UPDATE AND MERGE OF PARTITIONED DISTRIBUTED SYSTEMS

J.A. Anyanwu and B. Randell

Abstract

An Algorithm is presented for providing continued processing in a distributed system in the presence of network partitioning. It involves a protocol for automatic detection and resolution of mutual inconsistencies between replicated data items in partitioned distributed systems which does not require a prior knowledge of the semantics of operations on data structures. The algorithm, which is based on the use of majority and minority partitions, allows autonomous processing in individual nodes when a network partitions thereby providing continued processing despite network failures. Actual (as opposed to potential) conflicts between the operations in different partitions are detected and resolved during a merge action. The resolution of actual conflicts avoids the unnecessary UNDO processing which arises when potential rather than just those conflicts which actually occur are resolved. An implementation of the proposed algorithm in a hierarchical distributed system is discussed.

Series Editor: M. J. Elphick

© 1985 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

ANYANWU, Josephine Ada.

Update and merge of partitioned distributed systems.

By J.A. Anyanwu, B. Randell.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1985.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 204.)

Added entries
RANDELL, Brian.
UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 204.

Abstract

An Algorithm is presented for providing continued processing in a distributed system in the presence of network partitioning. It involves a protocol for automatic detection and resolution of mutual inconsistencies between replicated data items in partitioned distributed systems which does not require a prior knowledge of the semantics of operations on data structures. The algorithm, which is based on the use of majority and minority partitions, allows autonomous processing in individual nodes when a network partitions thereby providing continued processing despite network failures. Actual (as opposed to potential) conflicts between the operations in different partitions are detected and resolved during a merge action. The resolution of actual conflicts avoids the unnecessary UNDO processing which arises when potential rather than just those conflicts which actually occur are resolved. An implementation of the proposed algorithm in a hierarchical distributed system is discussed.

About the author

Dr. J.A. Anyanwu has now completed her Ph.D. study at the Computing Laboratory of the University of Newcastle upon Tyne. Professor B. Randell has been a Professor of Computing Science in the Computing Laboratory since 1969.

Suggested keywords

ACTUAL CONFLICTS, POTENTIAL CONFLICTS
CONFLICT DETECTION AND RESOLUTION
NETWORK PARTITIONING

Suggested classmarks (primary classmark underlined)

Dewey (18th): 001.64404
U.D.C. 519.687
UPDATE AND MERGE OF PARTITIONED DISTRIBUTED SYSTEMS

J. A. Anyanwu
B. Randell

Computing Laboratory
University of Newcastle Upon Tyne
NE1 7RU, U. K.

ABSTRACT

An algorithm is presented for providing continued processing in a distributed system in the presence of network partitioning. It involves a protocol for automatic detection and resolution of mutual inconsistencies between replicated data items in partitioned distributed systems which does not require a prior knowledge of the semantics of operations on data structures. The algorithm, which is based on the use of majority and minority partitions, allows autonomous processing in individual nodes when a network partitions thereby providing continued processing despite network failures. Actual (as opposed to potential) conflicts between the operations in different partitions are detected and resolved during a merge action. The resolution of actual conflicts avoids the unnecessary UNDO processing which arises when potential rather than just those conflicts which actually occur are resolved. An implementation of the proposed algorithm in a hierarchical distributed system is discussed.

1. INTRODUCTION

A major motivation for distributed systems is to encourage cooperation between individual nodes in a network with respect to the provision of user requested-services. In particular, computers in a distributed system may cooperate in order to improve the reliability and availability of resources. Some of the areas of cooperation between computers are with respect to (a) replicated data, (b) distributed data and (c) distributed tasks.

Data is often replicated in distributed systems in order to provide resiliency to node and network failures. Failure of the machine storing
the only copy (or copies) of a data item will prevent the completion of any task which needs that data. The provision of replicas on different machines means that any task which needs the replicated data has a chance to proceed as long as there is a copy on an accessible working node. Replication of data on different machines could also be used to make data entities more readily accessible by keeping data nearer to where it is being used. It also enables many nodes to service requests for the same information in parallel.

