Rufis: mobile data sharing using a generic constraint-oriented reconciler

Marc Shapiro
Microsoft Research Ltd., Cambridge, UK

Nuno Preguiça
Universidade Nova de Lisboa, Portugal

James O'Brien
Microsoft Research Ltd., Cambridge, UK

Abstract

Existing systems for disconnected data access and reconciliation are monolithic, complex and somewhat ad-hoc. In contrast, we demonstrate here a principled approach based on a general-purpose reconciliation engine. We describe the Reconcilable and Undoable File System, Rufis, implemented on top of the IceCube reconciler. IceCube is generic but supports application-specific reconciliation invariants. Consequently, the code for Rufis is quite small and simple, and the reconciliation logic is well separated from the main file system code. Furthermore, Rufis supports specialised reconciliation for files containing data of known types and enables ad-hoc user scenarios involving multiple applications.

1. Introduction

The Reconcilable and Undoable File System (Rufis) is a distributed file system supporting disconnected operation, reconciliation and selective undo. Rufis is built upon the application-agnostic reconciliation engine IceCube.

Rufis lets users replicate a shared file system on multiple machines. A user may tentatively update a replica while disconnected from the network. Upon reconnect, IceCube reconciles the tentative updates (called *actions*). The user is ultimately in control of which updates get committed.

Rufis and applications running on top of it declare to IceCube the invariants relating actions to one another, called *constraints*. Constraints enables powerful, well-behaved reconciliation and selective undo. Actions can be undone in any consistent order and by any user.

IceCube cleanly separates the application logic from reconciliation and removes the part of the burden of reconciliation from the application developer. We report here on how we first developed a non-replicated file system and extended it with ease to make Rufis, simply by declaring action semantics to IceCube.

Section 2, next, presents some example usage scenarios. Then we present the IceCube structures, APIs and algorithm in Section 3. Section 4 goes into the design and implementation rationale of Rufis. We study related work in Section 5. Finally, Section 6 concludes with lessons learned and future work.

2. Example scenarios

Consider the file system depicted in Figure 1. Initially Anne's and Brian's replicas both contain directories `/`, `/a` and `/b` and files `/b/c` and `/b/d`. While disconnected, user Anne creates a directory and moves file `/b/c` into `/a`. Concurrently user Brian creates directory `/a/c` and writes a string into `/a/d`.

In Rufis, Brian writing a file does not conflict with Anne renaming the same file. However, the name of the directory Brian created clashes with the file Anne moved. IceCube proposes schedules corresponding to the two non-conflicting outcomes depicted on the right-hand side of the figure. From the control interface, Brian experiments with undoing his directory creation (this does not undo his unrelated file write), redoing it, and finally committing a schedule that combines his and Anne's directory creations, thus aborting Anne's conflicting file move.

Now suppose that, after moving `/b/c` into `/a`, Anne had deleted directory `/b` and its contents `/b/d`. By default, Rufis considers that the delete depends on the previous move. (Anne could override the default if that was not her intent.) In order to satisfy the dependencies, when the move is aborted, the delete that depends on it is aborted is as well, and file `/b/d` will not be lost by mistake.

Brian could decide to group two commands into an an indivisible *parcel*, for instance: "`parcel {mkdir /a/c; write "I won!" /b/d}`." In this way, if Brian does not have permission to modify the file, the directory is not created; and vice-versa, when Brian undoes the directory creation, the write is also undone.

Proceedings of the 2004 IEEE International Conference on Mobile Data Management (MDM'04)
0-7695-2070-7/04 $20.00 © 2004 **IEEE**

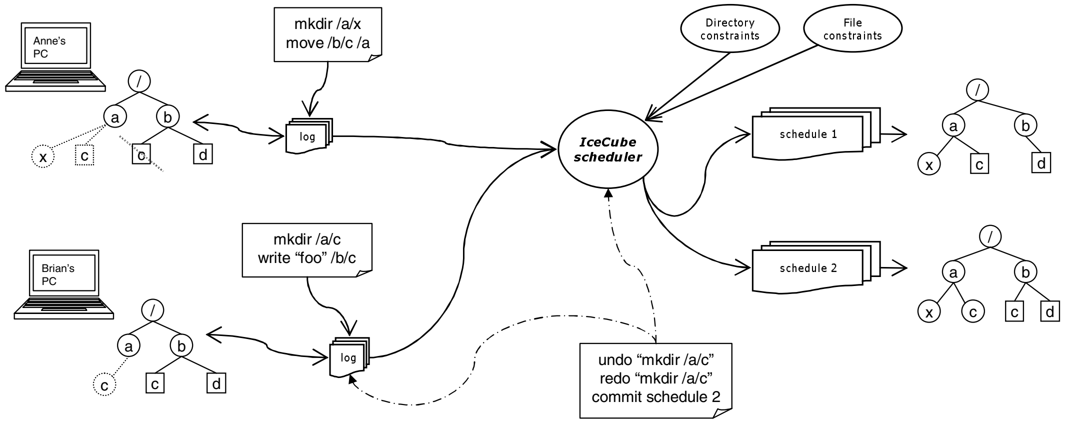

Figure 1. IceCube system structure and scenario (see text)

