Efficient semantics-aware reconciliation for optimistic write sharing

Nuno Preguiça
Departamento de Informática,
Universidade Nova de Lisboa, Portugal;
`nmp@di.fct.unl.pt`
Marc Shapiro
Microsoft Research Ltd., Cambridge, UK;
`Marc.Shapiro@acm.org`
Caroline Matheson
Microsoft Research Ltd., Cambridge, UK

17 May 2002
Technical Report
MSR-TR-2002-52

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

Distributed systems replicate shared data in order to improve read performance and availability. An optimistic protocol allows a user to also *update* a local replica without co-ordinating with other replicas. This improves write availability in the presence of high network latencies, failures, or parallel development, but allows replicas to diverge. This is especially useful when using mobile devices with intermittent connectivity.

Repairing divergence after the fact, called *reconciliation*, combines the isolated updates [17]. 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*.

Sometimes a set of updates *conflicts*, i.e., running them all would violate an invariant. Most systems avoid the violation by *dropping* one or more of the updates. For this reason disconnected updates are said tentative.

Dropping an action may have a high impact. Think of a calendar system dropping an important meeting, or a sales-support tool for a travelling salesman that would drop an order.

1.1 Limitations of existing systems

Many existing reconciliation systems (Section 7) are dedicated to a single application semantics.

Others are general-purpose and use a simple syntactic criterion such as timestamps to decide decide which updates to retain, and in what order to run them. This is inflexible and causes spurious conflicts. Consider the example of Figure 1, where two users make meeting requests to 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 in some simple way 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 is dropped. Running Log 2 first, or interleaving the two logs, has a similar problem. To satisfy all three requests requires reordering them, which syntactic systems can't do.

1.2 Challenges

Our goal is to provide a general-purpose system for optimistic write sharing. In this paper we focus on the challenges of scheduling, i.e., of deciding which actions a reconciler should run, and in what order to run them.

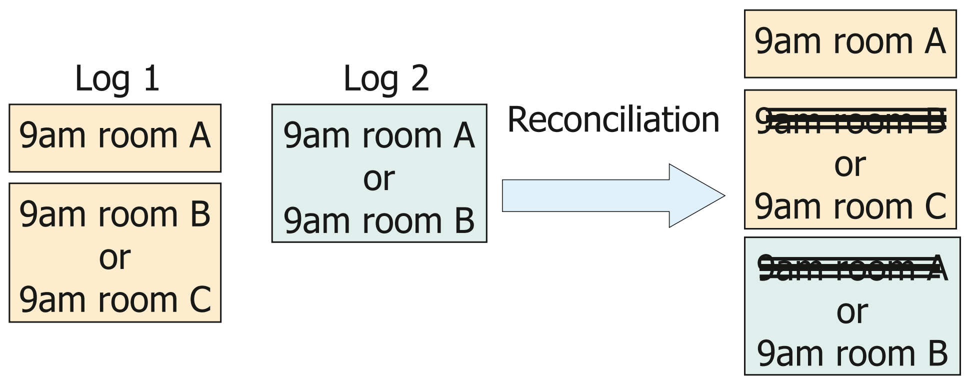


Figure 1: Syntactic scheduling spuriously fails on this example

Being excessively conservative and dropping actions spuriously, when there is no true conflict, should be avoided. One challenge then to minimise the number (or the value) of dropped actions. Therefore IceCube approaches scheduling as an optimisation problem.

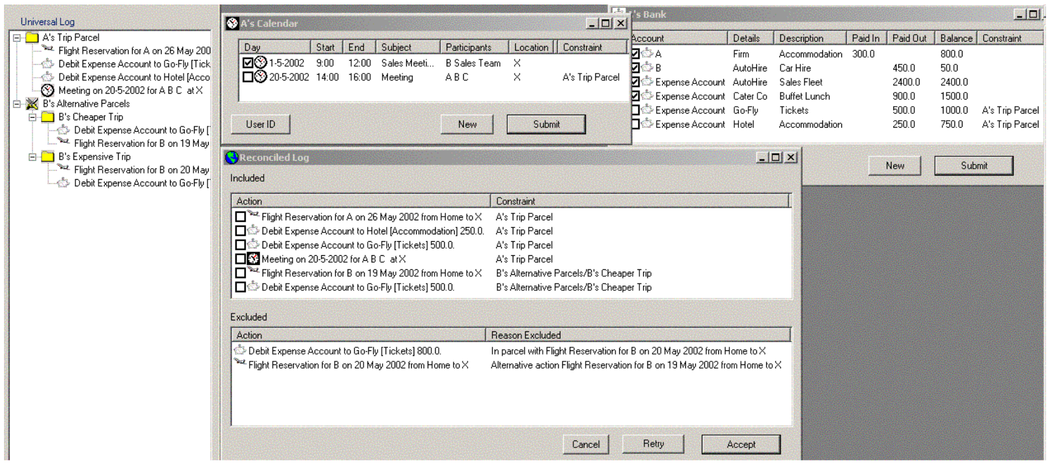
Only conflicting actions should be dropped; therefore the system must know something about the invariants to be respected, i.e., about the semantics. The second challenge for IceCube is to design an interface that is expressive enough and sufficiently abstract to support a wide range of applications. Our choice for IceCube is to abstract invariants as constraints.

A subclass of constraints, static constraints, direct the search of the optimisation engine. This helps answer the third challenge: efficient and scalable reconciliation. Furthermore, a number of techniques described hereafter further optimise IceCube's performance. Results given hereafter show that IceCube reconciles in reasonable time and scales nicely to large logs.

A final challenge is to demonstrate the practicality of the approach. We report in this paper on our experience coding a number of useful applications.

