) in the following context. The write quorum under MC(m, s) has constant size s in the best case and $s+m-1$ in the worst case. The read quorum under MC(m, s) has constant size s in the best case and size $s+m-2$ in the worst case.

3. The Problems

In this section, we identify three problems encountered in enforcing synchronous replication for structured P2P storage systems. The three problems are replica migration, replica acquisition, and state synchronization. Below, we elaborate the problems one by one.

- *Replica Migration*: When a node u newly joins the system and partially substitutes another node v to host some replicas, node v should transfer the replicas to u immediately. However, in a constantly changing P2P environment, the cost of transferring replicas may be too high. We need an efficient mechanism to allow replicas to migrate from substituted node to substituting node.
- *Replica Acquisition*: When an active node p substitutes a failing/leaving node q, node p needs to acquire all replicas hosted by q. The problem is that node q has no idea about which replicas are hosted by p. Thus, we need a mechanism to make node p know which replicas are hosted by node q and to acquire the replicas efficiently.
- *State Synchronization*: Suppose an active node p substitutes a failing/leaving node q, and p has acquired a replica r hosted by q previously. To make replica r effective, we have to synchronize r's state, i.e., to ensure that all the participating nodes have the same view with

respect to *r*'s states. We must ensure the acquired replica *r* is an up-to-date copy. Furthermore, since there may be a node that has locked replica *r* to make *r* in the locked state, we need a mechanism to ensure that the locked state is not violated after *p* acquires replica *r*.

4. The Proposed Scheme and the Correctness

In this section, we introduce the proposed scheme — MUREX, which uses the read/write quorum, the replica pointer, the on-demand replica regeneration, and the leased lock techniques to solve the three problems mentioned in the last section. We also show in this section that MUREX can ensure the one-copy-equivalence criterion for synchronous replication in P2P storage systems. Below, we first give an overview of MUREX and then elaborate its details.

4.1 Overview

For a data object, there are *n* replicas with hashed keys $k_1,\ldots,k_n$, where k_1=HASH$_1$(data object name), …, k_n=HASH$_n$(data object name). The replicas are disseminated to the nodes whose hashed ID are nearest to $k_1,\ldots,k_n$, respectively. Please refer to Fig. 2 for the illustration of the replica dissemination. Each replica has a version number which is 0 initially and will increase gradually. MUREX organizes the *n* replicas into a multi-column structure to help form read and write quorums. A quorum is a subset of the nodes storing the *n* replicas; it should satisfy the *intersection property* that any pair of a read quorum and a write quorum, and any two write quorums have at least one common member. MUREX provides the following operations:

- ***publish***(CON, DON): to place *n* replicas at the nodes associated with $k_1,\ldots,k_n$ for the object of name DON (standing for Data Object Name) with content CON (standing for CONtent) and version number 0.
- ***read***(DON): to acquire the up-to-date replica of the object of name DON by locking all replicas of a read quorum.
- ***write***(CON, DON): to update all the replicas of a write quorum with content CON for the object of name DON.

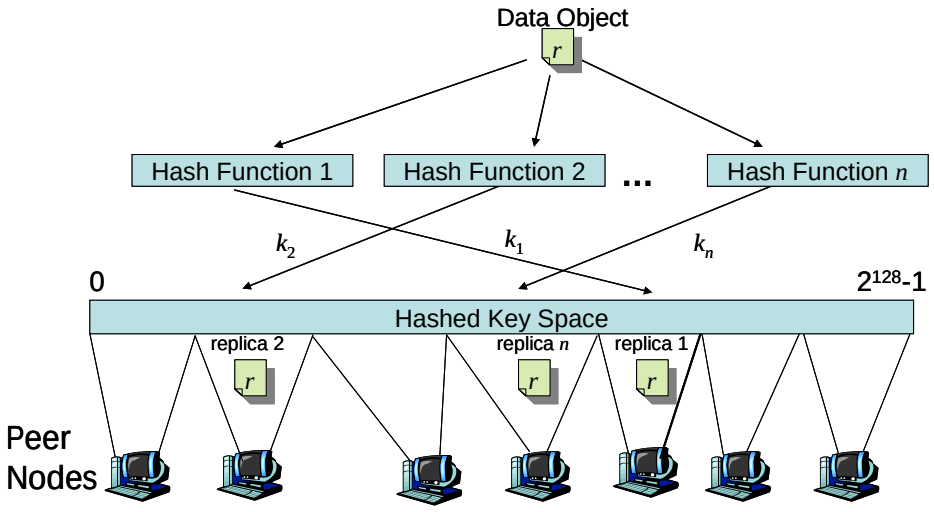

Figure 2. The dissemination of *n* replicas of a data object.