IceCube enforces application invariants, and can furthermore work across applications. Consider for instance a calendar implemented upon IceCube. Anne can request an appointment with Brian, proposing two possible times, say 10:00 and 11:00, using the *Alternatives* constraint. In each alternative, a parcel groups the appointment request with a write that stores a string into file `/a/c`, `"ten"` or `"eleven"` respectively. One of the two appointment requests (but not both) may succeed, depending on when Brian is free; then the file contains the appropriate time. If Brian is busy at both times, neither succeeds, nor does the write, and Anne is notified. If Brian commits Schedule 2 as above, then file `/a/c` doesn't exist, both appointement requests abort, and Anne is notified.

3. IceCube

IceCube fits into a distributed system supporting disconnected operation. IceCube combines actions from the different users to form a common, sequential *schedule*, to be replayed at every site. However, combining concurrent actions into a correct schedule is not trivial, as there may be conflicts.

To automate this process and to provide application independence, IceCube uses semantic information called *constraints*. The reconciler proposes and executes any number of schedules that obey the constraints. The user has ultimate control over the outcome. After viewing the results of different schedules he may *commit* one of them. Otherwise, he has several options, such as undoing some actions, committing only a subset of a schedule, asking for more schedules, or editing the logs and trying again.

3.1. Actions and constraints

An action represents an application-specific operation and it is defined by the application programmer.

Constraints are IceCube's central abstraction for conflict detection and scheduling. A constraint is the formal, "reified" representation of an application invariant. It is the responsibility of the IceCube system to maintain invariants despite concurrency.

A *static* constraint relates two actions unconditionally. As an example, Alice's and Brian's concurrent actions to create a file `/a/c` and a directory `/a/c` mutually exclude each other statically, because file-system invariants forbid two different objects to exist concurrently with the same name.

IceCube also supports dynamic constraints that test the success or failure of a single action, depending on the current state. A simple example is a bank account application that would use them a dynamic constraint to verify that a debit never violates the "no overdraft" constraint.

3.1.1. Log constraints A *log constraint* is a static constraint between actions of the same log. Users and applications use log constraints to make their intents explicit. Constraint `predecessorSuccessor`(α, β) establishes that action β executes only after α has succeeded (causal ordering). For instance, say a user tentatively updates a file, then copies the new version; to maintain this dependence upon reconciliation, the application records a `predecessorSuccessor` constraint between the write and the copy.

The `parcel` log-constraint is an atomic (all-or-nothing) grouping. Either all of its actions execute successfully, or none does.[1] For instance a user might copy two whole directory trees inside a third directory as a parcel. If any of the individual copies would fail (e.g., for lack of space, or because the user doesn't have the necessary access rights), then no copying at all occurs in the reconciled schedule.

The `alternative` log constraint provides choice of at most one action in a set. An example is submitting an appointment request to a calendar application, when the meeting can take place at (say) either 10:00 or 11:00. Users use `alternative` constraints to provide the scheduler with a fallback in case of a conflict. Rufis does not use `alternative` directly, but a number of applications running on top of Rufis do, for instance the calendar application.

3.1.2. Object constraints An *object constraint* is a static constraint between concurrent actions. Before reconciling, IceCube collects the object constraints for every pair of concurrent actions in its logs, by calling into application code. To this effect, application action pairs implement the following methods: `mutuallyExclusive` should return true if both actions cannot be in the same schedule; and `bestOrder` should return the preferred execution order of the action pair, if any exists.

Object constraints express (static) concurrency invariants, similarly to Schwartz [9] or Weihl [11]. For instance, creating a file and creating a directory with the same name is `mutuallyExclusive`. Another example: a bank account application indicates a `bestOrder` preference to schedule credits before debits.

3.1.3. Explicit commutativity We have found it useful to further differentiate pairs of commuting actions that have no mutual side effects. This is important in particular when an action's dynamic constraint is violated: actions that commute with it do not need to be rolled back. The following methods provide this information.

