A Replica Control Method for Improving Availability for Read-only Transactions[†]

Chang Sup Park Myoung Ho Kim Yoon Joon Lee
Department of Computer Science
Korea Advanced Institute of Science and Technology
373-1, Kusong-dong, Yusong-gu, Taejon, 305-701, Korea
{parkcs,mhkim,yjlee}@dbserver.kaist.ac.kr

Abstract

Data replication is often considered in distributed database systems to enhance availability and performance. The benefit of data replication, however, can only be realized at the cost of maintaining the consistency of data. In particular, network partition failures make it more difficult to achieve high data availability while ensuring strong correctness criteria such as 1-copy serializability. In this paper, we propose a replica control method to improve the availability of data in the presence of network partition failures. Our method extends the traditional primary copy method by using the relaxed correctness criterion called insular consistency for large-scale distributed systems, where partition failures frequently occur. We focus on increasing the availability of data for read-only transactions. We introduce a version vector as a tool for guaranteeing insular consistency and present a mechanism that allows read-only transactions to be executed at any partition as long as the insular consistency is satisfied. An asynchronous update propagation mechanism is also employed to improve the performance of update operations. We also show that the proposed method is correct and give some performance considerations.

1. Introduction

The main goal of data replication in a distributed database system is to enhance data availability and performance. By storing important data at multiple sites, we can continue to execute operations on the data even if failures occur at some parts of the system. Besides, performance of transactions can be improved because an efficient data access based on geographic proximity can be provided.

A replicated database should provide user transactions with transparency for replicated data. A read or write operation on a logical data item in a transaction should be transparently mapped into read or write operations on physical replicas of the data item, and consistency among the replicas must be maintained according to a predefined correctness criterion. A replica control protocol is required for the transparent and consistent management of replicas.

One-copy serializability(*1SR*)[3] is the most widely used correctness criterion in the literature on replicated databases. *1SR* means that the concurrent execution of transactions on a replicated database must be equivalent to a serial execution of those transactions on a non-replicated or *one-copy* database. It is the incorporated notion of serializability in non-replicated databases and one-copy equivalence, and it can be guaranteed by a concurrency control algorithm and a replica control protocol.

A distributed system consists of two kinds of components: sites, which process information, and communication links, which transmit information between sites. Both of them can experience system failures. As for site failures, we assume the fail-stop model [11]. The most critical communication failure is *a network partition failure* [5], where a network is partitioned into multiple sub-networks that cannot communicate with each other. If two transactions that update the same data item execute the update on different replicas in different partitions, an inconsistency can be introduced across the partitions. So difficulty lies in keeping consistency across all partitions in the face of system failures while at the same time enhancing data availability [5].

In this paper, we propose a data replication method that can improve data availability and system performance in a large-scale distributed database system where network partition failures frequently occur. We use as our correctness criterion the insular consistency [7] that is a relaxed correctness criterion from *1SR*. We mainly focus on increasing data availability for read-only transactions. In our method, most of read-only transactions can be executed at any partition in a network regardless of the number of partitions or the

[†]This work was supported by Korea Science and Engineering Foundation(KOSEF) through Center for Artificial Intelligence Research(CAIR), the Engineering Research Center(ERC) of Excellence Program.

size of each partition. Our protocol is based on the primary copy method, but has different update mechanisms that do not severely degrade performance of update transactions.

The remainder of this paper is organized as follows. In section 2, we discuss previous related works and present the motivation of our work. Section 3 describes the proposed replica control protocol in detail and Section 4 proves the correctness of our method. Finally, we conclude with a discussion of our work in section 5.

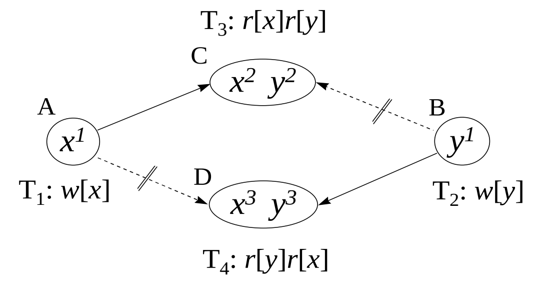
2. Related Works and Motivation

There are broadly two classes of consistency maintenance mechanisms of data replication, i.e., *pessimistic* and *optimistic*[5]. Pessimistic strategies keep a replicated database in a consistent state all the time by limiting the availability of data. They restrict the execution of update operations on a data item within only one partition. Most of the methods, including the primary copy method and the quorum consensus algorithm, belong to this class. On the other hand, optimistic strategies do not limit availability and allow updates on replicas of a data item in any partition. In these strategies, the system detects and resolves an inconsistency when it recovers from failures. Optimistic strategies in general are considered difficult to be applied because they require the rollback of the transactions that are already committed or the execution of appropriate compensating transactions.

[9] discusses *eager replication* and *lazy replication*. In the eager replication, an update operation is executed on all replicas of a data item synchronously in an atomic transaction, while the lazy replication applies an update operation to only one replica or a subset of replicas in a transaction and then propagates it to the other replicas asynchronously after the transaction commits. The difference in their update scheme has a great effect on consistency and update performance.