4.2 Read/Write Quorum Construction

Initially, a data object owner publishes the data object of name DON with content CON by calling ***publish***(CON, DON). Afterwards, any participant can call ***read*** (or ***write***) operation to read (or write) the data object by issuing RLOCK (or WLOCK) messages. With the help of the DHT and the multi-column structure, the messages will reach all members of a read (or write) quorum. The two functions *Get_Read_Quorum* and *Get_Write_Quorum* in Figure 3 try to issue RLOCK and WLOCK messages to nodes in a last-to-first column-wise manner to return respectively the read and the write quorums under a multi-column structure. It is noted that below we use the words “node” and “replica” interchangeably since a replica must be hosted by a node. Below, we also use LOCK message to stand for WLOCK message or RLOCK message.

When a node receives a LOCK message to request for locking a data object, it sends a MISS message to the requester if it does not own a replica of the data object. As we will show, the MISS message will cause the requester to send an up-to-date replica of the data object later. It is noted that a node has at most one pending MISS message for each replica. A MISS message is said to be pending if there is no replica sent in response to it. When a node has a pending MISS message for a replica and further receives LOCK message for locking the replica, it will send WAIT message to the requester. On the other hand, if the node owns the replica when it receives

a LOCK message for a data object, it then checks if there is a lock conflict. We say that there is a lock conflict if a read-locked replica receives a write-lock request, or if a write-locked replica receives a write-lock or a read-lock request. If there is no lock conflict, the node locks the replica and then replies with OK message containing the replica version number. On the contrary, if there is a lock conflict, the node replies with WAIT message.

```
Function Get_Write_Quorum((C1,...,Cm): MC Structure): Set;
   Var R = ∅: Set;
   For (i =m,...,1) Do
      Send WLOCK to all nodes in Ci and enter "wait period" for getting replies;
      If all nodes in Ci reply with WAIT or MISS
         Then {Send UNLOCK to nodes in C1∪…∪Ci; Exit;}
      If all nodes in Ci reply with OK Then Return R∪Ci;
      Else If a node u replies OK Then R=R∪{u};    //note: NO Return here
   EndFor
End Get_Write_Quorum

Function Get_Read_Quorum((C1,...,Cm): MC Structure): Set;
   Var R = ∅: Set;
   For (i =m,...,1) Do
      Send RLOCK to all nodes in Ci and enter "wait period" for getting replies;
      If all nodes in Ci reply with WAIT or MISS
         Then {Send UNLOCK to nodes in C1∪…∪Ci; Exit;}
      If i≠1 and all nodes in Ci reply with OK Then Return R∪Ci;
      Else If i≠1 and a node u replies with OK Then R=R∪{u}; //note: NO Return here
      Else If i=1 and a node u replies with OK Then Return R∪{u};
   EndFor
End Get_Read_Quorum
```

Figure 3. Two functions that can properly return a read and a write quorum, respectively.

After sending LOCK messages, a node enters the "wait period", which is of the length of a turn-around time. During the wait period, if a node has got any WAIT message, it can conclude that there is lock contention. For such a case, the node sends UNLOCK messages to all the nodes that it has sent LOCK messages. Only after a random backoff time, can the node start over again

to send LOCK messages for locking replicas of a quorum. The random backoff concept is borrowed from Either Net [5]. It is used to avoid continuous conflicts among contending nodes.

After *Get_Write_Quorum* or *Get_Read_Quorum* function returns a write quorum or a read quorum, it means that all replicas in the quorum have been locked. The node calling the function can then execute the desired operation. After the operation is finished, a node sends UNLOCK messages to all nodes that it has sent LOCK messages to unlock the replicas. A read operation in MUREX reads the replica of the largest version number from the read quorum. On the other hand, a write operation always writes all replicas of a write quorum with the version number one more than those ever encountered.

