Design and evaluation of semantic-aware reconciliation for disconnected computing

October 16, 2002

Abstract

Optimistic replication lets multiple users write to shared data with no remote synchronisation. However such replicas diverge and must be reconciled. IceCube is a general-purpose reconciliation engine, parameterised by "constraints" capturing data semantics and user intents. Given logs of update actions, IceCube suggests reconciliation schedules that combine the actions nearly-optimally and that honour the constraints. IceCube features a simple, high-level, systematic, and intuitive API for applications to express constraints. IceCube integrates together diverse applications, sharing various data, and run by concurrent users. This paper presents the IceCube API and algorithms. Application experience indicates that IceCube simplifies application design, supports a wide variety of application semantics, and seamlessly integrates diverse applications. On a realistic benchmark, IceCube runs at reasonable speeds and scales to large input sets.

1 Introduction

In mobile computing and other settings, replicating shared data allows users to access it readily even when network communication is unavailable, slow or expensive. To enable multiple users to also write the shared data requires an *optimistic* replication system, one where an update operates on a local replica, and is later propagated to other replicas as communication becomes available. However an update is tentative, since some concurrent, conflicting update might overwrite it. Given a set of concurrent tentative updates, a *reconciliation* algorithm selects, among all the possible combinations, one that contains no conflicts [15], possibly dropping some actions. In operation-based (or log-based) approaches, update actions are recorded in a log; reconciliation replays the combined actions, from the initial state, according to some *schedule*.

Existing reconciliation systems (see related work in Section 8) have several shortcomings. Those that wire in a single application semantics are severely

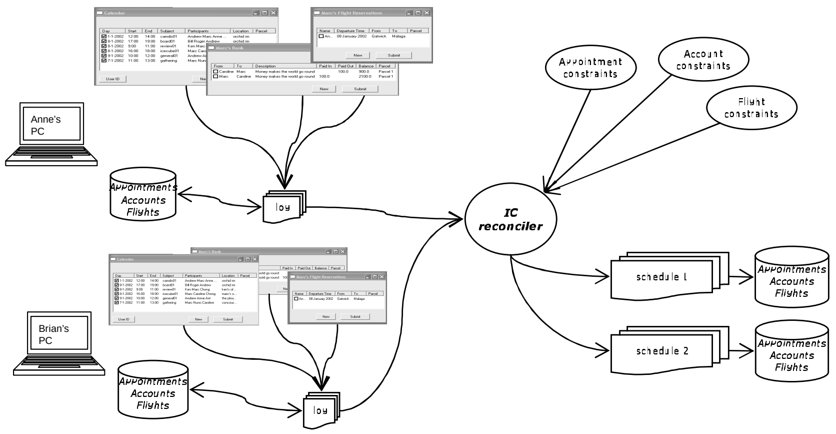


Figure 1: IceCube system structure

limited. Those that do not take semantics into account suffer spurious conflicts.[1]

In contrast, IceCube provides a common reconciliation layer that is generic, yet parameterised by application semantics. Most importantly, it reconciles harmoniously across users, applications and objects that need not be aware of each other's existence. An application may access several data objects; different applications may access the same data; a user might combine several mutually-ignorant applications in some particular task; and different users may work in parallel. IceCube properly reconciles across these borders.

1.1 Challenges and contributions

A conflict is defined as a set of actions that, if run together, would violate application invariants. To support a variety of applications, IceCube needs to know when this happens, i.e., it needs access to some of the application semantics. Our first challenge was to capture this information in a general and powerful way. Our contribution is an API whereby applications express precise dependencies and invariants as *constraints*.

Many systems are conservative in their conflict detection, dropping actions whenever in doubt. Our second challenge was to avoid dropping unless strictly necessary. Our contribution is to approach reconciliation as an optimisation problem. IceCube schedules are near-optimal, in the sense that they heuristically minimise the value of dropped actions.

The third challenge is efficiency and scalability. We address this primarily by encouraging applications to use so-called *static* constraints. Another tech-

[1]This is highly undesirable because the impact of dropping an action spuriously can be large. For real-life examples, consider yourself using a disconnected calendar that drops an important meeting, or a a travelling salesman's sales-support tool that would drop orders.

nique we use is to decompose large inputs into small independent reconciliation problems. Benchmarks show that IceCube reconciles in reasonable time and scales nicely to large logs.

A final challenge is practicality. We report our experience coding a number of useful applications. Using IceCube considerably simplifies application development, and furthermore enables users to interact with diverse applications and objects, with the assurance they will reconcile consistently and seamlessly.

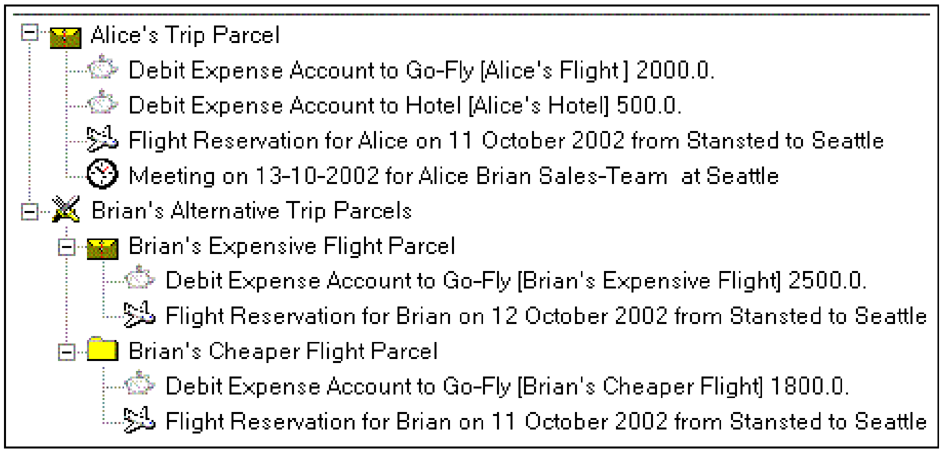
1.2 Outline

This paper is organised as follows. Section 1 is this introduction. In Section 2 we present our system model and give an example of the uses and capabilities of IceCube. Section 3 discusses the main concepts and APIs. The scheduler's algorithms are explained in detail in Section 4. Some applications that use IceCube are presented in Section 5. Some application design hints and a discussion of modes of operation around IceCube make Section 6. We evaluate performance and quality of IceCube in Section 7. Section 8 discusses related work, and Section 9 summarises conclusions and lessons learned.

