Flexible and Safe Resolution of File Conflicts

Puneet Kumar and M. Satyanarayanan

November 1994
CMU-CS-94-214

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the Proceedings of the Usenix Winter 1995 Technical Conference on Unix and Advanced Computing Systems, January 16-20 1995, New Orleans LA

Abstract

In this paper we describe the support provided by the Coda File System for transparent *resolution* of conflicts arising from concurrent updates to a file in different network partitions. Such partitions often occur in mobile computing environments. Coda provides a framework for invoking customized pieces of code called *application-specific resolvers (*ASR*s)* that encapsulate the knowledge needed for file resolution. If resolution succeeds, the user notices nothing more than a slight performance delay. Only if resolution fails does the user have to resort to manual repair. Our design combines a *rule-based approach* to ASR selection with *transactional encapsulation* of ASR execution. This paper shows how such an approach leads to flexible and efficient file resolution without loss of security or robustness.

This research has been supported by the Air Force Material Command (AFMC) and the Advanced Research Projects Agency (ARPA) under Contract F19628-93-C-0193. Support also came from IBM Corporation, Digital Equipment Corporation, and Intel Corporation.

Keywords: distributed file systems, resolution, application-specific resolvers, conflicts, safety, flexibility, transparency, robustness, Coda

1 Introduction

Optimistic replication has been shown to be a viable approach to high availability in distributed Unix file systems [15, 3]. It is especially valuable in mobile computing environments, where voluntary and involuntary disconnections are the norm rather than the exception [7]. The Achilles heel of optimistic replication is the need to cope with *conflicts* caused by concurrent updates in different network partitions. Such conflicts can hurt usability if they require frequent manual intervention by users.

In this paper we show how the Coda File System addresses this problem by providing for transparent handling, or *resolution,* of file conflicts. The essence of our approach is to provide a framework for installing and invoking customized pieces of code called *application-specific resolvers (*ASR*s).* Each ASR encapsulates knowledge specific to its application. If resolution succeeds, the user notices nothing more than a slight performance delay. Only if resolution fails does the user have to resort to manual repair.

Many practical considerations complicate realization of this simple idea. Users need to be able to control which ASR gets invoked for a specific application. Considerations of security imply the need to restrict the scope of damage caused by an errant ASR. Since intermittent connectivity is common in wireless communication, it is necessary to anticipate failures during the execution of an ASR and to encapsulate its effects in a way that permits easy cleanup. The Coda ASR mechanism addresses these and other related concerns by combining a *rule-based approach* to ASR selection with *transactional encapsulation* of ASR execution. Our implementation confirms that flexible and efficient file resolution is indeed possible without sacrificing security or robustness.

We begin the paper with an overview of Coda and a more detailed rationale for an application-specific approach to file resolution. The bulk of the paper consists of an overview of the ASR mechanism and details on how it achieves flexibility while preserving safety. To illustrate the use of the ASR mechanism, we describe some example ASRs that we have built. We conclude with an evaluation of performance and a discussion of related work.

2 Coda File System

Coda is a descendant of AFS-2 [4] that has *high data availability* as its main goal. Like AFS, it is based on the client-server model, provides a single shared, location transparent name space, and maintains cache coherence across clients using callbacks. Files are stored in *volumes,* each forming a partial subtree of the name space. Volumes are administrative units, typically created for individual users or projects. At each client, a user-level process called *Venus* manages a file cache on the local disk.

Coda uses two complementary strategies, both based on optimistic replication, to achieve high data availability: *server replication* and *disconnected operation*. Server replication allows volumes to be stored at a group of servers called the *volume storage group* (VSG). At any time, the subset of those servers available is called the *accessible volume storage group* (AVSG). Disconnected operation arises when the AVSG becomes empty. To prepare for disconnection, users may *hoard* data in the cache by providing a prioritized list of files. Venus combines explicit hoard information with LRU information to implement a cache management policy that addresses both performance and availability concerns.

Earlier papers [15, 7] provide more details on server replication and disconnected operation. Other papers [8, 10, 11] discuss broader aspects of Coda's approach to conflict resolution, and provide details on mechanisms such as directory resolution that are not covered here.

3 Motivation and Goals

An update conflict arises due to write-sharing of an object from partitioned clients. There is considerable anecdotal evidence and some quantitative evidence [7] to confirm that the average level of write-sharing in personal computing environments is low. This is, of course, what makes optimistic replication viable in such environments.

Unfortunately, certain applications can exhibit much higher than average levels of write-sharing. One example is the use of an online appointment calendar (such as Aldus Datebook [13]) by an executive and her secretary. Another example is the use of an online checkbook (such as Quicken [6]) for a joint account by a couple. Other examples can be drawn from the emerging body of software (such as Lotus Notes or DEC LinkWorks [2]) for computer-supported cooperative work. It is important to note that all of these examples are personal-computing applications; they are not drawn from the online transaction processing domain, where optimistic replication would clearly be inappropriate.

The importance of application assistance in file resolution can be seen by considering the example of a calendar management program. Suppose an executive and her secretary both make appointments while the former is disconnected. Upon reconnection, Venus detects that the file containing appointments is in conflict. But it has no knowledge of the format of file contents, nor of whether there is really a scheduling conflict. Only code specific to the calendar program can tell, for instance, that appointments for an hour each at 8am and 10am on the same day pose no problem if they are in the executive's office, while those same appointments are impossible to keep if they are in New York and San Francisco.

