Detection of Mutual Inconsistency in Distributed Systems

D. STOTT PARKER, JR., GERALD J. POPEK, GERARD RUDISIN, ALLEN STOUGHTON, BRUCE J. WALKER, EVELYN WALTON, JOHANNA M. CHOW, DAVID EDWARDS, STEPHEN KISER, AND CHARLES KLINE

Abstract—**Many distributed systems are now being developed to provide users with convenient access to data via some kind of communications network. In many cases it is desirable to keep the system functioning even when it is partitioned by network failures. A serious problem in this context is how one can support redundant copies of resources such as files (for the sake of reliability) while simultaneously monitoring their mutual consistency (the equality of multiple copies). This is difficult since network failures can lead to inconsistency, and disrupt attempts at maintaining consistency. In fact, even the *detection* of inconsistent copies is a nontrivial problem. Naive methods either 1) compare the multiple copies entirely or 2) perform simple tests which will diagnose some consistent copies as inconsistent. Here a new approach, involving *version vectors* and *origin points*, is presented and shown to detect single file, multiple copy mutual inconsistency effectively. The approach has been used in the design of *LOCUS*, a local network operating system at UCLA.**

Index Terms—**Availability, distributed systems, mutual consistency, network failures, network partitioning, replicated data.**

I. Introduction

A NUMBER of operating systems have been developed recently in which user files are distributed almost without restriction around a network. These systems range from network operating systems (NOS's) such as RSEXEC, NSW, ELAN [17], and DCS [4], to distributed database management systems (DDBMS's) like SDD-1 [5], [13] and INGRES [15]. These systems emphasize the uniform interfacing of multiple file systems. Files are to be accessible throughout the network, without regard to the accessor or file location.

Unfortunately, a file can be made inaccessible by network failures or crashes of the site where the file is located, so users may obtain randomly fluctuating views of the state of the network. To alleviate this problem, many of the systems propose to keep duplicate copies of files as a reliability mechanism. This solution engenders another problem. As soon as multiple copies of a file exist, the system must ensure the *mutual consistency* of these copies: when one copy of the file is modified, all must be modified correspondingly before an independent access can take place.

Much has been written about the problem of maintaining consistency in distributed systems, ranging from *internal* consistency methods (ways to keep a single copy of a resource looking consistent to multiple processes attempting to access it concurrently) to various ingenious updating algorithms which ensure mutual consistency [1], [2], [6], [8], [16], etc. We concern ourselves here with mutual consistency in the face of *network partitioning*, i.e., the situation where various sites in the network cannot communicate with each other for some length of time due to network failures or site crashes. This is a very real problem in most networks. For example, even in the Ethernet [10], gateways can be inoperative for significant lengths of time, while the Ether segments they normally connect operate correctly.

Network partitioning can completely destroy mutual consistency in the worst case, and this fact has led to a certain amount of restrictiveness, vagueness, and even nervousness in past discussions, of how it may be handled. In some environments it is desirable or necessary to permit users to continue modifying resources such as files when the network is partitioned. A network operating system would be a good example. In such environments mutual inconsistency becomes a fact of life which must be dealt with. This paper shows that in this case mutual inconsistency can be efficiently detected through the use of what we call *version vectors* and *origin points*. Once inconsistency is detected, some reconciliation steps are needed. In those cases where the semantics of the operations involved are straightforward, automatic reconciliation may be possible.

It is worth reflecting for a moment on the worth of keeping redundant copies. Although redundancy increases reliability and availability, and in most cases improves access time, it leads to mutual consistency problems when network partitions occur. When considering whether to store a file redundantly one must weigh the advantage of greater availability, the probability of a mutual inconsistency, and the ramifications of such an inconsistency. In many NOS environments, file update rates are moderate and "conflicts" would occur only rarely. However, in transaction-oriented DDBMS's update rates may be high, semantics of operations complex, and consistency extremely important.

Manuscript received November 11, 1980; revised November 13, 1981. This work was supported in part by ARPA Research Contract DSS MDA-903-77-C-0211 and in part by ONR Grant N00014-79-C-0866.

D. S. Parker, Jr., G. J. Popek, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, and C. Kline are with the Department of Computer Science, University of California, Los Angeles, CA 90024.

G. Rudisin was with the Department of Computer Science, University of California, Los Angeles, CA 90024. He is now with the Systems Technology Center, Western Digital Company, Pittsburgh, PA 15213.

S. Kiser was with the Department of Computer Science, University of California, Los Angeles, CA 90024. He is now with Xerox Corporation, El Segundo, CA 90245.

0098-5589/83/0500-0240$01.00

The results of this paper may be nevertheless useful in any system where mutual inconsistency, presumably due to network partitioning, is tolerated. Since our application (*LOCUS* [12], [14], [18]) is concerned with files, we will restrict our discussion henceforth to mutual consistency of *files* rather than of general resources. It is clear, however, that all results here may be applied to more general contexts.

