Supporting Application-Specific Resolution in an Optimistically Replicated File System

Puneet Kumar & M. Satyanarayanan
School of Computer Science
Carnegie Mellon University

In this position paper we describe an interface to incorporate application-specific knowledge for conflict resolution in an optimistically replicated file system. Conflicts arise in such systems because replicas of an object can be modified simultaneously in different network partitions. Application-specific knowledge is made available by the application writer in specialized tools called *Application-Specific Resolvers* (or ASRs). The interface we describe here is used to bind ASRs to objects and to specify the conditions under which a specific ASR is invoked. This allows the system to make conflict resolution transparent to the user, thus improving the usability of optimistic replication.

1. Motivation

The viability of optimistic replication strategies in providing high availability has been demonstrated by systems like Coda [5] and Ficus[2]. Optimistic replication is acceptable for Unix file systems because write-sharing between different users is low and consequently conflicting updates are rare. But when conflicts do occur, manually resolving them can be onerous.

In many important cases, an application itself may have enough knowledge to resolve the conflicts. Consider an appointment calendar maintained in a file with two replicas. A user makes appointments and updates one replica during a network partition while a secretary adds appointments to the other replica. If the appointments do not overlap, a utility program could merge the two replicas even though a conflict exists at the file granularity. Further, if this utility is invoked automatically by the system when the partition heals, conflict resolution can be made transparent to the user.

Electronic project diaries and cheque-books shared by users in the same organization are other examples of data that provide an increased opportunity for write-sharing. Fortunately, the applications that manipulate such data are typically unique and capable of being extended to perform application-specific resolution.

2. Coda File System

This work is being done in the context of the Coda file system. Coda clients are untrusted and provide a shared, location-transparent Unix file system stored on a much smaller number of trusted servers. The name space is mapped to individual servers at the granularity of subtrees called *volumes*. Coda provides high availability using two mechanisms: *server replication* and *disconnected operation*. The first mechanism increases the availability of data during partial network or server failure. If

all servers become inaccessible, the second mechanism allows *Venus*, the client cache manager, to service file requests using cached data. Details of server replication and disconnected operation are provided in [5] and [3, 4] respectively.

3. Overview of ASR mechanism

On every cache miss Venus verifies that all replicas of the object being fetched are identical. If the replicas diverge, Venus automatically forks an appropriate ASR to resolve them. The application for which the object is being fetched blocks until the ASR has completed. If no ASR can be found for the object or the conflict is unresolvable, the file's replicas are marked as inconsistent with a special flag. This makes the file inaccessible to applications until it is manually repaired by the user. The ASR mechanism is loosely analogous to a *watchdog* as proposed by Bershad and Pinkerton[1] in that it extends the semantics of the file system for specific files to be resolved transparently when needed.

In the following sections we will discuss our design from the viewpoint of the following questions:

- Where does an ASR run?
- How is an ASR bound to a file system object?
- How is replication exposed to an ASR ?
- How is the execution of an ASR made fault-tolerant?
- How do we cope with multiple machine types?

4. Where does an ASR run?

ASRs are executed at the client rather than the server for the following four reasons. First, our security model restricts us from running arbitrary application code at the server. Second, for scalability we push as much work to the client as possible. Third, the ASR needs to collect and examine all replicas of the file being resolved, the machinery for which is already present at the client. Fourth, servers do not currently have the ability to perform Coda name resolution.

Since clients are untrusted, the ASR must assume the rights of the user to make any modifications. If the user has insufficient priviliges to modify the file being resolved, the resolution will fail. In this case, we are willing to sacrifice some availability to avoid compromising security.

Running arbitrary code on a user's behalf could of course violate client security but the scope of the damage is confined to the user's domain. If desired, users can disable execution of all ASRs or allow execution of ASRs only from certain protected directories containing trusted programs.

5. Scope of an ASR

The bindings between files and the ASR needed to resolve them are specified as rules in a file named `RESOLVE`. The resolution rules in a `RESOLVE` file apply to objects in its directory and in

all descendant directories unless they contain overriding RESOLVE files. The RESOLVE entry in a directory can be a symbolic link so that users can share standard RESOLVE files provided by the application writer.

The rules in the RESOLVE file are based on a make-like language. Each rule has the following format:

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

The object-list is a non-empty set of object names for which this resolution rule applies. Object-names can contain C-shell wildcards like "*" and "?". The dependency-list can be empty or contain a list of objects whose replicas must already be equal when this rule is utilized. For example, recompiling foo.c to resolve a conflict on foo.o is reasonable only if foo.c does not have diverging replicas. To simplify the implementation resolution is not invoked recursively for objects in the dependency-list. The command-list consists of one command per line. Each command specifies a program to run along with its arguments. The language provides macros for specifying the object that matched a pattern and the directory component of the object's absolute pathname. These macros get expanded dynamically as the rule is utilized to resolve a file.