Even in the absence of computers, conflicts occur in application domains such as the appointment calendar and checkbook examples mentioned above. People are already inured to coping with occasional conflicts in such domains. Hence an acceptable goal for file resolution is to reduce the frequency of manual repair, rather than eliminating it altogether. Total elimination of conflicts is, by definition, an unattainable goal in an optimistically replicated environment.

4 Design Overview

A fundamental design choice pertains to the site of execution of an ASR: should it be executed on a server or on a client? Our choice was to execute ASRs on clients. The primary reason for this decision was to preserve the security model of Coda. Allowing arbitrary ASRs to execute on servers would have violated Coda's basic assumption that servers run only trusted software. Considerations of scalability were a second important factor in our decision. Since the computational resources needed by an ASR may be large, scalability is enhanced by off-loading this burden to clients. A third reason for executing ASRs on clients is the fact that much of the supporting machinery needed to execute an ASR is already present at the client but not at servers. For example, Venus already incorporates the code needed to perform pathname resolution.

Venus performs file resolution *lazily*. An ASR is only invoked in the course of servicing a system call, when Venus discovers that a file has divergent replicas. This is in contrast to an aggressive strategy, whereby execution of ASRs would be performed *en masse* upon recovery from a network partition. Our approach reduces the likelihood of recovery storms, a serious concern in environments with intermittent connectivity. But it does mean that the entire performance cost of ASR execution is incurred by the triggering system call, and can therefore not be hidden from applications and users.

Logically, the Coda ASR mechanism can be viewed as comprising three distinct parts: one part responsible for *invoking* an ASR when needed, a second part pertaining to *selection* of the correct ASR and a third part responsible for overseeing the *execution* of an ASR. In practice, of course, there is some interdependence between these parts. An underlying consideration in the design of all aspects of the ASR mechanism is the desire to preserve *transparency* from the viewpoint of users.

Viewed from a high level, the ASR mechanism works as follows. On every cache miss, Venus verifies that all replicas of the file being accessed are identical. If Venus detects divergence of replicas it searches for an ASR for this file using rules specified by the user. If an ASR is found, it is executed on the client. The ASR's mutations are performed locally on the client's cache and written back to the server atomically after the ASR completes. The application that requested service for the file is blocked while the ASR is executing. If a failure occurs, ASR execution is aborted and the results of partial execution are flushed from the cache. If no ASR was found or the ASR execution fails, an error code indicating a conflict is returned to the application process.

We elaborate upon this high-level description in the two sections that follow. Section 5 describes those aspects of our design that contribute to flexibility. Section 6 addresses aspects pertinent to safety.

5 Achieving Flexibility

5.1 Invoking an ASR

To execute an ASR on the client, Venus makes a request to a special process called the `ASR-starter`, as shown in Figure 1. As its name implies, the `ASR-starter` process is responsible for finding and executing an ASR for a file. An ASR cannot be executed on a client unless it is running an `ASR-starter`. The functionality of the `ASR-starter` could have been provided as a routine within Venus. However, we chose not to do so because Venus code is already complex. Implementation and debugging of the `ASR-starter` was greatly simplified by making it a separate process. The only disadvantage of this approach is slightly higher latency for starting an ASR.

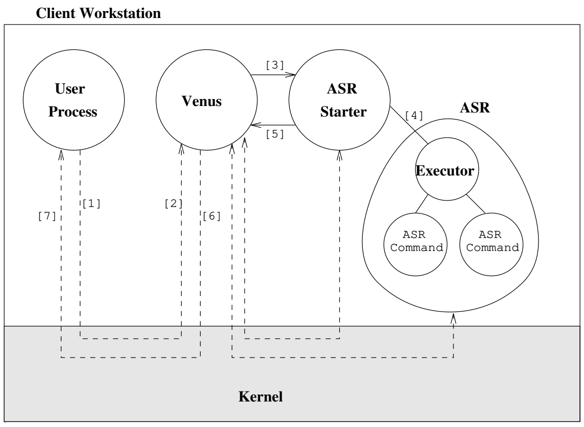


Figure 1: ASR Invocation

This figure shows the message exchange between processes involved in an ASR invocation on a client. Seven events responsible for the ASR invocation and execution are labelled in the Figure: events `[1]` and `[2]` are the user request for a file that has diverging replicas; event `[3]` is the `InvokeASR` RPC; event `[4]` is the execution of the ASR commands; the ASR result is returned to Venus via the `Result_Of_ASR` RPC labelled as event number `[5]`; events `[6]` and `[7]` return the result of the user request. The dotted lines show the message exchange necessary for servicing Coda file system requests made by the `ASR-starter`, the user process and the ASR processes.

A pool of threads in Venus, called *workers*, are responsible for servicing system calls. Once a worker is assigned to a system call, it is bound to that call until completion. The request to start an ASR is made by a worker when it notices that a file has diverging replicas. This request is made via an RPC, called `InvokeASR`, to the `ASR-starter` process. If the request is successful, the process-id (pid) of the ASR is returned to the worker. The worker now blocks, and awaits the result of the ASR's execution. The user process whose system call is being serviced by the worker thread also remains blocked for this duration. If an ASR cannot be started, resolution fails and a code indicating conflict is returned immediately to the user process.

The `InvokeASR` RPC has two parameters: the pathname of the file needing resolution, and the identity of the user on whose behalf the ASR is being executed. The pathname is used by the `ASR-starter` process to find an appropriate ASR. The user identity is used to restrict the privileges of the ASR via the Unix `setuid` mechanism. The `ASR-starter` itself runs as root.

