Panasync: Dependency tracking among file copies

Paulo Sérgio Almeida  Carlos Baquero  Victor Fonte
psa@di.uminho.pt  cbm@di.uminho.pt  vff@di.uminho.pt

Distributed Systems Group
Computer Science Department
Minho University, Braga, Portugal
http://gsd.di.uminho.pt/

Abstract

File copying is frequently used to implement ad hoc management of file replicas, backups and versions. Such tasks can be assisted by appropriate applications, at the expense of introducing some restrictions to the usage patterns. In particular, this is the case of interactions involving disconnected machines and transportable media. PANASYNC tries to support these actions by introducing a set of commands for file copying and re-integration that complement the file-system commands and provide support for dependency analysis among time-stamp assisted files.

1 Introduction

User interaction with the file system is supported by command line or by graphical user interfaces, both alternatives providing standard operations such as file and directory creation, renaming, copying and removal. In their normal activity users can resort to a given operation in order to achieve different purposes. For instance, the copy operation can be used to create a backup of a file, to branch a project or even to substitute a file with the contents of another. Due to the simplicity of the basic operations available to the user, the underlying purpose cannot be perceived by the system. The system treats the operations indistinguishably, thus having no provision to assist the user along its tasks. Consequently the user is on its own, regarding, for instance, file version management.

The common solution to this problem is to adopt version-control environments offering a special set of tools and often embedding their own filing structures in the underlying file system. However, the adoption of a special environment for versioning control is usually a matter of complexity assessment, and users tend to avoid it when they want to manage what they perceive as a simple file duplication or versioning task. Few users resort to versioning support tools when taking a draft document copy to eventually work on it in a weekend.

Additionally, the use of existing versioning and replication environments calls for a centralized or at least pre-set distributed configuration that is frequently inadequate to the user mobility needs. Missing features encompass the uncoordinated creation of new replicas and isolated forking of new versions. Together these restrictions can lead users back to basic file system operations, or at least make them think twice before switching into a coordinated environment.

In this article we discuss the design of a set of tools that provide autonomous file copying and versioning. These constructs can be implemented by a set of commands that manage additional time-stamping data over an off-the-shelf file system or by the design of a file system extension that manages and hides the time-stamp data. These tools can be used as a complement to the standard file system operations.

2 Related Work

Replicated file systems such as Coda [4], Ficus [3] and Rumor [8], are bound to rely on some notion of replication volumes (typically subtrees in a given machine). These systems can be related to version control systems if interpreted as providing version control over a well defined number of branches, when allowing optimistic replica evolution on each volume.

In these systems, the use of vector time-stamps for conflict detection requires either an indexing of the replication volumes or a way to univocally identify each volume. That assumption enables the design of file time-stamps as mappings from volume id (which can be a name or vector index) into an update counter [5, 7]. A consequence of this design is that a file cannot be replicated in the same volume, and in particular in the same directory.

Another problem is the transitive replication that can occur, for instance, when using transportable media.
this case, the availability of one replica is not sufficient for the autonomous creation of the new replication volume that would host a subsequent replica. All these patterns of usage, when actually needed, lead the users back to the use of uncoordinated copy.

The solution to the autonomous identity creation problem relies on a recursive construction of the new ids, in the presence of a single file replica. The BAYOU system [6] uses a similar technique for volume identity creation. PANASYNC will use a recursive technique that is based on previous work on autonomous causality [2] and autonomous file time-stamping [1] in order to provide single file replication and versioning.

Simple but well deployed forms of replication systems, such as Microsoft Windows Briefcase and the new off-line files and folders of Windows 2000, target optimistic replication between a mobile unit (or an instance of transposable media) and one host. This restricted form of replication fits the general case of replicated file systems.

