Typed Token Approach For Database Processing During Network Partitioning

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Shy-Renn Lian

Report Number:
90-987


This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
TYPED TOKEN APPROACH FOR DATABASE PROCESSING DURING NETWORK PARTITIONING
Bharat Bhargava
Shy-Renn Lian

CSD-TR-987
June 1990
Typed Token Approach for Database Processing during Network Partitioning

Bharat Bhargava and Shy-Renn Lian

Computer Science Department
Purdue University
West Lafayette, Indiana-47907

ABSTRACT

We propose a typed token scheme for managing replicated data in distributed database systems. Compared to previous schemes, for each object, a set of tokens is used. Each token represents a specific capability for the allowable operations on the object. By distributing tokens to different physical copies of the object, the object can be made available for different operations in various partitions of the network during a failure. Two types of replication for each of these tokens are proposed. One is based on the semantics of operations and the other is based on the semantics of the object. When failures are anticipated, tokens can be redistributed to maintain high availability. We present an algorithm for efficient recovery of database consistency to support the typed token scheme when partition merge. These ideas contribute towards increased availability during network partitioning while maintaining the consistency of the database.

This research is supported by National Science Foundation under grant number DRI-8821398.
1. Introduction

Replicated databases can increase availability during failures in distributed database systems [3,10]. The database system must ensure consistency among replicated copies when failures/recovery occur. To ensure the consistency during network partition, data availability and transaction processing is restricted. This research ensures the consistency among copies of the database while sustaining high availability during communication failures.

1.1. Related research

There are several replicated copy management approaches that have been proposed for partition processing [10]. The primary copy approach suggests that one copy of an object be designated as the primary copy [23]. All reads and writes for the object must be performed at the primary copy. Updates are propagated to all copies. When partition failure occurs, only the partition containing the primary copy can access the object. In the scheme of token, this approach has a single token per object and the token is always at one fixed location.

The true-copy token scheme proposed in [18] is similar to the primary copy approach. In this approach, one object copy has a token associated with it, permitting the user to access the object. The token can be transferred among copies and whichever copy bears the token becomes the primary copy. However, in the true-copy token approach, data accessibility to an object is still limited to one partition which contains the token during network partitioning.

In another direction of research, voting algorithms have been proposed in [12,8,20]. A transaction that needs to perform a read/write operation on an object, must gather enough votes from the sites storing the physical copies of that object. The proposed voting algorithms have different rules to guarantee the following characteristics: (1) At least one common voting site for two transactions requesting access to the same
object will be involved in deciding the ordering of these transactions. (2) At least one copy of the replicated object accessed by a transaction will contain the current value of that object. In the static voting algorithms, a few failures can render the data inaccessible because the vote assignment cannot change after the system has begun operation. Dynamic voting algorithms provide higher availability than static algorithms. However, the requirement of instantaneous state information by dynamic voting algorithms makes them complicated and costly in terms of message traffic.

General quorum consensus approach [13,4] extends the voting algorithm proposed by Gifford [12]. It uses type information to allow different operations to have different levels of availability.

1.2. Typed token scheme

In this paper, we propose a scheme, called the typed token approach, that has the merits of simplicity of the true-copy token approach. Our approach provides data availability by using type information as in the general quorum consensus approaches. To each data object, instead of assigning one single token, we assign a set of tokens. Each token represents some specific capability. Based on the semantics of operation and object, tokens can be further replicated to increase availability. This set of tokens are distributed among all physical copies of the object. If network partition does not occur, every copy of an object can execute any functions after collecting the necessary tokens from other copies. When partitioning occurs, each partition retains some subset of tokens for each object. Each partition can continue executing the functions endowed by tokens in its partition. Access to more than one copy of objects is allowed in different partitions to increase availability. However, the propagation and copying of values among different replicas of an object at partition merge is not enough to ensure consistency [3]. The total effects of transaction executions in different partitions should be reconciled for the updates on the database after recovering from partitioning failure.
The problem is more severe when partitioning and merging of multiple partitions occur dynamically. We propose an efficient algorithm to attack this problem. Unlike many other replication schemes that assume pairwise partitioning and merge [11,22], this algorithm handles dynamic partitioning and merge.

We assume that each site in the distributed system has a concurrency controller to schedule the local operations. We do not propose specialized concurrency control for transaction processing of abstract data types because each operation is eventually either a read or write to the database. In [5], a procedure is given to convert the dependency relation specification for an object into a mapping from typed operations on the object to read and write operations. Besides, typed-specific concurrency control mechanisms have already been proposed in the literature [14,21]. The concurrency controller ensures the atomicity property of transaction processing. We also assume a termination protocol[9] can be invoked to handle incomplete transactions when network partitioning occurs. Thus we need not worry about the serializability of transaction executed across a partitioned network. On top of concurrency control we have our replication control, the typed token approach, for correctly managing replicated data in spite of the presence of failures such as site failure and network partitioning.

In the failure treatment, we cannot distinguish a site failure from a partitioning failure. The worse case, i.e. a partitioning failure, has to be assumed. Our scheme does not require an instantaneous detection of network partition. During transaction processing, if some object copy cannot obtain a certain token, it becomes "aware" of a missing token. It then invokes the network partition treatment as described in Section 4.

The main questions that we must address for this scheme are how to assign tokens, how to possibly replicate tokens for further increasing availability, how to process transactions during both normal execution and network partition, and how to recover the database from dynamic partitioning and merge.
1.3. Organization of the paper

This paper is organized as follows. Section 2 contains the framework of our typed token scheme. Section 3 contains the details of transaction processing during both non-failure and network partition modes. In Section 4, we present an algorithm for handling dynamic partition and merge. Discussions and conclusions are stated in Section 5 and 6, respectively.

2. Framework of the Typed Token Scheme

We describe the framework of our typed token scheme in this section, including the object model, the assignment of typed token, and the replication of typed token.

