The Design and Implementation of Distributed, Disconnected Operation in PRCS

Josh MacDonald

Abstract. Traditional client-server version control applications have not addressed the need for *disconnected operation*. This mode of operation, applied to version control, allows users to use the version control utility to its fullest extent when disconnected from a shared repository. Version control applications demonstrate several features unique to *read-only* media that virtually eliminates changes to the operational model. The user is presented with the same interface, regardless of network connectivity, and changes in connectivity present themselves as the same operation one expects a version control utility to support, *merge* or *update*.

1 Previous Results

PRCS [9] is a version control application designed for simplicity and ease of use. Much like other systems, PRCS uses an optimistic concurrency model for organizing multiple, concurrent development efforts. The version model presented by PRCS is particularly simple: a project version is a snapshot of a set of files, organized into a directory subtree. Each project version has zero or more parent versions. This partial order on versions serves to define ancestry. Project versions are labeled with a branch (major) name and minor version number.

PRCS provides the operations to create, retrieve, compare, and reconcile changes in project versions. The operational model is, however, largely unchanged with the introduction of distributed capabilities. The overall goal is to provide a system design which gives users flexible control over the resources required to use the application.

2 Problems With The Current System

This work is motivated by several overwhelming requirements. As collaborative development efforts grow, their participants are increasingly less centralized. Additionally, groups desire to work over limited network connections. Wireless networks and dialup modem links have extremely limited bandwidth, making repeated, entire file or project transfers an unacceptable implementation technique. Unfortunately, the current implementation of PRCS forces all users to operate with access to a common file system, where the repository lies.

3 Require No Operating System Support

The research community has studied solutions to a similar problem, distributed and disconnected file systems. Distributed file systems are an inadequate solution to the problem at hand because file system semantics are inappropriate. One might claim that the application developer should focus on writing applications, and the operating system developer should focus on providing distributed, disconnected file system access, so that the application can operate in a distributed environment. Even assuming users have accepted optimistic concurrency, as found in Coda [8], a distributed file system does not seem to help.

A first approach might be to place a PRCS repository on a distributed file system. A file system to support such an arrangement would have to sacrifice availability for consistency, since the contents of the repository (as a result of the application semantics) do not lend themselves to optimistic concurrency or any sort of conflict resolution.

Another approach might be to place each developer's working files on a distributed file system. This file system does not need to sacrifice availability and the Coda approach would work well. Unfortunately, when the user is disconnected from the file system, the application is disconnected from the repository and the application can do little for the user.

Without consideration for disconnected operation, distributed file systems are of little help. Having turned over optimistic concurrency management to the version control utility, there is little need for a distributed file system. It is just as good to use a local file system and let the version control utility control synchronization with other, remote sites. Though proponents of distributed file systems claim that write-write conflicts are rare, they are accidents and must be avoided. Use of separate file systems makes concurrency management explicit and eliminates the both write-write and read-write conflicts.

Distributed file systems have focused on the question: "which files should I cache so that a disconnection will not affect the user?" By using a version control utility to explicitly manage projects and not using a distributed file system, this problem is eliminated. With the current trend in storage capacity and cost, many of the reasons for continuing to use distributed file systems are eliminated.

When file systems are read only, the problem of concurrencymanagement goes away. This is an important special case where the above arguments do not apply. Read only access is a very different problem and the available solutions are good. The property of an immutable media that makes its distribution acceptable is that immutable files can be cached with extremely simple caching policies. This property also makes the problem at hand, distributed version control, much easier: version control utilities record the history of a set of files, and history can't be changed. Since the data in a version control repository is never modified, replicated storage can be used to give each user control over how much disk space and network activity each operation will require to give an acceptable level of performance.

4 Distributed Branches

A branch is a division in project development, a logical split occuring when a particular ancestor has more than one descendent. Branches are typically used to manage and help reintegration of variations in a project's development.

The only major change to the operational model, from a user's perspective, is that branches are now first class repository objects. Branches are now the level of granularity at which a repository may be distributed. Each branch has a single home location, the machine owning that repository is the authority for all operations on that branch. Branches may be mapped into foreign repositories and may be exported with various permissions. Branch mapping is a part of repository administration. Branches are given symbolic names at each site, and these names need not agree between sites.

