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 Reseach 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 © 1983 IEEE

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-

pendently in two partitions $P1$ and $P2$, a version conflict must arise later whenever sites from these partitions inspect f. Condition 2) guarantees that partition P is the first point at which mutual consistency is again established.

An example of a partition graph is shown in Fig. 1. Here there are four sites, A, B, C, and D, which support f. Multiple partitions of these initially connected sites occur, so that at first sites A and B can communicate, but are isolated from sites C and D. Later A and B become isolated, as do C and D, but B and C resume communication. Ultimately, all four sites are reconnected in the bottom node of the graph. The file f is modified first in the $\{A, B\}$ partition, and subsequently in both the $\{A\}$ and $\{B, C\}$ partitions. Note that this sequence of modifications *should not* result in a version conflict in the BC or BCD partitions since site B at all times has the latest version of f; intelligent implementation of conflict detection should realize this fact and avoid notifying sites C or D that their f versions conflict with the current one. However, in the final $ABCD$ partition a conflict *is* (and should be) reconciled, since in this case both versions of f have incompatible modification sets.

Now, as mentioned above it is simple to provide some mechanism which detects all *possible* version conflicts; a simple timestamp algorithm will be adequate. What is more difficult is to find a mechanism which detects version conflicts *only* when they are real. In Fig. 1, for example, even though the first update may have been initiated by site A, this information is transitively passed by site B without conflict to sites C and D.

C. Version Conflicts and Version Vectors

Many possible approaches exist for attacking the problem of accurately detecting version conflicts. More elaborate timestamp schemes are a possibility, and there are a number of methods based on "update log files" (sometimes referred to as "journaling"). Unfortunately, these approaches suffer from either or both 1) a need to maintain some kind of global network time (in itself nontrivial [7]) and 2) a need to store the entire partition graph—or its equivalent—someplace where it may be accessed later on. Since the partition graph may get arbitrarily large, the latter requirement is undesirable. We now present instead a straightforward solution to this problem based on a version numbering scheme encoding just the necessary characteristics of the history graph.

One maintains a vector with each copy of each file. Within every partition (unit of mutual consistency), these vectors keep an update history for the file. As partitions merge, these vectors for the possibly inconsistent files are compared. It turns out that version conflicts are signalled when, and only when, the vectors are "incompatible." We formalize this as follows.

Definition: A *version vector* for a file f is a sequence of n pairs, where n is the number of sites at which f is stored. The ith pair $(S_i\colon v_i)$ gives the index of the latest version of f made at site S_i. In other words, the ith vector entry counts the number v_i of updates to f made at site S_i. We will use leters $A, B, C, \cdots$ to designate site names, and vectors will be written as $<A{:}9, B{:}7, C{:}22, D{:}3>$.

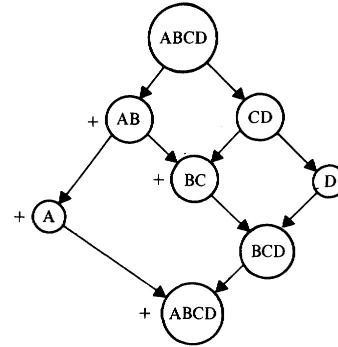

Fig. 1. Partition graph $G(f)$ for file stored redundantly at sites A, B, C, D.

Definition: A set of version vectors are *compatible* when one vector is at least as large as any other vector in every site component for which they each have entries. A set of vectors *conflict* when they are not compatible.

For example, the version vector $<A{:}1, B{:}2, C{:}4, D{:}3>$ dominates $<A{:}0, B{:}2, C{:}2, D{:}3>$ so the two are compatible; and $<A{:}1, B{:}2, C{:}4, D{:}3>$ and $<A{:}1, B{:}2, C{:}3, D{:}4>$ conflict, but $<A{:}1, B{:}2, C{:}4, D{:}3>$, $<A{:}1, B{:}2, C{:}3, D{:}4>$, and $<A{:}1, B{:}2, C{:}4, D{:}4>$ do not conflict, since the third vector dominates the other two. In Fig. 2 version vectors are given for f in every partition of Fig. 1. The vector $<A{:}2, B{:}0, C{:}1, D{:}0>$ associated with the node labeled BCD, indicates that f was modified twice at site A, once at site C, and nowhere else. Note in particular that during the $\{A, B\}$ partition, the file is modified twice at site A. The final merge results in a conflict.