4.3 Replica Pointers

When a node *u* newly join the system to share part of the load of node *v* by managing replicas of keys from *k* to *k′*, the replicas of keys from *k* to *k′* should migrate from *v* to *u*. To reduce the cost of transferring all the replicas, MUREX transfers replica pointers instead of the actual replicas. A *replica pointer* is a five-tuple of the form:

<hashed key, data object name, version number, lock state, actual storing location>.

It is produced when a replica is generated and can be used to locate the actual replica. When node *v* owns the replica pointer of replica *r*, it is regarded as *r*'s host, which can reply to the lock request for *r*. On the other hand, when node *v* sends out the replica pointer of replica *r*, it is no more the host of *r* and cannot reply to the lock request for *r* (even if it stores the actual replica of *r*).

The replica pointer is a lightweight mechanism for transferring replicas; it can be propagated from node to node in a very low cost. When a node *u* owing the replica pointer of *r* receives a lock request for *r*, it should check whether the node actually storing *r* is still alive. If so, *u* can behave as host of *r*. Otherwise, *u* regards itself as having no replica *r*. It is noted that every transfer of replica pointer between two nodes, say from *v* to *u*, should be recorded locally by *v* so that later messages, such as UNLOCK message, destined to *v* for replica *r* can be sent to the last node having the replica pointer.

4.4 On-Demand Replica Regeneration

When a node *q* fails/leaves and another node *p* substitutes node *q*, it is needed for node *p* to acquire all replicas hosted by *p*. However, we have the problem that node *p* has no idea about which replicas are hosted by *q*. Below, we show how the replicas can be acquired in an on-demand manner. The term "on-demand" means that node *p* only acquire requested replicas. When node *p* receives from node *u* LOCK message for locking a replica, it should send MISS message if it does not own the replica. Node *p* is assumed to have no replica *r* if the following conditions hold:

1. *p* does not have the replica pointer of *r*
2. *p* has the replica pointer of *r* and the pointer indicates that *w* stores *r*, but *w* is not alive.

After obtaining (resp., generating) the newest replica by executing a read (resp., write) operation, node *u* should send the replica to node *p*. It is noted that a node has at most one pending MISS message for a replica. Furthermore, when a node has a pending MISS message for a replica and further receives a lock request for the replica, it will send WAIT message to the requester. In such a manner, we can ensure that a node will only receive one replica in response to MISS message.

By the on-demand replica regeneration technique, node *p* passively acquires replicas only when the replicas are requested. For the replicas never requested, there is no need to acquire them to keep the overhead as low as possible. However, the number of replicas of a data object may decrease gradually and influence the persistency of the data object. Fortunately, the bad influence does not occur for replicas that are accessed frequently. Moreover, we can allow the publisher of a data object to periodically perform the "dummy read operation" for the data object, which will be described later. We even can demand each participating node to periodically perform the dummy read operation for rarely-accessed data object replica hosted by it. When a replica of a data object is not accessed for a specific period of time, the dummy read operation is performed once. The dummy read operation is similar to the read operation and plays the role of checking if replicas of the data object are still alive; it does not read the replica in practice and thus only incurs little overhead. When some replicas of the data object are missed, the node initiating the dummy read operation can re-disseminate the replica to the proper node. The persistency of the

data object can thus be ensured.

4.5 Leased Locks

When a replica *r* of a data object is re-disseminated to some node, we must ensure that all participating nodes have the same view with respect to the replica. We first need to ensure the replica is up-to-date. If the replica is re-disseminated due to a node's receiving MISS message, the replica is surely up-to-date. This is because a node re-disseminates the replica only after it has executed the read (or write) operation to acquire (or generate) the up-to-date replica. On the other hand, if the replica is re-disseminated due to a node's performing a dummy read operation, the node is demanded to first obtain the up-to-date replica and then to re-disseminate the replica.

The second thing for all participating nodes to agree with is the state of replica *r*. Since there may be some node that has locked replica *r* to make *r* in the lock state, we need to ensure that the lock state is not violated. To achieve this, each lock is assumed to be a *leased lock* that has a leased period of *L*. That is to say, after a replica is locked, it becomes unlocked automatically after a period of *L*. Assume that the critical section (CS) of a read or a write operation takes *C* time to complete. A node should release any obtained lock if it still has no chance to enter the CS and $H>L-C-D$ holds, where *H* is the holding time of the lock and *D* is the propagation delay for transmitting the lock. Please see Figure 4 for the relationship among *H*, *L*, *C* and *D*. The condition of $H>L-C-D$ can ensure a node to complete the desired operation before any lock expires.