The paper is organized as follows. Section II briefly surveys previous research on the partitioning problem. Section III then lays the formal groundwork on inconsistency detection. An accurate and easily implemented technique for detecting mutual inconsistency is developed. Section IV points out briefly what must be done in the reconciliation of inconsistent copies. Although the reconciliation of these conflicts must necessarily be left to the user in some cases, it is also demonstrated that for certain kinds of files (mailboxes, directories) the reconciliation may be performed automatically by the system. Finally, conclusions are offered in Section V.

II. Previous Work on Partitioning

Network partitioning is the situation occurring when a network is broken into logically separate components because of site or link failures. There are many partitioning-related issues which *must* be addressed in the design of distributed file systems. These issues include the relative importance of availability over mutual consistency of files, what occurs when one finds a file has become inaccessible or out of date, and so forth.

To our knowledge, however, partitioning has not been investigated very thoroughly. It has been mentioned in several proposed methods for updating files in distributed systems. The most typical response has been to enforce consistency by permitting files to be accessed only in one partition. Unfortunately, effective implementation of this policy can often result in the files being accessible in *zero* partitions. We outline several existing proposals below.

Voting: In voting-based systems such as proposed by Thomas [16] and Menasce *et al.* [9], mutual consistency is guaranteed at the expense of availability. Users desiring to modify a file must lock it by obtaining majority assent in a vote. Since there can be at most one partition containing a majority of the sites, any file will be accessible in at most one partition. Unfortunately, it is possible that there will be no partition which contains a majority of the sites, so in this case no updates could occur anywhere.

Tokens: Here it is assumed that each file has a token associated with it, which permits the bearer to modify the file. Obtaining the token is another issue, reducible more or less to locking. In this model only sites in the partition containing the token are permitted to modify the file, so using tokens is less restrictive than using voting. However, the problem of recreating lost tokens is nontrivial. Moreover, when a partition occurs, the token may happen to be resident in a rarely used part of the network, effectively making the resource unavailable.

Primary Sites: Originally discussed by Alsberg and Day [1], this approach suggests that a single site be appointed responsible for a file's activities. Upon partitioning (possibly involving a primary site crash) either 1) a backup site is elected as the new primary site and consistency becomes a possible problem (the proposed approach), or else 2) the file becomes inaccessible in all but the primary site partition.

Reliable Networks and Optimism: Communications in the SDD-1 system are based on the use of a "reliable network" [5], which guarantees the eventual delivery of all messages even if partitioning occurs. This delivery depends on "spoolers" which save messages to be transmitted following a break in communications. No guarantee of postpartition consistency exists; as with the primary site model, assuming consistent data afterwards is "optimistic" [6] in the sense that it may work out, but quite possibly the work done in different partitions will have to be detected in some way as inconsistent, and then undone or coalesced somehow by users.

Disk Toting: In this approach, employed at Xerox Parc and other installations where very intelligent terminals are linked via a network, files are not stored redundantly but are kept on removable storage media which can be carried around during prolonged partitions. Thus, availability and consistency are simultaneously achieved, but they are not achieved automatically. This approach is clearly only useful for local networks with compatible portable storage media at each site, where the delay and inconvenience implied is acceptable.

Note that none of these approaches openly states either 1) how conflicting versions of files are detected or 2) what is to be done when these conflicting files are detected upon merge of several partitions. Either the possibility of conflict is precluded by restricting file availability, or else any seemingly conflicting files must be "rolled back" to the most recent point at which there was no conflict. We show in the next sections how, *without* restricted availability, we can ensure correct propagation of updates in all cases except when unavoidably conflicting file versions are found.

III. Detection of Mutual Inconsistency

One of the reasons the partition problem is so difficult is that each partition can break into subpartitions and/or merge with other partitions many times before the entire network finally becomes connected. Indeed, it is possible that the network will *never* be completely reconnected! However, all messages sent might be delivered eventually through dynamically changing partitions. In this unpleasant eventuality, how can one hope to guarantee mutual consistency of files without restricting file availability as in Section II? We now show how inconsistencies or "conflicts" in the file system can be accurately *detected* easily; this solves the major part of the problem. The next section will discuss how these inconsistencies may then be *reconciled*.

We must formalize what we mean by a file "conflict" which arises after a partition, and pinpoint the kinds of inconsistency which partitioning can cause. This is important since, as mentioned above, many basic systems principles are invalidated in systems subject to partitioning. First, the semantics of renaming, deletion, and even creation of redundantly stored files or resources in systems which are partitioned are totally unclear. Second, and worse, user-visible names of entities in

the system may no longer be assumed to either uniquely specify, or even correctly specify, the entities themselves. After a partition, it may be discovered either that two files with the same name have been independently created, or that two independent updates to the same file have been made. In general, *names in one partition bear no relation to entities in another.* This is a principle reason for the difficulty in defining the semantics of renaming and deletion of files. We need some form of identification of system entities which is immune to partitioning. We achieve this below by using "origin points" and "version vectors."