Distribution of data and tasks is used to improve response time, since independent parts of a computation could be carried out in parallel on different machines. The service required by a task which originated in one machine is therefore provided by the cooperative activity of a set of computers. When there is such a dependency between computers, there is the possibility that the failure of the communication network could interfere with the service which a set of cooperating computers is expected to provide by obstructing communication between these computers. Such a communication failure divides (partitions) a distributed system into non-communicating sub-systems called partitions. The data which is stored by a distributed system may therefore be left in an inconsistent state because of lack of synchronisation between the update operations performed by computers in different partitions. Moreover, since many distributed systems use data replication for reliability and availability purposes, there is also the need to ensure that mutual consistency between replicated entities is maintained in spite of network partitioning.

Many algorithms have been proposed for maintaining mutual consistency of resources [1], [2], [3], [4], [5], [6]. However, these have not directly addressed the problem of resolving inconsistencies that can arise among replicated resources after a network had been partitioned. One drastic "solution" to the inconsistency problem resulting from network partitioning is to cease all operations in the distributed system until the network is fully reconnected. Another conventional, and almost equally drastic, solution is to allow processing to continue only in one partition. It is however often desirable to try to keep several or indeed all sites operational when a distributed system partitions. Such autonomous operation by various partitions clearly could result in mutual inconsistency among multiple copies of resources. It is therefore necessary that when the network reconnects, individual partitions should compare their update operations, detect conflicts and resolve them before partitions are allowed to merge.

Consider an airline reservation system which spans more than one node. The seat reservation activities of the various nodes have to be coordinated in order to avoid having flights overbooked. Network partitioning could interfere with the operation of such a system. However, many systems would require that processing (seat reservation in this
case) be continued during partitioning. Not allowing partitioned processing could result (in the worst case) in all the seats in a scheduled flight being empty. Another application area in which partitioned processing would be desirable is a military command and control system. In this environment an enemy attack could disrupt the communication system thereby resulting in a partitioning of the network. It is usually essential in such circumstances that processing in individual nodes be continued in the presence of such enemy activities. Distributed workstations are examples of systems where partitioned processing can in fact be considered to be part of the normal processing of a system. In these systems, individual nodes may be disconnected (at night and taken home, say) and processing continued in these individual nodes, using copies of generally available files. Usually, a workstation would have a central data store and each node would take copies of (possibly different) data items from the store so as to carry out individual processing on these items. In such environments, processing after network partitioning would yield to a more traditional solution, namely, the primary site approach. This would involve applying serially the update operations of each node to the central store. The algorithm which is presented in this paper, however, addresses the workstation problem in the worst case when every node takes a copy of the same data items from the central store and carries out autonomous processing. Then conflicts need to be detected and resolved. In each of these examples it is desirable to allow at least some degree of autonomous processing whilst a system is partitioned and to detect and resolve conflicts when the network reconnects.

This paper presents an algorithm for updating and merging a partitioned distributed system in such a way as to maintain the consistency of the data which is stored by the system. In particular, it addresses the problem of detecting and resolving mutual inconsistencies between replicated data items in a partitioned distributed system. Some previous work on conflict detection and resolution can be found in [7], [8], [9], [10]. However, it is usually accepted that automatic resolution of mutual inconsistency between replicated entities is not generally possible except when the semantics of operations on the entities are known a priori. Algorithms which rely on the knowledge of operation semantics often require an analysis of the operations which can be applied to a data type or prior declaration of the readset and writeset of a task. (The readset and writeset of a task refer to those data items which a task reads from or writes to.)

It is not always possible to determine beforehand all the resource requirements of an application. Also an analysis of all the operations which can be applied to a data type is usually rather difficult. Consider database applications where records to be read or written might depend on the value of fields in other database records. It would be difficult to know beforehand the complete set of resources which such an application program requires. In such circumstances, it might be
necessary to declare the entire database as the readset and writset of the application program. A more desirable solution is to allow application programs to acquire necessary resources as the need arises.

Another method which is used in conflict detection and resolution is the graph-theoretic approach such as is used in [11] and [12]. This approach is based on the ability to produce a global serialisable schedule of the combined set of transactions from all partitions. The algorithm which is presented in [12] guarantees that it can be determined in polynomial time which transactions must be backed out so as to keep the combined set of transactions serializable. There is the possibility of having large back-out sets when such a method is used and the problem of minimising the extensive UNDO (or cascading rollback) of transactions which might arise is difficult to solve except in very restricted cases.