Once the ASR completes execution, its result is returned to Venus by the `ASR-starter` via the `Result_Of_ASR` RPC. The pid of the recently completed ASR is also returned to Venus so that it can resume the worker waiting for this ASR. If the ASR return code indicates success, the worker retries servicing the user's request. Otherwise it returns an error code, `ESYMLINK`, to the user application.

5.2 Selection of an ASR

Resolution rules are stored in a file named `ResolveFile`, whose format is similar to a `Makefile`. The scoping mechanism for `ResolveFiles` is analogous to lexical scoping in common programming languages. Rules contained in a `ResolveFile` apply to all files in the subtree rooted at its directory, except where overridden by another `ResolveFile` lower in the tree. To find the resolution rule for a file, the `ASR-starter` searches for a `ResolveFile` upward from the file toward the root. The search stops at the earliest `ResolveFile` encountered, or at the root.

The advantage of the scoping mechanism is that resolution rules are automatically inherited as new objects are created. For example, a user may have only one set of resolution rules in a `ResolveFile` in his home directory, which applies to all his files. To simplify sharing of rule files, the `ResolveFile` entry in a directory can be a symbolic link. For example, an application-writer could provide a `ResolveFile` for the application's files. All users of this application

```
<object-list>:  <dependency-list>
                <command-list>
```

Figure 2: Format of rules in a `ResolveFile`

can share the `ResolveFile` by creating a symbolic link to it in their personal directory containing the application's data files.

Like `Makefiles`, `ResolveFiles` contain multiple resolution rules separated by one or more blank lines. Each rule has the format shown in Figure 2.

The object-list is a non-empty set of object names for which this resolution rule applies. Object-names can contain C-shell wild-cards like "`*`" and "`?`", thus allowing one rule to be used for multiple files. For example, a rule that specifies `*.dvi` in its object-list applies to all files with a `.dvi` extension.

The dependency-list contains a list of object names whose replicas must be identical before the rule can be applied. This provides a useful check for resolvers that regenerate a file based on the contents of another file. Consider once again the example of a file with a `.dvi` extension. Its contents can be regenerated by processing the source file with Latex. However, this strategy is usable only if the source file itself does not have divergent replicas, a condition that can be checked automatically by adding the name of the source file (i.e., the file with the `.tex` extension) to the dependency-list. In principle it would be possible to recursively resolve objects in the dependency-list. However, to keep the implementation simple, an ASR invocation is aborted if any object in the dependency-list has diverging replicas.

The command-list consists of one command per line. Each command specifies a program to be executed along with its arguments. Since the object-list may contain wild-cards, some file-names that need to be passed as arguments to the resolver programs are not known until the rule is being utilized to invoke the ASR. Therefore, the rule-language provides macros that serve as place holders for these file-names. The language provides three `make`-like macros, `$*`, `$>` and `$<`. The `$*` macro expands to the string that matches the wild-card in the object-list; `$<` expands to the pathname of the parent of the file being resolved and `$>` expands to the file's name. These macros are expanded dynamically when a rule containing them is used to execute an ASR. Figure 3 shows the macro expansions using a simple example.

```
*.dvi:  $*.tex
        file-resolve $</$>
        latex $*.tex
```

For a file `/coda/usr/pk/foo.dvi` the rule expands to:

```
file-resolve /coda/usr/pk/foo.dvi
latex foo.tex
```

And is executed only if the replicas of `foo.tex` are not diverging.

Figure 3: Macro-expansion of a Resolution Rule

To find a rule that applies to a file `foo`, the `ASR-starter` first parses the `ResolveFile`. The string `foo` is matched against the names in the object-list of each parsed rule. The first rule containing a match is used as the resolution rule for `foo`. If no match is found or a syntax error is found in the `ResolveFile`, the `ASR-starter` assumes no ASR exists for `foo` and returns an appropriate error code to Venus.

5.3 Exposing Replicas

Although Venus normally makes replication transparent to user processes, an ASR needs to be able to examine individual replicas of the file being resolved. To allow this, Venus temporarily modifies the name space seen by an ASR. The file being resolved appears to be a directory with each replica appearing as a child of that directory. For example, two replicas of file `/coda/usr/pk/foo` would be accessible as `/coda/usr/pk/foo/rep1` and `/coda/usr/pk/foo/rep2`. This in-place explosion of a file into a fake directory only occurs for files being resolved, and only at the client executing the ASR – replication continues to be transparent for all other files and at all other clients.

File replicas in a fake directory are accessible for reading via the normal Unix interface. However, the only mutating operation allowed on the fake directory is the `repair` pioctl, that takes the name of a replacement file as input. The replacement file can exist in the local Unix file system or it can be one of the replicas of the file being resolved. Its contents are used to atomically set all replicas to a common value. Once this operation succeeds, Venus collapses the fake directory back into a file. The implementation of fake directories makes use of the `mount` machinery already present in Venus, as elaborated elsewhere [9].

For the duration of the ASR's execution, the volume containing the file being resolved is forced into a special *fakeify* mode. In this mode, any file in that volume with diverging replicas, is converted to a fake directory upon access. This functionality is necessary for an ASR that simultaneously resolves a group of files with diverging replicas. For example, a user's calendar might be maintained in multiple files, and its ASR may need to examine the replicas of all files to perform resolution. Of course, this approach assumes that all files mutated by the application are contained in the same volume.