2.1. Insular Consistency

In most applications, the frequency of read-only transactions is much higher than that of update transactions. Hence, there have been many works that specifically focus on improving data availability for read-only transactions. [7] proposes three correctness criteria for read-only transactions in a fully replicated database, including *insular* consistency. The notion of insular consistency that has been shown to be effective in many applications[2, 7, 12] is as follows: an execution history H of some transactions satisfies insular consistency if and only if every sub-history that consists of all update transactions and a read-only transaction in H satisfies *1SR*. Figure 1 shows an example history that satisfies insular consistency. In the figure, each node



$\{T_1, T_2, T_3\}$: *1SR* $(T_1 \rightarrow T_3 \rightarrow T_2)$
$\{T_1, T_2, T_4\}$: *1SR* $(T_2 \rightarrow T_4 \rightarrow T_1)$
$\{T_1, T_2, T_3, T_4\}$: insular consistent

Figure 1. An example of insular consistency

represents a site where replicas of data are stored. x^1, x^2, and x^3 represent three replicas of a data item x, and y^1, y^2, and y^3 represent three replicas of a data item y. We suppose that update transactions T_1 and T_2 are executed at site A and site B respectively, and that after they commit at those sites, their results, or the new values of x and y are independently propagated to site C and site D. We also suppose that failures occur on the communication links between A and D, and between B and C so that the updates of x and y cannot be propagated to D and C, respectively. Now, if read-only transactions T_3 and T_4 are executed at C and D respectively, T_3 will see the result of only T_1, and T_4 will see the result of only T_2. Therefore, the execution history of $\{T_1, T_2, T_3, T_4\}$ is not one-copy serializable, while it satisfies insular consistency because the histories of $\{T_1, T_2, T_3\}$, $\{T_1, T_2, T_4\}$, and $\{T_1, T_2\}$ are all one-copy serializable.

[12] introduces three notions of consistency and proposes algorithms for executing read-only transactions in multiversion environment. [2] has also developed a replica control protocol based on insular consistency to enhance data availability for read-only transactions by using a new update propagation mechanism, called Commit Propagation Mechanism. Our method has some similarities with this method in that it adopts insular consistency as a correctness criterion and makes use of piggy-backing necessary information on the messages of the two-phase commit(2PC) protocol. This method, however, is fundamentally different from our method in three ways: (1) it is based on the standard quorum consensus protocol[8], (2) it guarantees insular consistency with respect to only insular transactions[1], and (3) its protocol becomes relatively complex for a partially replicated database.

[1] The insular transaction[7] means a read-only transaction that can be executed entirely at a single site.

2.2. Motivation

Most proposed methods for data replication adopt *1SR* as their consistency criteria. There are several problems to apply these methods in practice. First, the performance of a system degrades significantly because many replicas need to be synchronously accessed before committing a transaction. Second, they cannot cope with network partition failures effectively. When a network is partitioned, most methods allow read and write operations within only one partition or prohibit write operations in all partitions in order to prevent the occurrence of inconsistency among different partitions. Such approaches inevitably impose a severe restriction on data availability. Overhead from synchronous updates and vulnerability for a network partition failure are more serious in large-scale distributed systems and mobile computing environment.

It is important to make as many read-only transactions can be executed as possible when the frequency of read-only transactions is much higher than that of update transactions. Since *1SR* is considered too restrictive for read-only transactions in many applications, we need other correctness criteria to optimize the execution of read-only transactions.

In this paper, we propose a replication method that is appropriate for large-scale distributed systems or mobile computing systems, in which network partition failures frequently occur. Our method uses insular consistency as a correctness criterion and applies an asynchronous propagation scheme for the updates of replicas in remote sites, which may pay expensive communication cost and experience frequent communication failures. More importantly, our method improves data availability by allowing read-only transactions to execute in any partition including replicas of all data to read as much as possible.

3. Our Replication Method

3.1. Model and Assumption

Our replication method is for the large-scale distributed environment where many sites are distributed over extensive areas. Some characteristics of this environment are that communication cost between two sites that are remote from each other is expensive and that network partition failures frequently occur.

Without loss of generality, we make the following assumptions in this paper. First, the scheduler in each site uses a concurrency control algorithm that can guarantee serializable executions of transactions, such as the distributed two-phase locking algorithm. Second, there is no loss of the messages transmitted between two sites of different clusters. This can be realized with a system service such as a Recoverable Queuing System(RQS) [4, 6]. Finally, distributed sites are geographically grouped into several clusters. The communication cost is more expensive and the communication failures occur more frequently *between* clusters than those *within* a cluster. This is shown in Figure 2.