2.1. Object model

Each object is of an abstract data type for which a set of operations are allowed to be performed on objects of this type. Let the set of operations allowed on data object \( X \) be \( \text{op}^X = \{ \text{op}_1^X, \text{op}_2^X, ..., \text{op}_n^X \} \). A set of tokens \( t^X = \{ t_1^X, t_2^X, ..., t_m^X \} \) is assigned to \( X \). Each \( t_i^X \) is associated with a subset of \( \text{op}^X \) to represent the capability of executing this subset of operations. During transaction processing, to execute some operation \( \text{op}_j^X \), a site has to access a corresponding token \( t_i^X \) that allows \( \text{op}_j^X \). How to associate with \( t_i^X \) the subset of \( \text{op}^X \) will be discussed in Section 3.1. We shall drop \( X \) in the representation of operations and tokens when the context is clear.

Each object \( X \) is fully or partially replicated on the set of sites. The physical copy of object \( X \) stored at site \( S_i \) is denoted by \( x_i \). Each object is composed of the following fields: 1) a set of attributes, each assigned one value from its domain, 2) a set of operations allowed on \( X \), 3) an array, called TokenArray, of size \( n \), where \( n \) is the number of operations allowed on \( X \). If TokenArray\([i]\) = \( j \), that means the capability of executing \( \text{op}_i \) is represented by \( t_j \), 4) an array of linked lists, called TokenPool, of size \( m \), where \( m \) is the number of tokens assigned to \( X \). TokenPool\([i]\) is a linked list storing the ID's
of the sites where token $t_i$ resides. To store the site ID's, an array of linked lists is needed instead of a one-dimensional array like $TokenArray$. This is because a token may be replicated and distributed to reside at more than one site (as will be discussed in Section 3.2). With the information stored in token pool, an operational site can efficiently access any token needed for an object. An example of a token pool is shown in Figure 1, where $t_m$ is replicated and located at two sites $S_j$ and $S_l$.

![Diagram of token pool](image)

Figure 1. An example of a token pool.

The field of object attributes can be easily adapted to contain nested object with hierarchical structure. For the ease of discussion, we assume objects are composed of attributes with some values.

2.2. Typed token assignment

In our scheme, instead of only considering read/write operations, we utilize type information of an object to classify operations into conflict classes. Operations that may conflict with one another are put in the same class and each class is associated with one token.

Definition 2.1 The operation dependency relation is a binary relation `$\rightarrow$' defined on the set of operations for an object such that $op_i \rightarrow op_j$ if, and only if, $op_j$ must
observe previous (in the equivalent serialization order) executions of \( op_i \) in order to return correct results.

The dependency relation is type-dependent and must be specified by the system designer.

**Definition 2.2** Operations \( op_i \) and \( op_j \) conflict if either \( op_i \rightarrow op_j \) or the opposite holds. \( op_i \) and \( op_j \) are said to be in the same conflict class if they conflict.

To assign tokens, we start with the creation of a dependency graph \( G_d = (V_d, E_d) \) for each object \( X \), where \( V_d \) is the set of operations supported on \( X \) and \( E_d \) is the set of edges such that \( (op_i, op_j) \) is in \( E_d \) if and only if \( op_i \rightarrow op_j \). We can then use any graph traversal algorithm in [1] on \( G_d \) to divide \( G_d \) into weakly connected components \( C_1, C_2, C_3, \ldots \). For each \( C_i \), we assign a token \( t_i \) which endorses the owner of the right of executing any operations in \( C_i \). We use \( t_i = \{op_{i1}, op_{i2}, \ldots, op_{in}\} \) to express that \( t_i \) stands for the capability of executing \( op_{i1}, op_{i2}, \ldots, op_{in} \).

To execute some operation on an object copy, a site has to access the corresponding token representing this operation. Since conflicting operations are restrained to one partition during partition failures, operations belonging to different conflict classes can be allowed at the same time in different partitions. The distribution of tokens enables multiple partitions to preserve at least part of the capability during partition failure so that transaction processing is not restrained in a small portion of the system. The next two examples show how we can have multiple tokens using the operation dependency relation.

**Example 2.1:** Consider an object \( X \) which is fully replicated in a distributed database system. \( X \) is a special banking account and consists of three attributes: \( X.Balance \) contains the balance of the account, \( X.CreditRecord \) contains the credit record of the owner of \( X \) and \( X.Flag \) is a flag indicating whether the customer is allowed to overdraft the
account X. System manager sometimes decides to take the risk of allowing a customer to overdraft an account based on a careful review of the business situation and the customer's credit record. Operations that are allowed on X are the follows:

Deposit(a) : amount a is deposited to X and hence \( X.Balance = X.Balance + a \).
Withdraw(a) : amount a is withdrawn from X if the balance is enough.
  
  i.e. \( temp = X.balance - a \)
  
  if \( temp \geq 0 \) then \( X.balance = temp \) else reject.
SetFlag(P) : set the flag \( X.Flag = P \), P is one of the two values \{yes, no\}.
Overdraft(a) : if \( X.Flag = yes \) then \( X.Balance = X.Balance - a \) else reject.
ReviewRecord : display the credit record stored at \( X.CreditRecord \).
AddCreditRecord : add one more credit record to \( X.CreditRecord \).

Operation Overdraft is usually invoked only after Withdraw is rejected. AddCreditRecord can be done anytime as long as a corresponding token is obtained. The credit record to be reviewed by ReviewRecord is brought up-to-date on a monthly basis. The manager must see \( X.Flag \) to decide whether to allow X to be overdrafted or not, hence Overdraft depends on SetFlag. To set \( X.Flag \) properly, the manager must review the credit record of the customer hence SetFlag depends on ReviewRecord. The dependency relation for the set of operations is shown in Figure 2.

We can then classify the set of operations to four groups and assign each group a token as follows. \( t_1 = \{ \text{Deposit} \}, t_2 = \{ \text{Withdraw} \}, t_3 = \{ \text{ReviewRecord, SetFlag, Overdraft} \} \) and \( t_4 = \{ \text{AddCreditRecord} \} \).