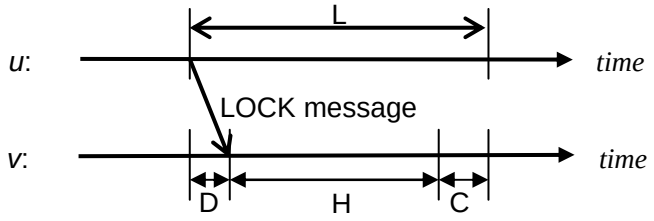

Figure 4. The relationship among *H*, *L*, *C* and *D*.

When a node detects that a lock of a specific replica is expiring (i.e., $H>L-C-D$ is going to hold), it is possible that the locks of other replicas will also expire in the near future. Thus, we demand a node to release all locks and start over to acquire the locks again. In this manner,

MUREX can avoid deadlock and starvation. Furthermore, we demand a node to wait for a random backoff time before acquiring the locks next time. This can alleviate the chance of repeatedly occurrence of contention-then-release-all-locks situation.

Now, we describe how to make all participating node have the same view for the lock state by the concept of leased locks. Suppose a node *p* substitutes a failing/leaving node *q* to host replica *r*, and node *p* has received the up-to-date replica of *r* at time *T*. After receiving the up-to-date replica, node *p* generates a replica pointer for *r* and can start to reply to LOCK message for locking *r* at time *T*+*L*, where *L* is the leased period of the lock. In this manner, all participating nodes have the same view with respect to *r*'s lock state.

4.6 Correctness

By now, we have elaborated the details of MUREX. Below, we show the correctness of MUREX by the following theorems.

Theorem 1. (Safety Property) MUREX ensures the one-copy equivalence consistency criteria.

Proof:

In MUREX, a read operation read-locks and accesses a read quorum of replicas, and returns the replica owning the largest version number. A write operation write-locks and updates a write quorum of replicas with the version number which is one more than the largest version number encountered. As shown in [12], multi-column quorums satisfy the intersection property: any pair of a read quorum and a write quorum, and any two write quorums have a non-empty intersection. By the read-write and write-write lock exclusions and the intersection property, a read operation always returns the replica with the largest version number, which is the most up-to-date. Furthermore, by the replica pointer, on-demand replica regeneration, and leased lock techniques, a replica can neither be locked by more than two write lock requests, nor be locked by a read lock request and a write lock request simultaneously. Thus, neither two write operations nor a write and a read operation can proceed at the same time. We can therefore conclude that one-copy equivalence is guaranteed. □

Theorem 2. (Liveness Property) There is neither deadlock nor starvation in MUREX.

Proof:

MUREX demands a node to relinquish all locks when any lock is going to expire. This eliminates the condition of hold and wait, which is necessary for a deadlock to occur. Thus, there will be no deadlock. Furthermore, there will be no starvation since the random backoff mechanism demands a node to wait for a random delay before reissuing lock requests. Thus, a node can eventually acquire enough locks to perform the desired operation, which means that no starvation will occur. □

5. Analysis and Simulation

In this section, we analyze and simulate MUREX for evaluating its performance. We first analyze the communication cost of MUREX for the case of no contention. In MUREX, a node sends request messages for locking replicas in a last-column-to-first-column order. For an MC(m, s) multi-column structure, an operation needs 3s messages in the best case: one LOCK, one OK and one UNLOCK messages sent to/by each member of the last column C_m.

When failures occur, the communication cost increases gradually. In the worst case, a node sends LOCK messages to all n replicas. Thus, the communication cost will be O(n) in the worst case. Fortunately, since MUREX demands a node to send LOCK messages to members of the last column first, the best case communication cost occurs much more frequently than the worst case.

When there are contending nodes, the communication cost also increases. This is because when a node receives WAIT message, it will release all the locks obtained, and start over again to send LOCK messages after a random backoff time. Fortunately, the random backoff mechanism can scatter the resending of LOCK messages and thus some node will complete its operation successfully. Consequently, the communication cost will not go too high.