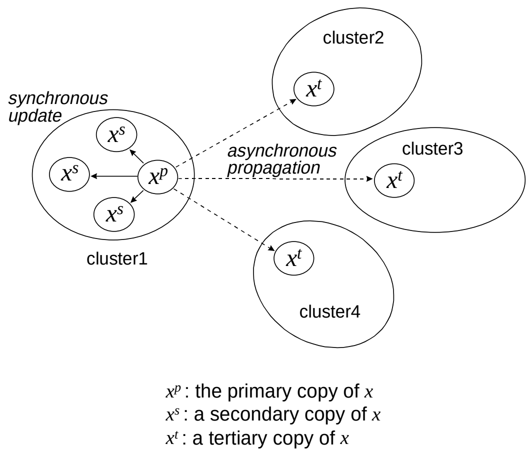

cluster2
cluster3
cluster1
cluster4
synchronous update
asynchronous propagation
x^p
x^s
x^t
x^p : the primary copy of x
x^s : a secondary copy of x
x^t : a tertiary copy of x

Figure 2. A data replication model in a large-scale distributed system

Our replication method is based on the primary copy method [1]. Replicas of each data item are divided into one primary copy and many backup copies, and all read and write operations on a data item are first transmitted to the primary copy site of it and then executed on the primary copy of the data item to guarantee *1SR*. Our method, however, takes further steps of dividing backup copies into two groups, i.e., *secondary copies* and *tertiary copies* according to whether they are contained in the cluster to which the primary copy belongs or not, and using different update propagation schemes for two kinds of replicas. To put it concretely, while secondary copies in the cluster in which a primary copy is contained are synchronously updated before a transaction commits, the result of the update is propagated to tertiary copies in the other clusters asynchronously after the transaction commits. By committing a transaction without waiting for expensive update propagation to the other clusters to be finished, we can decrease the response time of the transaction.

While a secondary copy of a data item always has the latest value, which is the same one as the primary copy of the data item has, a tertiary copy in a different cluster has a stale value until all the new values written are propagated to it. Therefore, when many transactions concurrently execute, many different versions of a data item may exist at the non-primary copies of it. These versions can be ordered by the time when they were created.

In our method, the location of the primary copy of each data item is determined as follows. Generally, for each replicated data item, there exists a site that plays a role of its owner. The owner site of a data item and the cluster that contains it are respectively called *home site* and *home cluster* of the data item, and a certain replica in the home site is designated as the primary copy of the data item. In this paper, we assume that most update transactions that update a data item are issued at the home site or at one of the other sites in the home cluster of the data item. For example, if a man that resides in an area A has an account at a bank, he will visit a branch of the bank in the area A more frequently than branches in the other areas. That means updates on his account will be originated mainly in A. In this case, we designate a site in the area A as the home site of the account and store the primary copy of it in that site. By using the notion of the home cluster, we can improve the availability of a data item in an area where the update requests on the data item occur most frequently.

On the other hand, when a failure occurs at the primary copy site of a data item, we select a new primary copy from the secondary copies in the same cluster along the predefined order of succession or by an election protocol. If a partition failure occurs in the home cluster of a data item, we choose a replica as the primary copy from only the majority partition, as the viewstamped replication method [10] does, in order to have always only one primary copy in the entire system for each data item. The new primary copy selected is sure to have the latest value that reflects the results of all the updates on the data item which had been executed before the occurrence of the failure.

Our method adopts insular consistency that is relaxed from *1SR* and improves the availability of data for read-only transactions against network partition failures. It allows read-only transactions that can afford to read stale data values to read replicas other than the primary copies. In other words, even if the primary copy of a data item is inaccessible by site failures or network partition failures, a transaction that must read the data item can continue to execute by reading an accessible non-primary copy in the local cluster or in one of the other near clusters. Our method guarantees insular consistency among transactions by exploiting version vectors, which are defined in the next section.

3.2. Versions and Version Vectors

In our method, each write operation on a data item produces a new *version* of it. Each version of a data item has the unique *version number* with the data value. Whenever a new version is created, it is assigned a version number that sequentially increases by one. Many different versions of a data item may exist in a replicated database at the same time because of asynchronouse update propagation to tertiary copies, and their version numbers imply the order in which they were created. Primary and secondary copies always have the recent versions of data items.

Notation	*Meaning*
DS	$\{x \mid$ a replicated data item$\}$
$VS(x)$	$\{x_i \mid$ a version of the data item $x \in DS\}$
$VN(x_i)$	i, the version number of $x_i \in VS(x)$
$RS(T)$	a readset, $\{x \mid r[x]$ is in the transaction $T\}$
$WS(T)$	a writeset, $\{x \mid w[x]$ is in the transaction $T\}$
$V_r(T)$	$\{x_i \mid$ the version of $x \in RS(T)$, read by $T\}$
$V_w(T)$	$\{x_i \mid$ the new version of $x \in WS(T)$, written by $T\}$

Table 1. Notations

Table 1 shows the notations related to data items and their versions. We define four basic relations on versions as follows.

Definition 1 *4 basic relations on versions*

1. $\prec_{wr}$ *is a binary relation on the set of versions of data items, such that* $x_i \prec_{wr} y_j$ *iff* $x_i \in V_w(T_m) \cap V_r(T_n)$ *and* $y_j \in V_w(T_n)$ *for two different transactions* T_m *and* T_n.
2. $\prec_{ww}$ *is a binary relation on the set of versions of data items, such that* $x_i \prec_{ww} x_j$ *iff* $x_i \in V_w(T_m)$ *and* $x_j \in V_w(T_n)$ *for two different transactions* T_m *and* T_n*, and* x_i, x_j *are two versions of a data item* x *such that* $VN(x_j) = VN(x_i) + 1$.
3. $\prec_{rw}$ *is a binary relation on the set of versions of data items, such that* $x_i \prec_{rw} y_j$ *iff* $x_i \in V_w(T_m)$ *and* $y_j \in V_w(T_n)$ *for two different transactions* T_m *and* T_n*, and there exists* y_{j-1} *such that* $y_{j-1} \prec_{ww} y_j$ *and* $y_{j-1} \prec_{wr} x_i$.
4. $=_w$ *is a binary relation on the set of versions of data items, such that* $x_i =_w y_j$ *iff* $x_i, y_j \in V_w(T_n)$ *for a transaction* T_n.