We adopt the following *usage of version vectors.*

1) Each time an update to f originates at site S_i, we increment the S_ith component of f's version vector by one. The vector is committed with the updated file.

2) File deletion and renaming are treated as file updates. Deletion results in a version of the file of length zero, for example; when all versions of a file are of length zero, information on the file may be removed from the system.

3) When version conflicts are reconciled within a partition, the S_ith entry of the version vector for the reconciled file is set to be the maximum of the S_ith entries of all of its predecessors, and in addition the site initiating the reconciliation increments its entry. This ensures future compatibility with any old versions of the file still remaining on the network.

4) When copies of a file are subsequently stored at new sites, the version vector is augmented to include the new site information. The definition of compatibility above still applies in this case.

Point 4) states that the vectors are not required to be of fixed length, but may grow (or shrink, actually) as long as the relevant site information is maintained. If a copy of f is added at a site E during some partition, the vector in the partition where the copy was obtained is simply augmented to reflect the existence of the E copy. Thereafter, sites merging with this partition will be required to augment their vectors accordingly. Also, note that the version counts should be of variable length, so running out of space will not be a problem.

Version vectors serve basically to encode the partial order

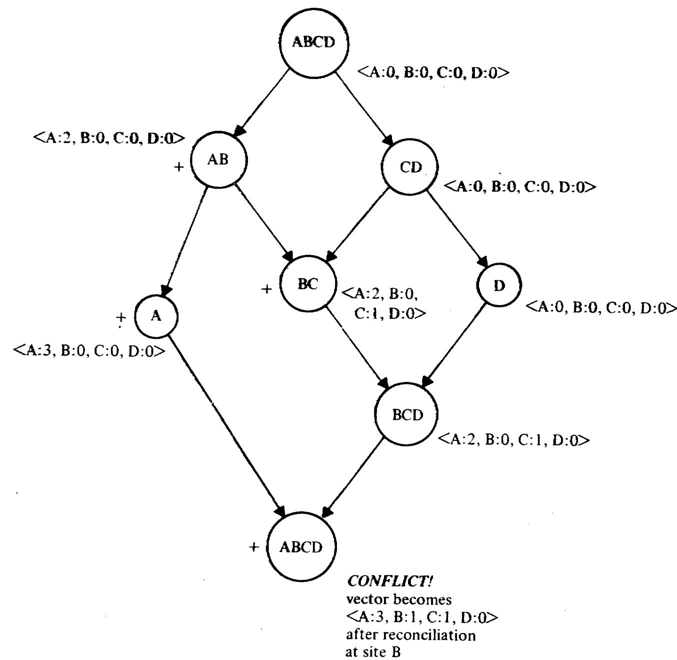

Fig. 2. Partition graph $G(f)$ for f with version vectors effective at the end of each partition.

defined by the partition graph: if one node in the graph "precedes" another, i.e., there is a path from the graph source through the former to the latter, then the version vectors of the two nodes will not conflict. This observation leads us to the following result, which shows us that version vectors are all we basically need to detect version conflicts.

Theorem: A version conflict must be reconciled at a node in $G(f)$ if and only if f's version vectors conflict at that point.