Example 2.2: Object Z is a 3-dimensional design unit in a CAD system and can be integrated in any part of a larger design. Z consists of the following attributes:
Z.Geometry contains the specification of the geometry of Z, Z.Belong indicates which part Z is integrated to the composite object, Z.LowerLeftCorner contains the \( x_{\text{min}}, y_{\text{min}} \)
coordinates for the object, $Z.Volume$ contains the volume of $Z$, and $Z.Area$ contains the area of the projection of $Z$ on the X-Y plane. $Z$ is equipped with the following operations:

- **Enlarge**($a$): enlarge $Z$ in all dimensions by a factor $a$.
- **Translate**($a,b$): translate $Z$ in X-direction for $a$ and in Y-direction for $b$ units.
- **ChangePart**($a$): Detach $Z$ and reattach it to part $a$ of the composite object.
- **Belong**: return $Z.Belong$.
- **Area**: return $Z.Area$.
- **Volume**: return $Z.Volume$.
- **LowerLeftCorner**: return $Z.LowerLeftCorner$.
- **Duplicate**: duplicate a new object from $Z$ with the same geometry and the same set of supporting operations.

The dependency relation for the set of operations on $Z$ can be described in Figure 3. Operation **Enlarge** changes the area as well as its volume. $Z.Belong$ obviously depends on **ChangePart**. The location of $Z$, represented by lower left corner $(x_{min}, y_{min})$, is affected by operations **Translate** and **Rotate** but not directly depends on...
We classify the set of operations into four classes and assign each class a token as follows. \( t_1 = \{ \text{Area, Volume, Enlarge} \}, t_2 = \{ \text{LowerLeftCorner, Translate, Rotate} \}, t_3 = \{ \text{Belong, ChangePart} \}, \) and \( t_4 = \{ \text{Duplicate} \} \).

These two examples show the assignment of typed token in different applications. The main advantage of the typed token approach is that with multiple tokens in the system, chances that multiple partitions can access different tokens concurrently during network partition is greatly increased.

2.2.1. Correctness criterion for typed token scheme

1-copy serializability is used as the correctness criteria in replication control[2,19]. We adapt this correctness criterion to abstract data types in our typed token scheme.

**Definition 2.3** The \textit{precedence} relation. In a replicated data schedule, \( T_a \) precedes \( T_b \) if there are some \( op_a \in T_a, op_b \in T_b \), and \( op_a \) is executed immediately before \( op_b \)
Definition 2.4 The depend-on relation. In a replicated data schedule, $T_b$ depend-x$_i$-on $T_a$ if there are $op_a \in T_a$, $op_b \in T_b$, $op_a \rightarrow op_b$, and $T_1, T_2, ..., T_l$, for $l \geq 0$, with $T_0 = T_a$, $T_{l+1} = T_b$, and $T_{j-1}$ precedes $T_j$, for $1 \leq j \leq l+1$, and there is no $op_j \in T_j$, for $1 \leq j \leq l$, such that $op_j \rightarrow op_b$.

In the replicated database using token approach, if the value of an object is changed after the execution of some operation, the new value has to be posted on the other copies in the same partition. We say the job of posting updates is done by a copier transaction[2], denoted $T_{copier}$.

Definition 2.5 The DEPEND-ON relation. In a replicated data schedule, $T_b$ DEPEND-X-ON $T_a$ if either (1) $T_b$ depend-x$_i$-on $T_a$ for some copy $x_i$, or (2) $T_b$ depend-x$_j$-on $T_{copier}$, for some $T_{copier}$ that copies the value written by $T_a$ on $x_i$ to $x_j$ and there is no $T_c$ which changes the value of $x_j$ before $T_b$ uses it.

In a serial one-copy schedule consisting of logical operations, with serial order given by $<, T_b$ DEPEND-X-ON $T_a$ if $op_a \rightarrow op_b$ for some $op_a$ and $op_b$ executed on $X$, where $op_a \in T_a$ and $op_b \in T_b$, $T_a < T_b$, and there is no $T_c$ such that $op_c \in T_c$, $op_c \rightarrow op_b$ and $T_a < T_c < T_b$.

Definition 2.6 The virtual serializability. A replicated data schedule is virtual serializable if it has the same DEPEND-X-ON relation for every object $X$ as some serial one-copy schedule.

Definition 2.7 A virtual serialization graph ($V-SG$) for a replicated data schedule $S$ is a graph with the nodes being the transactions in $S$ and the edges being the pairs of nodes $(T_a,T_b)$ such that $T_b$ DEPEND-X-ON $T_a$ for some $X$.

Theorem 2.1 A replicated data schedule $S$ of committed transactions is virtual
serializable if and only if $S$ has an acyclic $V$-$SG$.

Proof: $\Rightarrow$ Suppose $S$ is virtual serializable. Assume there is a cycle $T_i-T_{i+1}-...-T_{i+m}-T_i$ in the $V$-$SG$. Let $S_{serial}$ be an equivalent one-copy serial schedule of $S$. Since $S_{serial}$ has the same DEPEND-ON relations as $S$, there are $T_i < T_{i+1}$, ..., $T_{i+m} < T_i$, which is a contradiction.

$\Leftarrow$ Suppose $S$ has an acyclic $V$-$SG$. Let $S_{serial}$ be the serial one-copy schedule with serial order $<$ corresponding to any topological sort of the $V$-$SG$. We show that $S_{serial}$ has the same DEPEND-ON relations as $S$.

Suppose $T_b$ DEPEND-X-ON $T_a$ in $S$. Then $(T_a,T_b)$ is in the $V$-$SG$, since $<$ is a topological sort order, $T_a < T_b$ is in $S_{serial}$. Also since $op_b$ is the first operation, in the same conflict class with $op_a$, executed on $X$ after $op_a$, there is no $T_c$ such that $T_a < T_c < T_b$ and $T_b$ DEPEND-X-ON $T_c$, so $T_b$ DEPEND-X-ON $T_a$ is also in $S_{serial}$.

Suppose $T_b$ DEPEND-X-ON $T_a$ is not in $S$. $(T_a,T_b)$ is not in V-$SG$. From the definition, $T_b$ DEPEND-X-ON $T_a$ is not in $S_{serial}$. $\Box$

Theorem 2.2 The typed-token scheme assures virtual serializability.

Proof: When network does not partition, the concurrency control ensures serializability.

When network partitions, each partition allows only an acyclic partial conflict graph.