Figure 3 depicts the above basic relations as graphs, which represent relationships among the versions that are read or written by update transactions. A node x_i denotes a version of a data item x. A directed edge from x_i to y_j which is labeled with T_k means that there exists a transaction T_k such that $x_i \in V_r(T_k)$ and $y_j \in V_w(T_k)$, and a directed edge into x_i which is labeled with T_k only means $x_i \in V_w(T_k)$. Since all update transactions executed on the primary copies are serializable in our method, these graphs describe the serializable executions of update transactions. The relations $\prec_{wr}$, $\prec_{ww}$, and $\prec_{rw}$ imply that there exist respectively a write-read conflict, a write-write conflict, and a

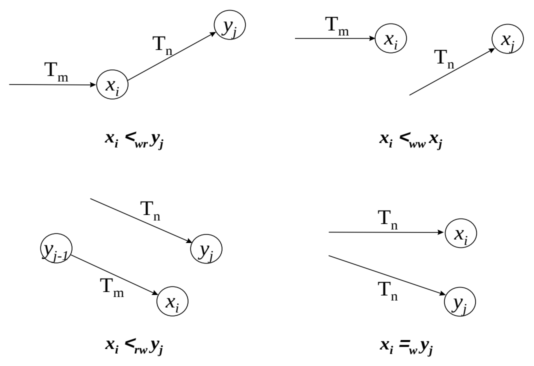


Figure 3. The basic relations on versions

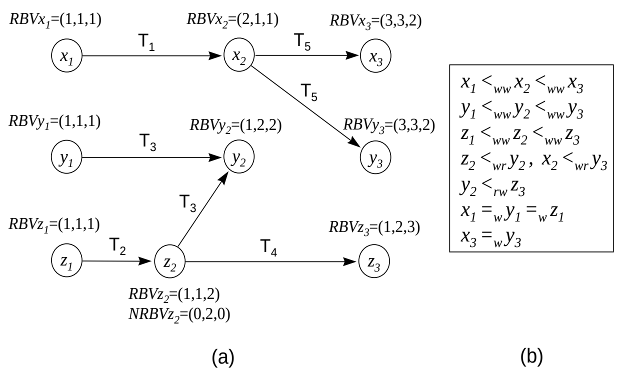


Figure 4. An example of RBV and $NRBV$

read-write conflict between two transactions, e.g., T_m and T_n in the figure, that created two related versions. So each of them determines a direct serialization order between T_m and T_n. The relation $=_w$ is an equivalent relation that means the creation of two versions of different data by an update transaction. Now, we define the following relation using the above four relations.

Definition 2 $\preceq_{nf}$ *is a relation which is defined by*

$$\preceq_{nf} \;\equiv\; \prec_{wr} \cup \prec_{ww} \cup \prec_{rw} \cup =_w$$

We denote the transitive closure of this relation as $\preceq^*_{nf}$. $x_i \preceq^*_{nf} y_j$ means that a transaction T_m that created x_i precedes a transaction T_n that created y_j *directly or indirectly* in a serialization order, or that x_i and y_j were created by the same transaction T.

A *version vector* is an ordered list of version numbers, in which a version number of a version for each data item is stored. For example, if n data items are replicated in a database, n version numbers, one for each data item, are stored in a version vector in a predefined order. There are two kinds of version vectors: *Read Bound Version Vector* and *Next Read Bound Version Vector*.

Definition 3 *Read Bound Version Vector(RBV)*
The Read Bound Version Vector of a version x_i is a version vector whose element for a data item y in DS is defined by

$$RBV_{x_i}[y]^2 = VN(y_j)$$

where $y_j \in VS(y)$, $y_j \preceq_{nf} x_i$, and there is no $y_k \in VS(y)$ such that $y_j \preceq^*_{nf} y_k$, $y_k \preceq^*_{nf} x_i$, and $y_k \neq y_j$.

RBV is defined for each version of a data item and is stored with each replica of the version. The version of y whose version number is $RBV_{x_i}[y]$ is the one that was created either by an update transaction that created x_i or by an update transaction that precedes the transaction that created x_i in the serialization order and most recently updated y. That means it is the oldest version of y that can be read with x_i in a read-only transaction while insular consistency is not violated. We call this version as *read bound version* of y for x_i. RBV stored with a replica is updated whenever a new version is created and stored in the replica, and it is used in the validation process of a read-only transaction.