A. File Conflict Types and Origin Points

A (network) *partition* is a set of sites which share a common, synchronized, view of some set of files.

An *origin point* OP(f) of a file f is a system wide unique identifier which is generated when f is created. It is an immutable attribute of f, although f's name is not (indeed f may have multiple system wide names). Thus, no number of modifications or renamings of f will change OP(f).

An origin point for a file might be something like a (creation time, creation site) pair. Now, just as names cannot uniquely specify files, origin points cannot either, but they do give us important information. Origin points tell us when two files are *based* on a common file, but do not tell us whether the two files are identical, since both could have been independently modified.

There are two types of conflicts that we wish to consider: *name conflicts* and *version conflicts.* A name conflict occurs when two files with different origin points have the same system wide name. In contrast, a version conflict occurs when two versions of the same file (same origin point) have been "incompatibly" modified. After some preliminaries, a version conflict occurrence is defined more precisely below.

A *modification id* for a version of a file f is a system wide unique identifier of a modification of f in some partition and at some time relative to that partition. A *modification history* for a version of a file f is the set of modification ids corresponding to the modifications of that version of f which have occurred. Two modification histories are *compatible* if they are identical or if one is an *initial history* of the other, and *incompatible* otherwise.

We define a version conflict to occur when two versions of the same file f (same origin point) have *incompatible modification histories.*

Note that when two versions of a file are not equal, their modification histories are always different. However, it *is* possible for two versions of a file to be equal yet have incompatible histories. For example, consider a file which contains a bank account balance. If the balance is $20 million initially, and both partitions decrease it to $0, then at partition merge time although both versions are $0 a conflict *will* be indicated. Further, if the semantics of "decrease" mean "withdraw," a conflict intuitively *should* occur.

We claim that this definition of version conflict occurrence is a reasonable one given that nothing is known about the file content's semantics.

Clearly, name conflicts are easy to detect. Version conflicts, however, are more difficult to detect efficiently. This latter problem is addressed in the following sections: modifying, deleting, or renaming the various copies.

B. The Problem of Version Conflict Detection

One might think that a simple timestamp scheme could be used to detect possible version conflicts among files: every time a file is modified in a partition, one marks it with an update time and the previous update time. Upon partition merge, one checks whether the timestamps on the copies of a file are either all identical (no update on the file occurred), or one copy of the files differs from the others by a single update. Thus, no conflict is signaled when at most one update is made, but in any more complex situation a version conflict condition is raised. This approach is deficient in general, since some nonconflict situations will be handled as conflicts.

Let us describe the version conflict problem in the following way. Think of a partition *for a file* as a subset of sites in the network in which all copies of the file may be maintained with mutual consistency. Note that this definition is not strictly tied to the physical details of network failure. Instead, here partitions are defined relative to files and to the higher concept of consistency. Although two sites with different versions of a file f may be communicating for some time, we do not consider the sites to be in a common partition relative to f unless this difference in the two versions is resolved.

Definition: A *partition graph* $G(f)$ for any file f is a directed acyclic graph (dag) which is labeled as follows. The source node (and the sink node if it exists) is labeled with the names of all sites in the network having copies of file f, and all other nodes are labeled with a subset of this set of names. Each node can only be labeled with site names appearing on its ancestor nodes in the graph; conversely every site name on a node must appear on exactly one of its descendants. In addition, a node is marked with a "+" if f is modified one or more times within the corresponding partition, and/or a version conflict had to be *reconciled.*

We define this latter situation recursively as follows. Let P be a node in $G(f)$. A version conflict had to be reconciled at P if there are backward paths from P to distinct nodes $P1$ and $P2$ in $G(f)$, such that

1) an update to f and/or a version conflict reconciliation for f occurred at both $P1$ and $P2$, and

2) there is no ancestor node of P having two backward paths to both $P1$ and $P2$.

Each node in $G(f)$ thus corresponds to a partition for f, a period of time during which the labeled sites maintain "synchronized" information about f. All sites appearing in the node label resolve any differences that might exist among their copies of f. All connections in $G(f)$ between nodes indicate transitions of the network under partitions or merges.

The definition of conflict and reconciliation models the notions of Section III-A for the following reasons. First, any version conflict that is reconciled must have been generated by two prior partitions $P1$ and $P2$, giving incompatible modification sets. Second, and conversely, if a file modification of some kind (update or reconciliation of updates) occurs inde-