Many of the algorithms which have been discussed so far address the problem of automatic resolution of inconsistency in partitioned systems only for special data structures (such as directories and mail-boxes) where the semantics of operations are known and, more importantly, simple. Other algorithms detect and resolve "potential" rather than actual conflicts (i.e. conflicts which actually occur) between transactions in different partitions. The algorithm which is presented in this paper is concerned with automatic detection and resolution only of actual conflicts between update operations in different partitions. The resolution of actual conflicts reduces extensive UNDO processing which could arise when potential rather than actual conflicts are resolved. The algorithm dynamically determines the writset of an application program thereby avoiding the need for an a priori knowledge of the semantics of operations on data structures. The method, which is based on the use of majority and minority partitions, permits autonomous operations in individual nodes of a partitioned system. (The concept of majority and minority partitions is explained in the next section.) The nodes in the majority partition would freely read from and write to accessible copies of objects. However, update operations of a minority partition are not committed (performed on the actual objects) until after a merge action. The result of such uncommitted operations are accessible only to tasks which produced them. All other users access the committed copies of objects which contain values of data items before a node became separated from the majority partition. When a network reconnects, conflicting update operations from different partitions are detected and conflicts are resolved by backing out uncommitted operations.

This scheme has the advantage of allowing processing in all partitions (not just in the majority partition) while avoiding the need to resolve conflicts by arbitrarily choosing transactions which are to be backed out. Resolving conflicts by arbitrarily choosing transactions which are to be backed out in order to start the resolution process.
could lead to unpredictable and expensive cascading rollback operations. There is however the possibility that a majority partition cannot be formed. In such circumstances, all partitions would continue minority partition processing and would delay commitment of update operations until a majority partition can be formed. This scheme is then somewhat similar to a transactional system which delays commitment of all operations until recovery is completed. Unlike a transactional system however, whenever there is a majority partition the operations of that partition are always committed. The work which is presented in this paper is in fact similar to that in [9] with regard to the use of a majority consensus algorithm for processing update operations. The majority consensus algorithm is also used in this paper by the merge protocol to determine the existence of the majority partition. Unlike the algorithm in [9] however, the merge protocol which is presented here considers the resolution of mutual inconsistency without an analysis of operations which can be performed on data structures.

The next section discusses the proposed partitioning algorithm. It first discusses some basic concepts and then presents the partitioned update and merge protocols. An implementation of the proposed partitioning algorithm in a hierarchical distributed system is considered in section 3. This section considers the creation and access strategies for replicated data. Section 4 presents some concluding remarks.

2. PARTITIONED PROCESSING

The partitioning issue is concerned with problems which arise when a communication system breaks down in environments where there are dependencies between nodes in a network of computers. The concept of a partition is therefore discussed only in the presence of such dependencies. In this paper, the dependency between individual nodes which is of concern is what we will call a replication dependency. Partitioned processing is therefore invoked only for replicated objects such as files. Partitioned processing as discussed here is therefore not concerned with the effects of communications between tasks which do not result in the reading or writing of replicated data items. In order to simplify the discussion which follows we assume that each replicated data item has a copy in the same subset of nodes in the network. Each node can however store both replicated and non-replicated data.

2.1. BASIC CONCEPTS

We shall consider a set of nodes which are in communication with each other to constitute a partition. It might be helpful to think of a partition as a single logical node consisting of a set of communicating nodes. The universal node will be used to refer to the set of all nodes in a distributed system which store replicated data items. The term "majority partition" will be taken to mean the set of nodes which store
replicated data items and which are in communication with a majority (absolute majority) of the nodes in the universal node. Any partition which is not the majority partition will be referred to as a minority partition. The assumption that replicas are stored on the same subset of nodes ensures that a partition is a majority or minority partition with respect to all objects in the network. A relaxation of this assumption so as to allow copies of objects to be stored on different nodes in the network could result in a partition being a majority partition for a set of objects and a minority partition for a different set of objects. The notion of a majority or minority partition will then have to be defined with respect to a specific object or set of objects. However, the use of the above assumption helps to simplify the discussion which follows.

All nodes maintain a "maj status" record which consists of a "maj switch" and a "maj number". The "maj switch" is turned on for all nodes in the majority partition. This switch is turned off when a node becomes a member of a minority partition. It is therefore easy to determine whether or not a node is in the majority partition. The "maj number" records the last majority consensus agreement action in which a node participated.

In order to allow minority updating and to be able to resolve conflicts which might arise, we use the following data structures: An entity list is of the form (own; inherited) where "own" is an ordered list of integers which contains the identification numbers of all objects which were modified by a partition since the partition was formed. The "inherited" part of the entity list contains the identification numbers of all objects updated by other minority partitions and which were passed to this partition after a minority partition had been further partitioned. The identity of the partition from which such objects were inherited is also recorded.

