A token-based independent update protocol for managing replicated objects

Wanlei Zhou* and Weijia Jia†

*School of Computing and Mathematics, Deakin University, Burwood, VIC 3125, Australia. Email: wanlei@deakin.edu.au
†Department of Computer Engineering and IT, City University of Hong Kong, Kowloon, Hong Kong. Email: itjia@cityu.edu.hk

This paper presents the design and evaluation of a token-based protocol supporting independent updates of replicated objects. The paper makes three major contributions. Firstly, a token is used to simplify the control of update propagation and ordering. Secondly a partial commit state is introduced to support independent updates and to enhance the efficiency of transaction processing. Thirdly, a data partition methodology based on priorities is adopted to reduce update conflicts. We evaluate our protocol in comparison of two existing replication-control protocols in terms of transaction processing efficiency.

Keywords: distributed systems, transaction processing, replication control protocol, fault tolerance, performance evaluation

1. INTRODUCTION

Replication is the key to providing high availability, fault tolerance, and enhanced performance in a computing system [17, 39]. Existing strategies used for managing replicated objects can be divided into two levels. At the system level, many strategies use multicast group communication systems to provide a run-time system support or toolkit for simplifying the design and implementation of reliable applications on replicated objects. Notable projects include Totem [2] from UCSB, ISIS and Horus [7] from Cornell University and Transis [13] from Hebrew University. At the business level, many strategies use transactional models to provide a framework for building reliable business operations on replicated objects. Notable transactional models include disjoint transactions, commutative transactions, timestamped transactions [16], and fulfillment transactions [28].

However, although considerable research effort have been directed towards the design of replication-control protocols, replication is still viewed as a 'necessary evil' [34, 15, 1]. Update propagation and transaction management in an environment of replicated objects are complicated issues. Most existing replication-control protocols are either inefficient or too complicated to be implemented [9, 12]. The efficiency and ease of implementation are still the main obstacles to the widely use of replication techniques.

1.1 An overview of the token-based independent update protocol

This paper proposes an approach based on the business level strategies for supporting efficient independent updates and maintaining transaction atomicity. Our proposed protocol is called the token-based independent update (TBIU) protocol [36]. We arrange all sites containing replicated objects as a logical ring and let a token circulate among these sites. Each site can execute transactions on a subset of their replicated objects locally, resulting partially committed transactions. Transactions accessing data objects outside the subset have
to be queued up until the site has the token. Only the site holding the token is allowed to propagate transactions to all sites. Compared with existing business level strategies, the main advantages of this scheme are (1) efficient: it does not use the distributed 2PC protocol, only the local 2PC protocol is used; (2) no global rollbacks: when there are conflict operations, only the partially committed transactions may be rolled back locally; (3) easy to implement: the proposed scheme is built on top of an existing efficient and reliable atomic multicast protocol [22].

We use a simple example, as shown in Figure 1, to illustrate our TBIU protocol throughout the paper. Assume some engineers are designing a new car. The car is divided into three components: engine, transmission and body. Each component consists of many parts. Engineers are also divided into three groups: the engine group, the transmission group, and the body group, where each group is responsible of the design of a component.

Now the objects of the car is replicated in three inter-connected sites: an engine site, a transmission site, and a body site. Engineers of each group connect their workstations to the corresponding site and they issue transactions to update their design. Each site contains the data about the whole car, but has different policies towards different types of data. For example, engineers of the engine group have a higher priority than engineers of other groups to update data about the engine.

The TBIU protocol works like this. A token circulates along the three sites. Transactions submitted by engineers of the engine group will be executed immediately if they only update data about the engine. These transactions are regarded to be partially committed. Otherwise, the submitted transactions will be queued up until the token arrives at the engine site. The same policy holds for the transmission site and the body site. When it has the token, the engine site propagates all transactions submitted to it to other sites. All partially committed transactions propagated by the engine site have to be committed by all sites, while other transactions without being partially committed will then be executed and results returned.

1.2 Related work

Most database operations are organised as transactions, which provide a way to structure interactions between autonomous agents in a distributed system with some exactly-once guarantees [26]. In classical research literature, transactions are characterised to have the properties of atomicity, consistency, isolation, and durability (ACID). The two-phase-commit (2PC) protocol is the most widely used protocol in commercial databases to maintain the ACID properties of transactions. Such a protocol has some intrinsic disadvantages, such as cost, delay, and reduced availability [4, 31]. Many applications, such as flight reservation system, inventory control system, computer-supportive cooperative system, etc., do not really require such restricted properties [25, 11]. Based on the classical transaction model, many novel transaction models have been proposed in recent research literature. Most of them are motivated by workflow and design applications, where long-lived activities need to be structured into semi-independent atomic steps [6, 37]. However, the efficiency and ease of implementation are still the main obstacles to the widely use of replication techniques [30].

Many update protocols for replicated databases and replicated objects have been developed [18, 33]. In the copy token protocol [29], only the replica holding the token is allowed to perform updates; it does not allow a replica without a token to perform an update. This protocol can meet the first target of tolerating certain failures, but it cannot support the second target well since it delays the processing of a transaction until a site becomes the token holder. This is a special case of our protocol where no access priority is defined.

The independent update protocol [11, 10] allows all replicas to proceed updates immediately and propagate them to all replicas. A reconciliation mechanism is used to bring all sites which cannot commit updates to the final state. This scheme is suitable if the majority of the updates do not access the same data objects. Otherwise, there will be a serious performance degradation during the reconciliation process since the reconciliation process involves some network communications. If the access priority of each site is the whole database and there is no token, the TBIU protocol can be reduced into the independent update protocol.

In the lazy replication protocol [25, 8] the primary is updated first, the update is then propagated to non-primaries asynchronously. The problem with this protocol is that the primary site is fixed and is the single-point of failure. Therefore it does not meet the first goal well. Instead of fixing the primary site, the TBIU protocol allows each site to have a chance to be the primary through the circulation of the token.

The location-based protocol [34] uses location information to facilitate the efficient updates to replicated copies. A location server lets the clients choose a leader replica which is supposed to be the closest site to execute a transaction and to send reply immediately afterwards. The leader replica is responsible for keeping consistency among all the replicas in a lazy manner. The protocol provides reliability similar to quorum-based replication protocols but with transaction delay similar to a one-copy system. However, the proposal cannot deal with network partitions properly. Another paper by the same authors [35] addressed the issue of achieving high availability in a partitioned distributed system through the use of transaction and replication techniques.

In quorum consensus protocols [14, 19, 20], a number of
read (write) quorum sites has to agree with the operation of a transaction before the operation can be proceeded. The read and write quorums are designed in such a way that they ensure that there is a nonnull intersection between any read quorum and any write quorum for a particular data item. As a result, whenever a read/write conflict occurs, it will be detected. However, mutual consistency is not maintained and replicas may have different values if the write quorum is less than the number of replicas.

The Coda File System [23] supports the replication of file volumes, managing the resulting multiple-update problem even in the presence of server failures. The replication of file volumes produces a fault-tolerant service. The most successful feature of the Coda system is its support for disconnected operations. However, the reintegration of files after a disconnection involves manual intervention in the case of conflicting updates.

The epsilon-serialisability (ESR) [32] allows some bounded slack of consistency among replicas. When this inconsistency slack converges to zero, ESR reduces to serialisability. The inconsistent data can be seen and controlled, and eventually converge to a consistent state. A set of protocols can be designed using the ESR concept.

In the quasi-copies [3] and the demarcation protocol [5], data values are allowed to have certain bounded deviation from the actual data value. Some degree of independence are allowed among sites, allowing some updates to execute locally and be propagated asynchronously to other sites.

The remainder of the paper is organised as follows. Section 2 introduces the replicated object model, the transaction model, and the failure semantics. Section 3 presents the architecture of the transaction management system. Section 4 describes the algorithms for the TBUI protocol. Section 5 presents some results of a correctness study of the TBUI protocol. Section 6 presents a few examples to explain the execution the TBUI protocol. Section 7 discusses the performance of the protocol. Section 8 concludes the paper.

2. SYSTEM MODELS

2.1 Replicated data and operations

We model a replicated distributed database system as consisting of many sites interconnected by a communication network, offering certain services to potential clients. A service is provided by a group of replicated objects located on some sites. These replicated objects manage some common data that can be shared by many clients. For simplicity, we assume that each replicated object knows the location of other replicated objects that store the same data. This assumption can be loosened if a replicated directory service is used.

We take an object-oriented view to model replicated distributed database systems in hoping that the proposed protocol will have a wider application. We first give two definitions on the objects in a replicated database system.