`GetDomain` returns any number of *domains*, where a domain is an opaque hash characterising a set of objects read or written by an action. Actions with no common domain are commutative. In Rufis, the domain of an action is the identifier of the file system that it operates upon.

If two actions have a common domain, IceCube calls their `overlap` method to tests whether they overlap; if not, they are commutative. In Rufis, two directory actions of the same file system do overlap only if they concern the same name inside the same directory; two file actions overlap only if they touch the same file.

When two actions overlap, method `commute` tests whether they commute semantically. For instance, two reads to the same file overlap but commute.

3.2. Reconciliation algorithm

We now describe very briefly the operation of IceCube for reconciliation and undo. As the scheduling problem is NP-hard, IceCube explores the space of possible schedules heuristically.

It first decomposes the inputs into independent sub-problems, such that the actions in one sub-problem commute with all actions in other sub-problems, and there are no static constraints connecting actions from different sub-problems. It follows that actions from different sub-problems may be scheduled in arbitrary order, and executing or rolling back an action belonging to some sub-problem does not affect actions of another sub-problem. In the common case, the combined complexity of the sub-problems is much lower than that of the original problem.

Then the scheduler performs an efficient, heuristic sampling of small portions of the search space for each sub-problem. If the user requests a new schedule, or the computation hits a dynamic constraint violation, the search restarts over an unrelated portion of the search space.

3.3. Undo

The user may request to undo an action. This also undoes all dependent actions. Undo is a particular application of the reconciliation algorithm. To undo some action α, the system adds an `alternative` constraint between a special "always-committed" action and α. This has the effect of rolling back execution to a state where α is not executed. Any actions that depend on α are excluded from the execution, as they now conflict with the always-committed action. Actions that do not depend on α are not undone, even if they were executed after α. This includes any action β that is concurrent with α or commutes with α, unless an explicit `predecessorSuccessor` or `parcel` connects α and β. Actions that were excluded by α may now be included (at the user's option) in the new schedule.

Redo-ing α is simply a matter of removing the exclusion between α and the always-committed action. The effect of undo-redo is not necessarily equivalent to doing nothing, as new conflicts may have appeared in the meantime.

[1] Unlike a traditional transaction, a `parcel` does not ensure isolation. In the absence of other constraints, its actions may run in any order, possibly interleaved with other `parcel`s.

4. Inside Rufis

Rufis emulates the usual Unix file system semantics. Operations include creating, deleting and renaming files and directories.

4.1. File system concurrency semantics

By default, a file is just an untyped byte string and its concurrency semantics are standard: conservatively, concurrent writes constitute a conflict. However a file can have its own invariants and weaker concurrency control. For instance, for a calendar file, two updates do not conflict unless they cause a double booking or other violation of calendar invariants.

Concurrent directory operations conflict only if combining them would would break the directory hierarchy, or would cause a user to lose work. For instance, one user editing a file while another deletes it, constitutes a conflict, whereas concurrently creating two files in the same directory does not conflict, as long as their names are different. More subtly, one user is allowed to edit a file while another renames the file or a parent directory.

Beyond the inherent file system invariants, applications or users can insert their own log constraints. For instance, grouping two copies and a delete into a parcel ensures that either all three operations can succeed, or none is executed.

Currently, commitment is centralised in IceCube. In this respect, Rufis follows the example of CVS, Bayou, or Microsoft Briefcase.

4.2. Object identification

We envisaged two design approaches to identify file and directory objects. In the first, each action in the log records its pathname arguments. This naturally captures standard file system invariants, but does not cope well with concurrent renaming. For instance if some user creates file `/a/b` while an other renames `/a` to `/c`, and the rename is scheduled first, then the new file should be created as `/c/b`. The chosen approach is to designate nodes by a unique identifier called RufisKey, independent of their pathname, much like Unix inode numbers. An action that operates on some file or subdirectory within a directory records the directory's RufisKey and the relative pathname of the object.

4.3. General structure

Internally, a file system is a tree of `DirectoryNodes` and `FileNodes`, and a hash table of the same nodes, indexed by RufisKey.