An entity list of the form \((E_1, E_2, \ldots E_m; \ E_k(A), \ E_j(B), \ldots)\) is therefore interpreted to mean that \(E_1, E_2, \ldots, E_m\) are identification numbers of the objects which were updated by this partition. \(E_k\) and \(E_j\) are identification numbers of objects updated by the minority partitions \(A\) and \(B\) respectively which were inherited by this partition.

An update set is defined to be a collection of records where each record contains the following:
n: node identifier
tl: task identifier
ob: object identifier
u: update request = (read, write)
d: data to be written

An update set is similar to an intentions list in [13] except that instead of identifying just a transaction, it also identifies a node which is involved in an update operation. The next section discusses the proposed update algorithm.

2.2. A PARTITIONED UPDATE ALGORITHM

The algorithm which is discussed in this paper can be used in any environment where fault tolerance for network partitioning is desired. However, since we will later be discussing the implementation of this algorithm in an operating system environment which stores files, our discussion will henceforth be restricted to files.

The algorithm presented here allows continued processing (reading and writing of files) in all partitions when a network partitions. However, only the partition which has a majority of the copies of a file (majority partition) will update the actual copies of that file. The update requests of a minority partition are delayed so as to be incorporated into the actual copy after that partition has rejoined the majority partition. (The actual copy of a file refers to the committed copy of that file.) This means that a node in the majority partition would freely read from and write to accessible copies of files. We shall however make the following assumptions about the activities of a minority partition:

AS1: The update operations of a minority partition are assumed to be initiated at the time when that partition rejoins the majority.

AS2: Until after a minority partition has merged with the majority partition the result of update activities of that partition are accessible only to tasks which produced them.

The algorithm for updating files is as follows: Each node in a minority partition maintains an update set for each file which it desires to update and stores its update requests in that set. (The
update set or entity list for a partition consists of the union of the update sets or entity lists respectively associated with the files in the partition.) The contents of an update set associated with a file are applied to the actual file copy during a merge action if no conflict is detected. (A minority partition is therefore not allowed to update the actual copy of a file.) An update set serves as a cache which is used to record requested operations on files by nodes in a minority partition. This allows a file to be left in the state it was in when the partitioning occurred, and enabling these update requests to be carried out later during a merge operation. Such a cache is also searched when a minority partition issues a read request. If the required file has an update set associated with it, the information in the update set is made accessible if the identification number of the task requesting the access is the same as the "task identification" entry of the update set. (By this means, and assuming that tasks communicate only via the use of shared replicated files, we can avoid causing inconsistent updates. Messages sent between tasks which do not result in the reading or writing of replicated files do not cause inconsistency in replicated data.)

Nodes in the majority partition apply their update operations to their local file copies after an agreement to perform the update operations has been obtained from every node in the majority partition. Obtaining agreement should not usually be difficult since nodes in the majority partition are in communication with each other. Such an agreement is obtained by executing a majority consensus algorithm which is initiated by the node that is requesting an update operation. If majority consensus is obtained and there is no "dissenting" vote from any node in the majority partition, an agreement is said to have been obtained. If agreement is not obtained, the update request is rejected and the user resubmits the request at a later time. Each partition (majority or minority) maintains an entity list for each file which was modified by the partition. When a file whose update operation was inherited from another minority partition is updated, the entity list is adjusted by erasing the entry associated with the file in the "inherited" part of the list and adding an entry for the file in the "own" part of the entity list. The use of an entity list enables the update and merge protocol to determine the update class of each partition dynamically and thus enables easy conflict detection. It is therefore not necessary that a task declares its update class beforehand.

The inherited part of the list is used to ensure that when a minority partition further divides into two or more minority partitions (and therefore ceases to exist) the updates which were performed by that minority partition are not lost but instead are inherited by the "child" minority partitions. It also enables the merge algorithm to recognise updates which are recorded in two separate partitions. This happens when two minority partitions inherit update operations from a parent minority partition. Inheritance from a majority partition is not recorded since
these do not cause any conflicts to be detected. This is because conflict detection and resolution are concerned with update operations which were carried out by a minority partition after that partition had been separated from the majority partition. Consequently, update operations which are inherited from a majority partition cannot cause a conflict with the majority partition itself.