We prove that the global conflict graph is also acyclic. Suppose the global conflict graph $G_g$ contains a cycle $T_i-T_{i+1}-...-T_{i+m}-T_i$. Without loss of generality, assume $T_i$, $T_{i+1}$, ..., $T_{i+k}$ are processed in a partition while $T_{i+k+1}$, ..., $T_{i+m}$ are processed in another partition. Since the edge $(T_{i+k},T_{i+k+1})$ exists in $G_g$, there are $op_{i+k} \in T_{i+k}$ and $op_{i+k+1} \in T_{i+k+1}$ such that $op_{i+k} \rightarrow op_{i+k+1}$. But then $op_{i+k}$ and $op_{i+k+1}$ are in the same conflict class and $T_{i+k}$ and $T_{i+k+1}$ should only be allowed in the same partition, a contradiction.$\Box$
2.3. Token replication

Several weaker forms of serializability have been used to increase data availability in the presence of system failures [3,8]. In this section, we propose several ideas to relax the correctness criterion of virtual serializability. The relaxation of virtual serializability leads to the possibility of token replication for further increasing data availability. We explore token replication in two dimensions: 1) based on the semantics of operations represented by a token, 2) based on the semantics of an object.

2.3.1. Operation-based token replication

In applications such as banks, airline reservations, and CAD database, there are many operations that are “benign” in a sense that the execution of these operations will not result in an undesirable state even when the current view of the database is not available. If a token contains more than one operation, some of these operations may be concurrently executed in different partitions without causing a bad state, while others not. This kind of token may be divided into fragments and each fragment is handled differently. We consider three types of operation-based token replication in the following subsections.

**Full replication of a token:** If a token $t_i$ contains only an operation which is revocable, $t_i$ can be replicated wherever the data object exists. In general, we define a revocable operation as an operation which has no external effects and whose effects on the database can be cancelled by executing an inverse operation (even manually). For example, in Example 2.1, *Deposit* is a revocable operation because subtracting the same amount added to an account cancels the effect of the *Deposit*. For a similar reason, *AddCreditRecord* is also revocable. *Withdraw*, on the other hand, is not a revocable operation because handing out cash to a customer is an external effect. Depending on the application, a system programmer may also define the revocable operation in his own ways. For example, he may consider an operation that causes a chain reaction or...
affects an especially important decision to be a non-revocable operation. With the required tokens replicated at all sites, a transaction containing only revocable operations is allowed in every partition.

Partial replication of a token without any extra restriction: In [3], view serializability has been defined. It requires two conditions to be satisfied for correct transaction processing: 1) The update transactions do not create a cycle in the conflict graph, and 2) the read-only transactions considered one at a time in the conflict graph do not create a cycle. This notion of correctness has also been used for locking based concurrency control in hierarchically structured database systems [24]. We adapt this notion of view serializability in replicating typed tokens. Our relaxed correctness criterion requires the following conditions be satisfied: 1) Transactions containing "conflict" operations do not create a cycle in the conflict graph, and 2) the read-only operations considered one at a time do not create edges to form a cycle in the conflict graph. We call this correctness criterion virtual view serializability.

Let $t_i = \{op_{i1}, op_{i2}, \ldots, op_{ik}\}$ be a token containing more than one operation. There are some operations in $t_i$ that do not depend on any other operation but are depended on by some other operations in $t_i$. We call this type of operation independent. If an independent operation $op_{ir}$ is also read-only in the sense that it does not change the state of an object $X$, $op_{ir}$ should be allowed in every partition during network partition. By defining a fragment of $t_i$ to contain this kind of operation, i.e., letting $t_i.read-only = \{ op_{ir} \mid \exists$ no $op_{il}$ such that $op_{il} \rightarrow op_{ir}$, for $1 \leq l \leq n$ and $op_{ir}$ does not change the state of an object $\}$, $t_i.read-only$ can be replicated at all sites. For example, in Example 2.1, ReviewRecord is a read-only independent operation since it does not change the state of $X$ and does not depend on any operations while its results are used by SetFlag and Overdraft. The freedom of allowing the fragment, $t_i.read-only$, of $t_i$ at every site is given by the second condition of virtual view serializability. We show how the freedom of allowing read-only independent operations in multiple partitions is gained in
Section 4.2 when we talk about transaction processing during network partition.

**Partial replication of a token with restriction:** Consider again $t_i = \{ op_{i1}, \ldots, op_{ik} \}$. Besides the $t_i.read-only$ fragment, $t_i$ usually can be divided into two other fragments. For the ease of explanation, we assume $t_i.read-only$ is empty in this part of discussions. The head fragment, denoted by $t_i.head$, contains any operations $op_{ij}$ such that there is some $op_{il}$ and $op_{ij} \rightarrow op_{il}$. The tail fragment, denoted by $t_i.tail$, contains those operations that are not in the head fragment. The head fragment, containing operations whose effects on the current view are crucial to the operations dependent on them, must be restrained to a partition so that the dependent operations are guaranteed to see the correct view. While the head fragment must be in the exclusive mode, the tail fragment can be in the shared mode, *i.e.* it can exist in multiple partitions, under the condition that the current view which it depends on does not change in some other partition. To enforce this condition, we divide the token into the head and tail fragments and only to make sure that at any given time only head or tail fragment exists. During the time when the tail fragment exists, it can be replicated to all sites. This approach provides an alternative use of tokens. We can choose between having a "whole" token or "head-tail" fragmented token. However, it does not necessarily increase the data availability to replicate the tail fragment, since the head and tail fragments cannot co-exist.

### 2.3.2. Object-based token replication

Another dimension of token replication comes from the semantic properties of the objects themselves. Objects may have different degrees of consistency requirements. The degree of consistency required for an object is determined by how it is involved in the specification of integrity constraints of the database. Integrity constraints are the predicates defined on the database which describe the relationships that must hold among the objects and their values. We shall assume all of our predicates are formed from atoms and clauses. It can be shown that all predicates can be expressed in
conjunctive normal form[17].

**Definition 2.8[17]** An atom is a comparison \( x \theta y \), where \( \theta \) is a comparison operator, i.e. 
\( \geq, \geq, =, \neq, < \) or \( \leq \), and \( x \) and \( y \) are either data objects or constants.