1.3 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. We present some


Universal Log

- A's Trip Parcel
  - Flight Reservation for A on 26 May 200
  - Debit Expense Account to Go-Fly [Tick
  - Debit Expense Account to Hotel [Acco
  - Meeting on 20-5-2002 for A B C at X
- B's Alternative Parcels
  - B's Cheaper Trip
    - Debit Expense Account to Go-Fly [
    - Flight Reservation for B on 19 May
  - B's Expensive Trip
    - Flight Reservation for B on 20 May
    - Debit Expense Account to Go-Fly [

A's Calendar

Day	Start	End	Subject	Participants	Location	Constraint
✓ 1-5-2002	9:00	12:00	Sales Meeti...	B Sales Team	X	
☐ 20-5-2002	14:00	16:00	Meeting	A B C	X	A's Trip Parcel

's Bank

Account	Details	Description	Paid In	Paid Out	Balance	Constraint
✓ A	Firm	Accommodation	300.0		800.0	
✓ B	AutoHire	Car Hire		450.0	50.0	
✓ Expense Account	AutoHire	Sales Fleet		2400.0	2400.0	
✓ Expense Account	Cater Co	Buffet Lunch		900.0	1500.0	
☐ Expense Account	Go-Fly	Tickets		500.0	1000.0	A's Trip Parcel
☐ Expense Account	Hotel	Accommodation		250.0	750.0	A's Trip Parcel

Reconciled Log

Included

Action	Constraint
☐ Flight Reservation for A on 26 May 2002 from Home to X	A's Trip Parcel
☐ Debit Expense Account to Hotel [Accommodation] 250.0.	A's Trip Parcel
☐ Debit Expense Account to Go-Fly [Tickets] 500.0.	A's Trip Parcel
☐ Meeting on 20-5-2002 for A B C at X	A's Trip Parcel
☐ Flight Reservation for B on 19 May 2002 from Home to X	B's Alternative Parcels/B's Cheaper Trip
☐ Debit Expense Account to Go-Fly [Tickets] 500.0.	B's Alternative Parcels/B's Cheaper Trip

Excluded

Action	Reason Excluded
Debit Expense Account to Go-Fly [Tickets] 800.0.	In parcel with Flight Reservation for B on 20 May 2002 from Home to X
Flight Reservation for B on 20 May 2002 from Home to X	Alternative action Flight Reservation for B on 19 May 2002 from Home to X

Figure 2: Multi-application travel scenario

measurements of write concurrency, and evaluate performance and quality of IceCube in Section 6. Section 7 discusses related work, and Section 8 summarises the conclusions and lessons learned.

2 System model and scenario of usage

2.1 System model

IceCube is intended as a component of a distributed system supporting disconnected operation. On each computer, a number of applications allow users to access the data shared with other computers. There may be several applications, each one accessing several sets of shared data. Each computer has its own replica of the shared data. Assume that all replicas start in an identical initial state.

While disconnected, an application accesses its local replicas, and records the actions describing its updates in a local log.

Eventually the computers reconnect. One site collects all the logs and submits them to IceCube. IceCube proposes one or more schedules combining the actions in the log, and replays them against the initial state of the shared data. The user commits one of the proposals. The committed schedule propagates to all machines, leading to a new common state of all replicas.

The system model has some obvious limitations. Although updating is peer-to-peer (any user can write without co-ordinating with the others), reconciliation is centralised. Reconciliation is not incremental and involves user intervention. In the conclusion, we present preliminary ideas for future work to overcome the limitations.

A schedule takes into account semantic constraints set by a user, an application, or a shared data type. The user can convey his intents, for instance may require that a request made via a flight reservation application be tied to another one made via a banking application. This is useful if the latter is a payment for a flight reserved in the former. The application may combine primitive actions into higher-level commands. For instance, the bank application may tie a credit action to a debit action to form a bank transfer. A shared data type may impose correctness conditions; for instance airline database forbids different users from reserving the same seat.

2.2 A multi-application, multi-data scenario

To give a feel of how IceCube operates, we now present a typical user scenario. Consider two users A and B planning a business trip to location X to meet another user C. They use three different applications: a calendar to organise the meeting, a flight-reservation system for their travel, and a bank account manager to pay for expenses. The users update their local replica of the the corresponding databases; when they reconnect they may experience conflicts such as double bookings, insufficient funds, or a cancelled flight.

Figure 2 shows a screenshot of this scenario from A's perspective. The top middle window shows A's calendar and the top right one A's bank account. Each line represents an action; ticked actions are those that have been committed in a previous reconciliation run, and the unticked actions are A's tentative requests. In the calendar window, A requests a meeting with B and C in location X on 20 May 2002 at 14:00; in the bank window there are two payment requests, for tickets and accomodation. If A's airline reservation window was visible it would show a flight request for 26 May. The rightmost column in an application window displays any constraints imposed on the actions; in this instance, all of A's tentative actions are in a "parcel," indicating that the schedule must contain all of them or none. Through the graphical user interface, A can add or remove constraints, even across different applications.

B has similarly arranged travel and payment, but has specified two al-

```
interface Action {                            // Java
   // preCondition has no side effects
   public boolean preCondition (ReplicatedState state);

   // execute modifies state; may save undo info; return postcondition
   public boolean execute (ReplicatedState state, UndoData info);

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

   // Do actions commute?
   public boolean commute (Action otherAction);

   // Do (overlapping, non-commuting actions) conflict?
   public boolean mutuallyExclusive (Action otherAction);

   // Is there a successful ordering of these (overlapping,
   // non-commuting, not mutually exclusive) actions?
   public int bestOrder (Action otherAction);

   // Return cluster identifiers
   public Object[] clusterIds ();
}
```

Figure 3: The action callback API

ternative possibilities, requesting the scheduler to help him choose between a cheap option, flying on 19 May, and a more expensive option on 20 May.

In this snapshot, A has asked to collect the logs and reconcile. The leftmost pane displays a tree representing the union of A's and B's logs. The first branch contains A's parcel. The second branch shows B's two alternatives, each of which is itself a parcel.

Finally, the bottom middle window displays the output from the reconciler. The top pane contains the actions included in the proposed schedule, the bottom one the actions dropped and the reason why they were dropped. As can be seen, the scheduler proposes to accept B's cheaper alternative and therefore to drop the expensive one. In this particular example there are no conflicts. At this point user A has a choice between three possibilities. He can either press the Cancel button to return to disconnected mode and continue adding (or removing) actions or constraints to his log; additionally he can select certain actions to force their inclusion in the next schedule (or exclusion from it). He can click on Retry to ask the scheduler to propose a different solution. Or he can choose Accept to commit this schedule, propagate it to B's machine to be replayed there, and iterate the process.