Traditional versioning systems, such as CVS1 and PRCS2, are targeted to manage short derivations from a central branch of development. In this sense they support an arbitrary number of concurrent evolutions but do not treat them as first class elements, and keep a centralized control on the versioning information. A recent trend in versioning systems, as depicted in BitKEEPER3 design, aims to support parallel lines of development that share code improvements with a more robust ‘diffing’ technique. This approach differs from the PANASYNC scenario as it targets the management of sets of files (repositories), in contrast with a file based approach, and focuses on ‘diff’ portability and not on causality based domination analysis between parallel evolutions. Nevertheless this trend appears to share common motivation with our approach whilst at a different level.

3 Revisiting Copy Constructs

PANASYNC intends to add file based replication and versioning constructs in a way that complements the usual file manipulation constructs. Since some of their functionality will be complemented, it is important to review the possible applications of existing operations. Considering the arbitrary syntax and Unix mapping shown in Table 1, we discuss how some user tasks expressing replication or versioning are typically performed.

If the intent is to create a new file unrelated to the other files, the operation could be create(content,new-file-name). However, it is sometimes useful to start the new file with the content of another file (for instance when starting a new LATEX document or a CGI script), which would lead to copy(file-name,new-file-name).

When the intent is to keep a backup copy of a given file, the operation is something like copy(file-name,file-name.orig) or slightly different, copy(file-name,file-name.v01), if several backup versions are to be created. If, latter on, the user wants to discard changes he issues something like copy(file-name.orig,file-name). Alternatively, if the user needs backup versions for a set of files, he might make use of a new sub-directory that gathers a given version of the files, and then use something like copy(*,dir-name.v01), which keeps the original file names but places them in a different name space.

Finally, when the intent is to replicate a file or set of files the practice is to keep the names and copy them into a different name space (disk and directory). Later on, replica identification and the possibility of replica re-integration must be evaluated by the user and lead to the appropriate copy or move operations.

In all these tasks the copy operation is heavily used and its different intents are only vaguely captured by the choice of names and name spaces that the user conducts. There is no way of providing system support to these operations and the users are on their own to make either correct decisions or mistakes.

It is also clear that the directory structure is used for several purposes, division of name space, classification, and identification of different physical storage devices. It can be the case that files like /floppy/panasync.tex and /home/cbm/psart.tex are versions of the same entity and /src/q/readme.txt and /tmp/qinst/readme.txt are totally unrelated.

4 Panasync Operations

In order to assist file replication and versioning tasks, we propose a set of commands that track dependencies between versions of files. A file system with PANASYNC extensions, or user level commands, manages ordinary as well as panasync-enabled files, the latter having an extra time-stamp attribute. For convenience of discourse we designate the panasync-enabled files pfiles and the others ofiles (ordinary files); we also use the term file to

| create(body,target) |
| echo "body" > target |
| copy(base,target) |
| cp base target |
| move(base,target) |
| mv base target |

Table 1: Classic file constructs.
refer to either class. The traditional commands apply to both file types, but pfles can also be manipulated by the PANASYNC operations as shown in Table 2 (where we also present possible Unix mappings).

The new operation is used to create a pfle. Its name, target, is mandatory and an optional file name for initialization is allowed by indicating a base file. Usage of this operation means that new a lineage of files is being created and that the target pfle is not comparable with other lineages. Ordinary files are all non comparable.

When a backup, versioning or replication action is needed, users can resort to the duplicate operation. This operation creates a target pfle from the base pfle and ensures that both share the same lineage. After duplication both pfles are equivalent, hold the same contents and should be regarded as siblings. In fact, although base was the starting point they do not hold a parent/child relation.

An immediately subsequent join operation with one of these pfles as base and the other as target would remove base since the system detects that they share unchanged content, as well as positions that can be determined to be equivalent in the version lineage.

By consulting the pfles time-stamps the join operation is able to relate any pair of files, verify if one of the following conditions holds, and advise appropriate action:

- **Condition**: base and target are in distinct lineages, or one or both of them are pfles.
  **Action**: Abort the join and do nothing.