Proof: It is clear that if there is a conflict reconciliation at some node P in $G(f)$, then the version vectors will conflict at P. (Version vectors detect real conflicts just as well as the simple timestamp algorithm: what must be shown is that they detect only real conflicts.) Conversely, suppose that f's version vectors conflict at some node P. Then two of the vectors must conflict and not be dominated by any third vector. These two vectors were generated in two earlier partitions $P1$ and $P2$—both having paths to P—where f was modified independently. All that must be shown is that there is no ancestor P' of P which also has backward paths to $P1$ and $P2$. (Note P' could be either $P1$ or $P2$.) Suppose P' exists. We know that in this case the $P1$ and $P2$ version conflict will be reconciled at P', giving a version vector whose components are the maxima of the components from the vectors in $P1$ and $P2$. But this vector will dominate both the $P1$ and $P2$ vectors at P, removing their conflict. This contradicts our original assumption. Hence, P' cannot exist, so the partition graph conditions are satisfied and there is a conflict reconciliation at P. □

D. Conclusions on File Conflict Detection

The theorem above shows us that version vectors may be used to detect version conflicts and request user reconciliation of the conflicts. Version vectors will detect only "real" conflicts, i.e., situations in which versions of a file were modified independently in separate partitions. Thus, our work differs from previous research in that, where many people have developed mechanisms (e.g., timestamps) which detect *sufficient* conditions for a conflict to exist, we have striven to provide a mechanism detecting *necessary and sufficient* conditions for conflict.

It should be noted that if an identical update is made in two separate partitions, version vectors will indicate a file conflict even though there may be none. In some applications, then, it may be desirable to actually check a file for differences when several copies are found to have conflicting vectors. Indeed, this cross-checking of copies may have to be done eventually if the user is to resolve the file conflict.

It is also important to recognize that what has been presented here is applicable only when *single* files are being processed. Consider the following example (of Mark Brown) where two "transactions" $T1$ and $T2$ execute in different network partitions. Let

$$\text{readset}(T1) = \text{readset}(T2) = \{f, g\}$$

$$\text{writeset}(T1) = \{f\}$$

$$\text{writeset}(T2) = \{g\}$$

and assume that both $T1$ and $T2$ complete prior to reconnection of the network. Then a conflict (serialization error) should occur after reconnection, but it is easily seen that in this case version vectors will not detect anything amiss. Many appropriate extensions immediately suggest themselves; one is presented in [11]. Note, interestingly, that many of the "solutions" for providing mutual consistency mentioned in Section II also *do not* solve this problem. In particular, such conflicts can still occur with the Token and Primary Site approaches, unless *all* updates are constrained to occur within a single partition.

We have shown in this section that file conflicts, whether

they are name conflicts or version conflicts, can be accurately detected by maintaining just two pieces of information with each file f:

1) an origin point
2) a version vector.

In the following section we take up the question of how to resolve file conflicts, now that the problem of detection has been clarified.

IV. Resolution of Mutual Inconsistency

A conflict detection mechanism, while valuable, has increased effect if there is also a method for reconciling conflicts automatically. From several conflicting versions of a file, this method should produce a subsequent version that dominates these versions, while preserving the operations which were done to them. Although this is certainly not possible in general, there are many cases which admit automated reconciliation.

Clearly, conflict reconciliation must take into account the semantics of the operations which were done to the data objects in conflict. This has been noted by many researchers (e.g., [1, p. 568], [5, p. 65], [13, p. 57]). In those cases where the nature of the semantics is sufficiently constrained, straightforward reconciliation algorithms can be given. For example, consider two important types of files in *LOCUS*, directories and user mailboxes. In both of these cases, there are just two available operations:

- insert an item (e.g., create a file, or receive a message)
- remove an item (e.g., delete a file, or process a message).

Such files have the characteristic that version conflicts can be reconciled simply by taking the union of the entries in the component files, then removing any entries which had been deleted. Reconciliation for both of these file types is handled automatically in *LOCUS*.[1]

Automatic reconciliation applies in much more far-reaching contexts that on the systems level. An instructive example can be found in electronic funds transfer. Consider a checking account, as proposed earlier in Section III-A. Credits and debits can be made to different copies of the account. Resolution is straightforward so long as the ith copy is represented as

$$x + \delta_i(x)$$

where x was the original account balance before partition and $\delta_i(x)$ is the change in that partition. Then the new balance is

$$x + \sum_i \delta_i(x).$$

This approach may be improper if we require the balance to remain positive. However, there are many ways to deal with this problem. When x is the balance of a large corporation, presumably the problem will not occur. More generally, one may operate the system in a more constrained fashion when it is partitioned, either by limiting withdrawals in those cases where the customer is not trusted, or by imposing quota-like limits on withdrawals within each partition.