2 Operation

2.1 System model

IceCube is the reconciliation component of a distributed system supporting disconnected operation, as sketched in Figure 1. In order to share data, each computer (Anne's and Brian's PC are shown in the figure) has its own replicas (Appointments, Accounts, Flights in the figure), which it can both read and update locally. An application applies updates tentatively to the local local replica and logs them. The system sends the logs (incrementally while the site is connected, or in a batch when a disconnected site reconnects) to the IceCube reconciler. The reconciler combines the concurrent tentative updates, and replays them against the initial state, in a sequential execution called a *schedule*. It consults semantic information, both recorded in the logs (log constraints), and characterising shared data (object constraints: see Appointment, Account and Flight constraints in the figure). If the logs are conflicting, the reconciler generates any number of non-conflicting schedules, and returns the proposed schedules and results. The user (or a process working on his behalf) may select a schedule to be committed from among the proposals. If the user is not satisfied with the proposed schedules (e.g., because they an unexpressed user preference, or because there are too many conflicts), he may either ask for more proposals, commit only a subset of actions, or edit the logs and try again.



```
Alice's Trip Parcel
    Debit Expense Account to Go-Fly [Alice's Flight] 2000.0.
    Debit Expense Account to Hotel [Alice's Hotel] 500.0.
    Flight Reservation for Alice on 11 October 2002 from Stansted to Seattle
    Meeting on 13-10-2002 for Alice Brian Sales-Team  at Seattle
Brian's Alternative Trip Parcels
    Brian's Expensive Flight Parcel
        Debit Expense Account to Go-Fly [Brian's Expensive Flight] 2500.0.
        Flight Reservation for Brian on 12 October 2002 from Stansted to Seattle
    Brian's Cheaper Flight Parcel
        Debit Expense Account to Go-Fly [Brian's Cheaper Flight] 1800.0.
        Flight Reservation for Brian on 11 October 2002 from Stansted to Seattle
```

Figure 2: The collected logs of the travel scenario

2.2 A multi-application, multi-data scenario

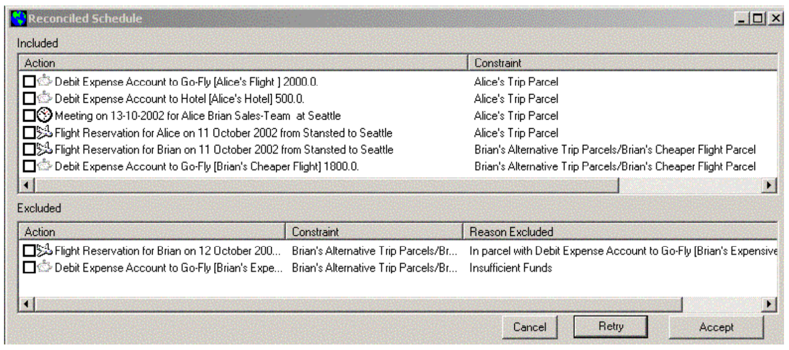
Let's consider a simple scenario of using IceCube to get a feel of how it operates. Two users, Anne and, Brian planning a business trip to meet their sales team in a distant city. They make their plans independently, updating the local replica of the corresponding databases. When they reconcile, they may experience conflicts such as double bookings, insufficient funds, or a cancelled flight.

While away from the network, Anne uses her calendar application to schedule a meeting, a travel reservation application to book flights, and a banking application to pay for travel and accommodation. Although these are separate applications, Anne indicates to the system that her actions are part of a single indivisible activity. This constrains the schedule to include either all of them or none.

Brian accesses much the same applications on his own disconnected personal computer. Brian wants the system to help him choose between two different possibilities. One is a convenient but expensive flight on 12 October, the other a less convenient but cheaper one on 11 October. Brian indicates his preference by giving a higher value to the 12 October flight.

Anne, as the manager, controls the reconciler. The collected actions appear, in a tree structure, in the window shown in Figure 2. The top branch shows Alice's four activities, collected in a "parcel" indicating indivisibility. Brian has submitted two parcels, and has indicated an "alternative", instructing the system to choose between them.

Figure 3 presents one output from the reconciler. The top pane contains a suggested non-conflicting schedule. The bottom pane shows the actions that were dropped from the schedule, and the reason why. Brian's payment for the expensive flight is dropped for insufficient funds (unbeknownst to Anne and Brian, another user has concurrently spent some of the travel account); the



Included

Action	Constraint
☐ Debit Expense Account to Go-Fly [Alice's Flight] 2000.0.	Alice's Trip Parcel
☐ Debit Expense Account to Hotel [Alice's Hotel] 500.0.	Alice's Trip Parcel
☐ Meeting on 13-10-2002 for Alice Brian Sales-Team at Seattle	Alice's Trip Parcel
☐ Flight Reservation for Alice on 11 October 2002 from Stansted to Seattle	Alice's Trip Parcel
☐ Flight Reservation for Brian on 11 October 2002 from Stansted to Seattle	Brian's Alternative Trip Parcels/Brian's Cheaper Flight Parcel
☐ Debit Expense Account to Go-Fly [Brian's Cheaper Flight] 1800.0.	Brian's Alternative Trip Parcels/Brian's Cheaper Flight Parcel

Excluded

Action	Constraint	Reason Excluded
☐ Flight Reservation for Brian on 12 October 200...	Brian's Alternative Trip Parcels/Br...	In parcel with Debit Expense Account to Go-Fly [Brian's Expensive
☐ Debit Expense Account to Go-Fly [Brian's Expe...	Brian's Alternative Trip Parcels/Br...	Insufficient Funds

Figure 3: Possible reconciliation for travel scenario

corresponding booking is dropped since it is in a parcel with the payment request; and the cheaper flight alternative is included in the schedule. At this point, Anne may press "Accept" to commit the decision, "Retry" to ask IceCube to propose a new schedule based on the same inputs, or "Cancel" to exit and reconcile later. In the last two cases, she may tick the boxes on the left to force a particular action to be included or dropped in the next proposal.

Although we presented an interactive reconciliation session, IceCube can also generate schedules under program control, which can then select among them without a user in attendance.

3 The IceCube system and APIs

IceCube explores the space of possible schedules heuristically. It selects and executes the highest-valued ones, subject to constraints specified by the application and the shared data. It can do so repeatedly until a satisfactory solution is found.

3.1 Shared data, actions and logs

A set of data managed by IceCube is abstracted by the `ReplicatedState` class. An application developer provides his `ReplicatedState` with methods to checkpoint and to return to a previous checkpoint. `ReplicatedState`s are disjoint and are intended to be coarse grained.

At any time a particular `ReplicatedState` exists in several versions: there is a copy at each site, and each copy can have multiple checkpoints, in addition to the current tentative state and/or the current replay state. The set of checkpoints includes a linearly-ordered subset of stable checkpoints.

The IceCube logging API has primitives for creating an action, recording it in the local log, and registering and combining log constraints. An action has an integer *value*, 1 by default.

An action is a Java object implementing a number of interfaces described in this and subsequent sections. The methods are call-backs invoked by the Ice-Cube system during the reconciliation process. The first interface is for action execution:

```
interface ActionExecution {
   // test precondition; no side effects
   public boolean preCondition (ReplicatedState state);

   // execute; update state; return postcondition + undo info
   public boolean execute (ReplicatedState state, UndoData info);

   // undo execution; return false if can not undo
   public boolean undo (ReplicatedState state, UndoData info);
}
```

Method `preCondition` is intended to test without side-effects whether the action is valid in the current state. Method `execute` is intended to update the state, accomplishing the action's purpose. It returns a boolean post-condition indicating success or failure. Optionally it may return information to be used by a subsequent call to the `undo` method. The latter is intended to roll back a previous update. We will see examples of these methods when we consider specific applications in Section 5.