Figure 4 shows an example of versions and their version vectors that are created and stored by some update transactions. In Figure 4-(a), the elements of version vectors are denoted in the order of x, y, and z. In this example, the basic relations on the versions exist as shown in Figure 4-(b). These relations determine the serialization order among the transactions and RBV of each version as shown in Figure 4-(a). For example, the serialization order among T_2, T_3, and T_5 is determined to $T_2 \rightarrow T_3 \rightarrow T_5$ by $z_2 \prec_{wr} y_2$ and $y_2 \prec_{ww} y_3$, and the values of $RBV_{y_2}[z]$ and $RBV_{y_3}[z]$ are set to 2, the version number of z_2. From the previously described meaning of the elements of RBV, these values imply that insular consistency is violated if y_2 or y_3 is read in a read-only transaction with a version of z that is older than z_2. For example, when a read-only transaction T_r reads y_2 and z_1, a cycle consisting of T_2, T_3, and T_r is generated in the serialization graph of committed transactions.

Definition 4 *Next Read Bound Version Vector(NRBV)*
Let V_c be the set of the most recent versions of all data items. The Next Read Bound Version Vector of x_c, the most recent version of a data item x, is a version vector whose element for a data item y is defined by

$$NRBV_{x_c}[y] \equiv max\{RBV_{z_c}[y]\}$$

for $\forall z_c \in V_c$ such that $x_c \prec_{wr} z_c$.

[2] In this paper, we index an element of a version vector by the name of a data item for convenience' sake.

$NRBV$ is defined for the most recent version of a data item and is stored with the primary and secondary copies of the data item. It is updated whenever the data item is read by an update transaction. $NRBV$ is an auxiliary version vector for RBV; when a new version is created, it is used in determining the value of a new RBV of the version. In the above definition, until the most recent version of x, x_c is updated to the next version, x_{c+1}, some necessary elements of RBV of z_c are stored in $NRBV$ of x_c on the ground that $x_c \prec_{wr} z_c$ directly induces $z_c \prec_{rw} x_{c+1}$. In Figure 4, for example, after T_3 reads z_2 and writes y_2, the update of z from z_2 into z_3 by T_4 induces $y_2 \preceq_{ru} z_3$. Since T_3 that created y_2 precedes T_4 that created z_3 in the serialization order, the read bound version of y for z_3 whose version number will be stored in $RBV_{z_3}[y]$ must be y_2. To reflect this fact when executing T_4 and computing RBV of z_3, we store in $NRBV_{z_2}[y]$ the version number of y_2 that was created by T_3 that read z_2, and then we make use of it later when determining the element of RBV of z_3.

3.3. Update Transactions

In this section, we describe how to manage version vectors when executing update transactions. An update transaction is issued from any client site and that site serves as the coordinator for the transaction. All read and write operations requested from the client site are transferred to and executed on the primary copies of the target data. A write operation is also transferred to the secondary copies of the data item in the home cluster. When all operations of an update transaction are issued and executed, the 2PC protocol starts at the client site with the participants, namely, all of the primary copies and the secondary copies that participated in the transaction. The additional computations to update version vectors at each site and the transmissions of necessary information between the client site and the participant site are included in the 2PC protocol as follows.

Phase 1: The client sends the PREPARE message to all the participants. The primary copy sites that can commit send the following data with the VOTE_COMMIT message to the client.

- RBVs of all the primary copies on which read or write operations were executed
- $NRBV$s of all the primary copies on which write operations were executed

After the client receives the VOTE_COMMIT messages from all the participants, it determines a new version vector $NewRBV$, which is defined by

$$NewRBV[x] = \begin{cases} RBV_{x_c}[x] + 1 \quad \text{(if } x_{c+1} \in V_w(T)) \\ \\ max\{RBV_{y_c}[x], RBV_{z_c}[x], NRBV_{z_c}[x]\} \\ for\ \forall y_c \in V_r(T)\ and \\ \forall z_{c+1} \in V_w(T) \quad \text{(otherwise)} \end{cases}$$

where $x_c \prec_{ww} x_{c+1}$ and $z_c \prec_{ww} z_{c+1}$.

Phase 2: When the result of the transaction is determined to COMMIT at the client, it sends $NewRBV$ to all the participants with the COMMIT message. Each site that received the message updates RBV of the primary copy or a secondary copy as well as $NRBV$ of the primary copy as follows.

```
for $\forall x_c \in V_w(T)$ and $\forall y \in DS$
    $RBV_{x_c}[y] = NewRBV[y];$
    $NRBV_{x_c}[y] = 0;$
for $\forall y_c \in V_r(T)$ such that $y_{c+1} \notin V_w(T)$ and $\forall y \in DS$
    $NRBV_{y_c}[y] = max\{NRBV_{y_c}[y],\ NewRBV[y]\};$
```

Then the participant site sends an acknowledgment to the client and commits the transaction. The client finishes the 2PC protocol after it receives the acknowledgments from all the participants. Meanwhile, update propagation starts from each primary copy which was updated by this transaction to the tertiary copies contained in the other clusters. In this propagation, the new data value and the new RBV are sent, and a local update transaction is originated at each tertiary copy site. This update propagation is assured of being executed only once and within a finite period of time by an order-preserving and eventual delivery mechanism.