- **Condition**: base dominates target, or target dominates base.
  **Action**: The normal action is to remove base and place in target the content that dominates (either from base or target), but an option can be provided to choose the dominated content.

- **Condition**: Neither base nor target dominates, which means that they hold concurrent updates.
  **Action**: Do nothing or prompt the user for a reconciliation file, in which case both base and target are removed and the new contents are stored in the position target. This new file dominates all files that is ancestors would dominate.

This description shows that names are not important in these operations since pfles have enough information to distinguish file instances as well as to compare them. As a consequence of this, the choice of pfle names need only address name clash avoidance in the directory system that stores them.

In fact it is possible to design an option that applies a join operation recursively to two whole subtrees. This would select all pfles from the subtrees, produce two flat lists of files and compare those in the same lineage, removing, for instance, the dominated files from the first subtree.

Another useful construct, although not a basic one, is a panasync base1 base2 command that produces a join of the two files in a temporary pfle and immediately duplicates it again into base1 and base2. The overall effect is to keep two copies with synchronized contents.

Renaming pfles can be done as usual with the original move command, as long as time-stamp association to file name can be tracked. Depending on the system, this need can lead to a simple patch to the native move command or to the introduction of a panamv construct.

Finally, the use of the native copy with a pfle as base produces a target ofile with unrelated lineage, which is a useful functionality.

5 Synopsis of Time-Stamping

The complex pattern of version and lineage control can only be achieved with a sound time-stamping technique that supports autonomous creation of a partial order among file replicas, and the identification of lineages. A presentation of the causality model and time-stamping technique is beyond the scope of this paper. Some insight on the technique can be found in [2, 1]. Here we will only address some significant points that characterize this time-stamp model.

Vector time-stamps, as originally shown in [5], allow the tagging of identical replicas with identical time-stamps. This is possible due to the fact that the identity of the replication volumes, and the information of the hosting volume for each replica, can complement the information stored in the time-stamp. On the contrary, if we wish to have autonomous time-stamps all the relevant information must be stored in each replica time-stamp. This leads to the existence of distinct time-stamps that identify equivalent replicas. The partial order algorithm must detect that simple replica duplication does not make them different but only raises the possibility of separate modifications. Such replicas cease to be equivalent once they suffer changes.

Unlike vector time-stamps, this scheme does not impose structural limits on the number of replicas, since
replica identity is recursively constructed with the information that is locally available.

6 Example Scenarios

.... Setup ..... 
1$ pananew mybib.bib pana.bib 
2$ panadup pana.bib /floppy/pana.bib 
3$ cat entry1.bib >> /floppy/pana.bib 
4$ panadup /floppy/pana.bib /zip/p.bib 

.... 1st Scenario ..... 
5$ mv /floppy/pana.bib /floppy/panasync.bib 
6$ cat DSM.bib >> /zip/p.bib 
7$ panajoin pana.bib /zip/p.bib 
Info: /zip/p.bib content dominates pana.bib 
8$ panajoin /zip/p.bib /floppy/panasync.bib 
Info: /zip/p.bib content dominates /floppy/panasync.bib 

.... 2nd Scenario ..... 
5$ mv /floppy/pana.bib /floppy/panasync.bib 
6$ cat DSM.bib >> /zip/p.bib 
7$ panajoin /floppy/pana.bib /zip/p.bib 
Warning: Files are concurrent 
use -s to specify substitute 
8$ adiff /floppy/pana.bib /zip/p.bib -o merge.bib 
9$ panajoin /floppy/pana.bib /zip/p.bib -s merge.bib 
10$ panajoin /zip/p.bib pana.bib 
Info: /zip/p.bib content dominates pana.bib 

Figure 1: Example runs with PANASYNC tools.

The example in Figure 1 (with first scenario in Figure 2) shows a hypothetical use of PANASYNC commands under the Unix environment. In the setup phase a new lineage is created together with the file pana.bib and its contents are initialized with mybib.bib content (1$). We recall that there is no ordering relation between these two files.