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 Actions and logs

An action is a tentative operation, executed by a program operating on an isolated replica, in order to update the shared state. The IceCube API has primitives for creating an action, recording it in the local log, and registering log constraints (defined later). An action is a Java object implementing the methods of Figure 3, which are call-backs invoked by the IceCube system during the reconciliation process. Their meaning will be explained later, as they become needed.

3.2 Shared data

A set of data managed by IceCube is abstracted as in the `ReplicatedState` class. An application developer provides a `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 subset of linearly-ordered *committed* checkpoints.

3.3 Constraints

A brute-force search through all possible schedules is infeasible; IceCube uses *static constraints* to limit the scope of search. Even so, the problem's complexity warrants a heuristic solution.

A static constraint is one that can be verified independently of the shared state. Furthermore, executing an action is subject to *dynamic constraints*, viz., its pre-condition and post-condition, verified against the current state of the shared objects. The heuristics take into account past dynamic constraint violations.

`predecessorSuccessor`(a, b)	$a \rightarrow b \wedge b \Rightarrow a$
`parcel`(a, b)	$a \Rightarrow b \wedge b \Rightarrow a$
`alternative`(a, b)	$a \rightarrow b \wedge b \rightarrow a$
`mutuallyExclusive`(a, b)	$a \rightarrow b \wedge b \rightarrow a$
`bestOrder`(a, b)	$a \rightarrow b$

Table 1: Static constraint equivalence relations

The system implements two primitive static constraints, order, noted $\rightarrow$, and implies, noted $\Rightarrow$. Relation $a \rightarrow b$ states that, if actions a and b are both scheduled, then a must be scheduled before b (although not necessarily immediately before). Relation $a \Rightarrow b$ means that, if a is scheduled, then b must also be (but not necessarily in that order nor contiguously) and must execute successfully.

The $\rightarrow$ relationship must be acyclic in any schedule. If cycles occur, they must be broken by dropping one or more actions. 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 [7]; 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 Ice-Cube API wraps them into higher-level abstractions, log constraints and object constraints, which we explain in the next sections. Table 1 shows the mapping between the two levels.

3.3.1 Log Constraints

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

The `predecessorSuccessor` constraint states that the successor action may be executed only after success of the predecessor (but not necessarily immediately after). This is useful when the predecessor produces some effect used by the successor. As an example consider that the user wants to copy a file after changing it. The write to the file must be a predecessor of the action that copies the file.

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, alternatives provide the scheduler with a fallback solution.

The `parcel` relation constitutes an all-or-nothing grouping — either all of its actions are executed successfully, or none are. (Unless otherwise constrained, actions of a parcel can run in any order, possibly interleaved with other actions.) Parcels make it possible to compose primitive actions into more complex ones. For instance, in our mail management application RMF (Section 5.3), moving a message between folders is a parcel unlinking it from its old location and linking it to the new one. Decomposing a complex action into more primitive ones makes it easier to reason about the correctness of constraints [15].

3.3.2 Object constraints

An *object constraint* expresses relations between classes of actions, and reflects the static semantics of shared data. Object constraints are computed by comparing each action with every other at the beginning of scheduling. Pairs of actions support the following comparison methods:

1. `commute`: returns true if the two actions commute. Actions that operate on different data objects generally commute; for instance debits to two different accounts. Often operations on the same object commute as well, for instance credits to the same account.
2. `mutuallyExclusive`: returns true if running both actions will violate an invariant. For example, in a file system, an action creating a directory with name N and another creating a file with the same name N are mutually exclusive.
3. `bestOrder`: Actions for which there is a favorable execution order return 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." This method is called only for pairs that are not mutually exclusive.

`commute` is not properly a constraint (indeed it indicates absence of constraints) but is included in this interface as an optimisation. The system avoids calling `mutuallyExclusive` and `bestOrder` for action pairs that commute. As we shall see in Section 4.3, commutability information enables the clustering optimisation. As a further optimisation, commutability is evaluated in several passes, from a coarse granularity to a fine one. The rest of this paper ignores this feature.

Object constraints	credit / credit	debit / debit	debit / credit
Same account	`commute`	`commute`	`bestOrder`
Different accounts	`commute`	`commute`	`commute`

Dynamic constraint: no overdraft

Table 2: Bank constraints

3.3.3 Discussion

In the absence of further constraints, commuting actions can be scheduled in any relative order; this is captured by the following relation:

$$\texttt{independent}(a,b) = \neg(a \rightarrow b) \wedge \neg(b \rightarrow a) \wedge \texttt{commute}(a,b)$$

Alternatives and `mutuallyExclusive` both cause choice of a single action from a set. However choosing among alternatives is normal behaviour, whereas `mutuallyExclusive` is reported back as a conflict.

If there is a best order between actions, they are always scheduled in that order.

As an example, Table 2 shows the object constraints in a banking application with credit and debit actions. More examples are given in Section 5.

4 Reconciliation scheduler

Static constraints limit the size of the scheduling space, but the problem remains inherently complex. Therefore our scheduler searches heuristically through the static constraint space. As it generates a schedule, it immediately executes it and checks the dynamic constraints. For improved performance, the search space is partitioned into independent subproblems. We now describe the design and implementation of the corresponding algorithms. Later (in Section 6) we benchmark their efficiency and quality.

4.1 Heuristic search

This scheduler performs efficient heuristic sampling of small portions of the search space. After a schedule has succeeded, or a partial schedule has failed, an unrelated portion of the search space is selected. Schedule selection and execution are repeated until some user-selected satisfaction criterion is satisfied.

Sampling combines search heuristics and randomisation. Initially each action is assigned a merit, estimated from its static constraints. Every time an action is selected and executed in 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; the selection procedure is called `selectActionByMerit` hereafter. This incremental approach circumvents the need for expensive graph analysis, as in Kermarrec et al. [10]. The downside is that we cannot guarantee that the system finds the absolute optimum.

The merit estimator computes the benefit of adding a given action to a given partial schedule. The lower the value of other actions this action prevents from being scheduled in the future, the higher its merit. More precisely, the merit of any 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, when an action fails dynamically, its merit drops drastically in the near future.