3.2 Constraints

Conflict detection and scheduling are based on *constraints*. Constraints express either user intents or object correctness invariants. In the travel scenario, Anne has expressed her intent that two apparently independent actions (booking a flight and making a payment) constitute a single activity. An example object correctness invariant would be forbidding two concurrent users from both reserving the last available seat in an airplane.

Constraints are either *static*, i.e., independent of the shared state, or *dynamic*.[2] The former serve to limit the scope of search; the latter (the pre- and post-conditions of the `ActionExecution` interface) are verified against the current state of the shared objects while reconciling.

3.2.1 Primitive static constraints

The two primitive static constraints are Order, noted $\rightarrow$, and MustHave, noted $\rightsquigarrow$. Any schedule s must satisfy the following soundness conditions:

1. Consistent with Order: For all actions $a, b \in s$, if $a \rightarrow b$ then a comes before b in the schedule (although not necessarily immediately before),
2. Closed for MustHave: For any $a \in s$, every action b such that $a \rightsquigarrow b$ is also in s (but not necessarily in that order nor contiguously).

The first condition implies that a schedule cannot contain a cyclic Order relationship. If cycles occur within or between logs, they must be broken by

[2]A better characterisation might be "declarative" and "imperative."

dropping one or more actions from the schedules. The breaking of (non binary) cycles is the main source of complexity of scheduling. Finding an acyclic subgraph of a given size is an NP-complete problem [6]; therefore the reconciliation problem, of finding an optimal such subgraph, is NP-hard.

The primitive constraints are too low-level to be used directly. The IceCube API wraps them into higher-level abstractions, log constraints and object constraints, which we explain next.

3.2.2 Log constraints

A *log constraint* is a dependency between two specific actions, stored with them in a log. A log constraint expresses how actions relate to one another, i.e., the user's intents.

The `predecessorSuccessor` constraint states that the second action may be executed only after (not necessarily immediately after) the first one succeeds. This usually indicates the second action consumes some effect produced by the first. For example consider changing a file, then copying the changed version. To maintain the same behaviour at reconciliation, the write action is made the predecessor of the copy action in the log. For two actions a and b the relation `predecessorSuccessor`(a, b) is equivalent to $a \rightarrow b \wedge b \rightsquigarrow a$, i.e., the conjunction of primitive Order and MustHave relations in opposite directions.

The `alternative` relation instructs the system to choose a single action from 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. In case of conflict, an `alternative` provides the scheduler with fallback. `alternative`(a, b) translates to $a \rightarrow b \wedge b \rightarrow a$, i.e., a and b cannot both be included in the same schedule.

The `parcel` relation constitutes an all-or-nothing grouping — either all of its actions are executed successfully, or the parcel fails. The actions may run in any order, possibly interleaved with other actions (unless otherwise constrained). `parcel`(a, b) is equivalent to $a \rightsquigarrow b \wedge b \rightsquigarrow a$.

3.2.3 Object constraints

An *object constraint* is a relation between action types, and reflects the static semantics of shared data. At the beginning of scheduling IceCube extracts object constraints by comparing each pair of actions (not already related by a log constraint) by the methods in the following interface:

```
interface ActionObjectConstraint {
   // test whether this and other action conflict
   public boolean mutuallyExclusive (Action otherAction);

   // Test successful ordering of this and other actions
   public int bestOrder (Action otherAction);
}
```

Method `mutuallyExclusive` returns true if running both actions would violate an invariant. For example, in a file system, an action creating a directory named N is mutually exclusive with one creating a file also named N. The relation `mutuallyExclusive`(a, b) is equivalent to $a \rightarrow b \wedge b \rightarrow a$.[3]

When there is a favorable execution order for a pair of actions, method `bestOrder` indicates that order. For instance, in a bank account application, `bestOrder` comparing a debit and a credit to the same account will return "credit before debit." The relation `bestOrder`(a, b) is equivalent to $a \rightarrow b$.

Object constraints only make sense between concurrent actions; therefore log constraints take precedence over object constraints. For example, two actions that already are related by `predecessorSuccessor` will not be reordered by `bestOrder`.

When there is no explicit Order constraint between two actions, they can run in any order. As an enhancement, the action interface provides more direct and efficient ways of indicating commutativity. `mutuallyExclusive` and `bestOrder` are called only after the following have been tested:

```
interface ActionEnhancements {
   // domain identifiers
   public long[] getDomain ();

   // test if this and otherAction (same domain) overlap
   public boolean overlap (Action otherAction);

   // test if this and other (overlapping) actions commute
   public boolean commute (Action otherAction);
}
```

`GetDomain`: An action has any number of domains, opaque identifiers that partition application objects. For instance the domain of a banking action might be a hash of the branch number-account number pair. Two actions with no common domain are commutative.

Method `overlap` tests whether actions (with a common domain) overlap. For instance a banking application action tests, first if the other action is also a banking action (because domains are not guaranteed unique), and whether it operates on the same branch and account number. Two actions that don't overlap are commutative.

Method `commute` tests whether (overlapping) actions commute semantically. For instance, two credits to the same bank account overlap but commute. Although ¬`overlap` and `commute` are functionally equivalent, developers find it easier to address the two questions separately, as will become apparent when we look at some applications. Two actions that `commute` are commutative.

As an example, Table 1 shows the object and dynamic constraints in a banking application with credit and debit actions, and a calendar application with actions to add or remove an appointment.

[3]The formulæ for `alternative` and `mutuallyExclusive` are identical. Indeed, both choose between actions. However they are reported back differently: the former as normal behaviour, the latter as a conflict.

Bank	credit/credit	debit/debit	debit/credit
Different accounts	¬`overlap`	¬`overlap`	¬`overlap`
Same account	`commute`	`commute`	`bestOrder`

Dynamic constraint: no overdraft

Calendar	`add/add`	`remove/remove`	`remove/add`
Other user, time	¬`overlap`	¬`overlap`	¬`overlap`
Same user & time	`Mut.Excl.`	`commute`	`bestOrder`

Dynamic constraint: no double-booking

Table 1: Bank and Calendar constraints

4 Reconciliation scheduler

Static constraints limit the problem size, but as it remains inherently complex, our reconciler searches heuristically through the static constraint space. As it generates a schedule, it immediately executes it and checks the dynamic constraints. We now sketch the design and implementation of the corresponding algorithms. Later (in Section 7) we benchmark their efficiency and quality.

The reconciliation engine consists of approximately 6,200 lines of Java.

4.1 Clustering

We first partition actions into disjoint subsets called clusters, dividing the search space into independent sub-problems. Then we will reconcile each cluster independently in arbitrary sequential order.

A cluster contains actions that are unrelated by static constraints to actions in another cluster. Given action a in cluster A and action b in cluster B, then $\neg(a \leadsto b) \wedge \neg(a \rightarrow b)$ is true, and vice-versa. This ensures that actions from different clusters may be scheduled in arbitrary order, and that the decision to include or drop an action from one cluster does not affect any other cluster. In particular a dynamic constraint violation will not cause actions from another cluster to roll back.

Clustering occurs in two stages. First, actions are partitioned by domain, with a complexity linear in the number of actions. Then, a subset is re-partitioned according to the static constraints between pairs in the subset, for a complexity quadratic in subset size. For space reasons, we omit further detail; interested readers are referred to our technical report [13]. Since, as we shall see shortly, the complexity of reconciliation is quadratic in cluster size, decomposing large inputs into smaller clusters is highly beneficial.