To determine the ASR for a file foo, the RESOLVE file is searched for the first rule in which foo appears in the object-list. The command-list of this rule constitutes the ASR for foo. If no RESOLVE file can be found or no rule exists for foo, a resolution failure is assumed. The following rules are some examples used in our system today:

```
!The following is a rule for recreating a .o file from the .c file.
!The rule will be used only if the .c file doesn't have diverging
!replicas.  First a ..o is created and then used to write over the
!replicas of the .o file using the fileresolve program
*.o: $*.c
     cc -c -o $*..o $*.c
     /usr/coda/etc/fileresolve $*.o $*..o
     rm $*..o

!The following specifies the ASR for a calendar program that keeps
!a foo.cb and a foo.key file for the calendar named foo.
!The merge program is the ASR that produces a new calendar from the
!diverging replicas in /tmp/newdb.  This calendar's files are
!then used to set the new contents of the diverging replicas.
!$# and [*] are special macros whose function is discussed in
!the next section.
*.key, *.cb:
     /usr/coda/etc/merge -n $# -f $</$*.cb[*] -db /tmp/newdb
     /usr/coda/etc/fileresolve $*.cb /tmp/newdb.cb
     /usr/coda/etc/fileresolve $*.key /tmp/newdb.key
```

6. Exposing replicated names

In normal operation, Venus makes replication transparent to applications. However, the ASR needs to examine the individual replicas of the file being resolved. Just before invoking an ASR Venus exposes the replicas of the file by expanding it in place as a *fake directory* with each replica of the file as its child. The children are named using the low-level identifiers of the replicas. Objects whose replicas are equal appear non-replicated to the ASR.

Fake directories are read-only objects that exist only at the client where the ASR is executing. The only mutating operation allowed on the fake directory is `repair` which atomically sets the contents of the diverging replicas to a common value. Once this operation succeeds, Venus collapses the fake directory back into a file.

Since replica names are not known *a priori*, the ASR specification language provides means to put place holders for these names in the rules. `$#` specifies the replication factor, `[*]` is used for naming all the replicas at once and `[`i`]` specifies the names of the individual replicas. For a file `foo` with three replicas, the rule

```
*:
    /usr/coda/bin/fileresolve $> $# $>[*]
```

results in
"`/usr/coda/bin/fileresolve foo 3 foo/cc0020 foo/cd0020 foo/ce0020`" being invoked as the ASR . Here `cc0020`, `cd0020` and `ce0020` are the identifiers of the individual replicas of `foo`.

7. Fault-tolerance

For purposes of fault-tolerance an ASR's mutations are *isolated* during its execution and committed *atomically* if and when it completes successfully.

To achieve isolation, Venus locks the object's volume exclusively for the ASR assuming it will only modify objects in that volume. The volume lock has a timeout interval to guard against errant ASRs. Non-ASR processes needing read or write access in the same volume are forced to wait until the ASR completes. Requests from the ASR and its child processes are distinguished by setting their process group identifier to a reserved value.

For atomicity, the ASR's updates are propagated to the server in a single transaction using the *reintegration* mechanism of disconnected operation[3]. When an ASR is forked, the object's volume is placed in a *writeback disconnected* state. In this state, the server is still available to service cache misses but mutations in the volume are reintegrated with the server in a single transaction only when the ASR completes.

8. Heterogeneity

In a distributed system environment with heterogeneous hosts we need the ability to choose an ASR binary depending on the client's architecture. For this we borrow the `@sys` pathname expansion capability of the Coda kernel. The Coda kernel evaluates the `@sys` component, if present in a path-

name, to a unique value on each architecture. Therefore, the same pathname for an ASR containing `@sys` as one of the components evaluates to a different object on each machine architecture type.

9. Conclusion

We have completed implementation of the parser for the ASR specification language and modified Venus to automatically find and invoke an ASR when needed. Venus is also able to expose the replicas before invoking the ASR. The fault-tolerance mechanism for ASRs is being designed and will be implemented next. To test the completeness of the ASR specification language, we have implemented resolvers for an application calendar program and an interactive X-based resolver.

In conclusion, our framework for specifying ASRs is general enough to be utilized by a wide variety of applications. It provides a simple and extensible interface which can easily incorporate new ASRs and make the resolution process transparent to most application users. Our methodology is general enough to be utilized not only for write/write conflicts but also for read/write conflicts which will be detectable in future versions of Coda.

10. Acknowledgements

This work was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465 and ARPA Order No. 7597. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies.

References

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

[2] Guy, R., Heidemann, J., Mak, W., Page, T., Popek, G., and Rothmeier, D. Implementation of the Ficus Replicated File System. In *Usenix Conference Proceedings* (June 1990).

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

[4] Kistler, J. J. *Disconnected Operation in a Distributed File System*. PhD thesis, School of Computer Scient, Carnegie Mellon University, Pittsburgh, 1993.

[5] 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).