The merit estimator executes in constant time, thanks to summary information computed and updated during scheduling (see next section).

4.2 Basic algorithm

Our reconciliation algorithm is displayed in pseudo-code[1] in Figures 4 and 5.

Procedure `reconcile` implements the complete reconciliation algorithm. Initially `computeInfo` computes a summary of static constraints, to be used in estimating the merit of each action. It calls `scheduleOne` repeatedly and remembers the highest-value schedule. It terminates when some application-specific selection criterion `happy` is satisfied — often a value threshold, a maximum number of iterations, or a maximum execution time. For the benefit of `happy` and as feedback to users, every time an

[1]Indented text indicates a nested block. Arguments and return values are passed by value. A procedure can return multiple values.

```
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
       toExclude := implyingSet (nextAction)
       toAbort := INTERSECTION (schedule, toExclude)
       IF NOT EMPTY (toAbort)
       THEN
          SIGNAL dynamicFailure (goodActions \ toExclude)
       ELSE
         summary.updateInfoFailure (actions, toExclude)
         actions := actions \ toExclude
         LOOP
     postcondition := nextAction.execute (state)
     IF postcondition = TRUE
     THEN // action succeeded
       actions := actions \ beforeSet( nextAction)
       summary.updateInfo (actions, nextAction)
       schedule := [schedule | nextAction]
       value := value + nextAction.value
     ELSE // post-condition failure
       toExclude := implyingSet (nextAction)
       SIGNAL dynamicFailure (goodActions \ toExclude)
   RETURN { state, schedule, value }
```

Figure 4: Executing a single schedule

```
reconcile (state, log1, log2) =
  state.setCheckpoint (init)
  bestValue := 0
  bestSchedule := []
  allActions := union (log1, log2)
  goodActions := allActions
  summary := computeInfo (allActions)
  DO
    summary.resetInfo (goodActions)
    state.restoreCheckpoint (init)
    { state, schedule, value } :=
        scheduleOne (state, summary, goodActions)
      CATCH dynamicFailure (goodActions)
        LOOP
    IF value > bestValue
    THEN bestSchedule := schedule
        state.setCheckpoint (best)
    // consider failed actions again
    goodActions := allActions
  UNTIL happy (bestValue)
  state.restoreCheckpoint (best)
  RETURN { state, bestSchedule, value }
```

Figure 5: Selecting best schedule (no clustering)

action is dropped from a schedule the scheduler generates an appropriate justification. For simplicity this is omitted from the pseudo-code.

4.2.1 Failure-free execution

The `scheduleOne` procedure creates a single schedule and executes it against the given data state. Its inputs are the data state and a set `goodActions` of actions to be reconciled. It returns a successful schedule, the value of the schedule and a new state, or exits with a `dynamicFailure` signal returning a reduced `goodActions` set.

The main loop repeatedly selects an action to add to the schedule selected so far, tests its pre-condition, executes its body and tests the post-condition. The selection procedure `selectActionByMerit` applies the heuristics explained previously in Section 4.1.

If both the action's precondition and the execute method return success, the action and any non-scheduled action that should have been executed before it (`beforeSet`) is removed from the set of candidates `actions`, `updateInfo` updates the summary of static constraints, the action is appended to the schedule, the value of the schedule is incremented, and the loop iterates.

4.2.2 Dynamic constraint violation

When there is a dynamic failure, the action that causes the failure and its implying-set are removed from the set `goodActions` of candidates for consideration for the next execution of `scheduleOne`.[2] The rationale is to avoid experiencing the same failures again, thus guaranteeing that some solution is reached, albeit a sub-optimal one. Once a solution has been found, these actions are put back into `goodActions` for consideration again.

If the precondition fails, we remove this action and its implying-set from the set of candidates `actions`. Procedure `updateInfoFailure` updates the summary information accordingly. If the implying-set contains an already-executed action (e.g., this action belongs to a partially executed parcel): (i) the current schedule is discarded; (ii) the actions that cause the problem and their implying-set are removed from `goodActions`, and (iii) `scheduleOne` aborts with a `dynamicFailure` signal, causing the state to roll back. Otherwise, as a pre-condition has no side effects, it is safe to iterate.

If the post-condition fails, the action and its implying-set are removed from `goodActions`. The state may be invalid, so execution aborts.

Note how `reconcile` uses checkpoints to record states and roll back. When a schedule fails that might have modified the state, we restore the initial state with `restoreCheckpoint (init)`. We also use checkpointing to record the state produced by the best schedule so far with (`setCheckpoint (best)`). A developer can implement checkpointing either by cloning `ReplicatedState` or by using an `undo` mechanism that we supply.

4.2.3 Complexity

The cost of of `scheduleOne` is dominated by `selectActionByMerit`. The overall complexity is $O(n^2)$, where n is the size of `allActions`. Intuitively, order n actions are scheduled, and for each one `selectActionByMerit` considers all the remaining ones.

The merit of an action evaluates in constant time thanks to the summary of static constraints. Initially it is computed by `computeInfo`, of complexity $O(n^2)$ because it compares every action with every other. When a very

[2]For some action a, `implyingSet`$(a) = \{b | b \Rightarrow a\}$. It is the set of actions that cannot be scheduled if a is dropped. By obvious extension to a set, `implyingSet`$(A) = \{b | \exists a \in A, b \Rightarrow a\}$.

```
reconcileWithClustering (initState, log1, log2) =
   clusters := clusterize (UNION (log1, log2))
   schedule := []
   value := 0
   state := initState
   FOR EACH c IN clusters
      { partialState, partialSchedule, partialValue } :=
              reconcile (state, c.log1, c.log2)
      state := partialState
      schedule := [ schedule | partialSchedule ]
      value := value + partialValue
   RETURN { state, schedule, value }
```

Figure 6: Selecting best schedule, with clustering

small number of schedules is created the relative contribution of `computeInfo` to total execution time increases.

The `reconcile` algorithm could be made $O(n \log n)$ by restricting the merit function and by using priority queues to maintain merits. The overall complexity would remain $O(n^2)$ because of `computeInfo`.