Definition 1 (logical objects)

A replicated distributed database system provides a set of (m) services: \( S = \{ S_1, S_2, \ldots, S_m \} \). Each service \( S_i \) is implemented through a logical object \( R^i = \{ P^i, D^i \} \), where \( D^i = \{ d^i_1, d^i_2, \ldots, d^i_S \} \) is the set of data managed by \( R^i \) and \( P^i = \{ p'^i_1, p'^i_2, \ldots, p'^i_s \} \) is the set of methods provided by \( R^i \) to manipulate \( D^i \).

We use \( R = \{ R_1, R_2, \ldots, R_m \} \) to denote the logical objects of a replicated database system and \( D_L = \{ D^1, D^2, \ldots, D^m \} \) to denote the data items of \( R \).

Using our car design example we can define: \( R = \{ R^1, R^2, \ldots, R^S \} \), where \( R^i, R^j, R^k \) are the sets of logical objects of the car, the engine, the transmission, and the body, respectively. If we further define the parts for each of the components as:

\[
D^1 = \{ e_1, e_2, e_3, e_4 \}, \quad D^2 = \{ x_1, x_2, x_3, x_4, x_5 \}, \quad D^3 = \{ b_1, b_2, b_3 \}
\]

then we have \( D_L = \{ D^1, D^2, D^3 \} \).

Definition 2 (physical objects)

Each logical object \( R^i \) is supported by a set of \( (k) \) replicated physical objects, where \( R^i = \{ R^{i_1}, R^{i_2}, \ldots, R^{i_k} \} \). Each replicated physical object \( R^{i_j} \) also has two parts, \( R^{i_j} = \{ P^{i_j}, D^{i_j} \} \), where \( D^{i_j} = \{ d^{i_1}_{j_1}, d^{i_2}_{j_2}, \ldots, d^{i_S}_{j_S} \} \) is the set of data managed by \( R^{i_j} \) and \( P^{i_j} = \{ p'^{i_1}_{j_1}, p'^{i_2}_{j_2}, p'^{i_3}_{j_3} \} \) is the set of methods provided by \( R^{i_j} \) for manipulating \( D^{i_j} \).

To simplify our description, we assume that the replicated database system has \( m \) sites (computers) \( \{ A_1, A_2, \ldots, A_m \} \) and a site has a physical object for every logical object. That is, each logical object \( R^i \) has \( m \) physical objects: \( R^i = \{ R^{i_1}, R^{i_2}, \ldots, R^{i_m} \} \). The physical objects running on site \( A_i \) can be expressed as \( \{ R^{i_1}, R^{i_2}, \ldots, R^{i_m} \} \), where \( n \) is the number of services (or, equivalently, the number of logical objects) of the system. Therefore the meaning of \( R^{i_j} \) is the physical object of the \( i \)th logical object running on site \( A_i \). The consistency constraint requires that (eventually) \( R^{i_j} \equiv R^i \) or equivalently, \( D^{i_j} \equiv D^i \), for \( i = 1, 2, \ldots, n \) and \( j = 1, 2, \ldots, m \).

In our example, we have three sites: \( \{ A_1, A_2, A_3 \} \), where \( A_1 \) is the engine site, \( A_2 \) is the transmission site, and \( A_3 \) is the body site. The physical objects stored on each site can be defined as:

\[
A_1: \{ R^{11}, R^{21}, R^{31} \} \\
A_2: \{ R^{12}, R^{22}, R^{32} \} \\
A_3: \{ R^{13}, R^{23}, R^{33} \}
\]

where \( R^{11}, R^{12}, R^{13} \) are the physical objects of logical object \( R^1 \) (the engine object), \( R^{21}, R^{22}, R^{23} \) are the physical objects of logical object \( R^2 \) (the transmission object) and \( R^{31}, R^{32}, R^{33} \) are the physical objects of logical object \( R^3 \) (the body object). The consistency constraint requires that (eventually):

\[
R^{11} \equiv R^{12} \equiv R^{13} \equiv R^1, \\
R^{21} \equiv R^{22} \equiv R^{23} \equiv R^2, \text{ and} \\
R^{31} \equiv R^{32} \equiv R^{33} \equiv R^3
\]

A site can be called a replica of the service or simply a replica since it contains a physical object for each logical object. We use these two terms (site and replica) interchangeably.

After we make a call to a method supported by a replica,
the call may return one of the following states (we ignore the computation results returned by the call):

**OK** This means that no failure occurred during the method's execution and the access to the data is successful.

**FL** This means that a failure occurred during the execution and the method was not executed. For example, if a transaction tries to withdraw more money than it is allowed from a bank account. Or a transaction tries to sell more units than the amount of current stock for some merchandise.

**TO** This means a time-out. We use this to indicate the replica is disconnected or is down.

Now we define the method calls to a replicated database system.

**Definition 3** (a method call or simply a call)

Let \( P = \bigcup_{i=1}^{n} P_i \) and \( D = \bigcup_{i=1}^{m} D_i \), for \( i = 1, 2, ..., n \) and \( j = 1, 2, ..., m \), then a method call to the replicated distributed database can be expressed as a mapping:

\[
c : P \times D \rightarrow \{ OK, FL, TO \}
\]

That is, \( c(p, d) \) \( p \in P, d \in D \) is a call which uses method \( p \) to manipulate data item \( d \). Without loss of generality we assume that all methods are update-oriented operations. That is, if \( c(p, d) \) is successful, it transforms the data item \( d \) from the existing state to a new state.

2.2 The transaction model

The following definition defines the transaction allowed in our replicated database system.

**Definition 4** (transaction)

We define a transaction as

\[
T = \{ c_1(p_1, d_1), c_2(p_2, d_2), ..., c_k(p_k, d_k) \}
\]

where \( c(p, d) \) is a method call, and \( p_i \in P, d_i \in D \). We express the set of all methods and the set of all data items of \( T \) as \( T.P = \{ p_1, p_2, ..., p_k \} \) and \( T.D = \{ d_1, d_2, ..., d_k \} \), respectively. The semantics of a transaction is that after issuing the transaction, the state of the transaction will be in one of the following:

- **OK**: there is no error occurs and all \( c_i \) of \( T \) have been executed on all sites (commit or OK).
- **FL**: if any one of the \( c_i \) fails, all executed method calls of \( T \) will be rolled back (abort or FL), and
- **PC**: the transaction is successfully executed on the local site, waiting to be propagated to other sites.

We assume that all method calls \( c(p, d) \in T \) are commutative, i.e. there is no ordering among these calls and they can be executed in parallel. The effect of ordered operations in a distributed replication system is investigated in [38].

In addition to the two normal states (commit and abort), we define a partial commit (PC) state. A call \( c(p, d) \) returns PC if and only if it has been performed locally within one replica and will be propagated to other replicas later. When the transaction is propagated to other replicas, the client which issued the transaction will be notified of the final result (OK or FL).

If we do not need to distinguish individual calls within a transaction \( T \), we then write \( T = c_i(p_i, d_i) \).

A client who issued a transaction \( T \) knows that when \( T \) returns OK (committed), then the transaction has transferred the replicated database from a consistent state to another consistent state for all alive replicas. That is, let \( c_i(p, d) \in T \) and \( d_i = d_i^k \in D_i^k \), then after the transaction, we have, \( d_i^k = d_i^l \in D_i^l \) for all alive replicas of \( d_i^l \). If \( T \) returns FL (aborted), then the replicated database is not changed by the transaction, i.e. we still have \( d_i^k = d_i^l \in D_i^l \) for all alive replicas.

When a failed replica re-joins the service, the system is responsible to make sure that the recovering replica performs all missed transactions in the right order. If \( T \) returns PC, then the data affected by \( T \) are in a temporary inconsistent state, i.e. \( \exists j, d_i^k \neq d_i^j \in D_i^j \). The transaction management system will be responsible to solve the temporary inconsistency and the client is warned that the use of the temporary inconsistent data may have some danger.

2.3 Access priority

In applications such as flight reservation systems, inventory control systems, and computer cooperative work systems, different transactions may have different priorities to access various data items. We associate such priorities to transactions originated from sites.

**Definition 5** (access priority)

Let \( A_i \) be a site of the replicated database system and \( D_i = \bigcup_{j=1}^{m} D_j \) be the set of data of all logical objects. Let \( O_i \subseteq D_i \),

We define that site \( A_i \) has an access priority to data items in \( O_i \), if all transactions originated from \( A_i \) have a higher or at least equal priority to update data items in \( O_i \) than transactions originated from other sites of the replicated database system. \( O_i \) is called the access priority set of \( A_i \).

The access priority expresses the possibility of a transaction being committed on a site. Let \( T_1 = \{ c_1(p_1, d_1) \} \) and \( T_2 = \{ c_2(p_2, d_2) \} \) be two transactions originated from sites \( A_j \) and \( A_i \), respectively. Then \( T_1 \) has a greater or equal chance to commit in \( A_j \) than \( T_2 \).