One motivation for this model of distribution comes from the following, common scenario. A project team develops a successful piece of software and makes their development repository, `MAIN-DEVEL`, publically available (read-only). A group dedicated to adding in some new feature, `SPIN-OFF`, takes up their task and soon realizes the complexity of synchronizing their work with the main development is very difficult, they need the version control utility's help at keeping their development synchronized with the main development effort. With existing tools, they are forced to either use the development repository (this requires the permission, acceptance, and resources of the main development group), or use their own repository and manually reconcile the changes made by the main development group. There is little or no support in the available tools for using multiple repositories and in the end the user is faced with the cumbersome process of synchronizing repositories by hand. By allowing the `SPIN-OFF` group to instantiate a repository and map some or all of `MAIN-DEVEL`'s branches, the `SPIN-OFF` group is able to use a repository which contains both of their work, without consuming the resources of the `MAIN-DEVEL` group.

The second motivation for this model is the desire to support disconnected operation on a client machine. For example, a laptop user may be physically separated from the network for some time and want to continue using the version control system. By creating a local branch work can proceed during network outages. When the network is reconnected, work may be merged under the control of the version control utility. The Coda file system supports a similar model, but reconnection is handled by the operating system. It seems better to use a system which was designed to manage optimistic concurrency, rather than one which claims conflicts are rare and accidental, for managing a reconnection.

Even with a fully connected network and access to a repository, users often do not make the most of their version control system because it is a shared resource. Since all the work they put in the repository is visible to other members of the team, they may hesitate to create as many intermediate or checkpoint versions as they would like since the work is not ready for sharing.

5 Requirements

Though the interface does not change much, the implementation has been completely redesigned with the following requirements in mind.

- The version control tool must remain useful for all levels of network connectivity, network topology, available disk space, and computational resources.
- The server must function as a master server (ownership of branches), a mirror (agent responsible for relieving load on server), a client cache, an effective transport participant, and various intermediate configurations. It should not be necessary to augment the operation of PRCS with such programs as `rsync` [16] or CVSup [11], a distribution protocol for CVS [1].
- Network communication must be efficient, optimized for the particular problem, and secure.

The above requirements suggest several implementation features, which will be discussed in the next section. When available, clients will use local disk to cache data. The use of local caches enables efficient network communication. Since the version control program is intimately aware of changes in the files it controls and already using deltas to encode these changes, it can also use deltas to communicate with and synchronize repositories.

Authentication and security must not be tied to operating system privileges, for this limits the effectiveness and participation of replicant, mirrored repositories. Currently, connections are authenticated using the SSL version 3.0 protocol [3]. Rather than use the proposed certificate authority infrastructure that comes along with SSL (and seems like quite a lot of baggage), certificate authority will be assumed by each individual server. Authorized users will register a message digest (such as SHA-1 [14]) of their certificate with the repository administrator. Only connections with certificate digests registered with the repository will be accepted.

6 Network Ancestry

Since branches are now capable of being available at one site and not another, care must be taken to insure that ancestry relations are properly preserved. If, for example, a local branch and network branch are merged, the new network version has a parent which is not available on the network computer. For this reason, the network computer must have at least a record of the ancestry of the local branch.

There is a more important issue, however, than the availability of the ancestry. It should not be possible for a version to be created on a foreign branch that makes that repository unusable to other users without access to the local branch. Simply stated, though a repository may contain versions without access to their ancestors, there should never exist a pair of versions in a repository such that their common ancestor is not also in the repository.

To prevent this situation from occurring, before a new version can be created a test is first performed to verify the availability of a common ancestor between the new version and all previous versions. Similarly, this test must be run before a branch may be mapped into or unmapped from a repository. This requirement produces a dependence between branches.

This is not a great restriction. The following two figures demonstrate a typical local-network scenario and one which is illegal according to the above rule. Figure 1 illustrates a typical scenario in which a user has created a local branch L, checked in several versions on that branch, and later merged with the network branch, N. Figure 2 illustrates an illegal configuration, in which a user has merged a local branch L against two network branches M and N. The common ancestor of $M.4$ and $N.4$ is $L.3$, and therefore cannot be admitted.