[1]Most directory systems, and some mail systems, permit additional operations. Therefore the automatic recovery software in *LOCUS* for these file types is more involved than indicated here.

A number of existing applications permit automated reconciliation while still allowing robust operation during partition. Two cases which have been studied carefully are banking and airline reservation systems [3]. Extensive, although not full, operation of these systems is quite feasible while partitioned.

A desirable characteristic of system operation semantics, or of the reduced *partition semantics*, is that reconciliation of a data item not necessitate the alteration of many other data items. In order to keep automatic reconciliation cost low in a database, for example, one might insist that most transactions executed during partitions not require undoing and redoing when their read sets are subsequently altered during a reconciliation. This is the case today for portions of banking systems such as automated tellers.

In general, it is often possible to break the semantics of operations into classes, and for each class give rules by which the reconciliation algorithms can be constructed. Simple semantic classes permit reconciliation in a straightforward way without keeping much history. As the semantics become more complex, more history and work is required.

Of course, even when the semantics of operations are clear, automatic reconciliation can be very difficult, expensive, and in some cases impossible. Reconciliation cannot be performed in those cases where, as part of the system's activity, an external action has been taken that cannot be undone nor can a compensating action be taken. These cases are the same ones for which general purpose data management recovery is impossible too.

One suspects that in many systems, automatic reconciliation will be feasible for the large majority of data items. However, there will remain cases that require human intervention.

Independent of the degree of automatic reconciliation, a consistent system policy must be defined for each of the following questions.

- When and how are data conflicts detected?
- Is permission to access a data item altered by the fact that the item is in conflict? No alteration of permission raises the question of which version to make available, and leads to the possibility of propagating inappropriate values.
- How are users informed of conflicts?
- What support does the system provide the user for reconciling conflicts?

These questions raise a number of architectural issues, some of which are addressed in [12], [14], [18].

V. Conclusions

We have developed an effective method for detecting mutual inconsistency in distributed systems. Here inconsistency has been assumed to be caused by multiple users modifying different copies of a common file without mutually excluding one another. Such a situation would arise, for example, when network failures isolate these users in different partitions of the network. The technique also applies when partitions are artificially introduced; for example, when stations in a connected network delay their transmissions to take advantage of

batching or lower communications rates at various times of day. The method used is simple, relying only on two newly introduced constructs, version vectors and origin points, for its operation. Although the method was discussed specifically in the context of file systems, it applies equally well to any class of resources for which occasional mutual inconsistency is tolerable for the sake of availability, or where the semantics of the allowed operations permit automated recovery.

The general problem of how to resolve mutual inconsistency of copies of a resource, once it is detected, is a complex question. We have only given it a summary treatment here, since it raises many design issues and can be answered thoroughly only when the semantics regarding the use of the resource are explicitly known. We have noted, however, that for some resources automatic reconciliation is straightforward to implement.

REFERENCES

[1] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proc. 2nd Int. Conf. Software Eng.*, Oct. 1976.

[2] C. A. Ellis, "A robust algorithm for updating duplicate databases," in *Proc. 2nd Berkeley Workshop Distributed Data Management and Comput. Networks*, 1977, pp. 1146-158.

[3] S. Faissol, "Operation of distributed database systems under network partitions," Ph.D. dissertation, Dep. Comput. Sci., UCLA, 1981.

[4] D. J. Farber and F. R. Heinrich, "The structure of a distributed computer system–The distributed file system," in *Proc. ICCC*, 1972.

[5] M. Hammer, and D. Shipman, "An overview of reliability mechanisms for a distributed data base system," in *Proc. Spring Compcon*, San Francisco, CA, Feb. 28–Mar. 3, 1978.

[6] H. T. Kung and J. R. Robinson, "On optimistic methods for concurrency control," *ACM TODS*, vol. 6, pp. 213–226, June 1981.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558–565, July 1978.