4.2 Heuristic search

Within each cluster, the scheduler performs efficient heuristic sampling of small portions of the search space. If the user requests a new schedule, or if a partial schedule has failed, an unrelated portion of the search space is selected.

Initially each action is assigned a merit, estimated from its constraints. Every time an action is added to a schedule, the merit of the remaining actions is re-evaluated.

At each step the scheduler selects (with randomisation) among the candidates with highest merit. Although our heuristics are not guaranteed to find the true optimum, their results are virtually indistinguishable from exhaustive search, and complexity is dramatically decreased (quadratic rather than exponential). This is confirmed by our benchmarks in Section 7.

The merit estimator computes the benefit of adding a given action to a given partial schedule. An action that enables many other actions to run has high merit. More precisely, the merit of a candidate action a is higher:

1. As the the value of actions that can only be scheduled before a is lower.
2. As the value of alternatives to a is lower.
3. As the value of actions mutually exclusive with a is lower.
4. As the value of actions that can only be scheduled after a is higher.

The above factors are listed in decreasing order of importance. Furthermore, as dynamic failures are expensive, when an action fails dynamically, its merit decreases drastically in the near future, to avoid visiting it again. The merit estimator executes in constant time.

Our scheduling algorithm, displayed in pseudo-code in Figure 4, selects (with randomisation) some action among those with highest merit, executes it, adds it to the schedule if execution succeeds, and drops any actions that would consequently violate soundness. When action a is dropped from schedule s, the algorithm also drops sets $\texttt{MustHaveMe}(a) = \{b | b \rightsquigarrow a\}$ and $\texttt{OnlyBefore}(a, s) = \{b | b \to a \wedge b \notin s\}$. If the dropped actions had side effects, these are rolled back.

The scheduler calls `scheduleOne` repeatedly and remembers the highest-value schedule. It terminates when some application-specific selection criterion is satisfied — often a value threshold, a maximum number of iterations, or a maximum execution time.

The overall complexity of `scheduleOne` is $O(n^2)$, where n is the size of its input. Readers interested in the full algorithm and justification of the complexity estimate are referred to our technical report [13].

5 IceCube applications

In this section we describe some applications implemented with IceCube. This is to give a flavour of how the IceCube abstractions are used in practice and of the complexity of building an application. In the next section we discuss the lessons learnt.

```
scheduleOne (state, summary, goodActions) =   // pseudo-code
   schedule := []
   value := 0
   actions := goodActions
   WHILE actions <> {} DO
     nextAction := selectActionByMerit (actions, schedule, summary)
     precondition := nextAction.preCondition (state)
     IF precondition = FALSE
     THEN // pre-condition failure
       // abort and exclude any partially-executed parcels
       cantHappenNow := OnlyBefore (nextAction, schedule)
       toExclude := MustHaveMe (nextAction)
       toAbort := INTERSECTION (schedule, toExclude)
       IF NOT EMPTY (toAbort)
       THEN // roll back
          SIGNAL dynamicFailure (goodActions \ toExclude)
       ELSE
         summary.updateInfoFailure (actions, toExclude)
         actions := actions \ toExclude \ cantHappenNow
         LOOP
     postcondition := nextAction.execute (state)
     IF postcondition = TRUE
     THEN // action succeeded
       toExclude := OnlyBefore (nextAction, schedule)
       toExclude := MustHaveMe (toExclude)
       actions := actions \ toExclude
       summary.updateInfo (actions, nextAction)
       schedule := [schedule | nextAction]
       value := value + nextAction.value
     ELSE // post-condition failure: roll back
       toExclude := MustHaveMe (nextAction)
       SIGNAL dynamicFailure (goodActions \ toExclude)
   RETURN { state, schedule, value }
```

Figure 4: Executing a single schedule

5.1 Calendar application

The calendar application maintains an appointment database shared by multiple users. User commands are to `request` a meeting, possibly proposing alternative times, or to `cancel` a previous request. A user command tentatively updates the database and logs the corresponding actions. Database actions are to either `add` or `remove` a single appointment. The user-level `request` command is mapped onto an `alternative` containing a set of `add` actions; similarly `cancel` maps to a set of `remove`s. The data part of an action contains the time, duration, participants and location of an appointment. The object constraint methods are the following:

1. The shared calendar of one institution constitutes a domain.
2. Two actions (of the same institution) overlap when either time and location or time and participants intersect.
3. (Overlapping) `remove` actions commute, as removing a meeting a second time is a no-op.
4. A pair of `add` actions at this point known to overlap, so it is mutually exclusive. An (overlapping) `add-remove` pair is not.

5. `remove`s execute before `add`s to increase the probability that `add`s can be accommodated.

These methods are particularly simple because logs are clean, i.e., contain no redundant actions.

The code for `add` and `remove` actions includes the following methods:

1. `add.preCondition` checks that the appointment doesn't conflict with anything currently in the database.[4] `remove.preCondition` returns true, since it can't fail (removing a non-existent appointment is a no-op).
2. The `execute` method performs the corresponding database updates and saves undo information.
3. The `undo` method uses the saved undo information to roll back a previous `execute`.

The whole calendar application is very simple, totaling approximately 880 lines of code. This application was used as one of our performance and quality benchmarks, as we report in Section 7.

5.2 Reconcilable File System

Our Reconcilable File System (RFS) emulates a file system with the usual Unix semantics.[5] There is no locking, since concurrency control is optimistic.

5.2.1 RFS design

Concurrent changes to the same object conflict if they violate the standard filesystem semantics: for instance creating two files in the same directory updates the directory object twice; but there is no conflict if the files have different names. A conflict occurs also if one user writes a file, while another user deletes it, because the second user's work would be lost. The code tests explicitly for this special case, as file deletion changes the state of the parent directory, not that of the file.

Internally a file system is a `ReplicatedState` containing a tree of `DirectoryNode`s and `FileNode`s, and a hash table of the same nodes, indexed by an internal node identifier, the RFSKey.

We decompose a user-level command into a high-level prelude that checks arguments and establishes what needs to be done, followed by low-level linking and unlinking nodes in the tree. The corresponding actions make up a parcel. For instance the `move` command, renaming a file or directory a file, is logged as a parcel, composed first of an action that checks which of nine possible cases to execute, then of up to three link and unlink actions.

[4]The static constraints appear to make this check unnecessary, but in some execution scenarios that are too complex to explain here it is indeed needed.

[5]The Unix link commands are not implemented.

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

Dynamic constraint: 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

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

Table 2: RFS object constraints

In a link or unlink action, the parent directory is identified by RFSKey and the file or directory being linked by its name. Using RFSKey to identify a parent directory has the advantage that the key does not change as the directory is renamed. A node creation action ensures that at reconciliation time, the new node is re-created with the same RFSKey as during tentative execution.

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.