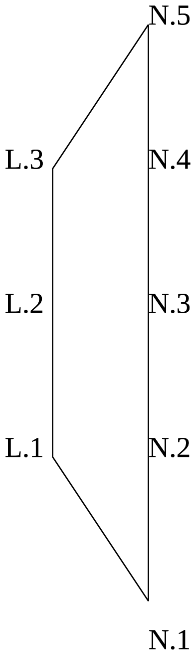

Figure 1: An acceptable local-network branch configuration

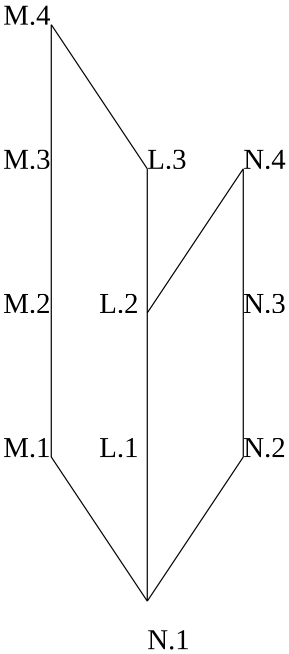

Figure 2: An unacceptable local-network branch configuration

7 Replacing RCS

There are three problems with RCS[15], and as a result, it will no longer be used as a back-end for the efficient, versioned storage of files. The first, a pragmatic issue, is that RCS is only available as executable programs, not as a library. This produces slow and inelegant code.

The second problem with RCS is that its version branching model does not cooperate with attempts to automatically allocate version numbers. This leads to unexpected performance implications, and it is difficult to optimize its use, either automatically or with user intervention. High level tools which have no notion similar to the RCS main-trunk do poorly at providing good performance from the underlying RCS storage system.

The third problem is RCS's delta mechanism. RCS uses the standard UNIX `diff` application to compute forward and reverse deltas, which it then stores. UNIX `diff` uses a Least Common Substring approximation to compute an edit sequence to transform one file into another. Least common substring algorithms solve a different problem than RCS needs solved. LCS-style deltas are useful for a human reader, but have more information than necessary for the reconstruction of a file: LCS computes the shortest sequence of *insert* and *delete* edits where RCS should really be using the shortest delta. Even the best of such algorithms [10] cannot perform as well as more modern delta algorithms which use techniques similar to file compression, generating a sequence of *copy* and *insert* instructions [6] [2]. These deltas are a set of instructions that describe how to construct a new file from an old file, rather than describing how to modify the old file into the new file. Finally, since LCS is more expensive to compute than these other delta mechanisms, it is forced to break the input into lines, leading to disastrous performance on non-line-oriented inputs.

7.1 Xdelta

A replacement for RCS, Xdelta, has been implemented to solve all three of these problems. It uses a delta algorithm which works well on binary files and still outperforms standard `diff` utilities on line-oriented files. Instead of using the RCS trunk organization with forward and reverse deltas, it stores only the most recent copy of a file and reverse deltas carefully computed to eliminate the need for branches. Since there are no branches, each version of a file is simply identified by a serial number.

In the equations that follow, the following conventions will be used. The letters f and d refer to byte sequences of arbitrary length and content. The letters A and D refer to sets of byte sequences which may be treated as the concatenation of their elements. The $\oplus$ operator indicates concatenation. Assume the function $d = \text{delta}(D, f)$ produces a delta d which can be used to construct f from D. The function $f = \text{patch}(D, d)$ accomplishes this reversal, yielding the identity $f = \text{patch}(D, \text{delta}(D, f))$.

For the archive containing n versions, the most recent version of the file, f_n, is stored verbatim (or compressed). One delta is stored for each previous version. A set of reverse deltas:

$$D_n \equiv \left\{ \bigoplus_i d_i : 1 \leq i < n \right\} \tag{1}$$

are stored to allow reconstruction of each previous file version. The XDELTA archive A is the most recent version and the set of deltas: $A_n = D_n \oplus f_n$.

Deltas are generated as follows: when F_{n+1} is inserted, the following delta is computed:

$$d_{n+1} = \text{delta}(f_{n+1} \oplus D_n, f_n) \tag{2}$$

The new archive is constructed, $A_{n+1} = D_n \oplus d_{n+1} \oplus f_{n+1}$. Extraction of the most recently inserted version is always trivial. To generate f_{n-1} from f_n, observe:

$$f_{n-1} = \text{patch}(f_n \oplus D_{n-1}, d_n) \tag{3}$$

where each argument to patch is a member of A_n.