**Definition 6** (degree of shared priority)

Let \( O_i \) and \( O_j \) be the access priorities of site \( A_i \) and site \( A_j \), respectively. We call \( \sigma_i = O_i \cap O_j \) the degree of shared priority for \( A_i \) and site \( A_j \).

The degree of shared priority shows the possibility that a transaction partially committed on one site may conflict with a transaction partially committed on another site. One extreme of the degree of shared priority is that \( O_j = \emptyset \) for all \( j \), which means that there is no access priority defined for any site. Or, \( \sigma_j = \emptyset \) for all sites \( A_i \) and site \( A_j \), \( n \neq j \). In that case, all access priorities are partitioned. Another extreme is that \( \sigma_j = D_i \), where every site has the same access priority, which is the total set of logical objects.

In our example we have defined the logical objects as \( D_l \).
\[ D^1, D^2, D^3 \]. If we define the access priorities for site \( A_1, A_2 \) and \( A_3 \) as

\[ O_1 = D^1, \quad O_2 = D^2 \quad \text{and} \quad O_3 = D^3 \]

then we have

\[ \sigma_{ij} = O_i \cap O_j = \emptyset, \quad i, j = 1, 2, 3, i \neq j \]

If transactions \( T_1.D = \{ e_1, e_3 \} \) (i.e. \( T_1 \) uses the two data items on engine) and \( T_2.D = \{ e_3, x_1 \} \) are originated from site \( A_1 \), then \( T_1 \) has a higher chance to commit in \( A_1 \) since \( e_1, e_3 \in O_1 \), but \( x_1 \notin O_1 \).

2.4 Failures

We consider three classes of failures in a replicated database system executing transactions: transaction abort, site-down, and network partition.

A transaction \( T \) aborts if any of its calls returns an FL (e.g. the requested update is not allowed), or \( T \) was in a PC state and then a propagated transaction accesses the same data items as \( T \) and \( T \) is then down-graded into an FL. The usual treatment for a transaction failure is to retry it after a random period of delay. The random delay is necessary to avoid oscillation when two or more transactions need to use the same data of a replica at the same time.

A site is down means that all physical objects running on that site are down. In that case, the site cannot accept any transactions from clients nor any propagated messages from other sites. During the time a site is down, other sites may have carried out some transactions. These missed transactions must be executed in the same order by the failed site when it recovers.

A network partition failure means that replicas of a service may belong to two disconnected partitions. In this case, the two sets of replicas should still provide the service to clients. However, when the two partitions are re-united, the two sets of replicas may have performed some conflicting operations. To solve this problem, we only let the partition with a majority of sites commit transactions. Other partitions can only partially commit transactions. When the partitions are re-united, those committed transactions during the partition period will be forced to commit in other partitions. While the partially committed transactions, if they have not been rolled back by the committed transaction from the majority partition, will be propagated to other partitions.

For simplicity, we only assume crash failures in this paper. Byzantine failures can also be dealt with when the number of replicas of a service is more than three and a majority voting scheme is used.

3. MANAGEMENT OF TRANSACTIONS

3.1 System components

Our TBIU protocol for transaction management in a replicated database system is token-based. A token circulates around all sites in an order say, \( A_1, A_2, \ldots, A_m, A_1, \ldots \). The token also carries a timestamp for ordering transactions. Only the token holder can propagate transactions to all sites. Other sites without the token can only locally process the transaction if the site has access priority on the transaction, resulting a partial commit state which may be rolled back if the transaction is conflict with a propagated transaction from the token holder. Each site has two queues that can only be accessed by system components:

- A transaction request queue (TRQ), ordered by their local arrival time, for all incoming transactions originated from that site.
- A PC transaction queue (PCQ), ordered by the local timestamp, for all partially committed transactions on that site.

The following system components running on site \( A_j, j = 1, 2, \ldots, m \) are used to manage transactions in our replicated database system:

- A transaction request manager (TrM) accepts a transaction

\[ T = \{ c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_k(p_k, d_k) \} \]

from clients. If \( d_i \in O_j \) for \( i = 1, 2, \ldots, k \), then \( T \) is executed locally and put into the PCQ. Otherwise, the transaction is queued up in the TRQ. The TrM algorithm is expressed as transaction_request_manager() in Section 4.1.

- A token-site manager (ToM) checks if \( A_i \) has the token or not. If it has, then it assigns global timestamps from the token to all PC transactions in the PC transaction queue (using its original order) and then assigns global timestamps to all transactions of the transaction request queue (using the same order). All transactions are ordered according to their global timestamps. These transactions are propagated to all sites, together with the message that the token has been assigned to the next site \( A_{i+1} \) (if \( i = m \) then to \( A_1 \)). The ToM algorithm returns an OK to all partially committed transactions and returns the execution results to all transactions without a PC state. The ToM algorithm is expressed as token_site_manager() in Section 4.2.

- A non-token-site manager (NTOM) has the functions of monitoring the liveness of the token holder, requesting a new token holder, acting as the new token holder, and most importantly, accepting transactions from the token holder and processing these requests according to the order given by the token holder. The NTOM algorithm is expressed as non_token_manager() in Section 4.3. If the NTOM finds out that a transaction propagated from the token holder is conflict with a PC transaction processed locally, then the local PC transaction has to be aborted and removed from the PC transaction queue. Aborting a PC transaction may also affect other PC transactions. The work for rolling back PC transactions is described in an algorithm called roll_back_transaction (RoT) and is expressed as roll_back_transaction() in Section 4.3.

When the logical ring has been established, each site runs the above system components as an endless loop, as shown in Listing 1.
Listing 1 The token-based independent update protocol

A client simply submits a transaction request to the TrM on a site and waits for the transaction result and the state. The returned state of a transaction at this stage can be one of OK, FL or PC. As soon as the client receives a returned state of OK or FL, it can continue with its next operation. That means, other operations, such as operations required for consistency, or for continuous committing will be dealt with by the system without the participation of the client. However, if a transaction returns PC, the client will be notified (via an asynchronous receive procedure, similar to an exception handler) later of the final state (commit or abort).

3.2 Data structures

We define some data structures and functions used in our algorithms. The use of these data structures and functions are exclusive.

For each queue (TRQ and PCQ), we define a head and a tail which point to the head and the tail of the queue, respectively. Each queue also has an enqueue() function that attaches a transaction to the tail of the queue and a dequeue() function which detaches a transaction from the head. These two functions will automatically update the values of the head and the tail.

We use a Partially Committed Transaction (PCT) table, which essentially is a checkpointing log, to record the events of partially committed transactions in a local site. The PCT table of each site is kept in stable storage [24]. Therefore, information stored in the PCT table will not be affected by system failures. The PCT table structure is listed in Listing 2, where TRANSACTION and DATA are the objects of transactions and data. We omit their definition here. We use a function previous(T) to compute the locations of existing PC transactions stored in the PCT table which affect T’s data items. Let X = previous(T), if T’ ∈ PCT and T’.D ∩ T.D ≠ ∅, then T’ ∈ X. That is, X contains those previously partially committed transactions which have updated some data items that T is now updating. Therefore, if T’ is to be rolled back, so does T. Another function, beforeImage(T.D), is used to compute all before images of transaction T.

The handler field stores the address of the asynchronous procedure used to notify the client of a transaction about the final result. If t ∈ PCT we use t.handler(OK) or t.handler(FL) to express the execution of the handler with the final result OK/FL. The corresponding PCT entry is deleted when the client has been notified.

Some sites may fail when a token holder propagates its transactions to all sites. We use a failed transaction (FAT) table to record all the sites and the propagated transactions that these sites have failed to execute. The FAT table has the data structure shown in Listing 3.

We use fat[A_j] to represent the FAT table contents for site A_j. The message is an object mainly used for packaging transactions and the token for transmission. If msg is a message object, then msg.append(T) is a function that appends transaction T to the end of msg and T = msg.getFirst() detaches the first available transaction of msg to T. After the initialization of the logical ring, fat[0].k will store the sequence number of the current token holder (assuming all A_i > 0).

A Token object is defined to hold the token and the information of the un-executed transactions by the failed sites. Let tk be a token object. Function tk.nextToken(A_j) sets the next transaction holder to the next site of the logical ring after A_j. Function tk.token(A) returns a boolean result, indicating if site A_j is the token holder. Without a parameter, tk.token() returns the site of the current token holder. Function tk.append(fat[A_j]) appends an FAT table item to the token object. Function tk.get() returns the information of the un-executed transactions at site A of the token. Function tk.newRing(flag, A) forms a new ring according to the flag. If flag=delete, then A_j is deleted from the existing ring; if flag=add, then A_j is added into the existing ring.

We attach two timestamp fields and a state field to each transaction T. The local timestamp, expressed as T.tls, is assigned by a site when the transaction is submitted to that site. The global timestamp, expressed as T.tgs, is assigned when the transaction is propagated by the token holder. The global timestamp is assigned by the token holder through the following function:

\[
T.tgs = tk.assignTS();
\]

where tk is a token object. And the token’s timestamp automatically increases after each assignment. The state, expressed as T.st, denotes if the transaction has been partially committed.