**Definition 2.9[17]** An integrity constraint, \( IC \), is a predicate having the form 
\( IC = \land_{i=1}^{n} (\lor_{j=1}^{q} P_{i,j}) \), where each \( P_{i,j} \) is an atom.

Let \( IC^X \) be the set of atoms in \( IC \) that mention data object \( X \). Some observations are made as follows.

1. If \( IC^X = \emptyset \), \( X \) can be considered of requiring no consistency at all. Every copy of \( X \) can have a replica of every token. Some statistics data objects, such as the average age of all employees in a company and the average GPA of all students in a class, etc, are served to let user store his computation and retrieve the data for having a rough view on the statistics. In this case, the object can be made as available as possible with no restriction.

2. If for each \( i \), there is a \( j \) such that any execution of the operations \( \in t_k \) always satisfies \( P_{i,j} \), for \( P_{i,j} \in IC^X \), then \( t_k \) can also be replicated at every copy of \( X \). As a simple example, consider two bank accounts, \( X \) being the savings account and \( Y \) being the checking account belonging to the same customer. Two operations are defined on each account: Deposit and Withdraw. Suppose the integrity constraints on \( X \) and \( Y \) are \( (X \geq 0) \land (X + Y \geq 0) \), i.e. \( IC^X = \{ (X \geq 0), (X + Y \geq 0) \} \) and \( IC^Y = \{ (X + Y \geq 0) \} \). Deposit\(^X \) always satisfies both of the disjunctive clauses in \( IC^X \) and Deposit\(^Y \) always satisfies the only clause in \( IC^Y \); hence, both can be allowed at every site.
3. Transaction Processing Using Typed Token Scheme

We now present the details of transaction processing during both non-failure and network partition modes.

3.1. Processing during normal operations

During normal execution, to carry out an operation $op_k$, a site $S_j$, which processes the transaction requesting $op_k$, needs to obtain token $t_i$, assuming $TokenArray[k] = i$. To obtain $t_i$, $S_j$ checks the linked list of $TokenPool[i]$ to find the first site that is in the same partition with $S_j$. A data structure, called a partition tree, defined in Section 5, contains the ID's of the sites in the same partition with $S_j$. $S_j$ then makes a request to this site, say $S_l$, for $t_i$. Every request for $t_i$ in the same partition is made to the same replica of $t_i$ so that concurrent requests for the same operation on the same object are serialized. If the value of an object attribute changes after the execution of some operation, the new value has to be posted on all copies.

In the implementation, there are two different ways that $S_l$ can grant $t_i$ to $S_j$: (a) physically transfer $t_i$ to $S_j$, then inform all sites to change the ownership of $t_i$ in their token pool, or (b) use the two-phase locking method to lock $t_i$ at $S_l$ for $S_j$. In the first implementation approach, tokens may be lost by a message link failure during communications and hence degrade the availability of the system. We may incorporate the reliable token transfer protocol [16] to prevent the loss of tokens. In the second implementation approach, tokens reside statically on data copies. If every operation allowed on object $X$ is equally likely to be executed on every copy of $X$, a straightforward policy can be used in assigning all $t_i$'s in the system. That is, arbitrarily assign each $t_i$ to a copy of the object until all the $t_i$'s are distributed. Alternatively, if the information about where each operation is likely to be invoked is available, a system designer can smartly distribute all $t_i$'s among copies using this information. Locating $t_i$'s at appropriate sites reduces the number of remote procedure calls and can improve the
efficiency of execution. Moreover, when network partitioning is about to occur, the tokens can be reassigned to the sites where the corresponding functions are more frequently invoked.

3.2. Processing during a network partition

When a failure occurs, if the requested tokens can be accessed, transaction processing proceeds as under normal processing. We now show the transaction processing during network partition by adapting an example from [3] to abstract data type. We show the freedom gained in accessing objects due to the replication of the independent read-only fragment of token.

Example 3.1 Let there be two bank accounts X and Y of the same type as X in Example 3.1. Assume they are fully replicated on sites A, B, and C. Also assume a severe case of network partitioning occurs when each site is in a separate partition. At the time of partition, let site A contain tokens $t_1^X$ and $t_2^X$ for X and $t_3^Y$ for Y, site B contain $t_3^X$ for X and $t_2^Y$ for Y, and site C contain $t_4^X$ and $t_1^Y$ and $t_4^Y$ for Y. Besides, let $t_3^X.read-only = t_3^Y.read-only = \{\text{ReviewRecord}\}$ be replicated at all sites.

The versions of X due to the execution of operations are represented as $X', X'', X'''$, and so on, and the versions of Y are represented as $Y', Y'', Y'''$, and so on. Assume the processing of transactions proceeds as in Figure 4.

Before partitioning occurs, the conflict graph on each site is as follows

\[
\begin{array}{c}
T_1 \\
\ast \\
T_2 \\
\ast
\end{array}
\]

There is no edge in this graph. After the processing of $T_3, T_4, T_5,$ and $T_6$, the conflict graph on each site is as follows

The global conflict graph is
Sites A ... connected ... Site B ... connected ... Site C

Object (Initial versions)  
\( T_1 \) (Deposit X, ReviewRecord X and Y)  
\( T_2 \) (Withdraw Y)  

\( X Y \)  
\( X' Y \)  
\( X Y' \)

---------------------------------- Network partitions ----------------------------------

\( T_3 \) (ReviewRecord X, SetFlag Y)  
\( T_4 \) (SetFlag X, AddCreditRecord Y)  
\( T_5 \) (ReviewRecord Y, Overdraft X)  
\( T_6 \) (Deposit X and Y)  

---------------------------------- Partitions merge ----------------------------------

(communication established)

Figure 4. An example of transaction processing

When the partitions merge, we have to add the edge \( T_5 \rightarrow T_3 \) to the global conflict
graph, since $T_5 \text{ReviewRecord}$ comes before $T_3 \text{SetFlag}$ for $Y$. Similarly, the edge $T_3 \rightarrow T_4$ should be included since $T_3 \text{ReviewRecord}$ comes before $T_4 \text{SetFlag}$ for $X$. The global conflict graph after the establishment of communication is as follows

![Graph](image)

Obviously, there is a cycle in this graph and the first condition of virtual view serializability is not satisfied. However, if only one read-only operation, in this case, $\text{ReviewRecord}$, is considered each time, either $T_5 \rightarrow T_3$ or $T_3 \rightarrow T_4$ does not exist, and the conflict graph is acyclic. Hence the conflict graph satisfies the conditions for virtual view serializability. In this case, to $\text{SetFlag}$ for $X$, $T_4$ sees the record reviewed by $T_1$ and, to $\text{SetFlag}$ for $Y$, $T_3$ also sees the record reviewed by $T_1$. Both $T_3$ and $T_4$ see a consistent state of the database even though it may not be current.