Conflict detection is carried out when a site in the minority partition requests a merge. A merge request is made after such a site has determined that it has regained communication with another partition in the network. The next section discusses when and how a merge operation is initiated and carried out.

2.3. A NETWORK MERGE ALGORITHM

A node in a minority partition periodically sends "are you alive" messages to other accessible nodes in order to determine whether it has regained communication with another partition. On determining that it has regained communication with another partition it sends a merge request to all accessible nodes. The merge protocol then determines whether a majority partition already exists and whether the requesting node is in communication with it. If the requesting node is in communication with the majority partition, a merge between the majority and the minority partition takes place. Otherwise, the scheme for merging minority partitions which is discussed in the next section is followed.

In order to ascertain the existence of the majority partition, the partitioning layer in the requesting minority node executes a majority consensus algorithm. (The partitioning layer is discussed in the section on "Implementation Issues".) The response by each node to a majority consensus request would contain a "yes" or "no" vote as well as the value of the "maj status" record for the node. It is then easy to determine whether a majority partition already exists by examining the status of each node. This will reveal the number of nodes which have their "maj switch" set. These nodes are also expected to have the same "maj number" since all nodes in the majority partition carry out the same update operations. Notice that the "maj switch" for each node is correctly adjusted after a majority consensus algorithm has been executed.

A merge request between the majority and minority partitions is accepted either conditionally or unconditionally. Unconditional acceptance is given when the update classes of the majority and minority partitions do not conflict; otherwise a conditional acceptance is given. (The update class of a partition is determined by the data items which it has updated as indicated in its entity list.) In order that the operations of two partitions may be considered conflict-free, it is usually required that the invariant
INV: $W_i \land R_j = 0 \land R_i \land W_j = 0 \land W_i \land W_j = 0$

hold. (Here $i$ and $j$ are the majority and minority partitions respectively; $R_i$ and $W_i$ are the readset and writerset of partition $i$). However, because of assumptions AS1 and AS2, the activities of the minority partition cannot influence the data which is read by the majority partition. We can therefore remove the first two requirements of the invariant INV and require only that $W_i \land W_j = 0$ holds. The activities of two partitions can therefore be considered to conflict if their associated entity lists have one or more "own" entries in common. Identical entries in the "inherited" part of two entity lists indicate that the associated partitions had inherited update operations from the same minority partition. Such entries do not cause a conflict to be detected but help ensure that the same update operation which is presented to the merge protocol by two different partitions is applied only once to the "actual" file copy.

This conflict detection process is carried out independently for each file which is modified by a minority partition. If no conflict is detected, and the merge request is unconditionally accepted, the update sets of the minority partition would be applied to the associated file copies in the majority partition and the entity lists of the minority partition would be added to the entity lists of the associated files in the majority partition. Conditional acceptance of a merge request requires that the affected minority partition discards its update sets and entity lists for which conflicts were detected before rejoining the majority partition. After a successful merge operation, a minority partition initialises its file copies and entity lists from the (merged) file copies and entity lists of the majority partition. The minority partition then ceases to exist.

Figure 1 gives an example of what happens during a merge operation in the case of a file which is replicated on five machines U1, U2, U3, U4 and U5. The notation name: Majp[node list] or name: Minp[node list] refers to a majority or minority partition respectively, called name. The "node list" contains the names of the nodes in the partition. The entity list "EL" is used to record the identification numbers of data items in a file which were updated by a partition. Update sets are used to store the update requests and data which are associated with such requests. Sub-partitions inherit the entity lists and update sets of their parent partition. An entity list associated with a file would be empty only if no update operations were carried out on the file by that partition or inherited from another minority partition.
A : Unode[U1,U2,U3,U4,U5]
    /  
   /   
  B : Minp[U3,U5]    C : Majp[U1,U2,U4]
  (EL = E1)  (EL = E3,E14,E20,E22)
  /      
 /        
(EL = E14; E1(B)) (EL = E6,E11; E1(B))
    /  
   /   
-      -
-      -
-      -
-      -
-      -
-      -
-      -
-      -

Figure 1: Merge Graph

In figure 1, the universal node (Unode) A consists of the five nodes which store copies of the file. From the figure, it can be seen that the minority partitions B, D and G each carried out update operations since their entity lists are non-empty. It can also be seen that the update operation carried out by B on data entity 1 as reflected in "E1" was inherited by the sub-partitions D and G.