This mechanism is radically different from the encoding used by RCS, and at first it seems to have disposed of exactly the mechanism by which RCS efficiently encodes diverging branches. However, it has not disposed of this property. Figure 3 demonstrates the deltas computed by RCS for the checkin of five file versions on two branches. RCS stores version 1.3 verbatim, reverse deltas d_1 and d_3, and forward deltas d_2 and d_4. RCS succeeds at efficiently storing these five versions because files on the 1.1.1.x branch do not interfere with deltas computed for files on the 1.x branch. The next delta computed on either branch is not affected by the differences between files at the head of either branch. In this way, parallel development does not impact the storage mechanism. The RCS mechanism does, however, have a great impact on the amount of work required to extract each version: files checked in on the 1.x branch are available in time proportional to the distance from the head, while files checked in on the 1.1.1.x branch are available in time proportional to the distance from the root version (1.1) plus the number of files on the 1.x branch. Unfortunately, this type of scenario degrades the performance of the RCS storage mechanism and is exactly what version control utilities are designed to handle. This feature of the RCS storage mechanism makes efficient allocation of new version numbers difficult for both users and higher level version control utilities.

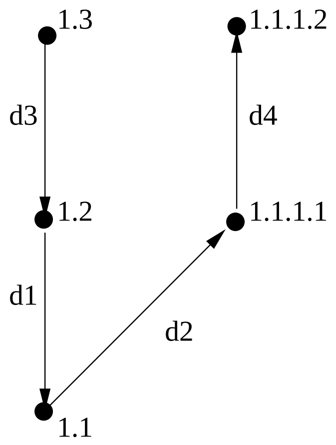

Figure 3: Sample RCS branch structure.

The XDELTA storage mechanism does not use branches and never stores forward deltas. To avoid accumulating multiple copies of deltas transforming one diverging branch into another, XDELTA computes a new delta based not only on two input files, but on previous deltas as well. The equivalent XDELTA encoding is illustrated in figure 4. The delta algorithm employed by XDELTA encodes deltas as a sequence of `copy` and `insert` instructions. A copy instruction occupies $O(1)$ space, while an insertion of length N requires $O(N)$ space. Note that forward insertions are encoded as reverse deletions and forward deletions are encoded as reverse insertions. We are only concerned with insertions, since deletions are not reflected in the sequence of `copy` and `insert` instructions. Suppose that a large deletion occurs between versions 1.1 and 1.1.1.1. The large deletion is stored as an insertion in d_1. It is important, without the presence of RCS-like branching, to insure that future deltas do not repeat the large insertion in d_1. d_2 encodes, as a deletion, the insertion required to bring 1.1.1.1 to 1.2. Were d_3 computed as $\text{delta}(1.1.1.2, 1.2)$, the insertion would be repeated. However, from the above formulae,

$$d_3 = \text{delta}(1.1.1.2 \oplus d_2 \oplus d_1, 1.2) \tag{4}$$

Since d_1 contains a similar insertion to $\text{delta}(1.1.1.2, 1.2)$, d_3 ends up containing a `copy` from d_1 instead of an `insert`. Thus, the XDELTA encoding encodes diverging branches with a similar efficiency to RCS.

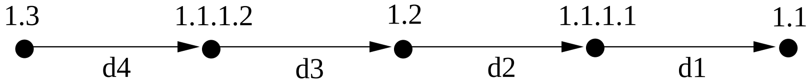

Figure 4: Equivalent XDELTA encoding.

In practice, it is not desirable to use the entire set of previous deltas as input to the delta algorithm, and the set may be trimmed to some previous history of length h. The XDELTA encoding has several other nice properties, not easily available in RCS. Since there are no branches, old versions can be flushed from the archive when they reach a certain age and are no longer referenced by future deltas. Limiting the history to length h, all files which are older than $2h$ may be deleted from the archive by simply removing the corresponding delta. No major reorganization of the archive is required. Similarly, old deltas can be compressed without affecting the efficient retrieval of newer versions. The RCS storage mechanism does not lend itself to either of these techniques.