The above example shows that during network partition, each partition preserves some capability although not all, and token replication further contributes toward freedom in allowing transaction processing during network partition.

4. Consistent recovery of database after the network merge

Partitioning and merge of partitions can occur dynamically due to various failures and repairs. We design a scheme to correctly handle it.

When two partitions $P_1$ and $P_2$ are merging, one partition, say $P_1$, has to obtain the recovery log from the other partition, say $P_2$, and rerun the log of $P_2$ so that the effects of the transactions committed in $P_2$ are also posted on $P_1$. However $P_1$ has to distinguish transactions that have also completed in its own log and avoid to commit
them twice. A transaction is completed in a partition if either it commits in the partition or the effects of its operations are posted in the database during partition merge. In other words, for a site $S_i$ to correctly update its database, the effect of each committed transaction should have been only imposed once after network merge. To achieve this, the histories of partitioning and merge occurring to the partitions where $S_i$ and $S_j$ currently reside should somehow be recorded.

We design a data structure, called a partition tree, to store the information about all the partitioning and merging occurred to a site. A similar data structure has been used in [15,16] to determine the majority partition during communication failure. Each site in the system maintains a partition tree, denoted by $p_{tree_{S_i}}$ for $S_i$. We call the period between the instant when a partition is formed and the instant when a change occurs due to a partitioning failure or a partition merge a partition session. Each partition session is identified by a partition ID.

Each node in the partition tree represents a partition session. We use node and partition ID alternatively to indicate a partition session. A node in a partition tree $p_{trees_{S_i}}$ is either undefined or constitutes of the following fields of information: (1) $p_{members_{S_i}}$: the set of sites in the same partition as $S_i$. (2) $p_{logs_{S_i}}$: a log pointer indicating the beginning of transaction execution in the partition session. (3) $p_{CommitTrans_{S_i}}$: ID's of the set of transactions committed during the partition session.

Initially, $p_{trees_{S_i}}$ has a root node representing the whole system as a partition. The root node is said to be at level 0. As soon as $S_i$ discovers that it cannot communicate with another site $S_j$, it starts to create two children for the root. The left child of root is defined and its $p_{members_{S_i}}$ contains all the sites that $S_i$ can communicate with after the first partition failure. The right child of root is undefined because $S_i$ does not know what happens to all the sites that it cannot communicate with.
If a partition failure occurs to $S_i$ afterward, $S_i$ adds two children to the left leaf node in $p_trees_i$: $p_members_{S_i}$ of the left child contains the new partition members and the right child is undefined. Likewise, $p_trees_i$ grows downward. As it grows, the number of levels increases. If a tree has $l$ levels, the nodes at level $l-1$ are said to be at the lowest level. For example, Figures 5(a) and 5(b) contain a partition tree maintained at sites $S_1$ and $S_3$, respectively, after several partitioning failures have occurred.

(a)

Figure 5. Partition trees maintained at $S_1$ and $S_3$ before any merge of partition occurs.

A partition tree $p_trees_i$ is always a binary tree. Before any merge of partitions $p_trees_i$ has the following properties: (1) There are two nodes at each level, the right node is always an undefined leaf and the left node is either a leaf or has two children. (2) The leftmost leaf of $p_trees_i$ represents the current partition in which site $S_i$ is included. (3) Let $l$ be the lowest level of $p_trees_i$. $l$ is not greater than the number of
partition failures occurring in the partition in which \( S_i \) exists, and number of nodes in the tree is \( 2^l + 1 \).

When partitions merge, some of the undefined nodes becomes "defined" because the sites in one partition now can obtain the knowledge of what has happened to the other partition. Figure 6 contains a partition tree which has been modified after a partition merge. In Section 5.1, we give an algorithm for modifying partition trees at merge for the general case. Figure 6 contains \( p\_trees_{1,3} \) which is maintained at \( S_1 \) and \( S_3 \) after (1) and (3) in Figure 5(a) and (b) are merged.

\[
P_{trees_{1,3}}:
\]

\[
\begin{align*}
&P_{trees_{1,3}}: \\
&\{1,2,3,4,5,6,7\} \\
&\{1,2,5,6\} \quad \{3,4,7\} \\
&\{1,2\} \quad \text{undefined} \quad \{3\} \quad \text{undefined} \\
&\{1\} \quad \text{undefined}
\end{align*}
\]

Figure 6. Partition tree maintained at \( S_1 \) and \( S_3 \) after (1) and (3) in Figure 5 merge.

In Figure 6, \( p\_trees_{1,3} \) contains two non-undefined leaves, which together make up the current partition. We call (1) and (3) basic partition units of the current partition \( \{1,3\} \). A basic partition unit is the smallest partition that does not partition any further and before a merger appeared as a leaf in the \( p\_trees_j \) for some \( j \). In general, all the leaf nodes of a partition tree that are not undefined are the basic partition units that
make up the current partition. A merged partition tree also contains all the intermediate nodes, which represent the partitioning histories leading to the basic partition units. For example, in \( S_1 \), \{3,4,7\} was undefined before the merge and is an intermediate node after the merge; \{1,2,5,6\} and \{1,2\} are the intermediate nodes known by \( S_3 \) after the merge.

### 4.1. Merging of partitions

When two partitions \( P_1 \) and \( P_2 \) are merging, we arbitrarily choose a site from each partition as the *partition manager*. Every site in the same partition can access the information available in the partition and acts as merely a representative. In the following, let \( S_1 \) be the partition manager of \( P_1 \) and \( S_2 \) be the partition manager of \( P_2 \). The merging of \( P_1 \) and \( P_2 \) is carried out as follows.