Since the number and identity of the diverging replicas are not known *a priori*, the rule-language syntax provides three additional macros, `[`i`]`, `[*]` and `$#`, that serve as *replica specifiers*. `[`i`]` is replaced by the name of the i'th replica, `[*]` by a list of names of all replicas, and `$#` by the replication factor of the file. Figure 4 shows the use of these macros with a simple example.

```
*.cb:
   merge-cal-reps $< $# $>[*]
```

expands to

```
   merge-cal-reps /coda/usr/pk 2 cal.cb/server1 cal.cb/server2
```

Figure 4: Macro-expansion of Replica Specifiers

This figure shows the macro-expansion for a file whose replicated pathname is `/coda/usr/pk/cal.cb`. The file's volume is replicated at two servers, `server1` and `server2`.

5.4 ASR Execution

The ASR is executed after all macros in the resolution rule have been expanded. Multiple programs may be executed in a single ASR invocation, since a rule's command-list can contain more than one command. Since these programs may take an arbitrary long time to execute, they are executed by a separate process, called the `executor` which is forked by the `ASR-starter` process. The `ASR-starter` process returns the pid of the `executor` to Venus. Running the ASR via the `executor` frees the `ASR-starter` process to continue servicing other requests from Venus. Once the ASR completes, its result is returned via the `Result_Of_ASR` RPC.

The `executor` runs with the identity of the user whose request triggered resolution. It executes the commands in the command-list sequentially, each in a separate process. If any command fails, the `executor` is aborted and an error returned to Venus.

Since Venus runs on multiple hardware platforms, the `executor` must have the ability to choose the binary appropriate for the client machine. To achieve this functionality, it uses the `@sys` pathname expansion capability of the Coda kernel. The kernel evaluates the `@sys` component to a unique value on each architecture. This mechanism allows a single resolution command pathname to be used for any client machine. For example, the pathname `/usr/coda/resolvers/@sys/bin/merge-cal`, translates to `/usr/coda/resolvers/pmax_mach/-bin/merge-cal` on DECstation 5000 clients and to `/usr/coda/resolvers/i386_mach/bin/merge-cal` on Intel-386 clients.

6 Preserving Safety

6.1 Security

Enforcing security is a critical issue because an ASR is not a piece of trusted system software. A wide range of catastrophes ranging from simple coding errors to full-fledged Trojan horse attacks have to be guarded against. The problem is especially tricky because ASRs are executed transparently. In other words, a user may be completely unaware that an innocent file reference by him caused a malicious ASR to be executed. Coda provides three levels of defense against this problem.

As the first level, which is the default, Coda uses the `setuid` mechanism to restrict the privileges of an executing ASR. Since the ASR only possesses the privileges of the user who triggered it, damage is limited to those portions of the Coda name space that can be modified by the user.

The next level provides control over which ASRs can be executed by a client. Using the `cfs` command, a user may specify a list of directories where trusted ASRs can be found. ASR invocation will fail if an attempt is made to execute an ASR from a directory not in the list. Even a trusted ASR is, however, subject to the `setuid` restrictions mentioned previously. It would be a fairly simple matter to extend this scheme to include a fingerprinting mechanism [18] to detect tampering of ASRs.

The third level prevents ASR execution altogether. As mentioned in Section 5.1, the presence of a `ASR-starter` process is essential to ASR invocation. If a user configures his client so that the `ASR-starter` is not run, he is assured that no ASR will ever be executed.

Concerns of security do cause some loss of transparency. Even the least onerous level of security involves some lost opportunities for file resolution because a user needs to have update access, not just read access, on a file to resolve it. Visibility of file conflicts, on the other hand, only requires read access. The second level involves further loss of transparency because it disallows ASRs from untrusted regions of the name space, even though many of them may really be safe. The loss of transparency is, of course, greatest at the third level – in effect, file resolution is avoided altogether.

There are no simple answers to the problem of preserving security while providing transparency in file resolution. Our approach is to allow the user to make the tradeoff, depending on his level of suspicion. Even at the weakest level, however, the user is no worse off than if he were to execute ASR binaries manually.

6.2 Robustness

Misbehaving ASRs can seriously affect the robustness of a client. For example, an ASR whose return code indicates success even though it is unable to resolve the file will cause the Venus worker thread to loop indefinitely: the worker, when resumed, will re-invoke the ASR since the file's replicas are still diverging. Resolvers with programming errors like infinite loops can end up starving user processes of critical resources. Since the number of worker threads is finite, and since a worker is blocked for the duration of an ASR multiple executions of a misbehaving ASR could block all worker threads, resulting in denial of service.

Coda addresses these problems by limiting the execution time of an ASR and the frequency of ASR invocations for an object. In our current implementation, these limits are two minutes and five minutes respectively. The system allows these limits to be changed dynamically for specific objects. Of course, no statically set limit can be perfect: one can always contrive situations where a correct ASR execution is aborted due to one of these limits being exceeded.

6.3 Isolation

To avoid interference between ASRs simultaneously executing on a client, the `ASR-starter` ensures that at most one ASR executes at a time. A worker thread requesting execution of an ASR is blocked if another ASR is already executing on the client. To avoid deadlock, all locks held by the worker are released before it is blocked. The blocked worker is resumed when the current ASR completes.

Executing only one ASR at a time implies that we cannot currently handle cascaded file resolution across volumes. Consequently, an ASR servicing one volume is aborted if it attempts to access a file with divergent replicas in a different volume. Note that, as described in Section 5.3, an access by the ASR to divergent files within the same volume does not abort the ASR – it merely results in an in-place explosion of those files.