[8] B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," Tech. Rep., Xerox PARC, 1976.

[9] D. A. Menasce, G. J. Popek, and R. R. Muntz, "A locking protocol for resource coordination in distributed systems."

[10] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 395–404, July 1976.

[11] D. S. Parker and R. Ramos, "A distributed file system architecture supporting high availability," in *Proc. 6th Berkeley Conf. Distributed Data Management & Comput. Networks*, Asilomar, CA, Feb. 1982.

[12] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "*LOCUS:* A network-transparent, high reliability distributed system," in *Proc. 8th Symp. Oper. Syst. Principles*, Asilomar, CA, Dec. 1981.

[13] J. B. Rothnie and N. Goodman, "A survey of research and development in distributed database management," in *Proc. 3rd VLDB*, Tokyo, Oct. 1977.

[14] G. J. Rudisin, "Architectural issues in a reliable distributed file system," M. S. thesis, Dep. Comput. Sci., UCLA, Rep. UCLA-ENG-8014 SDPS-80-001, Apr. 1980.

[15] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 188–194, May 1979.

[16] R. F. Thomas, "A solution to the concurrency control problem for multiple copy data bases," in *Proc. Spring COMPCON*, Feb. 28–Mar. 3, 1978.

[17] R. F. Thomas, R. H. Schantz, and H. C. Forsdick, "Network operating systems," Rome Air Develop. Cen., Tech. Rep. RADC-TR-78-117, May 1978.

[18] B. J. Walker, "Issues of network transparency and file replication in distributed systems: *LOCUS*," UCLA Tech. Rep., June 1981.

D. Stott Parker, Jr. received the A.B. degree in mathematics from Princeton University, Princeton, NJ, in 1974 and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana-Champaign, in 1976 and 1978, respectively.

Currently, he is an Associate Professor with the Department of Computer Science, University of California, Los Angeles. His main interests include distributed processing, algorithms, architecture, and logic programming.

Dr. Parker is a member of Sigma Xi and the Association for Computing Machinery.

Gerald J. Popek received the Ph.D. degree from Harvard University, Cambridge, MA, in 1972.

He then joined the faculty at the University of California, Los Angeles, where he is now Professor of Computer Science and in charge of the department's computing facilities. He is Principal Investigator of an ARPA contract, under which basic work in computer security was done, and which is now focused on distributed systems. He is the author of approximately 50 professional publications. He is also Vice President of Palyn Associates, Inc., and there has been responsible for major software efforts. Activities have focused on languages, operating systems, and distributed computing functions.

Gerard Rudisin received a B.S. degree from the Massachusetts Institute of Technology, Cambridge, in 1975 and the M.S. degree from the University of California, Los Angeles, in 1980, both in computer science.

He is currently a Ph.D. candidate in computer science at the University of California, Los Angeles, and also works for Western Digital Corporation's Systems Technology Center. His major research interests are local computer networks, operating systems, database technology, and Ada implementations and support environments.

Mr. Rudisin is a member of the Association for Computing Machinery and Sigma Xi.

Allen Stoughton received the B.S. degree in mathematics/computer science and the M.S. degree in computer science from the University of California, Los Angeles, in 1979 and 1981, respectively.

His current interests include operating system protection models and programming language semantics.

Bruce J. Walker received the B.Math degree from the University of Waterloo, Waterloo, Ont., Canada, in 1975 and the M.S. degree in computer science from the University of California, Los Angeles, in 1977.

He is currently finishing the Ph.D. degree at UCLA. The topic of his dissertation is the transparency and replication principles in distributed systems and in particular, LOCUS. Research interests also include computer security, about which he has published articles and a book.

Evelyn Walton was born in Little Rock, AR. She received the B.A. and M.S. degrees from the University of California, Los Angeles, in 1972 and 1975, respectively.

She has been working on research projects in the Department of Computer Science at UCLA since 1971. At present, she is working on the Ph.D. degree in computer science. Her research interests are in the areas of computer languages, systems, and networks.