**Step 1.** Choose a *merge manager* between \( S_1 \) and \( S_2 \) to handle the merge task.

Either \( S_1 \) or \( S_2 \) can serve as a merge manager, however, the one that needs to do less work for recovery is a better candidate. In the following steps, w.l.o.g., we assume \( S_1 \) is chosen as the merge manager.

**Step 2.** \( S_2 \) sends the recovery log maintained at its site, denoted by \( \log_{S_2} \), and the partition tree \( p_{\text{tree}_{S_2}} \) to \( S_1 \).

**Step 3.** \( S_1 \) runs the transactions logged in \( \log_{S_2} \) that are completed in \( P_2 \) but not in \( P_1 \).

A transaction is completed in a partition if either it commits in the partition or the effects of its operations have been accounted in the partition during some previous partition merge. First \( S_1 \) has to determine the set of transactions that have completed in \( P_2 \) but not in \( P_1 \). Next, it reruns this set of transactions and integrates the two partition trees to the one representing the merged partition. Details are described in Section 4.1.1.
Step 4. Update the database objects and the partition trees.

After running the set of transactions described in Step 3, $S_1$ has brought the database objects to their final values and updated $p_{trees1}$ to reflect the new partition status. $S_1$ sends the new values of objects and $p_{trees1}$ to every site in $P_1 \cup P_2$. The latter has formed a new partition.

4.1.1. Merge the partition trees and execute the recovery log

When partitions merge, $S_1$ has to: (1) find the set of transactions that completed in $P_2$ but not in $P_1$, (2) run this set of transactions on $log_{S_2}$, (3) modify $p_{trees1}$ to represent the new partition formed by merge.

In general, $P_1$ may be a partition formed by merging two previously existent partitions $P_1'$ and $P_1''$. Either one or both of $P_1'$ and $P_1''$ might in turn be partitions made by the merge of two other partitions. Similar arguments can again be applied to these partitions until the partitions are the basic partition units. Let $P_1^m = \{ P_{11}, P_{12}, P_{13}, ... \}$ be the set of basic partition units that determine $P_1$. $P_{1i}$'s are disjoint. Similarly, let $P_2^m = \{ P_{21}, P_{22}, P_{23}, ... \}$ be the set of basic partition units of $P_2$.

Before we describe how $S_1$ carries out the above three tasks, we need a notion of common ancestor.

A common ancestor $P_c$ of two partition sessions $P_i$ and $P_j$ is defined in the context of partition tree as follows: If there exists a node $N_1$ in $p_{trees1}$ and a node $N_2$ in $p_{trees2}$ such that $p_{members1}(P_i) \subseteq p_{members1}(N_1)$, $p_{members1}(P_j) \subseteq p_{members2}(N_2)$ and $p_{members1}(N_1) = p_{members2}(N_2)$, we say that $N_1$ and $N_2$ both represent a partition session $P_c$ which is a common ancestor of $P_i$ and $P_j$. For example, $\{1,2,5,6\}$ and $\{1,2,3,4,5,6,7\}$ in Figures 6 and 7(a) are the common ancestors of $\{1\}$ of Figure 6 and $\{5,6\}$ of Figure 7(a). A common ancestor at the lowest level is called the latest common ancestor.
Procedure *FindIncompletTran* (in Figure 9) finds the set of transactions, called *IncompletTran*, that completed in $P_2$ but in $P_1$. As indicated by the examples in Section 4.2, if a partition merge has occurred, the current partition is represented by multiple leaf nodes and we need to directly deal with these components for general merge cases. The procedure *FindIncompletTran* basically finds the right place in $p_trees_1$ for each basic partition unit $P_{2i}$ to fit. $P_{2i}$ will replace some undefined node in $p_trees_1$. Besides $P_{2i}$, some intermediate nodes, denoted by $P_{intermed}$, which indicate the histories of partitioning and merge in which $P_{2i}$ was involved and was missed by $S_1$, should also be added to $p_trees_1$. Since any common ancestor of $P_{2i}$ and $P_{1j}$, for any $j$, must contain the same information in both $p_trees_2$ and $p_trees_1$, $P_{intermed}$'s are those nodes lying on the path from the latest common ancestor $P_c$ of $P_{2i}$ and $P_{1j}$ to $P_{2i}$. Let $node_{new}$ indicate any of $P_{intermed}$'s and $P_{2i}$. Obviously, transactions that are in $p_CommitTrans_2(node_{new})$, for any $node_{new}$, are completed in $P_2$ but not in $P_1$. Moreover, because we cannot assume the instantaneous detection of network partition and terminate on-going transactions at the partitioning, there may be some transactions that were initiated in $P_c$, but committed only at $S_2$, before $S_2$ is aware that $S_1$ has been in a separated partition. Such transactions are, of course, included in $p_CommitTrans_2(P_c)$ but not in $p_CommitTrans_1(P_c)$. We denote this set of transactions by $DiffTrans_{P_c}$, i.e. $DiffTrans_{P_c} = p_CommitTrans_2(P_c) - p_CommitTrans_1(P_c)$. Hence during the incorporation of $P_{2i}$ into $p_trees_1$, we have found a subset of transactions that have completed in $P_2$ but not in $P_1$. We denote this subset of transactions by $IncompletTrans_{P_2}$. $IncompletTrans_{P_2}$ is the union of $DiffTrans_{P_c}$ and $p_CommitTrans_2(node_{new})$ for all $node_{new}$'s and $IncompletTrans$ is the union of $IncompletTrans_{P_2}$ for all $P_{2i}$'s.

To carry out the second task, $S_1$ needs to run $IncompletTrans_{P_2}$ for each $P_{2i}$. It must execute the entries between $p_logs_2(P_c)$ and $p_logs_2(node_{new^{1st}})$ on $logs_2$ for transactions in $DiffTrans_{P_c}$, and all the entries from $p_logs_2(node_{new^{1st}})$ to the last entry
logged during $P_{2i}$. Note that $node_{new}^{lst}$ is the first $node_{new}$ below $P_c$. When these entries are rerun by $S_1$, they are logged in logs, and the pointers $p_{logs}(P_c)$ and $p_{logs}(node_{new})$ are set to reflect their new addresses in logs. Resetting these pointers and the addition of $node_{new}$'s as described above are the necessary modifications needed on $p_{trees}$.