Enforcing serial execution of ASRs at a client only offers local isolation. It does not prevent multiple clients from running an ASR for the same file simultaneously. Coda uses an optimistic concurrency control strategy to handle this problem. Each ASR performs its updates in the client's cache assuming no other client is executing an ASR for the same file. When committing the updates made by an ASR, each server verifies that none of the objects in the ASR's write-set have changed since the ASR started executing. If this condition is violated, the ASR is aborted. Therefore, if multiple ASRs simultaneously attempt to resolve a file from different clients, only the first to finish will succeed.

To prevent other processes from accessing partial results of an ASR execution, an exclusive lock is held on behalf of the ASR by Venus. This lock applies to the volume containing the file being resolved, and is held for the entire duration of the ASR's execution. Requests from other processes are blocked until the ASR terminates. Venus uses the process group mechanism of Unix to distinguish between ASR and non-ASR requests – each ASR belongs to a new process group, and all processes created by it belong to this group.

6.4 Atomicity

An ASR execution may abort due to a variety of causes such as client or network failure, coding bugs in the ASR, and exceeding ASR time limits imposed by the `ASR-starter`. Coping with the partial results of an aborted ASR execution can be messy, especially if they lead to further conflicts. To alleviate this problem, Coda ensures that the updates performed by an ASR are made visible atomically. This transactional guarantee can be provided by Venus because it knows the boundaries of an ASR computation (bracketed by `InvokeASR` and `Result_Of_ASR` RPC requests) and because it can distinguish requests made by the ASR using the process group mechanism described above.

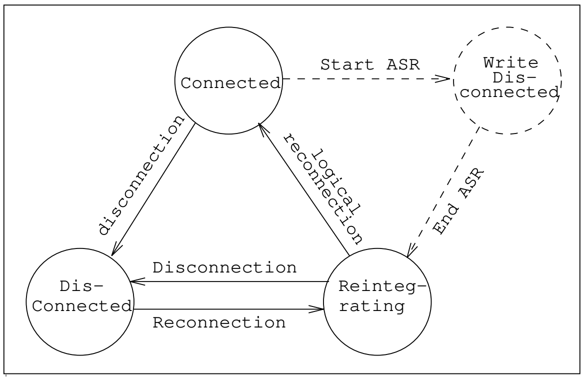
Venus exploits the mechanism already present for disconnected operation to make ASR execution atomic. To implement disconnected operation, Venus logs all mutations made at a client while it is disconnected from a server. This log is used to *reintegrate* the updates with the server when connection is re-established. The updates are committed at the server atomically, using a local transactional mechanism called *RVM* [16, 17].

To make the updates of an ASR atomic, Venus pretends that the ASR is executing while disconnected. Hence its updates are not written through to the server, but are logged. If the ASR aborts, its updates are undone by purging the log and the modified state in the client's cache. But if the ASR completes successfully, its updates are reintegrated atomically.

The support in Venus for disconnected operation had to be modified to allow servicing of cache misses during ASR execution. Originally, Venus operated in one of three states [7]: *connected* (also called the *hoarding* state), *disconnected* (also called the *emulating* state), or *reintegrating*, which is a transient state. We extended Venus to support a fourth state called *write-disconnected.* This state is a hybrid between the connected and disconnected states. Cache misses are transparently serviced, as in the connected state, but updates are only performed locally and logged, as in the disconnected state.[1] The volume containing the file being resolved is forced into the write-disconnected state before an ASR is invoked by Venus. The modified state transition diagram for a volume is shown in figure 5. Write-disconnected state is similar to the *fetch-only* mode proposed by Huston and Honeyman [5] for disconnected operation in AFS clients.

Two operations, *purge-log* and *commit-log* are provided to purge or commit mutations made by an ASR. `Commit-log` is invoked if the ASR computation was successful and the server is accessible: the volume transitions into the

[1] The dual of this state is read-disconnected. Both read- and write-disconnected states are collectively referred to as pseudo-disconnected states.



This figure shows the modified state transition diagram for a volume with the new *write-disconnected* state. The new state and its related transitions are shown in dotted lines/arrows.

Figure 5: Volume State Transitions

---

reintegration state, the ASR's updates are propagated to the server, and then the volume enters the connected state. `Purge-log` is used to flush the changes made by the ASR. Further details on the implementation of the write-disconnected state are provided in [9].

7 Example ASRs

This section shows the use of the resolution rule language with two example ASRs: a resolver for a calendar management program, and a resolver for a file created by the `make` utility. The former ASR merges the contents of the diverging replicas. The latter ASR, on the other hand, does not use the contents of the diverging replicas but reproduces the file's data from its source files.