4.3 Clustering

As the complexity of `scheduleOne` is polynomial in the size of its inputs, it helps to partition the problem into smaller, independent problems. We partition actions into mutually-independent and disjoint subsets, called clusters. Reconciling the initial logs is then equivalent to reconciling each individual cluster in some sequential order.

Actions that are `independent` of one another can be in different partitions, but only if neither implies the other. This is in order to limit the scope of a roll-back when `scheduleOne` signals a dynamic constraint violation. Clustering a set of actions A yields a set of clusters $c_1, ..., c_k$ that satisfy the following relation:

$$\begin{aligned} & A = c_1 \cup c_2 \cup \ldots \cup c_k \wedge \forall i, j, a \in c_i, b \in c_j, \\ & i \neq j \Rightarrow c_i \cap c_j = \emptyset \wedge \\ & \qquad \texttt{independent}(a, b) \wedge \neg(a \Rightarrow b) \wedge \neg(b \Rightarrow a) \end{aligned}$$

Figure 6 shows pseudo-code for our reconciliation algorithm using clustering. The set of actions to reconcile is clustered by `clusterise`. The reconciliation is executed incrementally calling `reconcile` sequentially, once for each cluster.

Clustering reduces complexity considerably; `reconcileWithClustering` is $O(|c_1|^2 + \ldots + |c_k|^2)$ (instead of $O(n^2)$ for `reconcile`). The performance measurements presented later are consistent with this estimate.

A straightforward clustering algorithm compares each action with every other, of complexity $O(n^2)$. For improved performance we combine this with a more coarse-grain algorithm of expected complexity $O(n)$. Here, an action returns a set of arbitrary identifiers (possibly identifying objects that the action modifies). Actions with no common identifiers are classified in different clusters. Multiple identifiers allow partitioning along different properties — e.g., a bank action might return the identifier of both the account and the branch involved. A coarse-grain cluster identified in this way is then partitioned again using the polynomial comparison-based algorithm.

5 IceCube applications

In this section we describe some applications implemented with IceCube: the demonstration application from Section 2.2, a calendar application used as a benchmark (evaluation results are the subject of Section 6), a larger mail folder application that interfaces to legacy applications, and a file system application. This is to give a flavour of how the IceCube abstractions are used in practice and of the complexity of building an application, and to convey some lessons learnt from the experience.

5.1 Multi-Application Demonstration

The multi-application demonstration presents an intuitive scenario that exemplifies the need for reconciliation across multiple applications. Log constraints are used to combine and define dependencies among actions from several applications and to specify alternative strategies to solve conflicts. In the example presented in Section 2.2, the plans to attend a meeting are expressed as a hierarchy of parcels and alternatives that bridge the three applications: calendar, bank and flight reservations.

Each application presents different intrinsic difficulty to the reconciliation process. Calendar actions that overlap in time and have a common participant simply exclude each other, which means that a "best" consistent schedule can potentially be determined from the static constraints. Bank actions (credit or debit) and flight reservations interact more subtly, and may fail dynamically because of insufficient funds or over-booking (it is impos-

Object constraints	add/add	remove/remove	remove/add
Same user&time	mutuallyExclusive	commute	bestOrder
Other user, time	commute	commute	commute

Dynamic constraint: none

Table 3: Calendar constraints

sible to express these constraints statically). Furthermore, we permit the committed state of the flight reservation system to diverge asynchronously from its replicas by the execution of actions not included in reconciliation (flights may be booked by some outside agency).

Finally, the scenario provides an effective example of reconciliation in the case where the engine can not automatically decide among multiple solutions. If conflicts arise it is essentially a human decision to decide which actions should take precedence, and consequently the reconciliation process must be guided by the user. During reconciliation, the user is permitted to select actions for preferential inclusion (or exclusion), thus ensuring a rapid approach to an acceptable schedule.

5.2 Calendar application

The calendar application maintains a shared calendar accessed concurrently by multiple users. A user can `request` a meeting, proposing one or more alternative times, or `cancel` a previous request.

A shared calendar consists of a set of entries, each describing an appointment: time, duration, participants, and location. The low-level actions add an appointment (the `add` action) or delete one (`remove`). The user-level commands are mapped in the obvious way onto `alternatives` of low-level ones.

The calendar uses IceCube's log cleaning mechanism (not described in this paper) to eliminate redundant pairs of `add` and `remove`. As illustrated in Table 3 the object constraints forbid double-booking a person or a room. `bestOrder` makes `remove`s execute at the beginning of the schedule, increasing the probability that `add`s can be accommodated.

The shared calendar is implemented in about 630 lines of code, blank lines and comments excluded (LOC). Each action averages 60 LOC. The calendar application totals approximately 880 LOC. This application was used as one of our performance and quality benchmarks, as we report in

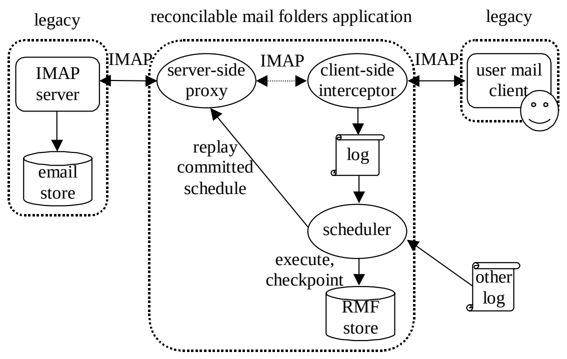


Figure 7: Structure of Reconcilable Mail Folders application.

Section 6. A further 250 LOC control the experiment, submitting requests and calling the reconciler.

5.3 Reconcilable Mail Folders

Our Reconcilable Mail Folders application (RMF) uses IceCube to manage replicas of e-mail folders and merge concurrent changes. RMF interposes between a mail client and a server by intercepting their communication using the standard IMAP (mailbox access) protocol [3]. The client and server are unmodified legacy systems; the server manages the storage of e-mail folders and messages.

As shown in Figure 7, an interceptor redirects IMAP interactions to the log. RMF's `ReplicatedState`, called hereafter `RMFState`, contains only meta-information about folders and messages. This avoids having to checkpoint the whole server state multiple times during reconciliation. Updates are reconciled initially against the `RMFState`, then the committed schedule executes against the IMAP server.