Before we formally give our algorithm in Figure 9, we illustrate the ideas via the following example.

Example 4.1 Let $P_1$ be $(1,3)$ in Figure 6 and $P_2$ be a partition $(4,5,6)$ presented in Figure 7. $P_{2m} = [(5,6), (4)]$. We need to find the proper undefined node in $p_{trees}$ and replace it by $(5,6)$. First we go down from the root $(1,2,3,4,5,6,7)$ of $p_{trees}$ to its left child $(1,2,5,6)$ since the latter contains $(5,6)$.

Next we notice the left child of $(1,2,5,6)$ is defined but does not contain $(5,6)$. We
cannot go further to the left child, \{1,2,5,6\} is the latest common ancestor of \{5,6\} with some basic partition unit of \(P_1\), so we go to the right child of \{1,2,5,6\}. The right child is undefined and we cannot go further hence \{5,6\} replaces the right child of \{1,2,5,6\}. Since in \(p_{trees_{4,5,6}}\), \{1,2,5,6\} is also the parent of \{5,6\}. Since there is no intermediate node between \{5,6\} and \{1,2,5,6\}, we do not need to add any intermediate node between these two nodes in \(p_{trees_{5,1}}\). Next we need to find the proper undefined node to be replaced by \{4\}. Again we start from \{1,2,3,4,5,6,7\} and go down to its right child since the right child \{3,4,7\} contains \{4\}. Then we find that the left child of \{3,4,7\} is defined but does not contain \{4\}, so \{3,4,7\} is the latest common ancestor of \{4\} with some basic partition unit of \(P_1\). We go to the right child of \{3,4,7\} and find that it is undefined. \{4\} should replace the right child of \{3,4,7\}. In \(p_{trees_{4,5,6}}\), \{4\} is in the left subtree but is not a child of \{3,4,7\}. Every node in the left subtree of \{3,4,7\} means some missing partitioning information to \(S_1\). Hence we replace the undefined child of \{3,4,7\} in \(p_{trees_{5,1}}\) by the left subtree of \{3,4,7\} in \(p_{trees_{4,5,6}}\). The resulted partition tree after \{1,3\} merged with \{4,5,6\} is shown in Figure 8. \{4,7\} is an intermediate node added between \{3,4,7\} and \{4\}.

4.1.2. Reset the recovery log and the partition tree

When the partitioned network is fully recovered, there is no "undefined" node in any partition tree and the partition trees can be deleted completely. At this time, by checkpointing the database state \[6,7\], the recovery log can also be reset to the beginning.

5. Discussion

The typed token scheme leads to increased availability, performance, and low message traffic. Using the semantic information of objects and operations, as shown in Example 3.1 and 3.2, more than one token can be assigned to an object. With multiple
tokens distributed in the system, the probability for each site to have some partial capability to allow transaction processing during network partition exists. Since to execute an operation, one only has to obtain the corresponding token without requesting the consensus from a significant number of sites, higher performance and lower message traffic can be expected than the various quorum methods. One flexibility provided by this scheme is it allows new copies of an object to be integrated into the system any time without affecting the execution at other sites. The quorum methods need reconfiguration or change of quorum assignment [4].

When a site discovers that it cannot communicate with some other node, it can invoke a simple protocol to determine the set of sites that are in its partition and can create a new node in the partition tree. Due to the lag of the detection of partitioning, the information stored at two sites, say \( S_1 \) and \( S_2 \), may be conflict in a sense that some nodes in \( p_{\text{trees}}_1 \) and \( p_{\text{trees}}_2 \) representing the same partition contain different values.
procedure FindIncompleTran (p_trees1, p_trees2)
  let $P^m_2 = \{ P_{21}, P_{22}, P_{23}, \ldots \}$;
  for $j = 1, 2, 3, \ldots$ do
    node = MergeTree($P_{2j}$, p_trees1);
    IncompleTran = p_CommitTrans2(node.parent) - p_trees1(node.parent);
    /* node.parent is a latest common ancestor of $P_{2j}$ and a basic partition
     unit of $P_1$ */
    while $p_{members_2}(node) \neq p_{members_2}(P_{2j})$ do
      IncompleTran = IncompleTran $\cup$ p_CommitTrans1(node);
      node = node.leftchild;
    endwhile
    IncompleTran = IncompleTran $\cup$ p_CommitTrans1(node);
  endfor
end

procedure MergeTree($P_{2j}$, p_trees1)
  node11 = root of p_trees1;
  node12 = node11;
  while node11 is not a leaf do
    node12 = node11;
    if $p_{members_2}(P_{2j}) \subset p_{members_1}(node12.leftchild)$ then
      node11 = node12.leftchild;
    else
      node11 = node11.rightchild;
    endwhile
  let node22 be the node in p_trees2 such that $p_{members_2}(node22) = p_{members_1}(node12)$; /* node12 and node22 represent the latest common ancestor */
  CopySubtree(node22.leftchild, node12.rightchild); /* copy the left subtree of node22 in p_trees2 to the right subtree of node12 in p_trees1 */
  return(node11); /* the first node_new added to p_trees1 */
end

Figure 9 Procedure FindIncompleTran

However, it is easy to see that the conflict is limited in the $p_{CommitTran}$ field of the latest common ancestor and does not propagate to other node. Hence the correctness of the partition tree maintained at each site does not rely on the instantaneous detection of network partition. When partitions merge, the traversal and the update on the two partition trees require work of $O(2n)$, where $n$ is the number of nodes in a partition tree and is no greater than $(2^{l+1} - 1)$ for $l$ being the total number of partitionings that occurred to the site since it started the current partition tree. The partition tree can be reset when there is no partitioning failure in the system by incorporating a checkpointing scheme as discussed in Section 4.1.2.
6. Conclusion

We have presented a typed token approach for managing replicated data in distributed database systems and a scheme for consistent recovery of database for dynamic partitioning and merging. The typed token approach allows multiple tokens and their replicas distributed in multiple partitions to increase data availability during network partition. At network merge, the scheme of using partition tree for consistent recovery of database allows missing updates to be efficiently identified in the recovery log.

8. References