Directory	link/link	link/unlink	unlink/unlink
Different parent, name	`¬overlap`	`¬overlap`	`¬overlap`
Same parent&name	`Mut.Excl.`	`bestOrder`	`commute`

Dynamic constraint: maintain tree; linked name doesn't exist

File	Read/Read	Read/Write	Write/Write
Other file	`¬overlap`	`¬overlap`	`¬overlap`
Same file	`commute`	`bestOrder`	`Mut.Excl.`

Dynamic constraint: none

Directory/File	Unlink/Write	other
Other file	`¬overlap`	`¬overlap`
Same file	`Mut.Excl.`	`¬overlap`

Table 1. Rufis object constraints

We decompose a user-level directory command into a parcel of several actions. The first checks arguments and establishes what needs to be done at a high level. The following ones link and unlink nodes in the tree. The last one might check the result. For instance the `move` command (which renames a file or directory) is decomposed into an action that checks which of nine possible cases to execute, followed by up to three link and unlink actions, followed by a a check that the tree structure is maintained (explained in Section 4.5).

In a link or unlink action, the parent directory is identified by RufisKey and the file or directory being linked by its string name. Using a RufisKey to identify a parent directory has the advantage that the key does not change as the directory is renamed. When a node is created, its RufisKey is logged so that at reconciliation time, it is is re-created with the same RufisKey.

The chances of successful reconciliation are improved by using `bestOrder` to move reads and unlinks to the beginning of the schedule, and writes and unlinks at the end.

4.4. Rufis actions

The low-level Rufis action types are the following. A directory action either creates a directory or links or unlinks a node (either a file or another directory) into a directory. A file action either creates a file or reads or writes it.

The object constraints and explicit commutativity methods are summarised in Table 1. In more detail, those on `DirectoryNode`s are as follows: Each file system constitutes a domain. Actions whose parent directory have the same RufisKey overlap. Also (special case mentioned above), writing a file overlaps with unlinking it from its parent directory. Otherwise, a `DirectoryNode` action does not overlap any other type of action. Actions that overlap commute if neither of them is a write, or if the object names differ. Overlapping actions that do not commute conflict if they are both writes. Writing a file conflicts with unlinking its directory (the special case again). Finally, we order unlink actions before links to reduce the chance of a dynamic constraint being violated.[2]

Similarly, for `File` actions: File actions in the same file system overlap if the files have equal RufisKey. A write action overlaps with a unlinking the node from its parent directory (the special case). Two overlapping file actions commute if either is a creation action, or if both are read actions. A write conflicts with unlinking the file (special case), and two non-commuting writes conflict with each other. Finally, reads are preferably scheduled before writes that do not commute with them.

4.5. Checking structural integrity

Rufis makes little use of dynamic constraints, except in the following case. At the end of a directory move, a final action checks that the file system structure remains a tree. This is necessary because a move parcel, unlike a transaction, is not isolated from concurrent moves. For example, consider a file system containing directories `/a/b/c/` and `/a/d/e/`. Two users could concurrently request to move `d/` under `c/` and `b/` under `e/`. The result would be a cyclic structure `b/c/d/e/b/...` unattached to `/a/`. The dynamic check will disallow this from happening and cause at least one of the moves to abort.

4.6. Implementation

The Rufis implementation is roughly 3,000 lines of Java. We first built a "solo" system that implements the centralised file system functionality in 1,700 lines. Converting this to optimistic replication was relatively simple: we added a layer to intercept and store successful solo commands in the IceCube log, and we exported appropraite log and object constraints. This constitutes the remaining 1,300 lines.

We choose to log by recording a trace of the actual side-effects, as opposed to "diff-ing" snapshots before and after tentative execution. Diffing is less successful at capturing user intents. For instance it does not easily differentiate renaming a file from two independent delete and create actions.

The logging code is essentially a modified copy of the solo code. Consider some representative command, for instance `mkdir` (create new directory). The solo `mkdir` first checks its arguments, then creates a new directory object, then links the new object into the parent directory. The replicated `mkdir` does the same, also remembering the RufisKey of the parent directory, the RufisKey of the new subdirectory, and the name of the subdirectory. If tentative (solo) linking was successful, it logs (1) an action to create a directory node with the same RufisKey and (2) an action to link the new node under the same name, into the parent directory identified by its own RufisKey. The two actions compose a parcel, and are `predecessorSuccessor` of one another.

By default, the replication code records a Predecessor-Successor log constraint between any action and the last action that modified the same object(s). Undo and reconciliation remain selective, because later unrelated updates are not constrained. However, this is still excessively conservative, so Rufis also provides an interface where the application provides log constraints explicitly.

4.7. RufisKey

All replicas of the same object are persistently identified by a common RufisKey, despite users concurrently mutating the file system tree. For instance consider a user creating a file while another changes the name of the parent directory. File creation will succeed despite the concurrent change, since the `mkdir` actions refer to the parent by RufisKey.