3.3 Transaction lifecycle

The lifecycle of a transaction in our TBIU protocol can be: 

Listing 2 The Partially Committed Transaction table
expressed in Figure 2. The dashed lines in Figure 2 represent asynchronous communications where the handler procedures defined in the PCT table are used.

A transaction $T$ is submitted by a client to the TrM of a site $A_j$. If $T$ can be executed locally, the TrM will perform $T$ locally. If $T$'s local execution is successful, the TrM will put $T$ into the PCQ, set up an entry for $T$ in the PCT table, and return a PC to the client immediately. If $T$'s local execution fails, an FL is returned. If $T$ cannot be executed locally, $T$ is put into the TRQ.

When $A_j$ gains the token, it uses the ToM algorithm to combine transactions in the PCQ and TRQ into a message $msg$ and sends $msg$ to all other sites. The message is ordered according to a global timestamp, with transactions in the PCQ ordered before transactions in the TRQ. The token holder also processes transactions with a PC state (i.e., in the PCQ originally) by returning an OK state to clients through the handler of each transaction stored in the PCT table. For transactions without a PC state (i.e., in the TRQ originally), the token holder will process the transactions locally using the local 2PC protocol and return the results back to clients.

When a non-token holder $A_i$ receives the above message $msg$ from the token holder, it uses the NToM algorithm to process the transactions contained in the $msg$ according to the order set by the token holder. Transactions with a PC state are executed before transactions without the PC state. All PC state transactions have to be forced to commit on $A_j$. That is, if $T \in msg$, $T_s = PC$, $T' \in PCQ$ of $A_j$ and $T.D \cap T'.D = \emptyset$, then $T'$ has to be rolled back. Also, any PC state transactions of $A_j$ which are affected by $T'$ have to be rolled back. That is, for any $T'' \in PCQ$, $T''.D \cap T'.D = \emptyset$, $T''$ has to be rolled back. The handlers in the PCT table are used to notify the clients that their transactions have been downgraded to FL. Transaction rollbacks are performed by algorithm RoT. The transactions in $msg$ without a PC state are executed on $A_j$ using the local 2PC protocol. The states of the execution (OK/FL) are ignored.

4. ALGORITHMS

4.1 The Transaction Request Manager

The transaction request manager has three basic functions to perform when it receives a transaction $T$:

- It checks the suitability of partial commit for the transaction. Transactions who only access data in the access priority set will be regarded suitable. According to the check result, the transaction will be either processed locally with a PC/FL state returned, or will be queued up in the TRQ.
- It uses the two-phase-commit protocol to process a transaction which is suitable for partial commit. All partial commit information is recorded in the log file (the PCT table) for further processing.
- It sets up an asynchronous receive procedure for the locally processed transaction $T$ which returns a PC. The procedure avoids messages regarding $T$ from ToM and the NToM. A message comes from the ToM about $T$ means that the current site is the token holder and $T$ has been propagated to all sites. So the procedure will up-grade the transaction to a commit state. If a message about $T$ from the NToM comes, that means $T$ needs to be rolled back and the procedure will down-grade the transaction to an abort.

Listing 4 shows the algorithm for the TrM. Here we assume the current site is $A_j$ and the access priority set of $A_j$ is $O_j$.

The TrM algorithm receives transactions from clients. The process suspends until a transaction comes. Incoming transactions are ordered using the local timestamps $T.its$. These local timestamps are used to queue up partially committed transactions in the PCT table. They can also be used to order local transactions when a timestamp-based local commit protocol is adopted.

When the data items accessed by an incoming transaction $T$ are within the access priority of the current site $A_j$, then $T$ is regarded to have a good chance to be committed. Therefore,

```c
transaction_request_manager()
{
    while (TRUE) { //
        receive(client, T = {c(p, d)});
        Assign local timestamp to T, denoted as T.its;
        if T.D \subseteq O_j // Can do it locally */
            process T locally using a local 2PC protocol;
        if (T is conflict with a propagated transaction)
            roll back T;
        switch (result of the local execution of T) {
            case (OK):
                form a PCT entry t with right t.ttl, t.ting, t.handler,
                t.pre and t.last fields;
                insert t into the PCT table;
                PCQ.enqueue(T);
                tell the client that the transaction returns PC;
                break;
            case (FL):
                tell the client that the transaction returns FL;
                break;
        }
        } else /* cannot do it locally, enqueue TRQ */
            TRQ.enqueue(T);
    }
}
```

Listing 4 Algorithm 1 – The TrM algorithm
fore \( T \) is executed locally. If the execution of \( T \) is successful, then a PCT table entry \( t \) for \( T \) is inserted into the PCT table. The fields for \( t \) are set as following:

\[
t.tr = T; \quad t.pre = \text{previous}(T.D); \quad \forall t' \in t.pre, \quad t'.\text{nxt} += t; \quad t.bimg = \text{beforeImage}(T.D); \quad t.\text{nxt} = 0;
\]

That is, \( t.pre \) points to all previously partially committed transactions \( T' \) that have updated some data items as \( T \) and one of the \( \text{nxt} \) pointers of all \( T' \) will point to \( t \).

\( T \) is also inserted into the the partially committed queue PCQ, awaiting to be propagated. The client is then informed that \( T \) is partially committed. \( T \) is aborted if the locally execution of \( T \) fails. \( T \) is inserted into the transaction request queue TRQ if it accesses some data items outside \( O \).

During the execution of the 2PC protocol, site \( A_j \) may be asked by the token holder to execute other transactions (the execution of transactions propagated by the token holder is done in the NToM algorithm, described in Section 4.3). If a propagated transaction and the local transaction has conflict in data access, then the local transaction is to be aborted, resulting an FL return in the local 2PC protocol of the TrM algorithm.

From the analysis of the TrM algorithm, we know the following assertion is true:

**Assertion 1**: A transaction \( T \) submitted to TrM has one of the following results:

1. TrM returns a PC where \( T \) is committed locally and \( T \) is queued up in PCQ.
2. TrM returns a FL and \( T \) is aborted, or
3. \( T \) is queued up in TRQ.

### 4.2 The Token-Site Manager

When a site has the token, the token site manager (ToM) is used to propagate and finalize transactions submitted through that site. Transactions in both PCQ and TRQ are to be propagated, while transactions in PCQ have a higher priority (smaller global timestamp). The timestamp carried by the token is assigned monotonically to every transaction (as the global timestamp) propagated by the ToM. Transactions in the PCQ are upgraded to OK state and transactions in the TRQ are processed locally and their results returned to the clients. The token is also passed to the next site of the logical ring by the ToM. Listing 5 presents the algorithm. In this algorithm, we assume that the current token holder is \( A_i \).

The ToM algorithm first assigns global timestamps \( T.gts \) to all transactions originated from the current site. The transactions that have been partially committed by the current site will have a smaller global timestamp than other transactions. The global timestamps will be used by each site to determine the execution order of transaction requests propagated by the token holder.

The ToM algorithm has to gain an exclusive use of the PCQ. This is done through a lock. All transactions in the PCQ will be upgraded to an OK state. All transactions in the TRQ will be executed locally by the ToM algorithm and the results returned to the clients.

The ToM algorithm is also responsible for detecting the liveness of other sites. After the propagation of transaction requests, the token holder collects responses from all sites. Any site which failed to respond within a time limit is eliminated from the current logical ring and the transactions just propagated by the token holder is inserted into the FAT table corresponding to the entry for the failed site. The FAT table information is passed together with the token to the next token holder.

When the token holder receives a re-joining message from a recovering site, the ToM algorithm is then responsible to add the recovering site into the logical ring and pass the FAT table information to the recovering site. The recovering site is then responsible for executing all missed transactions listed in the corresponding entry of the FAT table.

**Assertion 2**: Transactions passed by the token holder are ordered by the global timestamp.

**Proof**
The token holder uses the \( \text{msg} \) to pass transactions to all
sites. The msg is formed after the current site \( A_j \) knows it is the token holder. In that case \( A_j \) uses two while statements to order transactions originated from \( A_j \). The first while statement orders all partially committed transactions stored in the PCQ and the second while statement orders other transactions that were not processed. The \( tk.assignTSK() \) function is defined in Section 3.2. It assigns the global timestamp carried by the token and increases the timestamp monotonically. Therefore all transactions have been ordered before they are multicast to all sites.

We use \( T_1 < T_2 \) to express the fact that \( T_1 \)'s global timestamp is smaller than \( T_2 \)'s global timestamp.