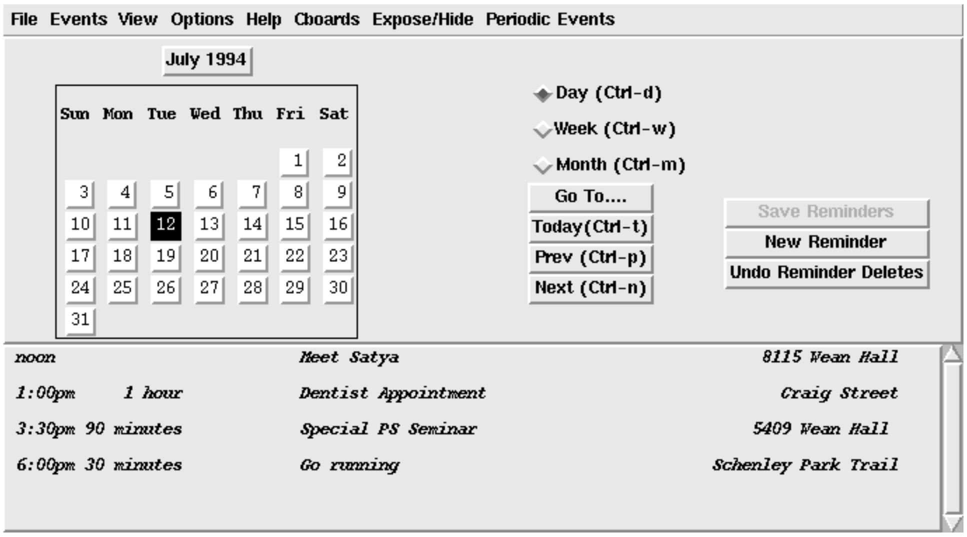
7.1 Calendar Application

The `cboard` application is one of the calendar management programs available in our environment. Using this application, each user can store her appointments in multiple databases. For example, a user may have a `personal` database for private appointments, and an `official` database that she shares with her secretary, for recording business appointments. Furthermore, a `system` database is shared by users to store announcements for public events. The `system` database and a user's `official` database, exhibit write-sharing and are thus prone to concurrent partitioned updates.

Each database is maintained in two files, an *events* file and a *key* file. The former, stored in ASCII format, contains one record per event. The key file, on the other hand, is stored in binary format and contains an index of the events file. The index is used for efficient retrieval of a day's events. These two files are distinguished by special name extensions – a `.cb` extension for the events file and a `.key` extension for the key file. For example, the files storing the `system` database, are named `system.cb` and `system.key`.

The user interface for the calendar, implemented in Tcl/Tk [12], is shown in Figure 6. A user may browse through a database or mutate it in one of three ways: *insert* a new event, *remove* a cancelled event and *update* a changed event. An event is inserted by appending a new record to the events file. To remove an event, its record is invalidated, i.e. logically removed from the calendar, but not deleted physically from the events file. Finally, an event is updated by invalidating its record and inserting a new one with the updated data. In all three cases, the index is changed and both files are written to disk.

Since each mutation modifies the events and key file, a concurrent partitioned update to the calendar causes both these files to have diverging replicas. Therefore, a resolution rule for the calendar application, as shown in Figure 7, contains both files in its object-list. The dependency-list of the rule is empty, since the ASR does not require any other file to have non-diverging replicas. The ASR is executed in three steps. First, the `merge-cal` program merges the diverging replicas of the events file and produces a temporary database. The number of replicas and their names,



This figure shows the Tcl/Tk based interface for browsing events in the calendar. The menu is used to invoke functions like selecting a database, inserting and deleting events etc. The top frame shows the calendar and highlights those days whose events are being viewed in the lower frame. The user may view events for a day, week or month. The lower frame shows a one line summary for each event. Details for an event can be seen by clicking on its summary line. The application uses a dialog-box to remind users about an event and a window-based form to receive user input.

Figure 6: User Interface for the Calendar Manager

as well as the name of the temporary database are provided as arguments to this program. In the example shown in Figure 7, the temporary database is stored in `/tmp/newdb`. In the next two steps, the temporary database is used to atomically update the contents of the diverging files using the `file-resolve` utility. This utility sets the contents of the diverging replicas of a file to a common value. The new contents are provided in a file whose name is supplied as an argument to this utility. If this argument is missing, the replicas are replaced with an empty file.

The `merge-cal` program, performs its task as follows. It builds an index of each replica of the events file. The index includes deleted events, even though their records are invalid, so that the resolver can disambiguate between recent deletes and new insertions. The indices are merged using a straightforward algorithm – a unique copy of every valid record is preserved.

```
*.key, *.cb:
        merge-cal-replicas -n $# -f $</$*.cb[*] -db /tmp/newdb
        file-repair $*.cb /tmp/newdb.cb
        file-repair $*.key /tmp/newdb.key
```

Figure 7: Resolution Rule for the Calendar Application

The original calendar application was implemented more than a decade ago. We chose this as one of our test applications due to its wide-spread use in our community. There are approximately 90 regular users of this application. The user interface, when it was originally implemented, was based on terminal input/output. The more recent implementation based on Tcl/Tk and shown in Figure 6 is being used regularly by ten users. We expect this number to increase as the software becomes more stable.

7.2 Make Application: Reproducing a File's Data

Another kind of ASR, commonly used for a file produced by `make`-like applications, reproduces the file's data from a *source* file. Of course, such ASRs can succeed only if the source file itself is conflict-free.

Common examples of files that could benefit from such ASRs are files with a `.dvi` or `.o` extension. The former are created by Latex and the latter are produced by the C compiler. The resolution rules for these two file types are shown in Figure 8. Note that both rules have a non-empty dependency-list, which is typical for ASRs that regenerate the file's contents. The resolver for `foo.dvi` reproduces its contents by processing `foo.tex` with Latex, provided `foo.tex` itself does not have diverging replicas. Similarly, if `bar.c` has identical replicas, it is processed by the C compiler to generate the contents of `bar.o`.

```
*.dvi:  *.tex
      file-resolve $</$>
      latex $*.tex

*.o:  *.c
      cc -c $*.c -o /tmp/$*..o
      file-resolve $> /tmp/$*..o
```

Figure 8: Make Application's Resolution Rules

Recall, that no mutation except `repair` is allowed for a file with diverging replicas. Since `foo.dvi` is updated by Latex while processing `foo.tex`, its replicas are first truncated using `file-resolve`. Alternatively, the functionality of the `file-resolve` utility could be incorporated into Latex. However, our strategy allows us to use the Latex software off the shelf without any modifications. This intermediate step is not necessary for the latter example, since the C compiler allows its output to be redirected to any named file, e.g. `/tmp/bar..o` in the example above. This file is used to set the replicas of `bar.o` to a common value using `file-resolve`.