3.4. Read-only Transactions

In our method, all read operations in a read-only transaction can be executed on any replica. If all reads are executed on the primary copies, they always get the recent values of data by concurrency control processes in the primary copy sites. But if a read operation in a transaction is executed on a non-primary copy, the results of all read operations in the transaction must be validated before the transaction commits in order to guarantee the insular consistency criterion to be satisfied. Therefore, our strategy for executing a read-only transaction operates under the optimistic assumption that at least insular consistency can be mostly satisfied in the history consisting of it and other transactions. The validation rule for deciding whether a read-only transaction T_r can commit or not is as follows.

Validation Rule:

```
**if** for any two versions $x_i$ and $y_j$ in $V_r(T_r)$,
    $RBV_{x_i}[x] \geq RBV_{y_j}[x]$ and $RBV_{y_j}[y] \geq RBV_{x_i}[y]$
```

```
then commit $T_r$
else abort $T_r$
```

The condition in the above rule means that, for each version of a data item read in a read-only transaction, all versions of the other data items that were read with it are the same versions as the read bound versions of the data items in its RBV, or the later versions than them. In Figure 4, for example, we suppose a read-only transaction T_r that reads both y and z selects y_1 and z_3. Then $RBV_{y_1}[y] = 1$ and $RBV_{z_3}[y] = 2$ lead to $RBV_{y_1}[y] < RBV_{z_3}[y]$, and insular consistency is violated since a cycle $T_r \rightarrow T_3 \rightarrow T_4 \rightarrow T_r$ is generated in the history. On the other hand, if T_r chooses y_3 and z_1, $RBV_{z_1}[z] = 1$ and $RBV_{y_3}[z] = 2$ lead to $RBV_{z_1}[z] < RBV_{y_3}[z]$, and insular consistency is also violated because a cycle $T_r \rightarrow T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_r$ is generated in the history. However, reading a pair of y_3 and z_3, or y_2 and z_3, or y_3 and z_2 in T_r satisfies the condition of the **Validation Rule**, and the transaction can commit with insular consistency maintained. If the results of a read-only transaction fail in the validation process, we should re-execute the transaction on other replicas after aborting it.

4. Proof of Correctness

The correctness criterion which is used in our replication method is insular consistency, which requires any history consisting of each read-only transaction and all update transactions should be one-copy serializable. In our method, all read and write operations in update transactions are executed first on the primary copies, so that all update transactions are guaranteed to be one-copy serializable by concurrency control processes in the primary copy sites. In this section, we prove that our method guarantees insular consistency for any execution of transactions by showing that a read-only transaction passed the **Validation Rule** of the previous section satisfies *1SR* with all update transactions.

Lemma 1 *for* $\forall x_i \in V_w(T_i)$, $\forall y_j \in V_w(T_j)$ *such that* $T_i, T_j \in H$ *and* $T_i \neq T_j$,

$$x_i \preceq^*_{nf} y_j \iff T_i \rightarrow^* T_j \in SG(H)^3$$

PROOF
(*If*) We can prove by the mathematical induction on the path length from T_i to T_j in $SG(H)$.

Basis of induction. If $T_i \rightarrow T_j \in SG(H)$, by the definition of the serialization graph, there must exist at least one of three conflicts, i.e., a write-read conflict, a read-write conflict, and a write-write conflict, between T_i and T_j.

1. write-read conflict:
   There exists a version of a data item z, z_i, such that $z_i \in V_w(T_i) \cap V_r(T_j)$. $x_i =_w z_i$ and $z_i \prec_{wr} y_j$ infer $x_i \preceq^*_{nf} y_j$.
2. read-write conflict:
   There exist two versions of a data item z, i.e., z_{j-1} and z_j, such that $z_{j-1} \in V_r(T_i)$ and $z_j \in V_w(T_j)$. $z_{j-1} \prec_{ww} z_j$ and $z_{j-1} \prec_{wr} x_i$ infer $x_i \prec_{rw} z_j$, and it and $z_j =_w y_j$ subsequently infer $x_i \preceq^*_{nf} y_j$.
3. write-write conflict:
   There exist two versions of a data item z, i.e., z_i and z_{i+1}, such that $z_i \in V_w(T_i)$ and $z_{i+1} \in V_w(T_j)$. $x_i =_w z_i$, $z_i \prec_{ww} z_{i+1}$, and $z_{i+1} =_w y_j$ infer $x_i \preceq^*_{nf} y_j$.

Induction step. we suppose that $T_i \rightarrow^n T_j \in SG(H) \Longrightarrow x_i \preceq^*_{nf} y_j$ for any positive integer n. If $T_i \rightarrow^{n+1} T_j \in SG(H)$, there exists an update transaction T_k such that $T_i \rightarrow^n T_k \in SG(H)$ and $T_k \rightarrow T_j \in SG(H)$. By the above assumption and *Basis of induction*, $x_i \preceq^*_{nf} z_k$ and $z_k \preceq^*_{nf} y_j$ for $\forall z_k \in V_w(T_k)$. Therefore, we have $x_i \preceq^*_{nf} y_j$ from the transitiveness of the relation $\preceq^*_{nf}$.
(*Only if*) We can prove by the mathematical induction on the number of times of the relational products of $\preceq_{nf}$.