Message-related IMAP commands map directly onto a single RMF action. This is made possible by the simple semantics of most IMAP commands. For instance creating a new message never conflicts with another operation. Mail folder operations are more complex. Creating a mailbox is idempotent.[3] Renaming a folder rename is a parcel linking into the new location and unlinking from the old one.

[3]An idempotent action is one that has the same effect whether executed once or several times.

Conflicts include concurrently renaming the same mail folder with different names, changing a message while concurrently removing the folder, etc. Other concurrent events are reconciled; for instance one user may copy a message while another user discards the message.

Space limitations forbid any greater detail. RMF totals 2625 LOC approximately. RMF supports seven different actions, implemented in 590 LOC; about one-third of this represents repetitive code that could be generated automatically if we had better tools. The code size for `RMFState` is 505 lines. The rest of RMF adds up to approximately 1530 lines, which includes the IMAP interceptor, the proxy to execute a schedule in the IMAP server, and the code that controls reconciliation.

Our approach can be contrasted with disconnected operation as implemented by some IMAP clients [6]: when a user reconnects, his disconnected log is replayed against the current server state. While adequate for a single user, this causes trouble in some concurrent update situations. Suppose a user copies a message to another folder. Another user deletes the same message. If the second user's actions are replayed first, the copy fails. In contrast, in RMF the semantic ordering will ensure the copy is scheduled before the delete.

5.4 Reconcilable File System

Our Replicated File System (RFS) emulates a file system in memory, with the usual semantics. In many respects RFS is similar to RMF; we focus here on some differences and lessons learned.

RFS semantics are somewhat more complex than RMF; for instance the `move` command has nine different cases depending on the nature of the source and destination. There is also more opportunity for conflict, for instance, creating a file and a directory under the same name constitutes a conflict.

We decompose a user command into a parcel, first establishing a high-level assertion, then linking/unlinking nodes in the filesystem tree. For instance the `move` parcel first checks which of the nine possible cases to execute, then does the corresponding set of links and unlinks.

RFS is approximately 1660 LOC over 33 files. Most of the code is shared between a non-replicated and a replicated version. There are 9 concrete action classes; actions account for 500 lines in 6 files; much of this code is shared across several actions. 250 more lines over 3 files are specific to replication.

Currently we have no performance results for RFS, but we expect to include them in the conference version of this paper.

5.5 Application design

Our experience shows that using IceCube drastically simplifies the development of reconcilable applications. Indeed, developers do not need to create ad-hoc mechanisms for every different application. They only need to convey some simple facts about their application, and have tools to structure the application in the right way. However, the application must be designed to be aware of, and to tolerate, disconnected operation, roll-back, replay, and reconciliation, and this is by no means easy. 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. Use static constraints as much as possible, and avoid dynamic constraints; the former direct the search while the latter cause schedule execution to roll back. Individual actions should be simple, because in this way it is possible to characterise their static properties [15]. Also to enable static reasoning, the log should be clean, i.e., should contain no redundant actions.[4] Furthermore, the probability of successful reconciliation is increased when actions commute, and when they are idempotent.

The RMF experience shows how it is possible to interpose reconciliation inside 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 Measurements and evaluation

This section contains statistics collected on some CVS repositories, justifying the need for reconciliation, and two experiments evaluating the performance and scalability of IceCube.

6.1 Empirical observations of CVS repositories

Previous work [11, 16] on file systems shows that concurrent writes are rare. While true for the general file population, this conclusion can be mislead-

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

Trace	snmp	chord	ron	ucl	JOnAS	Jonathan	cm	rochester	hugo	JORM	OwS
Duration (days)	795	467	553	584	674	704	1099	232	390	186	412
Users	14	29	18	9	8	9	21	5	11	11	7
Non-concurrent M	8696	3808	1555	779	3398	1369	1824	1388	2593	2258	557
Merges G	1170	449	82	55	24	198	163	39	93	75	4
Collisions C	557	261	70	49	42	116	143	27	53	38	10
Concurrency (%)	16.57	15.71	8.90	11.78	1.91	18.66	14.37	4.54	5.33	4.77	2.45

Table 4: CVS statistics.

ing. The measurements hereafter show that, for sets of files designated as write-shared, the rate of concurrent access is far from negligable.

CVS (Concurrent Version System) [2] is widely used to manage replicated text files for concurrent code development.

A CVS repository is a collection of write-shared files. Each user has his own replica and edits it without co-ordinating with others. Occasionally he "updates" against the central repository to synchronise recent changes. When a single replica has changed, the repository incorporates the updates, and records the event by an "M" record in its history log. When two replicas have changed concurrently, CVS attempts to reconcile them. Concurrent changes to overlapping sets of text lines are considered conflicting, and must be resolved manually ("C" record). Disjoint changes are merged automatically, and are logged with a "G" record.

We analysed eleven history files, one (snmp) publicly available from SourceForge[5] [18], the others provided to us by various volunteers. They come mostly from academic institutions but vary in size and complexity.

Table 4 presents some statistics. Here, Duration is the time recorded in the history file. Concurrency is the ratio $(C + G)/(C + G + M)$, where C, G and M represent the number of corresponding records in the repository. Concurrency measurements only take into consideration those files that are actually editable (for instance, system files and derived files are excluded from the calculations).

Concurrency rates are far from negligeable, ranging from 1.91% to 18.66%. We also note that CVS's merging is relatively successful.

6.2 Benchmarking IceCube

In this section we present measurements to evaluate both the quality of reconciliation (by the size of the schedules), and its efficiency (by execution time).

[5]Most SourceForge projects have the history feature turned off.

This uses the calendar application described in Section 5.2. The benchmark inputs are based on a trace from actual Outlook calendars, modified to control the rate of conflicts and to include alternatives. 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 average of add alternatives per requestis two. Note that execution times include both system time (scheduling and checkpointing), and application time (executing the actions); however the latter is negligeable because the code is very simple.

In each cluster, the number of different adds across all actions is no larger than the number of Requests. For instance, in the example of Figure 1, in the three Requests, there are only three different adds ('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 said *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 averages 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.

6.2.1 Single cluster

We first evaluate the IceCube heuristics without clustering. Our first set of inputs gives birth to a single cluster; however execution time does include execution of clusterise.