We conduct a simulation for MUREX with regard to success rates of operations for the purpose of evaluating the influence of different multi-column quorums. An operation is considered to be successful if it can finish before any leased lock expires. The simulation assumes that the underlying DHT is Tornado [11], which is developed by ourselves. We adopt four multi-column structures, namely MC(5, 3), MC(4, 3), MC(5, 2) and MC(4, 2), for the

construction of read/write quorums. When we simulate the case for MC(m, s), the leased period is assumed to be m×(turn-around time). We also assume that there are totally 2000 nodes in the system. There are three experiments in our simulation. For each experiment, we perform the simulation for 3000 seconds, during which 10000 operations are requested, half for reading and half for writing. Each request is assumed to be destined for a random file (data object); thus, when the number of files increases, the degree of contention decreases.

In the first experiment, we assume there are 200 nodes that may join or leave the system randomly during the experiment. In Figure 5, we can see that the success rate increases as the number of files increases. This is because the degree of contention decreases when there are more files. Among the four multi-column structures, we can see that MC(5, 3) achieve the best performance and MC(4, 2) achieves the worst, while MC(4, 3) and MC(5, 2) achieve in-between and resembling performances. From this experiment, we can check that lower contention renders higher success rates.

In the second experiment, we assume there are 250 files in the systems and 0, 50, 100 or 200 nodes may leave during the experiment. In Figure 6, we can see that the success rate decreases as the number of leaving nodes increases. This is because more leaving nodes can cause more unsuccessful lock requests. Among the four multi-column structures, we can see that MC(5, 3) renders the best performance and MC(4, 2) renders the worst, while MC(4, 3) and MC(5, 2) render in-between and resembling performances. From this experiment, we can see that higher node leaving rates cause worse performances.

In the third experiment, we assume that no node joins or leaves. In Figure 7, we can see that the success rate increases as the number of files increases. This is because the degree of contention decreases when there are more files. We can also see that the performances for the four multi-column structures are resembling. By this experiment, we can see that the degree of contention is a dominant factor in the success rate.

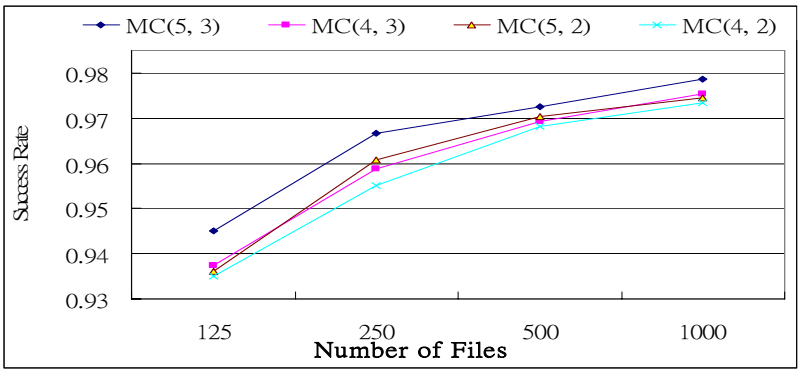

Figure 5. Simulation results for the 2nd experiment, in which 200 nodes may join or leave randomly.

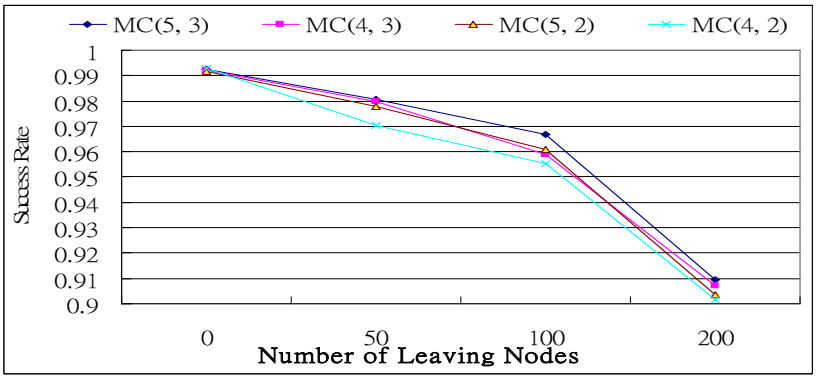

Figure 6. Simulation results for the 2nd experiment, in which 0, 50, 100, or 200 nodes may leave.