### 4.3 The Non-Token-Site Manager

A non-token site needs to perform the following tasks:

- To monitor the liveness of the token holder. Each site sets up a timer to wait for the token holder to propagate messages. The token holder is assumed to have failed if a time-out occurs. In that case, the non-token site manager has to notify the potential next token holder to assume the duty. The information for selecting next token holder is stored in the FAT table (as \( fat[0].k \)).
- To assume the token holder duty if the token holder fails and the current site has been chosen as the next token holder.
- To determine if the current site is the token holder and stores information about missed transactions by individual failed sites into the FAT table.
- To process the propagated transactions requested by the token holder. These transactions are ordered by their global timestamps and the NToM is responsible to make sure that all transactions requested by the token holder will be performed in the local site. A local 2PC protocol is used in processing these transactions.

Listing 6 shows the NToM’s algorithm. We assume that the current site is named \( A_j \).

The NToM algorithm receives a message or token from the token holder, or a token requests from non-token holders. The process suspends / times out until a message comes.

The NToM algorithm monitors four types of situations. First, it alerts the predecessor of the current token holder when a time-out occurs. In that case, the token holder is assumed to be failed and the predecessor of the current token holder is to be the next new token holder.

Second, if a request from a site arrives, asking the current site to be the next token holder, then the current site will become the new token holder.

Third, if a token message arrives, the current site will check if itself is to be the token holder and save any FAT entries (these entries represent the missed transactions of each failed site). The reason of doing this is to make sure that the FAT information is kept in every site and can be used when a site is selected to be the new token holder.

And fourth, if the message contains transactions propagated by the token holder arrives, the current site will process these transactions according to their global timestamps. All transactions with PC state will be force to commit on \( A_j \). A local 2PC protocol is used to process transactions without a PC state. We assume that the local processing of the propagated message is atomic.

The NToM algorithm has a higher priority than the TrM algorithm for accessing data items. That is, if a transaction \( T \) executed by the NToM algorithm is conflict with another transaction \( T' \) executed by the TrM algorithm, then \( T' \) will be forced to roll back. Also, if the data items accessed by \( T \) has been accessed by a partially committed transaction \( T'' \), then \( T'' \) will be rolled back. The NToM algorithm is also responsible for letting the client know that the partially committed transaction has been rolled back and an FL is returned (through the handler field of the corresponding entry of the PCT table, this entry is deleted after the notification to the client).

The algorithm RoT for rolling back a partially committed transaction is simple. Let \( T \) be a global transaction requested by the token holder and \( D' \subseteq T.D \) be the set of data items of \( T \) that have been partially committed by some transactions in \( A_j \). The algorithm searches through the beginning of the PCT table and matches the data items of the partially committed transactions with the data items in \( D' \). If \( T' \in PCT \) is a match, then the \( T' \) and all the following transactions of \( T'.next \)
roll_back_transactions(T)
{
    if (TrM is processing T)
        notify TrM to roll back T;
    if (T ∈ PCT) {
        follow T′.nxt link (say, T′′) until T′′.nxt = 0;
        roll back T′ and T′′.pre till T′′.pre = T′;
        roll back T′;
    }
}

Listing 7 Algorithm 4 – The RoT algorithm

should be rolled back, starting from the last one of the link (i.e. following the T′′.nxt link to the transaction T′′ with T′′.nxt = 0). Then starting from T′′ to roll back all transactions following the T′′.pre link, till the link reaches T′ (T′ is also rolled back). Listing 7 shows the RoT algorithm.

We want to look at the effect of the NToM algorithm on the data items stored on a site. Let us define some terms first.

Definition 7 (Logical view)
Let $D_{lj}$ be the set of data items of all physical objects on site $A_j$, and $PCQ_j$ be the partially committed transaction queue of site $A_j$. $A_j$’s logical view of $D_{lj}$ (logical view in short) is defined as the result of $D_{lj}$ after rolling back all transactions in $PCQ_j$, in reversed order (i.e. from the tail to the head). We also call $D_{lj}$ as $A_j$’s physical view.

Let $D$ be a set of data items and $T$ be a transaction accessing $D$, we use $D + T = D'$ to represent the effect of executing $T$ on $D$ and $D'$ has been transferred into $D'$. Similarly, let $m = \{T_1, T_2, ..., T_k\}$ be a set of transactions accessing $D$, we use $D + m = D'$ to represent the effect of executing $m$ in the order of $T_1$, then $T_2$, ... then $T_k$ on $D$ and $D'$ has been transferred into $D'$. Therefore, if $D_{lj}, D_{lj}$ and $PCQ_j$ are the logical view, the physical view, and the queue of partially committed transactions of site $A_j$, respectively, then we have $D_{lj} + T + T' = D_{lj} + T + T'$.

Assertion 3
Let $D_{lj}, D_{lj}$, and $PCQ_j$ be the logical view, the physical view, and the queue of partially committed transactions of site $A_j$. If $A_j$ is not a token holder and $msg$ is the transactions propagated by the token holder, then after the execution of the NToM algorithm the new physical view for $A_j$ is $D_{lj}' = D_{lj} + msg + PCQ_j'$, where $PCQ_j' \subseteq PCQ_j$ and if $T < T'$ and $T, T' \in PCQ_j$, then $T, T' \in PCQ_j'$.

Proof
The effect of the NToM algorithm is

$D_{lj} + msg = D_{lj} + PCQ_j + msg$

If transactions in $PCQ_j$ and $msg$ do not interfere to each other, that is, $\forall T \in msg, T' \in PCQ_j, T.D \cap T_1.D = \emptyset$. Then the assertion is true:

$D_{lj} + msg = D_{lj} + PCQ_j + msg = D_{lj} + msg + PCQ_j$

Now we assume that $T$ and $T'$ access some common data items. In the NToM algorithm, transactions in $msg$ have a higher priority than transactions in $PCQ_j$. That is, if $T \in msg, T' \in PCQ_j$ and $T.D \cap T_1.D = \emptyset$ then $T'$ will be rolled back. Also, all $T'.nxt \in PCQ_j$ will be rolled back. These rollback will not affect the original ordering in $PCQ_j$. So we have $PCQ_j' \subseteq PCQ_j$, and if $T < T'$ and $T, T' \in PCQ_j'$ then $T, T' \in PCQ_j'$.

After the rollback, $T$ is then executed on $D_{lj}$. Since $T$ does not access any data items accessed by $PCQ_j'$, therefore,

$D_{lj} + PCQ_j' + T = D_{lj} + T + PCQ_j'$

for all $T \in msg$. So, $D_{lj}' = D_{lj} + msg + PCQ_j'$

4.4 Recovery from failures

When a site fails, the token holder detects the site failure through the TO returned by the transaction propagation to that site. The failed site is then eliminated from the logical ring and the result is multicast to all members of the new ring when the token is passed to the next token holder (refer to the ToM algorithm).

When a failed site re-joins the logical ring, it multicasts the re-joining message to all members of the logical ring. The token holder then responds to this message by sending to the re-joining site its corresponding messages stored in the FAT table. The re-joining site has to perform all the transactions it missed during its down time. A new logical ring which includes the re-joined site is formed and the information is multicast to all site by the token holder when it passes the token to the next site. The process is shown in the algorithm shown in Listing 8 (assuming $A_j$ is the recovering site).

When the token holder $A_j$ fails, other sites will detect that fact by a time-out since they all knew who will be the next token holder (through the fat[0] entry, refer to Algorithm 3 of Section 4.3) and were expecting a multicast message from it. As described in Section 4.3 and following [21], a message is sent to site $A_{lj}$, asking it to be the new token holder. $A_{lj}$ has the highest probability of being alive since it just passed the token out. $A_{lj}$ also has the up-to-date information about the token (such as the last timestamp and the FAT entries). In the case of $A_j$’s failure, $A_{lj}$ is the optimal choice to be the next token holder without an election. If $A_{lj}$ fails then the site recovery()
    if (msg ≠ ∅)
        process transactions in msg (as in the NToM algorithm);
        multicast to all sites, request for re-joining;
        receive(kk) \$
    perform all transactions in tk.fat[A_j], using the global timestamp ordering;

    cobegin
        transaction_request_manager();
        token_processing_manager();
        token_request_manager();
    coend

Listing 8 The Site Recovery algorithm
duty as the token holder, site \( A_{i,2} \) (and the process continues if \( A_{j,2} \) fails) will be selected by all alive sites to be the next token holder.

When a network partition occurs, the replicated sites may be partitioned into smaller subgroups. We consider the following two situations:

- When one of the subgroups has the majority of sites. In this case, the subgroup with the majority of sites will be allowed to form a new logical ring and resume the function of propagating transactions globally. If the subgroup contains the token holder, then the token holder will be responsible for the reconforming of the ring, as indicated in Algorithm 2. Otherwise, a new token holder has to be decided using Algorithm 3. Transactions propagated in this subgroup will be recorded into the FAT table and then propagated to other sites when the network is recovered. Other subgroups without the majority will not be allowed to propagate their transactions. However, they are allowed to partially commit their transactions locally.