8 Performance

The cost of executing an ASR is visible directly to the user because his request is suspended for the duration of the ASR execution. The time overhead occurs during each of the three parts of the ASR mechanism, i.e. invocation, selection and execution of the ASR. While the time overhead of the first two parts is application-independent, the time overhead of the last part depends on the complexity of the ASR's algorithms and the number of updates made by the application. The purpose of this section is to quantify the overhead of only the first two parts, those concerned with providing flexibility and safety for the ASR mechanism.

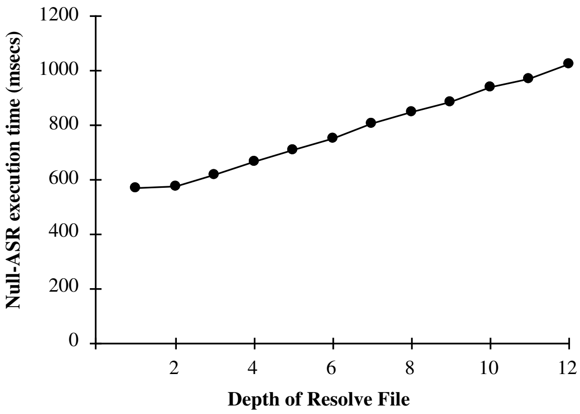
To measure this overhead we conducted a series of experiments. An experiment consisted of triggering an ASR and measuring the elapsed time between Venus requesting the ASR and receiving notification from the `ASR-starter` that the ASR had completed. The ASR was triggered by issuing a `stat` request for a file with diverging replicas. The ASR consisted of running the Unix command `echo` without any arguments. Since we are interested in measuring the overhead only for invoking and selecting an ASR, we could have used an empty command-list for the resolution rule. Instead we used a null-program so that our measurements included the cost of a `fork` system call which would be made even by a minimal ASR.

A measure of the overhead must take into account the fact that the time to find an ASR depends on the location of the `ResolveFile` with respect to the file being resolved. A longer distance between these files lengthens the time needed by the `ASR-starter` to find the ASR. To study the effect of changing the *depth* of the `ResolveFile`, i.e. the distance between the `ResolveFile` and the file being resolved, the above experiment was conducted with depth levels from 1 to 12. A depth-level of n implies the ASR-starter had to lookup n ancestral directories to find the `ResolveFile`.

Resolution rule used for the experiments:

```
*:
        /bin/echo -n
```

Depth of ResolveFile	ASR Exec. Time (milliseconds)
1	571.5 (4.9)
2	576.6 (4.5)
3	620.8 (5.2)
4	665.0 (3.2)
5	708.5 (5.2)
6	754.1 (13.9)
7	806.5 (10.9)
8	847.1 (4.3)
9	888.9 (4.0)
10	940.7 (16.7)
11	970.2 (5.3)
12	1026.4 (6.8)



The table shows the elapsed time for executing the null-ASR. The graph shows the increase in overhead with increasing depth of the `ResolveFile`. Each additional lookup in an ancestral directory takes 44.8 milliseconds. The file being resolved was replicated at two servers. The experiments were performed on a DECstation 5000/200 with 32 Megabytes of memory. The time values in milliseconds are the mean value from nine trials of each experiment. Figures in parentheses are standard deviations.

Figure 9: Execution Time for a Null-ASR

The experiments were conducted on a DECstation 5000/200 with 32 Megabytes of memory running version 2.6 of the Mach kernel. In each experiment the latency of the ASR execution was measured using a microsecond timer.

8.1 Results

The results of our experiments, shown in Figure 9, confirm that the framework for invoking and selecting an ASR has a small overhead. In most cases, it takes between one-half and one second for the ASR to be invoked and its results returned to Venus.

The minimum overhead of 571.5 milliseconds occurs when the `ResolveFile` and the file being resolved are in the same directory. Detailed measurements of this experiment configuration show that the RPC request from Venus to the `ASR-starter` costs 12.5 milliseconds. The `fork` request that starts the `executor` causes a delay of 50 milliseconds. The `executor` takes 496 milliseconds to perform its work – 180 milliseconds to parse the `ResolveFile` and 316 milliseconds to lookup the parent directory for the `ResolveFile`, lock the volume and `fork` the `echo` program. Finally, the RPC from the `ASR-starter` to Venus costs 12.5 milliseconds.

Although the cost of invoking and selecting an ASR may seem high, it is usually lower than the time needed to execute the resolver. Further, the flexibility of the ASR mechanism combined with the major improvement in usability resulting from ASRs being invoked automatically, rather than manually, renders this cost entirely acceptable.

9 Related Work

The work reported in this paper only represents one part of the overall support for conflict resolution in Coda. Resolution of *directories* is performed using a server-centric, log-based scheme [10]. The radically different approaches to file and directory resolution arise due to their very different characteristics. Directories are objects whose semantics are entirely known to the system; files, on the other hand, are treated as untyped byte streams. Directories are more critical to availability, because a directory conflict denies access to an entire subtree while a file conflict only denies access to a single object. Finally, to safeguard against structural damage caused by directory corruption, Coda servers never accept entire directory contents from clients as they do file contents; rather, directory updates are performed individually on servers.

These differences lead to Coda's use of very different resolution strategies for directories and files. Directory resolution is performed entirely by servers, although it is triggered by clients. The code for performing directory resolution is part of the trusted system software on a server, in contrast to ASRs which are untrusted. A similarity between directory and file resolution is that both function lazily, and resolve on demand rather than resolving *en masse*. As mentioned earlier, this approach minimizes the likelihood of recovery storms.