Figure 8, compares IceCube with a syntactic scheduling algorithm, to justify our optimisation approach. We choose the simplest one, concatenat-

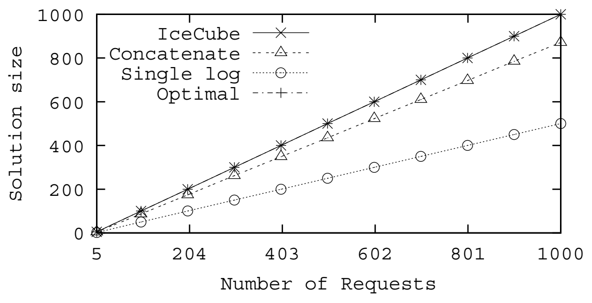


Figure 8: Size of reconciliation.

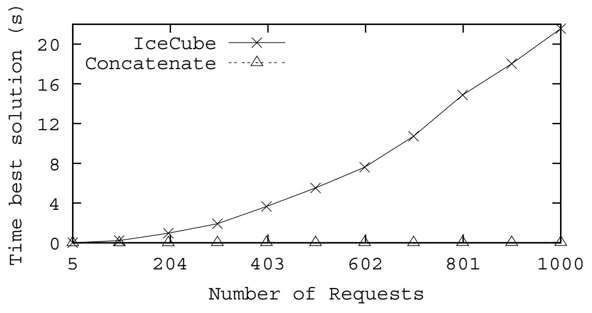


Figure 9: Execution time to solution (single cluster).

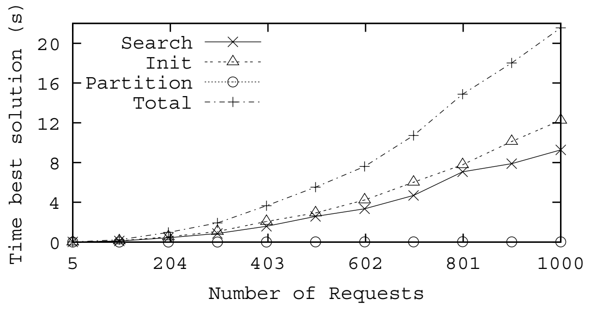


Figure 10: Decomposition of reconciliation time (single cluster).

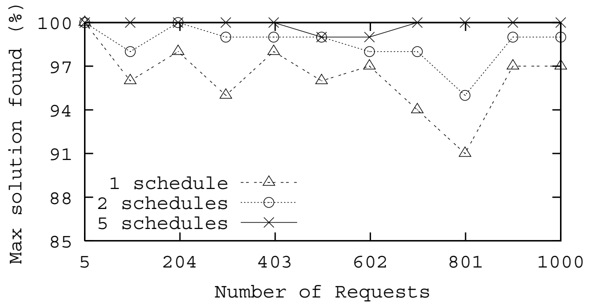


Figure 11: Percentage of cases where a max-solution was found in the first 1, 2 or 5 schedules created.

ing the logs, but note that any syntactic algorithm will have similar worst-case performance. For instance in Bayou [20] the schedule is ordered the way updates are received at certain designated servers, which may be exactly the same as concatenation order.

As expected, the results of semantic-directed search are better than syntactic ordering. In our tests, IceCube could always find the best solution. With syntactic ordering, approximately 12% Requests are dropped, contrasted to close to none for semantic-directed search. Compare against the baseline marked “Single log” in the graph, which represents the simplest non-trivial scheduler that guarantees absence of conflicts (it selects all actions from a single log and drops all actions from the other).

The drop rate grows very slightly with size. Although the improvement may appear small, remember that dropping a single action may have a high cost.

Figure 9 shows the execution time of `reconcile` vs. 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 and $O(n)$ for concatenation.

Figure 10 decomposes the execution time into its major parts. First, the time to cluster the actions; as expected, this represent a small fraction of the overall execution time. Second, the time to compute the initial summary of static constraints. Third, the search time, i.e., the time to create and execute the schedules (action execution time is negligeable). As expected, the latter two components are of the same order of magnitude.

In these examples, the best result is reached in a small number of schedules, allowing initialisation time to slightly dominate search time. Even if

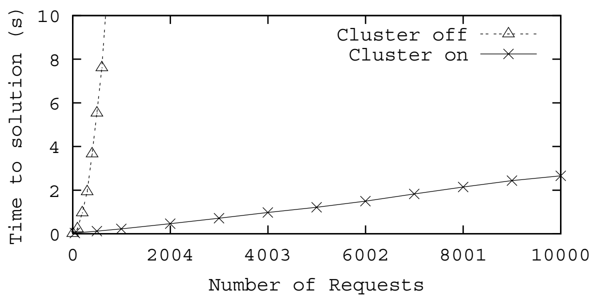


Figure 12: Execution time to solution (with and without clustering)

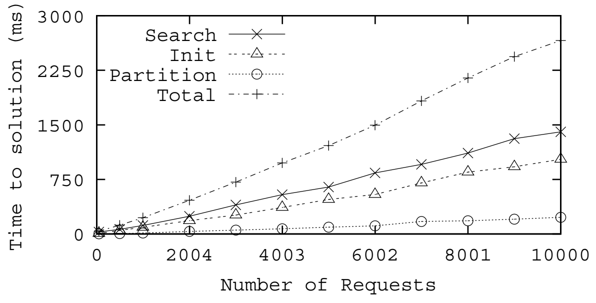


Figure 13: Decomposition of reconciliation time (multiple clusters).

the heuristic scheduler were made infinitely fast, a bottleneck would remain in initialisation time.

In these experiments, happy is designed to stop either when a max-solution is found, or after a given amount of time. Figure 11 shows how quickly a max-solution is reached. 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 tests. A related result is that in this experiment, even non-max-solutions were all within 1% of the max size.