- When there is no majority in any subgroup. In this case, none of the subgroups is allowed to propagate their transactions globally. However, they are allowed to make partial commit locally for their transactions. When the network recovers, these partially committed transactions will be processed by Algorithms 2 and 3.

Assertion 4

After the recovery of a site failure or a network partition failure, all sites will have the same logical views.

Proof

Let us look at a site failure first. Let \( D_L, D_{LU} \) and \( PCQ \) be the logical view, physical view and the queue of partially committed transactions of site \( A_j \), respectively.

Suppose at the point of its failure, site \( A_j \) is not a token holder. When \( A_j \) recovers, \( A_j \) will have its PCQ (together with the FAT table and TRQ) restored to the state before the failure. That is, \( D_{LU} \) and \( PCQ \) will be the same as before. \( A_j \) then sends a rejoining message to all sites and receives the token from the token holder. The FAT item of \( tk, \text{fat}[A_j] \) contains all transactions that \( A_j \) has missed during this failure. That is, for site \( A_j \), its logical view will be \( D_{LU} + tk, \text{fat}[A_j] \), and its physical view will be \( A_j \)'s logical view before \( A_j \)'s failure. By executing the transactions, \( A_j \) will have a new physical view of

\[
D_{LU}' = (D_{LU} - PCQ) + \text{fat}[A_j] + PCQ',
\]

where \( PCQ' \) is a subset of \( PCQ \) and maintains the original order. So \( A_j \) has the same logical view of \( D_{LU} + tk, \text{fat}[A_j] \) as \( A_j, j \neq i \).

Now let site \( A_j \) be a token holder when it fails. If it fails before its propagation message \( msg \) is sent (that is, before the PCQ and TRQ are changed), then the situation reduces into the same as a non-token holder. If \( A_j \) fails after the message is propagated, then the transactions contained in the message have been executed by alive sites \( A_j, j \neq i \). Therefore, \( A_j \)'s logical view before \( A_j \)'s failure will be \( D_{LU} + msg \). The transactions that \( A_j \) has missed during its failure is stored in \( tk, \text{fat}[A_j] \). So the logical view of \( A_j, j \neq i \) when \( A_j \) recovers is \( D_{LU} + msg + tk, \text{fat}[A_j] \). The \text{site\_recovery()} function requires that when \( A_j \) recovers, it restores the \( msg \) and executes the transactions contained in \( msg \) before it executes transactions in \( tk, \text{fat}[A_j] \). So the logical view of \( A_j \) will be

\[
(D_{LU} - PCQ) + msg + tk, \text{fat}[A_j] + PCQ',
\]

where \( PCQ' \) is a subset of \( PCQ \) and maintains the original order. So \( A_j \) has the same logical view of \( D_{LU} + msg + tk, \text{fat}[A_j] \) as \( A_j, j \neq i \).

Next we look at the recovery of a network partition failure. Let \( D_L \) be the logical view of all sites before the partition. If the partition does not produce any majority subgroup, then there is no propagation of transactions during the partition. Therefore \( D_L \) does not change when the failure recovers.

If the partition produces a majority subgroup \( G \), then the subgroup will carry out transaction propagation during the partition. Let \( m \) be these propagated transactions. Then \( \forall A_j \notin G, \exists \text{fat}[A_j] = m \). The logical view of all sites in the majority subgroup will be \( D_L + m \) when the partition failure recovers. Each \( A_j \notin G \) will execute the \text{site\_recovery()} function when it recovers and the logical view after the execution of \( tk, \text{fat}[A_j] \) will be

\[
(D_{LU} - PCQ) + tk, \text{fat}[A_j] + PCQ' = D_L + m + PCQ',
\]

So, \( A_j \) has the same logical view of \( D_L + m \) as \( A_j \in G \).

5. CORRECTNESS

In this section we outline the informal analysis of the correctness, proving that the proposed scheme is correct in terms of serialisability and atomicity. We assume the lifecycle of the system entities (sites, links, and replicas) is

\[
\text{work} \rightarrow \text{crash} \rightarrow \text{repair and restart} \rightarrow \text{work}
\]

Without loss of generality, we assume that the maximum down time (including crash, repair and restart time) is finite, and is denoted as \( \Gamma_e \).

Assertion 5

Let \( D_L \) be the logical view of all sites \( A_1, A_2, ..., A_m, A_i \) be the token holder, and \( msg \in \{ T_1, T_2, ..., T_l \} \) be the transactions propagated by \( A_i \) (ordered by their global timestamps, \( T_i \) is the smallest). After all sites executed \( msg \), their logical views will be the same.

Proof

If \( msg \) is empty, the assertion holds, since the logical view of the token holder is \( D_L \) and the logical view for site \( A_j \neq i \) is \( D_L - PCQ = D_L \). This is because all sites other than the token holder can only locally commit transactions and queue them in their own PCQ.

If \( msg \neq \emptyset \). After the execution of \( msg \), the physical view of the token holder is \( D_L + msg \), which is the same as the logical view since \( PCQ \) is empty. According to Assertion 3 in Section 4.3, the physical view of \( A_j \neq i \) after \( A_i \) executed \( msg \) is: \( D_L' = D_L + msg + PCQ' \), where \( PCQ' \subseteq PCQ \) and \( PCQ'_j \) keeps the ordering of \( PCQ_j \). Therefore the new logical
view of \( A_j \) is \( D'_L = D_L + \text{msg} \). So all sites have the same logical views.

**Assertion 6**

If the execution of a transaction \( T \) returns OK, \( T \) has been committed in all updatable replicas. A site is regarded to be updatable if it is alive and it is connected to the majority of sites. Therefore a site which is alive but is disconnected from the majority of sites is not updatable.

**Proof**

According to Assertion 1 in Section 4.1, \( T \) may have one of the three possible results after it is submitted to TrM of site \( A_j \). Only two of these results (\( T \) is either in PCQ or in TRQ) may lead to an OK return through the ToM algorithm when \( A_j \) becomes the token holder:

- \( T, st = PC \). According to Assertion 1, \( T \) has been executed on \( A_j \) successfully by TrM. The ToM algorithm puts \( T \) in front of any transactions of the TRQ and \( T \) is propagated to all updatable sites. In this case, \( A_j \)'s logical view has included the effect of \( T \). An OK is returned by the ToM through the handler of \( T \)'s entry in the PCT table.

- \( T, st \neq PC \). The ToM algorithm of \( A_j \) will execute \( T \) locally. If the execution is successful, the ToM algorithm returns an OK and the effect of \( T \) is made into the logical view of \( A_j \). \( T \) is also propagated by \( A_j \) to all updatable sites.

According to Assertion 5, after the execution of the propagated message from \( A_j \), all sites will have the same logical view. Since \( A_j \)'s logical view has included the effect of the successfully execution of \( T \), this implies that all sites have executed \( T \) successfully.

**Assertion 7**

If a transaction returns FL, no operations of the transaction have been executed in any replica.

**Proof**

The analysis is the same as Assertion 6.

**Assertion 8**

All sites commit transactions in the same order.

**Proof**

The only way to commit transactions by a site is that these transactions are propagated by the token holder, and the order of execution of these transactions follows their global timestamps. According to Assertion 2 in Section 4.2, all transactions propagated by the token holder have been ordered by the global timestamp. Therefore the assertion holds.

**Assertion 9**

If a transaction’s execution returns PC, the transaction will be notified an OK or an FL return in a finite time.

**Proof**

A partially committed transaction \( T \) submitted through site \( A_i \) may be notified of the final result (OK/FL) in two ways. First, \( A_i \) is not a token holder and is executing a transaction \( T' \) propagated from the token holder. When \( T.D \cap T'.D \neq \emptyset \), \( T \) will be aborted and an FL is returned by \( A_i \) through its handler in the PCT table. Second, \( A_j \) is the token holder. \( T \) will be propagated to all other sites for execution and \( A_j \) will return an OK to the transaction. An alive site is either a token holder or a non-token holder and it will be a token holder in a finite time \( W + \Gamma_\alpha \), where \( W \) is the maximum cycle time of the token. Therefore the assertion holds.

6. **EXAMPLES**

In this section we illustrate the TBIU protocol using our example. In our example we have defined the parts for each logical object as:

- \( D^1 = \{e_1, e_2, e_3, e_4\} \)
- \( D^2 = \{x_1, x_2, x_3, x_4\} \)
- \( D^3 = \{b_1, b_2, b_3\} \)

and \( D_L = \{D^1, D^2, D^3\} \). We have define the access priorities for site \( A_1, A_2 \) and \( A_3 \) as:

- \( O_1 = D^1 \)
- \( O_2 = D^2 \)
- \( O_3 = D^3 \)

We first consider the situation where there is no failure. Then we discuss the case when a site fails.