A RufisKey also serve a different purpose. The Rufis application has access to several (virtual) copies of the file system. For instance, at some point in time might coexist the last committed state, the current tentative version being modified by the user, and one or more reconciled states being proposed by IceCube. What is logically a single logical file, with different versions, becomes at runtime unrelated Java objects, one for each file system version. The RufisKey serves to identify the single logical file object relative to the different versions. This tends to confuse application programmers, but could undoubtibly be alleviated by specific compiler support.

5. Comparison to related work

Optimistic replication has been widely researched; we refer to Saito and Shapiro [7] for a comprehensive study. Relevant work includes the file systems Coda [3] and Ficus [6]; the Concurrent Versions System CVS [2]; formal studies of file replication semantics by Balasubramaniam and Pierce [1] and by Ramsey and Csirmaz [5]. In all these systems, there can be no dependencies between files or objects, and application-specific invariants are either ignored or difficult to describe.

[2] Despite the strong static constraints, it does occur in some corner cases that a dynamic constraint is violated.

The most similar system is Bayou [10], a general-purpose replicated database system. Specialised applications, such as a calendar or a mail folder manager, can run on top of Bayou, each with their own invariants. An action consists of a conflict test, an update procedure, and a "second-chance" merge procedure in case of conflict. In contrast to IceCube however, the semantics are opaque to the system, and schedules run strictly in timestamp order.

The novel contribution of IceCube is that invariants have a formal representation (as constraints) and that maintaining invariants is delegated to the system. Furthermore, in Rufis (in contrast to most file systems), writes do not necessarily conflict, and files are not necessarily independent. Support for application invariants is a native feature, and IceCube will reconcile seamlessly data from very different application domains.

Constraints are used widely in many application areas [8], but we believe IceCube is the first to use them for replication.

Presenting alternative reconciliation schedules and their outcomes to the user appears to be a unique feature of our system. Similarly, we have not seen other systems that let the user combine application actions in complex scenarios.

6. Conclusion and future work

Developing a replicated application is creative work and remains difficult. The application must execute user commands tentatively, log them, and respond to conflicts by undoing some actions and replaying others under system control.

The IceCube and Rufis frameworks make this work somewhat easier. The IceCube approach is to make the application invariants explicit (reified as constraints) and to leave enforcement of the invariants to the system. Rufis provides the standard naming and persistence services of a file system, seamlessly integrated in the IceCube environment.

Constraints are a convenient, intuitive representation of application semantics. In particular, the object constraint API (i.e., the concurrency control interface) makes explicit the questions that the developer would need to ask anyway: do these operations conflict? do they commute? etc. This encourages developers to design independent, commutative and idempotent actions whenever possible.

Reconciliation logic is well separated from application logic. Adding reconciliation to the solo version of Rufis was fairly mechanical. Today unfortunately, the process remains verbose, tedious and error-prone; we would welcome specialised language support, which would make it easier to write action code, could automate object identification (see the RufisKey discussion earlier), and could generate the logging code automatically. We have implemented a compiler to automatically extract static constraints from dynamic preconditions in the SQLIceCube project [4].

Acknowledgements

This work benefited greatly from discussions with Nikolaj Bjorner, Ant Rowstron and Caroline Matheson. Flavio Junqueira implemented persistence of files and logs, and fixed many problems.

References

[1] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Int. Conf. on Mobile Comp. and Netw. (MobiCom '98)*. ACM/IEEE, Oct. 1998.

[2] P. Cederqvist, R. Pesch, et al. Version management with CVS, date unknown. `http://www.cvshome.org/docs/manual`.

[3] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Usenix Tech. Conf.*, New Orleans, LA, USA, Jan. 1995.

[4] N. Preguiça, M. Shapiro, and J. Legatheaux Martins. Automating semantics-based reconciliation for mobile transactions. In *CFSE'3 : conférence française sur les systèmes d'exploitation*, pages 515–524, La-Colle-sur-Loup, France, Oct. 2003.

[5] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. In *9th Foundations of Softw. Eng.*, pages 175–185, Austria, Sept. 2001.

[6] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *Usenix Conf.* Usenix, June 1994.

[7] Y. Saito and M. Shapiro. Optimistic replication. Technical Report MSR-TR-2003-60, Microsoft Research, Oct. 2003.

[8] V. Saraswat and P. V. Hentenryck, editors. *Principles and Practice of Constraint Programming.* MIT Press, 1995.

[9] P. M. Schwartz and A. Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, Aug. 1984.

[10] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO (USA), Dec. 1995. ACM SIGOPS.

[11] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, Dec. 1988.