Delta computation involves computing checksums at various offsets in the file. These checksums may be pre-computed and stored along with each segment. This allows data to be compressed in the archive; when a pre-computed checksum matches during delta computation, segments may be lazily extracted and uncompressed. A great deal of research has been done in text-retrieval algorithms, any of these algorithms may be used for limiting the set of segments which might contain matches prior to delta computation. This technique has not been explored. An even greater generalization of the above technique is incremental, multi-file compression, which does not seem to have been studied.

Another common requirement of a version control archive format is the ability to annotate each version of a file with the name of the author who most recently altered ranges of the file. This attribution record can be kept for the most recent version and reverse attribution deltas can be stored to compute previous attributions. Current implementations of annotate in RCS require generating each parent version beginning with the root.

An experiment was done comparing the RCS and XDELTA archives for each of the RCS files extracted from FreeBSD's CVS repository in the `src/sys/vm` subdirectory, taken on May 3, 1998. The collection totaled 35 files and 1608 versions, of which only 134 versions were off the main ($1.x$) branch. The average number of versions per file is 45.9, with a standard deviation of 39.6. The RCS log messages were stripped before comparison, since XDELTA does not have many of the features of RCS. The resuling archive sizes are as follows:

Archive size	Compressed (bytes)	Uncompressed (bytes)
RCS	429320	1617850
XDELTA ($h = 10$)	913213	1686808

The results show that for an uncompressed archive, the performance of XDELTA is not much worse than RCS. However, XDELTA is capable of efficiently operating on compressed archives and RCS is not, so the advantages seem to outweigh the disadvantages. The difference in compressed archive sizes is so great because the compression of many small files is not likely to be as good as the single compression of one large file.

8 A Global Cache and Transport Mechanism Based on Delta Communication

PRCS servers may be configured to exchange and synchronize data with peer servers. Much like the NNTP protocol [7] [5], these servers may be configured to feed neighbors in a logical server network. This network of PRCS servers provides a distributed, replicant file repository functionality which supports the following, abstract operations:

```
get (file, branch-name)
```

```
put (file, branch-name)
commit ()
```

Though this functionality is could be provided on top of existing NNTP implementations such as INN [13], the effort required to implement the necessary modifications seems as great as the work required to adhere to existing standards. This distribution network is managed separately from and, indeed, unrelated to the semantics of PRCS. Files are named as follows:

```
file-name = <prcs-project, unique-name, md5-checksum>
```

a repository consists of a set of files. The prcs-project component of file-name serves only to group files with their project. The unique-name component serves to group files related by content. All files with the same values of prcs-project and unique-name are treated as related, much like all versions in an RCS archive. The MD5 [12] checksum serves to distinguish variants of the same file `<prcs-project, unique-name>` group. Though file-name contains no sequence number, each server maintains its own, internal record of the order in which each `md5-checksum` variant was created. It will use this sequence to determine the sequence of deltas to transmit to a peer during synchronization. The branch-name argument supplied to each get and put operation allows servers to determine which files they are interested in.

8.1 Synchronization

Each server maintains a log of the files it can produce. Entries in this log are include the name of a file, whether it was created or deleted, and for create operations, the branch name provided with the put or get operation that caused the server to obtain a copy of the file.

The algorithm for synchronizing two servers assumes that both servers have identical copies of several data structures, including a list of interested branches and available files. When two servers are initially connected, they must transmit this information in its entirity. Subsequent transmissions will only send updates to these data structures. To implement this communication efficiently, each server maintains a cookie supplied by the other which indexes a certain point in the remote server's logs. Upon reconnection, each server transmits the new log entries and a new cookie.

Branch names are used to filter new log entries. If a server is not interested in a particular branch, its peers will not transmit log entries for file creations which were associated with that particular log. Internally, each server maintains the set of branches each file belongs to and any change in this state is also transmitted.

To synchronize two servers, each compares the set of files it has belonging to the remote server's branch set with the remote server's set of files. For server A to send a file F to B, if B already contains a file related to F then it sends a forward delta capable of transforming one of the versions B can produce into F. If multiple, related files must be transmitted from A to B, A selects F to be the newest of these files and transmits reverse deltas capable of transforming F into each other related file. If B does not contain a file related to F, A must send the entire file. Later, during description of the PRCS protocol, one more case will be introduced in which A and B

must communicate a file and neither have a version common to the other. In this case, the *rsync* algorithm [1] will be employed to synchronize files.