5.2.2 RFS actions

File system commands operate on one or two parent directories, linking or unlinking nodes within them. Low-level code either creates a node, links a node into a directory under some name, or unlinks a named node. The object constraints for RFS are summarised in Table 2. In more detail, those on `DirectoryNode`s are as follows:

1. Each file system is a distinct domain.
2. Actions with the same parent directory (i.e., same RFSKey) 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.
3. Actions that overlap commute if neither of them is a write, or if the object names differ.
4. Overlapping actions that do not commute conflict if they are both writes. Writing a file conflicts with unlinking its directory (the special case again).
5. Actions that overlap but do not commute nor conflict have the same parent directory. We order unlink actions before links to reduce the chance of dynamic failure.[6]

[6]As in the calendar application, dynamic failures might occur, despite the static constraints, in some corner cases.

There are further object constraints on `File` actions:

1. The domain of a File action is its file system.
2. File actions in the same domain overlap if the files have the same RFSKey. A write action overlaps with a unlinking the node from its parent directory (the special case). Otherwise, a `File` action does not overlap with any other type of action.
3. File actions commute if either is a creation action, or if both are read actions.
4. A write conflicts with unlinking the file (special case), and two writes conflict with each other.
5. Reads are preferably scheduled before writes to the same file.

Finally `Directory` actions are of two types: either creating a directory, or a prelude action for a parcel of `DirectoryNode` links and unlinks. The former never conflict, and the latter might fail dynamically but have no side effects. Hence: the domain of a Directory action is its file system; no `Directory` action overlaps with any other; and the other object constraint methods are never called.

For RFS, we chose to log by recording a trace of the actual side-effects (then cleaning up the trace), as opposed to "diff-ing" snapshots before and after tentative execution. Diffing does not properly record user intents; for instance it cannot differentiate renaming a file from two independent delete and create actions.

5.2.3 Implementation

We now study the RFS code. The RFS code is based on a "solo" system that implements the centralised file system functionality. A "replicated" interface provides reconcilable semantics. Each command in the replicated version first executes the solo code on the local replica, then (but only if solo execution succeeded) traces the corresponding actions. For every side effect (viz., node creation, linking and unlinking) performed by the solo version, the replicated version records the corresponding action in the trace. The trace code is essentially a modified copy of the solo code.

We focus on the directory creation command `mkdir`, because it is simple yet representative. 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 RFSKey of the parent directory, the RFSKey of the new subdirectory, and the name of the subdirectory. If linking was successful it then records to the log, first an action to create a directory node with the same RFSKey, then an action to link the new node under the same name, into the parent directory identified by its own RFSKey. The two actions are both part of a single parcel, and `predecessorSuccessor` of one another. `Mkdir` is sufficiently simple that this parcel needs no prelude

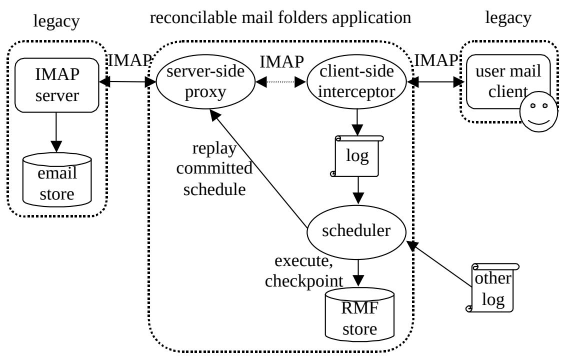


Figure 5: Structure of Reconcilable Mail Folders application.

(in particular it is not necessary to check the arguments again at reconciliation time).

5.2.4 Discussion

Thanks to parcels, it is possible break down a complex action, such as `move`, into a series of simpler links and unlinks.

Identifying nodes by RFSKey and name helps make the problem more tractable. Consider one 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 RFSKey. On the other hand, if users concurrently create a file and a subdirectory with the same name, within the same parent directory, the `mutuallyExclusive` method of the link action will indicate a conflict.

Currently RFS has no provision for activities that might change between tentative and reconciliation time. For instance, renaming all files ending with `".c"` to `".cc"` is logged as a series of individual moves. If a new `".c"` file has been added, it won't be renamed in the reconciled schedule. If this is not the desired effect, a new kind of action could be added to accommodate pattern renaming; this would make the object constraints more complex.

Currently, any concurrent writes to a file are considered in conflict. But it should be easy to layer object constraints with higher semantics above RFS: for instance CVS-like object constraints applying to source files, or an RMF object constraints applying to mail folder files.

RFS is 2,974 lines of code. Of these, roughly 57% are for the solo code; only 43% (1,283 lines) are specific to replication. The RFS developer added reconciliation with a small piece of code, thanks to our systematic approach. His task was made easier by a well-defined framework designed as a series of straightforward and relatively intuitive questions.

5.3 Reconcilable Mail Folders

Our Reconcilable Mail Folders application (RMF) uses IceCube to merge concurrent changes on replicas of mail folders. In most aspects related with semantic reconciliation, RMF is similar to RFS, although its semantics are somewhat simpler. An interesting aspect is that RMF interposes between legacy, unmodified mail client and server, by intercepting the standard IMAP messages [3].

As shown in Figure 5, each replica maintains a local IMAP server and an equivalent IceCube state, called hereafter `RMFStore`. The `RMFStore` contains only meta-information about folders and messages; the legacy client caches message contents and the server stores the mail database. At a client site, an interceptor logs IMAP operations. Updates are reconciled initially against the `RMFStore`, and only then the committed schedule executes against the IMAP server. This avoids having to checkpoint the whole server state to try multiple schedules during reconciliation.

RMF totals 2,625 lines of Java. RMF differs from ordinary IMAP clients that implement reconciliation by replaying a user's log against the current server state [5]. This approach, akin to log concatenation (Section 8), suffers from false conflicts. Consider a user copying a message while another deletes it: this fails if the second user reconnects first. In contrast, RMF uses the semantically correct ordering.

6 Discussion

In this section we present some guidelines for application design, based on our experience. We also discuss issues related with deployment of IceCube.

6.1 Application design

Our experience indicates that IceCube considerably simplifies the development of reconcilable applications. Indeed, developers do not need to re-create ad-hoc mechanisms for each application. They only need to convey some simple facts about their application, and they have tools to structure the application in the right way.

The application must be designed to tolerate tentative update, roll-back, and replay. Many modern applications, such as Microsoft Word, already support logging, undo and redo, but they do not record precise MustHave and Order dependencies.

Based on our experience, we offer some further design hints that can serve as guidance for future developers.

High-level entities should be decomposed into small, manageable units. This applies to both data and actions. Small data objects reduce false sharing, thus improving concurrency. Small and simple actions simplify static properties [14]. Complex actions can always be created using the composing mechanisms defined in the IceCube API.

Developers are encouraged to use static constraints and to avoid dynamic ones, because the former direct the search and the latter cause schedule execution to roll back.

To simplify static reasoning, the log should be clean, i.e., should contain no redundant actions.[7] Avoid actions that overlap. Design actions to commute and/or be idempotent.

The RMF experience shows how it is possible to interpose reconciliation onto a legacy application by keeping only a compact representation of the legacy state. However this approach can work only if every committed schedule executes without failure in the legacy application.

6.2 Incremental operation

IceCube supports either a batch style of operation, or an incremental style where the system transmits and collects actions in the background. As users work, the system continuously detects conflicts and proposes solutions; users have the option to ignore or to apply the proposals, without interrupting their work flow. This style is appropriate, either when connections are slow, or when users work in isolation by choice (e.g., during cooperative development or to test out some hypothesis).