6.1 **Example: No failures**

Initially each site stores the same logical view of all logical object: \( D_L = \{D^1, D^2, D^3\} \). Assume that \( A_3 \) (the body site) has the token. \( T_1 \) and \( T_2 \) are submitted to \( A_1 \) and \( T_3 \) and \( T_4 \) are submitted to \( A_2 \). The data sets accessed by each transaction are:

- \( T_1.D = \{e_1, e_2, e_3\} \)
- \( T_2.D = \{e_2, x_3, b_4\} \)
- \( T_3.D = \{x_2, x_3\} \)
- \( T_4.D = \{x_1\} \)

That is, \( T_1 \) only updates data about the engine, while \( T_2 \) requires the access to data related to the engine, the transmission, and the body components. Similarly, \( T_3 \) and \( T_4 \) only access data about the transmission component. Therefore, on the engine site \( A_1 \), \( T_1 \) can be executed, queued up in PCQ and a PC is returned, while \( T_2 \) has to be queued into TRQ. After the execution of \( T_1 \), \( A_1 \) has a new physical view in which \( e_1, e_2 \) and \( e_3 \) have been changed by \( T_1 \). Similarly, \( T_3 \) and \( T_4 \) can be partially committed on \( A_3 \) and queued up in PCQ. \( A_2 \) has a new physical view in which \( x_1, x_2 \) and \( x_3 \) have been changed by \( T_3 \) and \( T_4 \).

Next moment \( A_3 \) passes the token to \( A_1 \). When \( A_1 \) has the token, it returns an OK to \( T_2 \) since \( T_1 \) has been partially committed. Then \( T_1 \) and \( T_2 \) (from the TRQ) are packaged together and multicast to \( A_2 \) and \( A_3 \). In the mean time, transactions \( T_3 \) and \( T_4 \) are submitted to the body site \( A_3 \):

- \( T_5.D = \{b_1, b_2\} \)
- \( T_6.D = \{b_3\} \)

Since both transactions only access data in the body component, they are partially committed and queued up in PCQ.
When \( A_3 \) and \( A_3 \) receive the propagated message, they need to commit partially committed transactions (in that case, \( T_1 \)). They also have to execute other propagated transactions \( (T_2, \text{in this case}). \) Since \( T_2 \) and \( T_3 \) conflict with each other \( (T_2.D \cap T_3.D = \{ x_1 \}), \) \( T_3 \) is rolled back in \( A_2 \), and \( T_2 \) is forced to commit in \( A_3. T_2 \) can be executed on \( A_3 \) without any trouble. After the execution of \( T_1 \) and \( T_2 \), all sites have the same logical view again (where \( e_1, e_2, e_3, x_1, \) and \( b_1 \) have changed by \( T_1 \) and \( T_3 \)). After site \( A_1 \) receives the acknowledgement from \( A_2 \) and \( A_3 \), it passes the token to \( A_2 \).

When \( A_2 \) has the token, it returns an OK for \( T_2 \) and passes it to \( A_1 \) and \( A_3 \; \text{where \( T_2 \) will be forced to commit. Figure 3 illustrates the above process.} \)

### 6.2 Example: Site failures

A site can be a token holder or a non-token holder when it fails. Let us look at the case of the failure of a non-token holder first. Assume initially \( A_1 \) has the token and \( T_1 \) and \( T_3 \) are submitted to the engine site \( A_1 \), where \( T_1.D = \{ e_1, e_2, e_3 \} \), \( T_2.D = \{ e_2, x_1, b_1 \} \). Similar with Section 6.1, on \( A_1 \), \( T_1 \) can be executed, queued up in PCQ, and a PC is returned, while \( T_2 \) has to be queued into the TRQ. After the execution of \( T_1, A_1 \) has a new physical view in which \( e_1, e_2 \) and \( x_1 \) have been changed by \( T_1 \).

When \( A_1 \) becomes the token holder, it commits \( T_1 \) and \( T_2 \) and packages \( T_1 \) and \( T_3 \) into a message \( m \) for multicast. \( m \) arrives at \( A_2 \) but \( A_1 \) fails before it gets \$m\$. \( A_2 \) will perform \( T_1 \) and \( T_2 \) and return an acknowledgement to \( A_1 \). Since \( A_1 \) fails to receive an acknowledgement from \( A_2 \), it will put \( \{ T_1, T_2 \} \) into the token message \( t.f.a[t_A] \) and passes the token to its next site \( A_3 \).

When \( A_3 \) recovers, it sends a re-join message to all sites. The current token holder, say, \( A_p \), responds to this message by including the recovering site into the logical ring. \( A_1 \) will receive the \( t.f.a[t_A] \) message when the token is passed by \( A_2. A_3 \) then analyses \( t.f.a[t_A] \) and executes the missed transactions (in that case \( \{ T_1, T_3 \} \). After that all sites will have the same logical view. Since we have assumed that propagated transactions will be processed by non-token holders in an atomic manner, the case of \( A_3 \) fails after it receives \( m \) is simple. Figure 4 illustrates the situation of a non-token holder failure.

Next we analyse the case of the token holder failure. We use the same assumption as we used for analysing a non-token holder failure. That is, on the engine site \( A_1 \), \( T_3 \) has been partially committed and \( T_2 \) has been queued up in the TRQ.

When \( A_1 \) has the token, it commits \( T_1 \) and \( T_3 \) and propagates the message with \( T_1 \) and \( T_3 \) packaged in it to \( A_2 \) and \( A_3 \). Both \( A_2 \) and \( A_3 \) commit \( T_1 \) and \( T_3 \) and the three sites will have the same logical view. Now assume \( A_1 \) fails at this moment, before receiving the acknowledgement from other sites. \( A_2 \) and \( A_3 \) will time out. Both of them will find out that the previous token holder before \( A_1 \) is \( A_2 \). So \( A_3 \) will take the responsibility to generate the new token and the logical ring will be re-formed without \( A_1 \). When \( A_1 \) recovers, it notifies.
all sites about its intention of re-joining the ring. The current token holder (A₂ in this example) will add A₁ into the logical ring. A₁ will be able to perform all the transactions it has missed when it is down from tk.fat[A₁]. Figure 5 illustrates the situation of the token holder failure.

7. PERFORMANCE ISSUES

In this section we first analyse the performance of the TBIU protocol and then compare our TBIU protocol with two similar protocols: the copy token and the independent update protocols.

7.1 Analysis

In all analysis of this paper, only point-to-point network is used in which a single source site sends the same message to m sites by transmitting m messages, one to each receiver. We assume that the communication time (including the token passing time and the transaction propagation time) of each message transmission is a constant C. The following parameters are used:

- \( \lambda \): the arrival rate of transactions.
- \( T \): the mean processing time of a transaction.
- \( u \): utilisation of the transaction processing system: \( u = \lambda T \).
- \( p \): the probability that a transaction is queued up in the TRQ. Therefore, \( (1 - p) \) is the probability that a transaction is to be partially committed and then queued up in the PCQ.
- \( q \): the probability that a transaction is to be rolled back from the PCQ because of conflict.

We use induction to derive the time needed to process transactions in a site. Without loss of generality, let Site 1 be the token holder initially. The sequence of token passing is 1, 2, ..., m, 1, ..., where m is the number of sites. In that case, all incoming transactions to Site 1 within a unit time will be processed and then propagated. Let \( D_i \) be the transaction processing time at Site \( i, i = 1, 2, ..., m \). We have:

\[
D_1 = \lambda T = u
\]

For Site 2, it has to wait for \( D_1 \) time before it gets the token. During that time, \( D_1 \lambda p \) transactions will come, of which \( D_1 \lambda p = \lambda^2 Tp \) of them will go into the TRQ and the cost of processing them is \( \lambda^2 T^2 p \). Similarly, \( (1 - p)\lambda^2 T \) transactions will go into the PCQ. The cost of performing these transactions, then rolling back some of them, and then re-submitting these rolled back transactions can be expressed as \( kq(1 - p)\lambda^2 T^2 \), where \( k \geq 2 \). So,

\[
D_2 = \lambda T + (\lambda^2 T^2 p + kq(1 - p)\lambda^2 T^2) = \lambda T + (p + kq(1 - p))\lambda^2 T^2 = D_1 (1 + \lambda T(p + kq(1 - p)))
\]
Let $D_{i-1} = D_{i-2} (1 + \lambda T(p + kq(1 - p)))$, for Site $i$, we have:

$$D_i = D_{i-1} + (D_{i-2} + \lambda p + kq(1 - p)D_{i-1} + \lambda T(p + kq(1 - p)))$$

By induction, we have:

$$D_m = D_{m-1} (1 + u(p + kq(1 - p)))$$

Let

$$D_m / D_{m-1} = (1 + u(p + kq(1 - p))) = \Delta$$

Then

$$D_m = \Delta D_{m-1} = \Delta^2 D_{m-2} = \cdots = \Delta^{m-1} D_1 = \Delta^{m-1} u$$

It is easy to show that if we include communication time $C$ into the above analysis, then we have:

$$D_m = (1 + u(p + kq(1 - p)))^{m-1} (u + C)$$

### 7.2 Evaluation

In this section we look at the effect of changes in $p$ and $q$ to the response time of the entire system, since $p$ and $q$ are two key factors of our protocol. In the following evaluation, we set $\lambda = 1$, $T = 0.5$, $m = 3$, $k = 2.5$ and $C = 10$. Figure 6 shows the change of the response time $R$ when $p$ varies. We have the following observations from Figure 6:

- When $q = 0.8$, the response time $R$ decreases as $p$ increases. $q$ is large means that most of the transactions conflict with each other. Therefore increasing $p$ means putting more transactions in the TRQ and subsequently avoiding the cost of rolling back many of these conflict transactions.