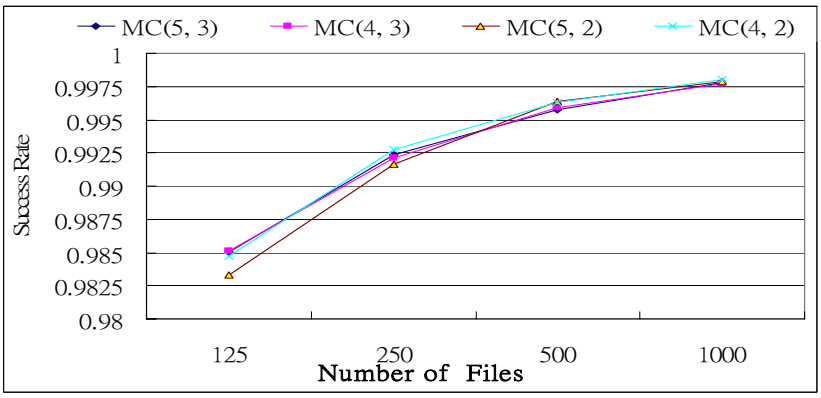

Figure 7. Simulation results for the 3rd experiment, in which no node join or leave during the experiment.

6. Related Work

As far as we know, there are four existent mutable P2P storage systems: Ivy [16], Eliot [21], Oasis [19], and Om [24]. The key concepts of these systems are P2P logs, replicated metadata service, dynamic quorum membership, and replica membership reconfiguration, respectively. In additional to these four systems, we also review a scheme called “Sigma Protocol” [15], which

intelligently collect replica states to achieve mutual exclusion among replicas. The scheme can be used as the basis of replica locking control and is thus worth investigating.

Below, we first introduce Ivy [16], which stores a set of logs with the aid of DHTs. Ivy keeps, for each participant, a log storing all its updates, and maintains data consistency optimistically by performing conflict resolutions among all logs. The logs should be kept indefinitely and a participant must scan all the logs associated with a file for the up-to-date data. Thus, Ivy is only suitable for small groups of participants.

Eliot [21] relies a reliable, fault-tolerant, and read-only P2P storage substrate — Charles to store data blocks, and uses an auxiliary metadata service (MS) for storing mutable metadata. It supports NFS-like write-through consistency semantics; however, the traffic between MS and the client is high for such semantics. It also supports AFS-like open-close consistency semantics; however, this semantics may cause the problem of lost updates. The MS service is provided by a conventional replicated database, which may be unfit for dynamic P2P environments.

Oasis [19] is based on Gifford's weighted voting quorum concept and allows dynamic quorum membership. It spreads versioned metadata along with data replicas over the P2P network. To complete an operation on a data object, a client must first find a metadata related to the object to figure out the total number of votes, required votes for a read and a write operation, replica list, and so on. It then forms a quorum according to the acquired metadata. Data consistency may be violated if a node happens to use a stale metadata.

Om [24] is based on the concepts of automatic replica regeneration and replica membership reconfiguration. The data consistency is maintained by two quorum systems: a read-one-write-all quorum system for accessing replicas, and a witness-based quorum system for reconfiguration. Om forwards all writes to the primary replica to serialize them, and uses a two-phase procedure to propagate the writes to all secondary replicas. In this manner, Om allows replica regeneration from a single replica. However, the primary replica may become a bottleneck and the overhead incurred by the two-phase procedure may be too high. Furthermore, the reconfiguration by witness-based quorum system has the probability of violating data consistency.

The Sigma protocol [15] uses Byzantine agreement algorithm [3, 14] to achieve mutual exclusion among replicas for P2P storage systems. The basic concept of the protocol is described

as follows. A node *u* wishing to be the winner of the mutual exclusion should send a timestamped request to all *n* replicas ($n=3k+1$, where *k* stands for the maximum number of nodes that may leave the system) and waits for replies. On receiving a request from *u*, a node *v* should put *u*'s request in a local queue in accordance with the timestamp order, takes the node as the winner whose request is in the front of the queue, and replies the winner ID to *u*. When the number of replies received by *u* exceeds $2k+1$, *u* acts according to the following conditions: (1) if more than $2k+1$ replies take *u* as the winner, then *u* is the winner. (2) if more than $2k+1$ replies take *w* ($w \neq u$) as the winner, then *w* is the winner and *u* just keeps waiting. (3) if no node is regarded as the winner by more than $2k+1$ replies, then *u* sends YIELD message to cancel its request temporarily and then resends its request again. In this manner, one node can eventually be elected as the winner even when communication delay variance is large. However, Sigma protocol requires a node to send timestamped requests to all replicas and to receive advantaged replies from a large portion ($\geq 2/3$) of replicas to be the winner of the mutual exclusion. This may incur large overhead; the overhead will even be larger under an environment of high contention.