Suppose now that the minority partition G requests a merge with the majority partition C. The entity lists of the two partitions will be compared. From figure 1 it can be seen that these two entity lists have no entry in common which means that there is no conflict between the update operations of these two partitions. The merge request will therefore be accepted unconditionally. The contents of the update set of the minority partition G will be applied to the majority partition's file copy and nodes which were in G will reinitialise their entity lists with respect to this file from the majority partition's (merged) entity list.

Now consider instead a merge request by the minority partition D with the majority partition C. The entity lists of these two partitions have an "own" entry in common namely, "E14". This means that a conflict exists. A conditional acceptance to the merge request will be given. The minority partition D will be required to discard the update set associated with the file before rejoining the majority partition. It will also reinitialise its entity list from the entity list of the majority partition. Notice that the majority partition does not maintain
an update set, since its operations are applied directly to the majority partition's file copy.

2.4. MERGING OF MINORITY PARTITIONS

The merge protocol which was described in the previous section allows a merge action between a majority and a minority partition. However, a network could partition in such a way that there would be no majority partition though one could come into existence again through the merging of minority partitions. An extension to the basic algorithm which would allow the merge of minority partitions is generally not difficult.

One approach would be to merge minority partitions by combining their update sets and entity lists so that each site in the merged partitions contains the same information in these two data structures. This would be done after the entity lists of the two minority partitions have been compared so as to ensure that there is no conflict between the operations of the two partitions. If a conflict is detected, the operations of the partition which carried out the least number of update operations will be backed out by discarding its update set. The partition which was backed out would then initialise its entity list and update set from those of the partition with which it has been merged. Then a majority consensus algorithm would be executed to determine if the newly merged partition has formed a majority partition. Each site knows the number of sites in the universal node (Unode) so it is easy to determine when a merge action results in the majority partition being formed. If the newly merged partition contains a majority of the nodes in Unode, then that partition constitutes the majority partition, in which case its update set would be applied to the local file copy of each site in the newly formed majority partition. The update set would then be discarded since majority partitions do not maintain update sets. On the other hand if the newly merged partition does not form a majority partition, it would remain as a minority partition and would continue to follow minority processing procedures.

3. IMPLEMENTATION ISSUES

This section considers the implementation of the proposed algorithm in a UNIX-based distributed system called UNIX United. A brief overview of the UNIX United distributed system which appears in [14] is given here for the benefit of the reader. A full description of the architecture of a UNIX United system is given in [15].

3.1. OVERVIEW OF A UNIX UNITED DISTRIBUTED SYSTEM

A UNIX United system is usually made up of a (possibly large) set of standard UNIX systems interconnected by a communication network. The
naming scheme used by such a system joins together the naming structures of the individual UNIX systems into a single naming tree such that these component systems appear as directories in that naming tree. This enables a legitimate user on any of the UNIX systems to access files or devices of any other component system within the UNIX United framework as though these devices were part of the user's own system. Figure 2 shows a UNIX United naming tree containing five UNIX systems U1, U2, U3, U4, U5.

```
(base)
   /\  /
 U1 U4 /\  /
   /\  U3 /\  /
 U2 /\  U5 /\  f1 f2
```

Figure 2 An Example of a UNIX United Naming Tree

Following normal UNIX naming convention, names starting with "/" indicate that the name starts at a root directory and the symbol ".." is used to indicate a parent directory. The root for any process is at the UNIX system in which the user "logged in" unless the process changes its root with a "change root" command. From figure 2 it follows that a user process on UNIX system U4 can access file "f1" as "/U5/f1". A user on U2 can access this file as "/.../U4/U5/f1". This implies that a user can access files in the network using standard UNIX system calls with the file-name being interpreted as a route through a naming tree, each element specifying the next branch to be taken.

The UNIX United naming scheme is implemented by means of communication links and the inclusion of a software subsystem, called the Newcastle Connection, in each of the individual UNIX systems. This software subsystem is located either within the UNIX kernel, or between it and the rest of the operating system and user programs. It intercepts system calls and determines which of the calls are local and which are for remote UNIX systems. It also incorporates UNIX servers which accept calls that have been re-directed to it from other systems. Communication between the Connection layers in the individual UNIX systems is performed by a remote procedure call mechanism [16].