A peer-to-peer architecture, in which sites send their logs to each other and run IceCube locally, is a natural complement to incremental operation. In this situation, reconciliation results are advisory only, for two reasons. First, because of asynchronous communication, at any point in time a local reconciler has only partial information, and what appears optimal at time t may be sub-optimal at $t' > t$. Second, it is difficult to commit in a peer-to-peer setting. The next section examines this difficulty.

The user interface used for the travel scenario (Section 2.2) demonstrates an intermediate style. It relies on a central reconciler, but gives the controlling user options to express preferences. The user might accept or reject part of a proposed schedule, change the weights of actions, or even edit the log, changing, adding or removing actions and constraints. The system ensures that any resulting schedule remains sound, e.g., committing an action commits its whole parcel.

6.3 Distributed operation

In the current prototype, commitment is centralised. A primary site collects update logs over some interval, runs the reconciler, possibly commits a schedule, and propagates it to the other sites. The interval period can be varied to suit different work habits. This approach could be extended to a CVS-style system [2] where any user can reconcile, but commitment by different users is serialised at a primary.

[7] IceCube implements a generic log cleaning mechanism, which we do not describe here for lack of space.

IceCube could also be used as the reconciler of a peer-to-peer communication architecture like Bayou [17]. In Bayou, any site may update any object and reconcile locally. However, objects are partitioned into disjoint subsets, called databases. A single primary site commits actions operating on a given database. Furthermore, Bayou forbids an action or a transaction to operate on objects from two databases (in IceCube terms this is disallowing log constraints across database boundaries). These restrictions ensure that a primary may commit independently from the others, but limit the user's freedom to work with several objects.

It would be nice to relax the above restrictions, and to support reconciliation across widely-distributed shared objects and across non-trusting organisations. The system should of course ensure global correctness. Hence the following requirements:

1. To support partial replication.
2. To support decentralised commitment.
3. To enable the computation of optimal (or near-optimal) schedules.
4. To ensure eventual convergence [15].

Unfortunately the requirements appear to be at odds with one another. It is doubtful they can all be fulfilled at once. For instance, a necessary condition for convergence is that sites agree on which actions are included in a committed schedule (committed action) or dropped from it (aborted action); no action may be both committed by one reconciler and aborted by another. Maintaining this global invariant is hard when commitment is decentralised.

One approach to maintaining the invariant is to centralise reconciliation at a single site, but this is at odds with a multi-organisation system. The Bayou architecture with its partitioned objects and actions supports multiple primaries, but since it disallows log constraints between subsets, it does not support application composition.

A possible decentralised approach is the following. Any reconciliation site is allowed to abort an action independently of others (as long as it maintains soundness). An action is committed only when all reconcilers have seen it, and none has aborted it. The shortcoming of this approach is that schedules tend to be non-optimal. For instance, two reconcilers might each abort a different side of a conflict. In the limit this approach might abort everything.

7 Measurements and evaluation

This section reports on experiments that evaluate the quality of IceCube reconciliation (by the size of the schedules, and by comparison with other approaches), and its efficiency and scalability (by execution time). Our two benchmarks are the calendar application, described previously, and an application described by Fages [4]. The calendar inputs are based on traces from actual Outlook calendars. These were artificially scaled up in size, and were modified

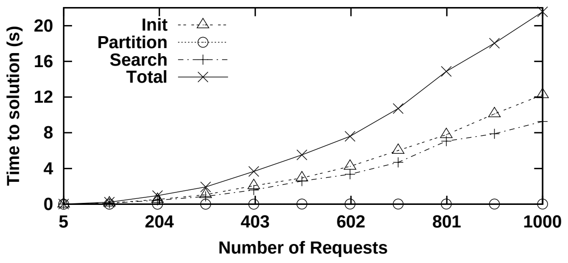


Figure 6: Decomposition of reconciliation time (single cluster).

to contain conflicts and alternatives and to control the difficulty of reconciliation.

The logs contain only Requests, each of which contains one or more `add` alternatives. We varied the number of Requests and the number and size of possible clusters. The average number of `add` alternatives per `request` is two.

In each cluster, the number of different `add`s across all actions is no larger than the number of Requests. For instance, in the example of Figure 13, in the three Requests, there are only three different `add`s ('9am room A', '9am room B' and '9am room C'). This situation represents a hard problem for reconciliation because the suitable `add` alternative needs to be selected in every `request` (selecting other alternative in any `request` may lead to dropped actions).

In these experiments, all actions have equal value, and longer schedules are better. A schedule is called a *max-solution* when no `request` is dropped. A schedule is optimal when the highest possible number of Requests has been executed successfully. A max-solution is obviously optimal; however not all optimal solutions are max-solutions because of unresolvable conflicts. Since IceCube uses heuristics, it might propose non-optimal schedules; we measure the quality of solutions compared to the optimum. (Analysing a non-max-schedule to determine if it is optimal is an offline, *a posteriori* process.)

The experiments were run on a generic PC running Windows XP with 256 Mb of main memory and a 1.1 GHz Pentium III processor. IceCube and applications are implemented in Java 1.1 and execute in the Microsoft Visual J++ environment. Everything is in virtual memory. Each result is an average over 100 different executions, combining 20 different sets of requests divided between 5 different pairs of logs in different ways. Any comparisons present results obtained using exactly the same inputs. Execution times include both system time (scheduling and checkpointing), and application time (executing and undoing actions). The latter is negligeable because the `add` code is extremely simple.

7.1 Single cluster

To evaluate the core heuristics of Figure 4, we isolate the effects of clustering with a first set of inputs that gives birth to a single cluster.

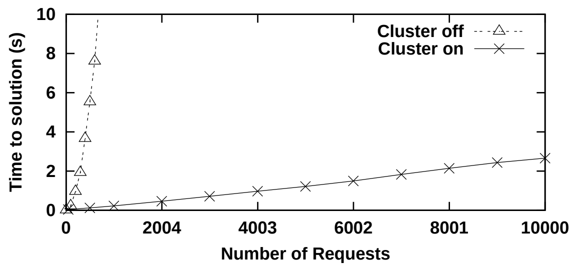


Figure 7: Performance improvement with clustering

Figure 6 measures the major components of IceCube execution time as log size increases. The "Init" line plots the time to collect object constraints and compute the initial summary of static constraints. "Partition" is the time to run the clustering algorithm (although the experiment is rigged to generate a single cluster, the clustering algorithm still runs). "Search" is the time to create and execute schedules. "Total" is the total execution time. As expected, clustering takes only a small fraction of the overall execution time. Init and Search are of comparable magnitude. The curves are consistent with our earlier $O(n^2)$ complexity estimate.

These experiments are designed to stop either when a max-solution is found, or after a given amount of time. Analysis shows that the max-solution is reached very quickly. The first schedule is a max-solution in over 90% of the cases. In 99% of the cases, a max-solution was found in the first five iterations. This shows that our search heuristics work very well, at least for this series of benchmarks. A related result is that in this experiment, even non-max-solutions were all within 1% of the max size.