6.2.2 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 Re-

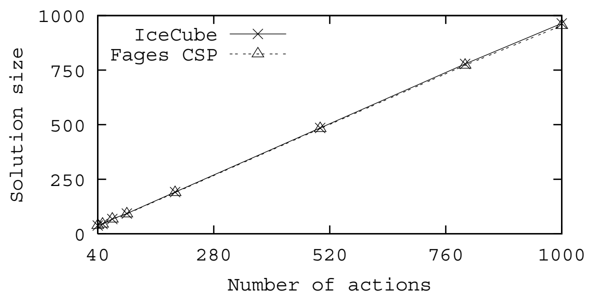


Figure 14: Solution size for Fages' benchmarks.

quests, and are constructed to that 25% of the adds can be clustered alone; 25% of the remaining adds are in clusters with two actions; and so on. Thus, as problem size increases, the size of the largest cluster increases slightly, as one would expect in real life. For instance, when the logs contain 1000 actions, the largest cluster contains the adds from 12 Requests, and 18 when the logs total 10000. 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 the optimal solution, whether clustering is in use or not. In contrast, using log concatenation, between 6% and 8% of the requests were dropped. (This value is smaller than in the previous section because a large fraction of the requests have zero or close to zero related actions; e.g., 25% of the actions commute with any other action, thus, they can be scheduled in any mutual order.)

Figure 12 shows the time to find a max-solution, with clustering turned on or off. As expected a solution is obtained much more quickly in the former case than the latter. 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 13, shows that all components of the reconciliation time grow linearly, as expected.

6.3 Comparison with Fages

Fages [4] solves reconciliation problems using constraint logic programming techniques; his study uses synthetic benchmark problems. Here we study how well IceCube solves Fages' benchmarks, and how well Fages' algorithm solves our calendar benchmark.

We ran IceCube against a subset of Fages' benchmarks. The constraints included in each problem are generated randomly with a density of 1.5 for each type of constraints, meaning that there are $1.5 \times$ size constraints of each type on average.

Figure 14 compares the quality of Fages' CSP solutions with IceCube's. The results are similar, but notice that IceCube performs slightly better on average on large problems. This shows that the IceCube heuristics perform well on Fages' inputs as well. Since the execution environment is very different, it would make no sense to compare 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' constraint-solving algorithm, execution time grows very quickly with problem size. For instance, for only 15 `requests`, Fages' algorithm cannot find a solution within an (arbitrary) timeout of 2 minutes. We believe that the problem lies in the existence of alternatives.

7 Related Work

Several systems use optimistic replication and implement some form of reconciliation for divergent replicas. Many older systems (e.g., Lotus Notes [8] and Coda [12]) reconcile by comparing final tentative states. Other systems, like IceCube, use history-based reconciliation, such as CVS [2] or Bayou [20]. Recent optimistically-replicated systems include TACT [21] and Deno [9]. Balasubramaniam and Pierce [1] and Ramsey and Csirmaz [15] study file reconciliation from a semantics perspective. Operational Transformation techniques [19] re-write action parameters to enable order-independent execution even when they do not commutate. 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 [17].

Bayou [20] 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 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 easier to use.

Lippe et al. [13] search for conflicts exhaustively comparing all possible schedules. Their system examines all schedules that are consistent with the original order of operations. A conflict (to be resolved manually) is declared when two schedules lead to different states. Examining all schedules is untractable for all but the smallest problems.

Phatak and Badrinath [14] 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.

The original IceCube is due to Kermarrec et al. [10]. They are the first to distinguish static from dynamic constraints. However their engine only supports $\rightarrow$ (not $\Rightarrow$), does not distinguish between log and object constraints, and does not scale as it exhaustively searches the static constraint space. The current system has a more powerful and easier-to-use API, and is much more efficient. The quality of the solutions is virtually indistinguishable from the original system.

8 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, Icecude 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.

Making IceCube work as a completely decentralised system is our main item of future work, the key issue being the problem of making reconciliation fully peer-to-peer. Our current prototype centralises reconciliation in the hands of a single user at some unique location. If however several users at different locations are to share the authority to commit, how are we to decide between possibly competing commitment decisions, and how should we to order them? In the limit this might entail a consensus over the whole distributed system.

A first possible solution is inspired by CVS [2], where different users share the responsibility of commit, but commits are serialised at the central repository. A more fully decentralised approach might distribute the responsibility over a small number of core sites, along the lines of Gray *et al.* [5], and make decisions by quorum consensus, as in the Deno system [9].

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

Acknowledgements

Many thanks to Yek Chong for his study of CVS repositories reported here. 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. `http://www.cis.upenn.edu/~bcpierce/papers/snc-mobicom.ps`.

[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] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. Dangers of replication and a solution. In *Int. Conf. on Management of Data*, pages 173–182, Montréal, Canada, June 1996.

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

[7] 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.

[8] 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.-Supp. Coop. Work*, Portland, OR, USA, September 1988.

[9] Peter J. Keleher. Decentralized replicated-object protocols. In *18th Symp. on Princ. of Distr. Comp. (PODC)*, Atlanta, GA, USA, May 1999. `http://mojo.cs.umd.edu/papers/podc99.pdf`.

[10] 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. `http://research.microsoft.com/research/camdis/Publis/podc2001.pdf`.

[11] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)*, 10(5):3–25, February 1992. `http://www.acm.org/pubs/contents/journals/tocs/1992-10`.

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

uary 1995. http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/usenix95.pdf.

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

[14] Sirish Phatak and B. R. Badrinath. Transaction-centric reconciliation in disconnected databases. In *ACM MONET*, July 2000. http://www.cs.rutgers.edu/~phatak/monet.ps.

[15] Norman Ramsey and Előd Csirmaz. An algebraic approach to file synchronization. In *9th Int. Symp. on the Foundations of Softw. Eng.*, Austria, September 2001. http://www.eecs.harvard.edu/~nr/pubs/sync-abstract.html.

[16] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *Usenix Conf.* Usenix, June 1994.

[17] Yasushi Saito and Marc Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, Hewlett-Packard Laboratories, March 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-33.html.

[18] SourceForge. http://sourceforge.net/.

[19] 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. http://www.acm.org/pubs/articles/proceedings/cscw/289444/p59-sun/p59-sun.pdf.

[20] 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*, pages 172–183, Copper Mountain, CO, USA, December 1995. http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf.

[21] 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. http://www.cs.duke.edu/~yhf/icdcsfinal.ps.