The Ficus File System also uses optimistic replication and provides facilities for automating the resolution of file conflicts [14]. Like Coda, Ficus uses a rule-based approach for selecting a resolver. But it is less flexible along many dimensions. For example, the parameter list to a resolver is assumed to have a fixed format; a Ficus user can have only one personal resolver list; it is not possible to specify that a group of programs are to be executed as one logical resolver, nor can a group of files be resolved together. There are also important differences in the execution models of resolvers in Ficus and Coda. Since Ficus uses a peer-to-peer rather than a client-server model, it is more liberal in its choice of execution site – any site with a replica of a file can run a resolver for it. If one resolver fails, others are tried in succession.

The Ficus design pays less attention to issues of security and robustness. Ficus resolvers are run on behalf of the owner of the file and not the user accessing it. Therefore, a malicious user could cause serious damage by using a misbehaved resolver on a file owned by another user. There are no specific mechanisms in Ficus to provide atomicity or isolation. Coda, in contrast, takes these issues much more seriously and provides specific mechanisms to improve safety.

The ASR mechanism in Coda is loosely analogous to a *watchdog* as proposed by Bershad and Pinkerton [1]: it extends the semantics of the file system for specific files. Of course, since the watchdog mechanism is not specifically intended for conflict resolution, it does not incorporate many important mechanisms needed by us. For example, it does not use a rule-based approach for selection of watchdogs, provide a mechanism for exposing file replicas, or pay particular attention to the issues of security, robustness, isolation and atomicity.

10 Conclusion

The importance of optimistic replication as a technique for providing high availability in distributed systems has been known for over two decades. But the use of this technique in actual systems has been minimal. One reason for this has been the fear of designers that conflicts, an inevitable consequence of optimistic replication, might hurt usability unacceptably. A second reason has been concern that the machinery needed to cope with conflicts might be excessively complex and unwieldy.

Our work puts both these fears to rest in the context of distributed personal computing environments. We have shown how one can build a practical system that provides support for resolving file conflicts. A key aspect of our approach is that the semantic knowledge needed for resolution is provided by application writers rather than being wired into the system. The challenge with this approach is to ensure that security, robustness and other safety properties are not compromised intolerably. This is indeed possible, as we have shown in this paper.

References

[1] Bershad, B., and Pinkerton, C. Watchdogs - Extending the UNIX File System. *Computing Systems 1*, 2 (Spring 1988).

[2] Eldred, E., and Sylvester, T. A Groupware Duet with Gusto. *Client/Server Today 1*, 3 (July 1994).

[3] Guy, R., Heidemann, J., Mak, W., Jr., P., T.W., Popek, G., and Rothmeier, D. Implementation of the Ficus replicated file system. In *USENIX Summer Conference Proceedings* (Anaheim, CA, June 1990).

[4] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[5] Huston, L., and Honeyman, P. Disconnected Operation for AFS. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing* (Cambridge, MA, August 1993).

[6] INTUIT. *User's Guide: Your Day to Day Reference Guide to Quicken*, September 1992.

[7] Kistler, J., and Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[8] Kumar, P. Coping with Conflicts in an Optimistically Replicated File System. In *Proceedings of the IEEE Workshop on Management of Replicated Data* (Houston, TX, November 1990).

[9] Kumar, P. *Mitigating the Effects of Optimistic Replication in a Distributed File System*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1994.

[10] Kumar, P., and Satyanarayanan, M. Log-based Directory Resolution in the Coda File System. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems* (San Diego, CA, January 1993).

[11] Kumar, P., and Satyanarayanan, M. Supporting Application-Specific Resolution in an Optimistically Replicated File System. In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems* (Napa, CA, October 1993).

[12] Ousterhout, J. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[13] Parkinson, K. Remote Users Get in Sync with Office Files. *Macweek 8*, 28 (July 1994).

[14] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. Resolving File Conflicts in the Ficus File System. In *USENIX Summer Conference Proceedings* (Boston, MA, June 1994).

[15] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers 39*, 4 (April 1990).

[16] Satyanarayanan, M., Mashburn, H., Kumar, P., Steere, D., and Kistler, J. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems 12*, 1 (February 1994), 33–57.

[17] Satyanarayanan, M., Mashburn, H., Kumar, P., Steere, D., and Kistler, J. Corrigendum: Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems 12*, 2 (May 1994), 165–172.

[18] Tygar, J., and Yee, B. Strongbox: A System for Self Securing Programs. In *CMU Computer Science: 25th Anniversary Commemorative*. Addison-Wesley, 1991.

Author Information

Puneet Kumar received a PhD in Computer Science from Carnegie Mellon University in 1994, after a B.S. degree in Computer Science from Cornell University. His PhD thesis was concerned with automating resolution in optimistically replicated distributed file systems. He is one of the original implementors of the Coda File System.

Mahadev Satyanarayanan is an Associate Professor of Computer Science at Carnegie Mellon University. He is currently investigating the connectivity and resource constraints of mobile computing in the context of the Coda File System. Prior to his work on Coda, he was a principal architect and implementor of the Andrew File System. Satyanarayanan received the PhD in Computer Science from Carnegie Mellon University in 1983, after a Bachelor's degree in Electrical Engineering and a Master's degree in Computer Science from the Indian Institute of Technology, Madras. He is a member of the ACM, IEEE, Sigma Xi, and Usenix, and has been a consultant to industry and government.