- When $q = 0.1$, the response time $R$ decreases as $p$ decreases. $q$ is small means that most of the transactions do not conflict with each other and therefore they can be per-
formed independently. Decreasing $p$ means putting more transactions in the PCQ and executing them independently. The time saved by executing transactions independently is greater than the time used to roll back a small number of transactions. So the response time decreases as $p$ decreases.

- When $q = 1/k = 1/2.5 = 0.4$, the response time $R$ does not change as $p$ varies. $q = 1/k$ is a critical point where $p$ becomes irrelevant.

The conclusion is that if $q > 1/k$, we should try to avoid the use of the PCQ. However, if $q < 1/k$, then more transactions should be performed independently. Figure 7 shows the change of the response time when $q$ varies.

In all the three cases ($p = 0.1$, $p = 0.5$ and $p = 0.8$) shown in Figure 7, the response time $R$ increases as $q$ increases. However, the pace of increase is different when $p$ has different values. When $q < 0.4$, the best performance (i.e. with the smallest $R$) is the case of $p = 0.1$, then $p = 0.5$, and the worst is $p = 0.8$. However, such an order changes to $p = 0.8$, $p = 0.5$, $p = 0.1$ when $q > 0.4$. Again we see that $q = 1/k = 0.4$ is a critical point.

Figure 8 shows the change of the response time when both $p$ and $q$ change. It confirms our previous observations that, to keep the response time $R$ small,

- when $q$ is small, $p$ should also be small. That is, if the probability of transaction conflict is small, more transactions should be executed independently; and
- when $q$ is large, $p$ should also be large. That is, if the probability of transaction conflict is large, less transactions should be executed independently to avoid the cost of rolling back many of them.

To verify the correctness of our theoretical model, we have implemented a simulator for the TBIU protocol using the SMPL simulation package [27]. Figure 9 shows a comparison of the results between the simulator (marked as ‘Simulation’) and the theoretical model (marked as ‘Calculation’). The result of the theoretical model is adopted from Figure 6, where $q = 0.8$ and we show the changes of $R$ when $p$ changes. The simulator uses the same parameters as that of the theoretical model.

It can be seen from Figure 9 that the two results are very close, especially in the middle and lower parts of the curves. Only when $p$ is small, the simulated model increases dramatically compared with the theoretical model. The reason is that in our theoretical model, we assumed that the rollback effect is a constant ($k$) times the rollback probability and cost. While in our simulator, we used a more practical approach by letting the failed transactions re-enter the system for processing. As $q$ is high in our case ($q = 0.8$), when $p$ is low, most of the transactions will go into the PCQ. Many of these transactions will be rolled back and ultimately will re-enter the system again. The effect of these re-entering transactions causes a congestion in this section (e.g. $p < 0.2$) and therefore the curve increases dramatically.

7.3 Comparison: Copy Token vs. TBIU

In this and the next sections we compare the TBIU protocol with two similar update protocols: the copy token protocol [29] and the independent update protocol [11, 10].

A transaction $T$ submitted to a site $A$ of the token copy protocol enters a queue to wait for the site to become the token holder. When $A$ becomes the token holder, $T$ is executed locally and a result returned to the client immediately. $T$ is packaged into a multicast message and sent to other sites for forced execution. This is a special case of the TBIU protocol. In fact, Equation (1) of Section 7.2 has two special cases:
When \( p = 1 \), no partial commit is permitted. All transactions have to be put into the TRQ, waiting for the token. This is the original copy token protocol. In that case,
\[
D_m = (1 + u)^{m-1}u
\]
(3)

The TBIU protocol is better if
\[
(1 + u) > (1 + u(p + kq(1 - p)))
\]
That is, if \( q < 1/k \). Figure 10 confirms this situation, where \( k = 2.5 \). The TBIU protocol is expressed as normal lines, and the copy token protocol is expressed as dots. Figure 10 shows that TBIU is better when \( q \) is small.

When \( p = 0 \), all transactions are to be partially committed and are subject to the possibility of being rolled back. The transactions commit when the site becomes the token holder. This is a revised copy token protocol. In that case:
\[
D_m = (1 + kqu)^{m-1}u
\]
(4)

The TBIU protocol is better if
\[
(1 + kqu) > (1 + u(p + kq(1 - p)))
\]
That is, when \( q > 1/k \). Figure 11 shows this situation where \( k = 2.5 \). The TBIU protocol is also expressed as normal lines, and the revised copy token protocol is expressed as dots. Figure 11 shows that TBIU is better when \( q \) is large.

### 7.4 Comparison: Independent Update vs. TBIU

Now we compare the TBIU protocol with the independent update protocol [11, 10]. A transaction \( T \) submitted to a site \( A \) of the independent update protocol is executed and propagated to other sites immediately for execution. Each site \( A' \) will execute the propagated transaction and return an acknowledgement. If the execution of \( T \) on \( A' \) conflicts with other transactions already executed on \( A' \), the reconciliation process is launched, which involves the exchange of information between \( A' \) and other sites whose transactions conflict with transactions executed on \( A' \).

Figure 12 shows that the TBIU protocol is better when \( 2 \leq m \leq 13 \) if \( q = q_1 = 0.5 \). Also, when \( m > 13 \), the response time of the TBIU protocol increases dramatically. However, if \( q = 0.3 < q_1 \), the range of \( m \) in which the TBIU protocol is better becomes \( 2 \leq m \leq 19 \). Furthermore, if \( q = 0.1 \ll q_1 \), then the above range increases to \( 2 \leq m \leq 29 \). As we mentioned earlier, we did not consider the arrivals of transactions during the reconciliation process when we analysed the independent update protocol. That is the main reason that the result of the independent update protocol is almost linear.

Even with the above analysis, we can conclude that the TBIU protocol is better if the number of replicas is small or medium. Fortunately in practice, most replicated systems will have a small number of replicas.
8. SUMMARY

The reconciliation problem is inherent in any independent update protocols. If conflicting updates are issued by transactions that execute on different replicas, then those updates will need to be reconciled when they arrive at each replica.

Increasing the number of replicas increases the number of updates that need to be propagated because each update must be propagated to all replicas. That is, if the number of replicas is increased and each replica is allowed to independently propagate new update transactions, then the load increases quadratically [6]. Therefore, when conflicting updates are likely, primary-copy approaches generally work better.

However, independent update protocols provide a better response to client requests since the requested updates can be carried out immediately. Our effort is to combine the advantages of independent update and primary-copy protocols and yet to avoid the problems associated with these protocols. That is, we want our protocol to be able to carry out immediate update requests, and yet to avoid the cost of a large-scale reconciliation process.

What we have achieved in this paper is a protocol that allows independent updates locally and reduces the cost of reconciliation. These locally performed updates are propagated when a site becomes the token holder. All propagated update transactions have a higher priority than any locally performed transactions and will be committed. Those locally performed transactions whose updates conflict with any propagated updates will be rolled back locally. The access priority mechanism is defined to control the number of independent updates in a site. More specifically, the paper makes the following major contributions:

- A token is used to simplify the control of update propagation and ordering. Transactions propagated by the token holder have higher priorities and the timestamp carried by the token simplifies the way of ordering transactions.
- A partial commit state is introduced to support independent updates and enhance the efficiency of transaction processing. Clients now have an option of continuing their work after a partial commit is received and they know that the chances of final commit of their transactions are very high.

- A data partition methodology based on access priorities is adopted to reduce update conflicts. Transactions updating the access priority set can be carried out independently, whereas other transactions have to wait until the site becomes the token holder.
- Proved the correctness of the proposed protocol, analysed the performance and compared the protocol with some similar schemes.

ACKNOWLEDGEMENT

The paper is partially supported by the following grants: (1). CityU Project No. 7001355 while the first author was visiting City University of Hong Kong, and (2). Deakin University Research Grant No. 0504-23434-31-01.

REFERENCES