Basis of induction. if $x_i \preceq_{nf} y_j$, namely, $x_i \prec_{wr} y_j$ or $x_i \prec_{ww} y_j$ or $x_i \prec_{rw} y_j$, $T_i \rightarrow T_j \in SG(H)$ by the definition of the serialization graph and the meanings of the relations $\prec_{wr}$, $\prec_{ww}$, and $\prec_{rw}$.

Induction step. For any positive integer n, we let the *n*-th transitive extension of $\preceq_{nf}$ be $\preceq^n_{nf}$ and assume $x_i \preceq^n_{nf} y_j \Longrightarrow T_i \rightarrow^* T_j \in SG(H)$. If $x_i \preceq^{n+1}_{nf} y_j$, (1) $x_i \preceq^n_{nf} y_j$ or (2) there exists z_k satisfying $x_i \preceq^n_{nf} z_k$ and $z_k \preceq^n_{nf} y_j$. In the case of (1), by the above induction hypothesis, we have $T_i \rightarrow^* T_j \in SG(H)$. In the case of (2), by the above induction hypothesis, we have $T_i \rightarrow^* T_k \in SG(H)$ and $T_k \rightarrow^* T_j \in SG(H)$. Consequently, we have $T_i \rightarrow^* T_j \in SG(H)$. □

Theorem 1 *a read-only transaction* T_r *in* H *passes the* **Validation Rule** *if and only if the serialization graph of the sub-history* H_s, $SG(H_s)$, *consisting of* T_r *and all update transactions in* H *is acyclic.*

PROOF
(*If*) Suppose that T_r fails to satisfy the validation condition. Then, as Figure 5-(a) shows, there exist two versions, i.e., $x_i \in V_w(T_i)$ and $y_k \in V_w(T_k)$, such that $x_i, y_k \in V_r(T_r)$ but $RBV_{y_k}[x] = VN(x_j)$ where $x_j \in V_w(T_j)$ and $VN(x_i) < VN(x_j)$, that is, $RBV_{x_i}[x] < RBV_{y_k}[x]$. From $x_i \in V_r(T_r)$, we have

$$T_r \rightarrow T_j \qquad (1)$$

^{3} $SG(H)$ denotes the serialization graph[3] for a history H, and $T_i \rightarrow^* T_j \in SG(H)$ means that there exists a path from T_i to T_j in SG(H).

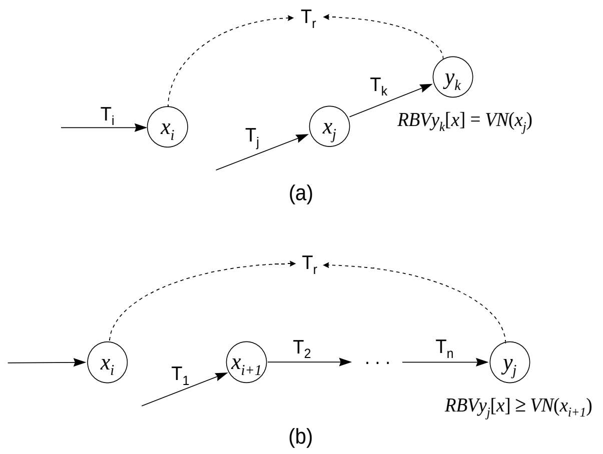


Figure 5. The proof of Theorem 1

$RBV_{y_k}[x] = VN(x_j)$ infers $x_j \preceq_p^* y_k$ by the definition of RBV, and by **Lemma 1** we have

$$T_j \rightarrow^* T_k \;\; or \;\; T_j = T_k \qquad (2)$$

In addition, $y_k \in V_w(T_k) \cap V_r(T_r)$, namely, there exists a write-read conflict between T_k and T_r on y_k, so that we have

$$T_k \rightarrow T_r \qquad (3)$$

From (1), (2), and (3), we have $T_r \rightarrow T_j \rightarrow^* T_k \rightarrow T_r$ or $T_r \rightarrow T_j (= T_k) \rightarrow T_r$. Hence, there exists a cyclic path in $SG(H_s)$, which is a contradiction.

(*Only if*) Suppose that there exists a cyclic path in $SG(H_s)$. Since the sub-history of all update transactions is one-copy serializable, T_r must be included in that cyclic path. Without loss of generality, let that cyclic path be $T_r \rightarrow T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T_r$. From $T_r \rightarrow T_1$, there exists a data item x such that $x \in R-Set(T_r) \cap W-Set(T_1)$. As Figure 5-(b) shows, if we choose $x_i \in V_r(T_r)$ then $x_{i+1} \in V_w(T_1)$. From $T_n \rightarrow T_r$, there exists y_j such that $y_j \in V_w(T_n) \cap V_r(T_r)$. Since $T_1 \rightarrow^* T_n$, $x_{i+1} \preceq_p^* y_j$ by **Lemma 1**, and then by the definition of RBV, $RBV_{y_j}[x] \geq VN(x_{i+1}) = RBV_{x_i}[x] + 1$. Consequently, we have $RBV_{y_j}[x] > RBV_{x_i}[x]$ for $x_i, y_j \in V_r(T_r)$, so that T_r does not satisfy the validation condition, which is a contradiction. □

The above **Theorem 1** and the serializability theorem of [3] directly infer the next corollary.