Here is how the inputs are constructed. On average, each `request` is an alternate of h `add`s; each `add` in one `request` conflicts with a single `add` of another request. A log of x `request`s contains hx actions. To put the performance figures in perspective, consider that a blind search that ignores static constraints would have to explore a search space of size $(hx)!$ which, for $h = 2$ and $x = 1000$, is of the order of 10^{2061}. A more informed search that takes advantage of commutativity of actions would still have to explore a space of size $2^{hx} \approx 10^{600}$ for $h = 2$ and $x = 1000$. In fact there are only x distinct max-solutions.

7.2 Multiple clusters

We now show the results when it is possible to cluster the actions. This is the expected real-life situation.

The logs used in these experiments contain a variable number of Requests, and are constructed to that 25% of the `add`s can be clustered alone; 25% of the remaining `add`s are in clusters with two actions; and so on. Thus, as problem

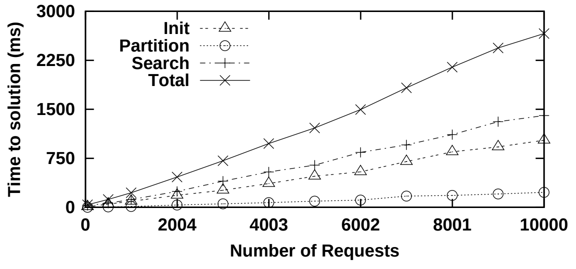


Figure 8: Decomposition of reconciliation time (multiple clusters).

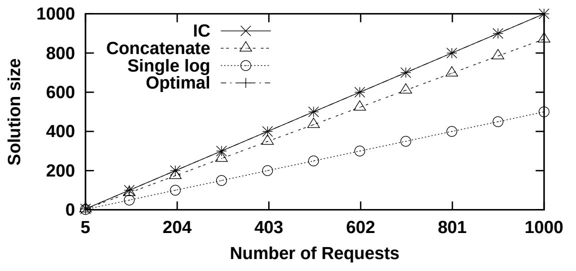


Figure 9: Syntactic vs. semantic, comparing schedule quality.

size increases, the size of the largest cluster increases slightly, as one would expect in real life. For instance, when the logs contain 1,000 actions, the largest cluster contains the `add`s from 12 Requests, and 18 when the logs total 10,000. The number of clusters is approximately half of the number of actions; this ratio decreases slightly with log size. The average number of alternatives per `request` is two.

IceCube always finds a max-solution, whether clustering is in use or not.

Figure 7 shows the time to find a max-solution, with clustering turned on or off; note the increased scale of the x axis. As expected a solution is obtained much more quickly in the former case than the latter. A running time under 3 s for a log size of 10,000, much larger than expected in practice, is quite reasonable even for interactive use. As the number of clusters grows almost linearly with the number of actions and the size of the largest cluster grows very slowly, reconciliation time is expected to grow almost linearly. The results confirm this conjecture. Moreover, the decomposition of the reconciliation time of Figure 8, shows that all components of the reconciliation time grow approximately linearly, as expected.

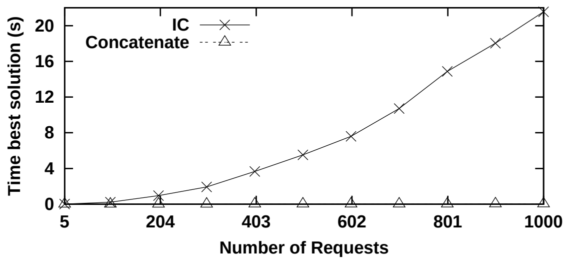


Figure 10: Syntactic vs. semantic, comparing performance.

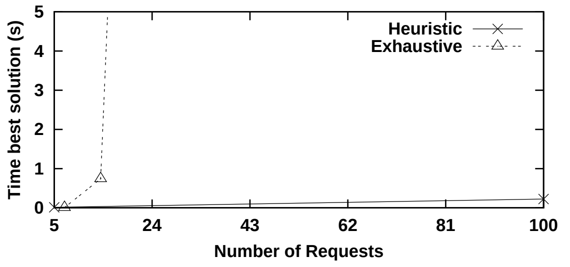


Figure 11: Exhaustive vs. heuristic search, comparing performance.

7.3 Comparisons

Here we compare the quality and performance of IceCube to competing approaches. Results in this section pertain to the non-clustered problems of Section 7.1.

Most other systems use a syntactic scheduling algorithm (see Section 8) such as timestamp order or log concatenation. To justify our optimisation approach, Figure 9 compares IceCube with log concatenation. Concatenation is representative of syntactic algorithms (see Section 8), which all suffer from false conflicts equally badly.

As expected, the results of semantic search are better than syntactic ordering. Whereas log concatenation drops approximately 12% of Requests, semantic-directed search drops close to none (although IceCube's drop rate grows very slightly with size). Remember that dropping a single action may have a high cost.

The baseline for comparison is the line marked "Single log." This scheduler selects all actions from a single log and drops all actions from the other; it is the simplest non-trivial syntactic scheduler that guarantees absence of conflicts.

Figure 10 shows the execution time of our engine versus a log-concatenation (hence suboptimal) scheduler. As expected, IceCube is much slower. This is in line with the expected complexities, $O(n^2)$ in IceCube without clustering, and

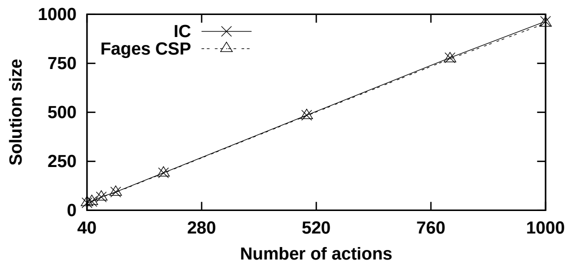


Figure 12: IceCube vs. Fages, comparing schedule quality (Fages' benchmarks).

$O(n)$ for concatenation.

Figure 11 compares execution time of our heuristic with an exhaustive search algorithm [9]. Given unlimited time, exhaustive search is guaranteed to find the optimal schedule, but the figure shows this is not feasible except for very small log sizes (up to 20 actions or so). When execution time is limited, exhaustive search yields increasingly worse quality solutions as size increases. For instance, exhaustive searches of five different logs, each containing 30 `requests`, and each admitting a max-solution (size 30), returned schedules of size 28, 17, 6, 30, and 4 (average = 17) when limited to a very generous 120 s. With size 40 the average is 18, and for size 100 the average is only 28, under the same time limit.

Fages [4] studies a constraint satisfaction programming (CSP) reconciliation algorithm, with synthetic benchmarks. We now compare the quality of the two approaches by submitting one of Fages' benchmarks to IceCube, and our calendar benchmarks to Fages' system.

Fages' benchmark contains randomly generated Order constraints with a density of 1.5, meaning that there are $1.5 \times$ size constraints on average. Figure 12 compares the quality of Fages' CSP solutions with IceCube's. The results are similar, but notice that IceCube appears to perform slightly better on large problems. This shows that the IceCube heuristics perform well on a different kind of input. As Fages' execution environment is very different, it would make no sense to compare absolute execution times; however we note that IceCube's execution time grows more slowly with size than Fages' constraint solver.