A partitioning layer which would handle the management of
replicated files (under normal or partitioned processing) can be conveniently placed between the user and the Newcastle Connection software as portrayed in Figure 3.

```
User
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning Layer</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Newcastle Connection</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Kernel</td>
</tr>
</tbody>
</table>
```

Figure 3: Position of the Partitioning Layer

The aim is to provide file replication facilities while maintaining the standard UNIX system interface. This is achieved by making replication transparent to the user so that a replicated file can be accessed and updated with the same system calls as an ordinary UNIX file. However, it is suggested that the actual creation of a replicated file should not be transparent to the user since we believe that a user should be aware of the fact that he is creating a replicated file. (An alternative would be to provide particular directories within which all the files are automatically replicated.)

3.2. FILE CREATION

To create replicated files a user would invoke a new "mkrep" primitive which is provided by the partitioning layer. Such files are created as special objects. The "mkrep" primitive would mark these files to ensure that they are detectable as replicated files. It creates a file copy on its node and replicas on other nodes in the network. Since the "mkrep" primitive creates a replicated file as a special object, the partitioning layer sees such a file as one requiring special processing. Further investigation into the "specialness" of such a file can reveal that the file is a replicated file and not a "special file" in UNIX terms. (Special files in UNIX are usually associated with devices.) When it has been determined that a file is replicated, the file access strategy which is described in the next section would then be invoked.

3.3. FILE ACCESS

File access is (apparently) the same for replicated files as for ordinary UNIX files. A user calls the operations of the partitioning layer in order to open, read and write files, using standard UNIX system calls. Replication is therefore transparent to the user. The
partitioning layer intercepts all system calls so as to detect access requests to replicated files. This layer is also responsible for obtaining majority consensus during the update and merge operations. It also ensures that all accesses to remote nodes are passed to the Newcastle Connection software which handles accesses in a UNIX United system.

In accessing a file which is replicated on N machines, this layer would map the user's single access request into N requests (one to each copy) in the case of a write request and one request in the case of a read request. For “write” requests the partitioning layer then issues requests to all sites in the partition to write to their local file copy or to their update set, depending on whether or not a site belongs to the majority or minority partition respectively.

A read request from the majority partition is first directed to the local file copy, if a copy is available at the user's node, otherwise it is directed to any accessible copy belonging to the majority partition. A read request from a minority partition for replicated data first searches the update set associated with a file. If there is no entry in the update set for the requested values, the actual file copy is used. The result of such minority read requests are made available in a special minority buffer whose contents are not allowed to be written on an actual file copy. This is to prevent a task which is using both replicated and non-replicated data in a minority partition from writing uncommitted data (contents of an update set) to a non-replicated file. In environments which support a fully replicated system (i.e. in which all data items are replicated) there would be no need for a special minority buffer.

The partitioning layer intercepts all system calls and can therefore control the contents of the information which a user sees. This layer ensures that a user sees a replicated file as an ordinary UNIX file. Replication transparency could thus be achieved without modifying the UNIX kernel, although as with the connection layer, it would be possible to install the partitioning layer within rather than above the kernel. The partitioning layer at each node could seek majority consensus for each separate access or at periodic intervals. This would depend on the consistency requirements of the application. (If an application seeks majority consensus periodically, it ought to assume that it is part of a minority partition until such consensus is sought and obtained.)

4. CONCLUSION

The problem of providing continued processing in a distributed system in the presence of network partitioning, while maintaining the mutual consistency between replicated data, has been considered. An algorithm for detecting and resolving "actual" inconsistencies between
replicated objects in a partitioned system which does not require a knowledge of the semantics of operations on data structures has been presented. The resolution of "actual" conflicts helps minimise the extensive UNDO (or cascading rollbacks) which could arise when potential rather than just those conflicts that actually occur are resolved. An implementation of the proposed algorithm in a UNIX-based distributed system has been discussed.

The partitioning protocol which was presented here is resilient to multiple network failures. By this we mean that the protocol is well suited to environments in which network failures could result in one majority partition and several minority partitions. It is also useful in environments where the network could fail such that there is no majority partition, and can deal with the subsequent re-formation of a majority partition from the merger of minority partitions.

The proposed algorithm can be used in any environment which desires continued processing during network partitioning. Even in the worst case when a majority partition cannot immediately be formed, most environments would be able to carry out useful processing for a reasonable time interval until the network is reconnected. The algorithm which has been discussed in this paper is presently being considered for implementation in a UNIX United distributed system.
References