7. Conclusion

In this paper, we have identified three problems for synchronous replication in DHT-based mutable P2P storage systems. The problems are replica migration, replica acquisition and state synchronization. We have proposed MUREX, a mutable replica control scheme, to solve these problems by the concepts of multi-column read/write quorums, replica pointers, on-demand replica regeneration and leased locks. We have proved that MUREX guarantees one-copy equivalence and causes no deadlock. Furthermore, we have analyzed and simulated MUREX. As we have shown, MUREX has constant communication cost in the best case and has good operation success rate.

References

1. D. Agrawal and A. E. Abbadi, "The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data," in *Proc. Of the 16th VLDB Conf.*, Brisbane, Australia, 1990.
2. R. Bhagwan, D. Moore, S. Savage, and G. Voelker, "Replication Strategies for Highly

Available Peer-to-peer Storage," in *Proc. of International Workshop on Future Directions in Distributed Computing*, 2002.

3. M. Castro, B. Liskov, "Practical Byzantine Fault Tolerance," in *Proc. of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, February 1999.
4. S. Y. Cheung, M.H.Ammar, M.Ahamad, "The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data," *IEEE Transactions on Knowledge and Data Engineering*, 1992.
5. G. Chockler, D. Malkhi, and M. K. Reiter, "Backoff Protocols for Distributed Mutual Exclusion and Ordering," in *Proc. of the 21st International Conference on Distributed Computing Systems*, pp. 11-20, April 2001.
6. E. Cohen and S. Shenker, "Replication Strategies in Unstructured Peer-to-peer Networks," in *Proc. of SIGCOMM*, 2002.
7. F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area Cooperative Storage with CFS," in *Proc. of SOSP*, 2001.
8. P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *Proc. of HotOS VIII*, Schoss Elmau, Germany, May 2001.
9. D. K. Gifford, "Weighted Voting for Replicated Data," in *Proc. of 7th ACM Symp. on Operating Systems*, pages 150–162, 1979.
10. V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher, "Adaptive Replication in Peer-to-peer Systems," in *Proc. of International Conference on Distributed Computing Systems*, 2004.
11. H.-C. Hsiao and C.-T. King, "Tornado: A Capability-aware Peer-to-peer Storage Overlay", *Journal of Parallel and Distributed Computing*, 2003.
12. J.-R. Jiang, "The Column Protocol: A High Availability and Low Message Cost Solution for Managing Replicated Data," *International Journal of Information Systems*, Vol. 20, No. 8, pp. 687-696, 1995.
13. T. Klingberg and R. Manfred, "The Gnutella 0.6 Protocol Draft," *http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html*, June 2002.
14. L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, 4(3):382-401, July 1982.
15. S. Lin, Q. Lian, M. Chen, and Z. Zhang, "A practical distributed mutual exclusion protocol in dynamic peer-to- peer systems," in *Proc. of 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, 2004.
16. A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, "Ivy: A Read/write Peer-to-peer File System," in *Proc. of the Symposium on Operating Systems Design and Implementation*

(OSDI), 2002.

17. Napster, Napster Website, http://www.napster.com
18. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network", in *Proc. of ACM SIGCOMM*, 2001.
19. M. Rodrig, and A. Lamarca, "Decentralized Weighted Voting for P2P Data Management," in *Proc. of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pp. 85–92, 2003.
20. A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems", in *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.
21. C. Stein, M. Tucker, and M. Seltzer, "Building a Reliable Mutable File System on Peer-to-peer Storage," in *Proc. of 21st IEEE Symposium on Reliable Distributed Systems (WRP2PDS)*, 2002.
22. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", in *Proc. of ACM SIGCOMM*, 2001.
23. Robert H. Thomas, "A Majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems (TODS)*, 1979
24. H. Yu. and A. Vahdat, "Consistent and Automatic Replica Regeneration," in *Proc. of First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.
25. B. Zhou, D. A. Joseph, and J. Kubiatowicz, "Tapestry: A Fault Tolerant Wide Area Network Infrastructure," in *Proc. of ACM SIGCOMM*, 2001.