When we submit our calendar problems to Fages' system, execution time grows very quickly with problem size. For instance, for only 15 of our `requests`, Fages cannot find a solution within a timeout of 120 s. The explanation, we suspect, is that Fages' system does not deal well with alternatives.

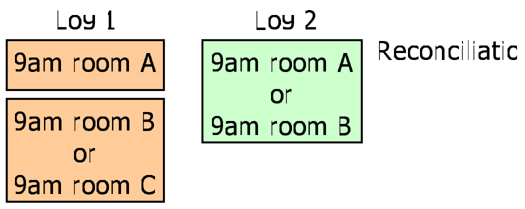


Figure 13: Syntactic scheduling spuriously fails on this example

8 Related Work

Several systems use optimistic replication and implement some form of reconciliation for divergent replicas. Many older systems (e.g., Lotus Notes [7] and Coda [10]) reconcile by comparing final tentative states. Other systems, like IceCube, use history-based reconciliation, such as CVS [2] or Bayou [17]. Recent optimistically-replicated systems include TACT [18] and Deno [8]. Balasubramaniam and Pierce [1] and Ramsey and Csirmaz [14] study file reconciliation from a semantics perspective. Operational Transformation techniques [16] re-write action parameters to enable order-independent execution of non-conflicting actions, even when they do not commute. For lack of space we focus hereafter on systems most closely related to IceCube. For a more comprehensive survey, we refer the reader to Saito and Shapiro [15].

A number of systems base resolve conflicts and schedule based on simple syntactic properties of the updates: for instance timestamp ordering or log concatenation. This is inflexible and causes spurious conflicts, as Figure 13 illustrates. Two users are using a calendar program. One user requests room A at 9:00, and either room B or C, also at 9:00. Meanwhile, the other user requests either room A or B at 9:00. Combining the logs syntactically does not work. For instance running Log 1 then Log 2 will reserve rooms A and B for the first user, and the second user's requests are dropped. Running Log 2 first has a similar problem; so does any other syntactic approach. To satisfy all three requests requires reordering them, which syntactic systems can't do.

Bayou [17] is a replicated database system. Bayou schedules syntactically, in timestamp order. A tentative timestamp is assigned to an action as it arrives. The final timestamp is the time the action is accepted by a designated primary replica. Bayou first executes actions in their tentative order, then rolls back and replays them in final order. A Bayou action includes a "dependency check" (dynamic constraint) to verify whether the update is valid. If it is, the update is executed; otherwise, there is a conflict, and an application-provided merge procedure is called to solve it. Merge procedures are notoriously hard to program. IceCube extends these ideas by pulling static constraints out of the dependency check and the merge procedure, in order to search for an optimal schedule, reconciling in cases where Bayou would find a conflict. IceCube's alternatives are less powerful than merge procedures, but provide more information to the scheduler and are easier to use.

Lippe et al. [11] search for conflicts exhaustively comparing all possible schedules. Their system examines all schedules that are consistent with the original order of operations. A conflict is declared when two schedules lead to different states. Conflict resolution is manual. Examining all schedules is untractable for all but the smallest problems.

Phatak and Badrinath [12] propose a transaction management system for mobile databases. A disconnected client stores the read and write sets (and the values read and written) for each transaction. The application specifies a conflict resolution function and a cost function. The server serialises each transaction in the database history based on the cost and conflict resolution functions. As this system uses a brute-force algorithm to create the best ordering, it does not scale to a large number of transactions.

IceCube is inspired by Kermarrec et al. [9]. They were the first to distinguish static from dynamic constraints. However their engine only supports Order (not MustHave), does not distinguish between log and object constraints, and does not have clean logs. Most importantly, an exhaustive search algorithm like theirs cannot not scale beyond very small log sizes.

9 Final remarks

For an environment where concurrent writes to shared objects cannot be neglected, we presented a general-purpose, semantics-aware reconciliation scheduler that differs from previous work in several key aspects. Our system is the first to approach reconciliation as an optimisation problem and to be based on the true constraints between actions. We present novel abstractions that enable the concise expression of semantics of these constraints. This simplifies the development of applications using reconciliation, as demonstrated by several prototype applications, and enables the reconciler to deliver high-quality solutions efficiently: although reconciliation is NP-hard, our heuristics find near-optimal solutions in reasonable time, and scale to large logs. Finally, IceCube is application-independent, and bridges application boundaries by allowing actions from separate applications to be related by log constraints and reconciled together.

Designing an application to be tolerant of disconnected operation and reconciliation still remains a demanding intellectual task, but our system has simplified this problem and provides a general tool so that application developers need not develop their own reconciliation mechanism.

The source code for IceCube is available from URL `http://research.microsoft.com/camdis/icecube.htm`.

Acknowledgements

We thank François Fages for studying the complexity of reconciliation and proposing alternative solutions. Peter Druschel, Anne-Marie Kermarrec and Antony Rowstron provided much of the initial design and implementation.

References

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

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

[3] M. Crispin. Internet Message Access Protocol, version 4rev1. Request for Comments 2060, IETF, December 1996.

[4] François Fages. A constraint programming approach to log-based reconciliation problems for nomadic applications. In *Proc.* 6^{th} *Annual W. of the ERCIM Working Group on Constraints*, June 2001.

[5] Terry Gray. Status of Disconnected Operation in IMAP, July 1996. `http://www.imap.org/papers/docs/disc-status.html`.

[6] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Tatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

[7] Leonard Kawell Jr., Steven Beckhart, Timoty Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *2nd. Conf. on Comp.-Supported Coop. Work*, Portland OR (USA), September 1988.

[8] Peter J. Keleher. Decentralized replicated-object protocols. In *18th Symp. on Princ. of Distr. Comp. (PODC)*, Atlanta, GA, USA, May 1999.

[9] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of diverging replicas. In *20th Symp. on Princ. of Distr. Comp. (PODC)*, Newport, RI, USA, August 2001.

[10] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter Tech. Conf.*, New Orleans, LA, USA, January 1995.

[11] E. Lippe and N. van Oosterom. Operation-based merging. *ACM SIGSOFT Software Engineering Notes*, 17(5):78–87, 1992.

[12] Sirish Phatak and B. R. Badrinath. Transaction-centric reconciliation in disconnected databases. In *ACM MONET*, July 2000.

[13] Nuno Pregui ca, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge (UK), May 2002.

[14] Norman Ramsey and Előd Csirmaz. An algebraic approach to file synchronization. In *9th Int. Symp. on the Foundations of Softw. Eng.*, Austria, September 2001.

[15] Yasushi Saito and Marc Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, Hewlett-Packard Laboratories, March 2002.

[16] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Conf. on Comp.-Supported Cooperative Work (CSCW)*, page 59, Seattle WA (USA), November 1998.

[17] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th Symp. on Op. Sys. Principles*, Copper Mountain CO (USA), December 1995. ACM SIGOPS.

[18] Haifeng Yu and Amin Vahdat. Combining generality and practicality in a Conit-based continuous consistency model for wide-area replication. In *21st Int. Conf. on Dist. Comp. Sys. (ICDCS)*, April 2001.