8.2 Location Service

The previous section describes a mechanism for servers to synchronize the contents of their repository. It is not, however, necessary for a server to maintain a complete copy of every file. A server is only responsible for being able to provide copies of the files belonging to branches it owns. When a server receives a request for a file which does it does not have locally, it contacts the branch's owner. The owner must reply with the file or another authoritative source for the file. The reason for this indirection of authority, not yet introduced, is that it is possible for one server to extend trust to another. Trust, a reflexive, transitive relation between servers, allows one server to claim responsibility for a file on behalf of the branch's owner. By extending trust to peer servers, it is possible for branch owners to only manage transactions on branch-specific meta-data, such as the PRCS project file corresponding to each version. Branch owners are not forced to obtain and where-house all files related to their branch. As a result, the centralized management (and point of failure) is not responsible for the additional work-load of retrieving and storing all of the data it has authority over. Failure of a branch-owner only means that no new versions may be created on that branch.

9 Changes to PRCS

The implementation of PRCS does not change significantly with this reorganization. It requires very little of a repository other than the operations listed above, `get()` and `put()`. The checkin is the only operation which requires `put()`, and must be handled by contacting a server trusted by the branch owner. This communication can be handled by proxy through intermediate, untrusted servers, or by direct communication with the trusted server.

Availability. Information about PRCS and its source and binary distributions are available at `http://www.xcf.berkeley.edu/~jmacd/prcs.html`. Information about XDELTA and its source and binary distributions are available at `http://www.xcf.berkeley.edu/~jmacd/xdelta.html`.

References

[1] BERLINER, B. CVS II: Parallelizing software development. In *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA* (Berkeley, CA, USA, Jan. 1990), USENIX Association, Ed., USENIX, pp. 341–352.

[2] BURNS, R. C., AND LONG, D. D. E. A linear time, constant space differencing algorithm. Master's thesis, University of California at Santa Cruz, Department of Computer Science, 1997.

[1] The rsync algorithm is a coarse delta algorithm which operates over the network by exchanging checksums on relatively large segments of each file. XDELTA can be easily extended to perform the rsync algorithm.

[3] Freier, A., Karlton, P., and Kocher, P. The ssl protocol: Version 3.0, November 1996.

[4] Horton, M. R. RFC 850: Standard for interchange of USENET messages, June 1983. Obsoleted by RFC1036 [5]. Status: UNKNOWN.

[5] Horton, M. R., and Adams, R. RFC 1036: Standard for interchange of USENET messages, Dec. 1987. Obsoletes RFC0850 [4]. Status: UNKNOWN.

[6] Hunt, J. J., Vo, K.-P., and Tichy, W. F. An empirical study of delta algorithms. *Lecture Notes in Computer Science 1167* (July 1996), 49–66.

[7] Kantor, B., and Lapsley, P. RFC 977: Network news transfer protocol: A proposed standard for the stream-based transmission of news, Feb. 1986. Status: PROPOSED STANDARD.

[8] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the coda file system. *SOSP 8*, 2 (October 1991), 213–2251.

[9] MacDonald, J., Hilfinger, P. N., and Semenzato, L. Prcs: The project revision control system. *Lecture Notes in Computer Science (to appear)* (1998).

[10] Myers, E. W. An $O(ND)$ difference algorithm and its variations. *Algorithmica 1*, 2 (1986), 251–266.

[11] Polstra, J. Program source for CVSup. ftp://ftp.cvsup.freebsd.org/pub/CVSup, 1996.

[12] Rivest, R. RFC 1321: The MD5 message-digest algorithm, Apr. 1992. Status: INFORMATIONAL.

[13] Salz, R. InterNetNews: Usenet transport for Internet sites. In *Proceedings of the Summer 1992 USENIX Conference: June 8–12, 1992, San Antonio, Texas, USA* (Berkeley, CA, USA, Summer 1992), USENIX Association, Ed., USENIX, pp. 93–98.

[14] Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2 ed. John Wiley & Sons, Inc., 1996.

[15] Tichy, W. F. RCS: A system for version control. *Software—Practice and Experience 15*, 7 (July 1985), 637–654.

[16] Tridgell, A., and Mackerras, P. The Rsync algorithm. Tech. Rep. TR-CS-96-05, The Australian National University, June 1996.