Afterwards the pana.bib file is duplicated to a directory mapping a floppy device (2$) and its contents are changed with the concatenation of entry1.bib (3$). Finally this file is duplicated into /zip/p.bib (4$). We can expect that /zip/p.bib and /floppy/pana.bib are equivalent, and that both dominate the local pana.bib content.

For simplicity all examples have been illustrated in a single machine. It must be kept present that all operation steps are possible on any arbitrary machine that accesses the used persistent store. The use of floppy and zip names emphasizes this possibility since they designate transportable persistent media.

6.1 First Scenario

Now we change the name of the floppy resident file into /floppy/panasync.bib (5$). In fact we can move this file to any place or system since its identity does not depend on its name. Next we add some content to p.bib and try to join it with the local pana.bib (6$).

This join is straightforward since one of the files dominates the other. As usual, the two files supplied as arguments to panajoin are checked for their relative order and the join outcome is written to the second file argument. This is the case even if the second file is the dominated one.

After the last panajoin invocation the three replicas from the start of this first scenario have been collapsed into a single replica at /floppy/panasync.bib. Since there were no concurrent changes the convergence was trivially accomplished. A simple way to check for the presence of concurrent changes, in Figure 2 as well as the second scenario figure, is to track the arrows (●) that indicate changes. This can be done by following the arrows, from the replicas, in the reverse direction and check if both have changes that the other has not seen.

6.2 Second Scenario

In this second scenario, with its evolution outline in Figure 3, we make sure that some concurrency of changes does occur, by adding content to both /zip/p.bib (5$) and /floppy/pana.bib (6$). Consequently, the join tentative over these two files fails and issues a warning identifying the occurrence of concurrency, and asking for the provision of a content that reconciles the files.

The user is free to choose the content that is to be provided. In this case, the user resorts to the adiff tool in order to select the merged content from the two concurrent files and to supply it to the next panajoin invocation.

The last panajoin invocation illustrates that when merges of concurrent evolutions occur, the order of the new file is such that it dominates all the files that were previously dominated by either of the merged files. This factor empowers the user decisions when supplying a merge and helps future replica convergence, thus constituting a very powerful property that is particular to this system.
(floppy) panasync.bib
↓$ mv$
(floppy) pana.bib

(floppy) • pana.bib
↓$ cat >>$
(floppy) pana.bib

(floppy) • pana.bib
↓$ cat >>$
(floppy) pana.bib

(local) mybibs.bib

(local) pana.bib

Figure 2: First Scenario. Here a single branch dominates the other branches. The mv action that renames one of the pfiles does not change its identity and time-stamp.

Figure 3: Second Scenario. Two parallel branches suffer concurrent changes and are re-conciliated with a merge content. The resulting pfile inherits existing domination relations and supersedes an early branch.

7 Design Issues

The basic PANASYNC implementation will be built over a set of portable command-line tools. The purpose of these tools is not only to test the use of this dependency tracking system, but also to ease its integration into existing file managers. A second phase will encompass the exploration of an adaptation technique for native file systems, eventually with the use of a reflection mechanism.

In PANASYNC, file naming is a user convenience although the system does not rely on it to track dependencies between pfiles. Evidences from observation of typical patterns of usage suggest that PANASYNC users should be able to change pfile names at will and the system must still be able to ensure correct dependency tracking. To achieve this purpose, PANASYNC will rely on a mapping from a special pfile identifier into its name. This will enable the system to assess if two pfiles belong to the same lineage independently of their current names. The identifier will be given to the pfile upon creation and associated to the specified name. Each time a move is issued the pfile identifier mapping will be updated. For practical purposes this identifier can be generated from traditional techniques based on existing hardware settings (e.g., ethernet address), file creation time and a random value.

To achieve its purposes, PANASYNC needs to store this extended information about pfiles. In fact, not only it will need to record the current name of each pfile, but also its time-stamp and an MD5 digest to actually track its dependencies.