Johanna M. Chow received the B.S. degree (summa cum laude) in mathematics/computer science from the University of California, Los Angeles, in 1980.

She is currently working on the Ph.D. degree in computer science at UCLA. Her thesis deals with the issues of transaction management and recoverable processes in distributed systems.

David Edwards was born on June 15, 1957 in Glendale, CA and grew up in La Crescenta. He received the B.S. degree in mathematics/computer science from the University of California, Los Angeles, in 1979.

As a graduate student at UCLA, he has worked on the LOCUS Distributed Operating System project under Gerald Popek. He has been a departmental scholar and the recipient of a fellowship from System Engineering Laboratories. His current interests are in operating system design, especially distributed and in the interaction between the designs of hardware architecture and system software.

Stephen Kiser received the B.S. degree in mathematics/computer science from the University of California, Los Angeles, in 1979.

He is currently working towards the M.S. degree at UCLA and is also employed by the Xerox Corporation, El Segundo, CA. His interests are in distributed computing, computer theory, and programming languages.

Charles Kline received the B.S., M.S., and Ph.D. degrees from the University of California, Los Angeles, in 1970, 1971, and 1980, respectively.

His Ph.D. research was in the areas of computer and network security, especially the use of security kernels and encryption. He is the author of numerous papers on the issues of computer security, security kernels, and the use of public key and conventional encryption in computer network security. He has been employed as a researcher in the Department of Computer Science at UCLA since 1966. During that period his research areas have included operating systems, languages, networks, distributed systems, and computer security problems.

Input–Output Tools: A Language Facility for Interactive and Real-Time Systems

JAN VAN DEN BOS, MARINUS J. PLASMEIJER, AND PIETER H. HARTEL

***Abstract*—A conceptual model is discussed which allows the hierarchic definition of high-level input driven objects, called input–output tools, from any set of basic input primitives. An input–output tool is defined as a named object. Its most important elements are the input rule, output rule, internal tool definitions, and a tool body consisting of executable statements. The input rule contains an expression with tool designators as operands and with operators allowing for sequencing, selection, interleaving, and repetition. Input rules are similar in appearance to production rules in grammars. The input expression specifies one or more input sequences, or input patterns, in terms of tool designators. An input parser tries, at run-time, to match (physical) input tokens against active input sequences. If a match between an input token and a tool designator is found, the corresponding tool body is executed, and the output is generated according to specifications in the tool body. The control structures in the input expression allow a variety of input patterns from any number of sources. Tool definitions may occur in-line or be stored in a library. All tools are ultimately encompassed in one tool representing the program.**

The input–output tool model offers a nonprocedural input specification language with a parser provided by the run-time system. It forces clean and structured programs and allows for easy definition of abstract input devices and simulation of physical devices on other devices. Implementations have been completed and are being evaluated.

***Index Terms*—Computer graphics, dialogue, input functions, input tools, interaction language, process control, programming language, real time, specification language.**

Manuscript received October 2, 1981; revised October 1, 1982. This work was supported in part by a grant from The Netherlands Organization for the Advancement of Pure Research (ZWO).

J. van den Bos and M. J. Plasmeijer are with the Department of Computer Science and the Computer Graphics Group, University of Nijmegen, Nijmegen, The Netherlands.

P. H. Hartel was with the Department of Computer Science and the Computer Graphics Group, University of Nijmegen, Nijmegen, The Netherlands. He is now with the Department of Computer Science, University of Amsterdam, Amsterdam, The Netherlands.

I. INTRODUCTION

INTERACTIVE computing, in which we include real-time systems and process control, forms a sizable, perhaps more than 50 percent, share of all computing. Most interactive systems have been programmed using regular (or slight derivatives of) batch programming languages. Unfortunately, these languages are badly lacking in provisions for handling input and output on an advanced level. Few, if any of these languages, have, for example, provisions to read one input source out of several specified. Facilities to define named abstract input devices in terms of (a collection of) existing physical devices

0098-5589/83/0500-0247$01.00 © 1983 IEEE