Corollary 1 *If all read-only transactions in a history H pass the **Validation Rule**, H satisfies insular consistency.*

5. Discussion and Conclusion

In this paper, we proposed a replication method which is applicable to large-scale distributed database systems. While our method bases on the traditional primary copy method, it uses insular consistency as a correctness criterion for the execution of transactions. Moreover, It updates synchronously only the replicas in the cluster that contains the primary copy and then asynchronously propagates the update to the other replicas. Our method can improve data availability for read-only transactions more than other traditional methods can do.

We introduced the notion of version vectors to maintain in each replica the necessary information to guarantee insular consistency for execution of transactions. By integrating all necessary transmissions of information between sites into the general 2PC protocol, our method does not need any extra phase for exchanging messages related to version vectors during the execution of a transaction.

The characteristics of our method in regard of four important measures are discussed as follows.

- Consistency: Insular consistency is a correctness criterion that can be used more generally than other application-specific correctness criteria. It guarantees *1SR* for any execution of transactions including a read-only transaction and all update transactions. This implies the result of a read-only transaction is the values of data items in a database in a feasible consistent state. Using insular consistency can improve data availability for a read-only transaction by ignoring its relation with the other read-only transactions.

- Availability: Traditional replication methods guaranteeing *1SR* have a drawback that they degrade data availability when a network partition failure occurs. In our method using insular consistency, many read-only transactions that cannot be executed to the end with other replica control protocols because of a network partition failure can be executed and committed, so that data availability can be much improved.

  An update operation in an update transaction cannot be executed if the primary copy of a target data is inaccessible. That is a restriction of the primary copy method that guarantee *1SR*. While our method, having regard to update performance, uses the asynchronous update scheme for replicas out of the home cluster, it copes with a site failure of the primary copy by keeping the secondary copies in the home cluster equivalent to the primary copy. As we suppose in this paper, when more update requests are issued within the home cluster than from the other clusters, we can maintain update availability in the home cluster in a high degree as compared with that in the other clusters.

- Performance: By using an asynchronous update propagation scheme for replicas remote from the primary copy, response time of transactions can be highly improved. On the other hand, throughput of transactions is closely related to data availability. Since our method improves data availability particularly for read-only transactions as mentioned above, it can also increase throughput of the system where read-only transactions are predominantly issued.

- Storage and communication cost: The additional data structures that are used for guaranteeing insular consistency like version vectors inevitably lead to an increase in the amount of storage and communication messages. Their overheads are as follows.

  1. Storage cost:
     The storage cost of version vectors in the primary copy is $O(2n)$ and that of a secondary or tertiary copy is $O(n)$, where n is the number of replicated data items.
  2. Communication cost of read-only transactions:
     The amount of the additional messages transmitted is $O(mn)$, where m is the average number of data items read by a read-only transaction.
  3. Communication cost of update transactions:
     When we denote the average numbers of read operations and write operations in the update transaction as r and w respectively, and denote the numbers of secondary copies and tertiary copies as s and t respectively, the amount of the additional messages transmitted at commit is as follows:

     cost in Phase 1 + cost in Phase 2 + cost of update propagation
     $= O(rn + 2wn) + O(wn(s + 1)) + O(nwt)$
     $= O(rn + 2wn + wnd)$

     where d is the average number of replicas of a data item, i.e., $d = s + t + 1$.

  As represented above, the storage and communication cost of our method depends on the number of replicated data items and the number of replicas. The number of replicated data items is closely related to the granularity of replication. Generally, it can be variously determined by applications, and our method is appropriate for a relatively coarse granularity such as a fragment or a relation. As for the number of replicas, a trade-off with data availability is needed.

On the other hand, our method can be incorporated with other traditional replica control protocols for improving data availability of read-only transactions against network partition failures.

As for further works, It is needed to define more practical and more useful correctness criteria to improve performance of data replication. In addition, considerations about scalability of a system are required to design a replication method for large-scale distributed systems.

References

[1] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proc. of 2nd IEEE Int'l Conf. on Software Engineering*, pages 627–644, 1976.

[2] P. Aristides and A. Abbadi. Fast read-only transactions in replicated databases. In *Proc. of 8th IEEE Int'l. Conf. on Data Engineering*, pages 246–253, Tempe, AZ, Feb. 1992.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proc. of ACM-SIGMOD Int'l Conf. on Management of Data*, pages 112–122, 1990.

[5] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, Sept. 1985.

[6] S. Dietzen. Distributed transaction processing with Encina and the OSF DCE. Technical report, Transarc Corporation, Sept. 1992.

[7] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Trans. on Database Systems*, 7(2):209–234, June 1982.

[8] D. Gifford. Weighted voting for replicated data. In *Proc. of 7th Symp. on Operating System Principles*, pages 150–162, Dec. 1979.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of ACM-SIGMOD Int'l Conf. on Management of Data*, pages 173–183, Montreal, Canada, June 1996.

[10] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proc. of 7th ACM Symp. on Principles of Distributed Computing*, Toronto, Canada, Aug. 1988.

[11] R. Schlichting and F. Schneider. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Trans. on Computer Systems*, 1(3):222–238, 1983.

[12] W. Weihl. Distributed version management for read-only actions. *IEEE Trans. on Software Engineering*, 13(1):55–64, Jan. 1987.