Another issue in the design of PANASYNC is the trans-
parent detection of modification of pfiles’ content. This 
objective cannot be reliably achieved by evaluating the 
creation and modification time-stamps provided by most 
of the traditional file systems. Instead, PANASYNC calcu-
lates a MD5 digest for each pfile upon its creation, and 
also stores this information on the mappings discussed 
above. Each time panadup and panajoin are issued the 
MD5 is recalculated, enabling the detection of modifica-
tions on the pfies with the setting of a dirty attribute 
that is used in the time-stamp construction algorithm.

To ensure portability PANASYNC will provide an exter-
nal representation of pfles’ attributes. This will enable 
transferring pfies through non-supported systems, such 
as email, for instance.

8 Conclusions and Future Work

We have presented the motivation behind the concep-
tion of PANASYNC and shown usage scenarios. The sys-
tem aims to support common tasks of file replication and 
versioning, which could be done either manually, with-
out system support, or under control environments that 
are focused toward coarser grain scenarios. Also, these 
environments cannot track dependencies among an ar-
binary number replicas. We believe that the addressed 
patterns of fine grain file copying are bound to increase 
with ongoing trends of increased user mobility and in-
formation sharing among mobile and fixed units. The 
PANASYNC approach does not intend to substitute the 
functionality of versioning systems or replicated file sys-
tems, but rather act as orthogonal support for a partic-
ular and common class of use cases.

Apart from the practical design issues, the central 
point that enables the conception of a system with these 
characteristics is the underlying time-stamping scheme. 
Presently, we have reached a time-stamp design that al-
 lows identifier simplification upon joins. This design for-
mat allows us to start the construction of the first com-
mand prototypes.

Having designed the time-stamping mechanism, the 
next step will be the study of time-stamp size impact 
on the system under an average work pattern. Although 
not comparable with the small size of standard time at-
tributes we are confident that the extended control pos-
sibilities will make the use of pfies worth in a significant 
set scenarios.

Acknowledgments

The authors would like to thank Rui Oliveira and the 
amonymous referees for their comments.

References

[1] Carlos Baquero and Paulo Sérgio Almeida. Towards ef-
cient time-stamping for autonomous versioning. In Ac-
tas informais do EPCM’99, Encontro Português de Com-
[2] Carlos Baquero and Francisco Moura. Causality in au-
tonomous mobile systems. In Third European Research 
Seminar on Advances in Distributed Systems. Broadcast, 
EPFL-LSE, April 1999.
[3] Richard G. Guy, John S. Heidemann, Wai Mak, 
Thomas W. Page, Gerald J. Popek, and Dieter Roth-
meier. Implementation of the ficas replicated file sys-
USENIX, June 1990.
ation in the Coda file system. In Thirteenth ACM Sym-
podium on Operating Systems Principles, volume 25, pages 
213–225, Asilomar Conference Center, Pacific Grove, US, 
Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, 
David Edwards, Stephen Kiser, and Charles Klime. 
Detection of mutual inconsistency in distributed sys-
tems. Transactions on Software Engineering, 9(3):240– 
246, 1983.
[6] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, 
Marvin M. Theimer, and Alan J. Demers. Flexible update 
propagation for weakly consistent replication. In Sixteen 
ACM Symposium on Operating Systems Principles, Saint 
Malo, France, October 1997.
[7] David Ratner, Peter Reiher, and Gerald Popek. Dy-
namic version vector maintenance. Technical Report 
CSD-970022, Department of Computer Science, Univer-
sity of California, Los Angeles, 1997.
[8] Peter Reiher, Jerry Popek, Michal Gunter, John Sal-
omone, and David Ratner. Peer-to-peer reconciliation 
based replication for mobile computers. In Max Mulhauser, 
editor, Special Issues in Object-Oriented Programming, ECOOP’96 II Workshop on Mobility and 
Replication. Dpunkt Verlag, 1996.