

Semantics-based reconciliation for collaborative and mobile environments

Nuno Preguiça¹, Marc Shapiro², and Caroline Matheson²

¹ Dep. Informática, FCT, Universidade Nova de Lisboa, Portugal

² Microsoft Research Ltd., Cambridge, UK

Abstract. IceCube is a system for optimistic replication, supporting collaborative work and mobile computing. It lets users write to shared data with no mutual synchronisation; however replicas diverge and must be reconciled. IceCube is a general-purpose reconciliation engine, parameterised by “constraints” capturing data semantics and user intents. IceCube combines logs of disconnected actions into near-optimal reconciliation schedules that honour the constraints. IceCube features a simple, high-level, systematic API. It seamlessly integrates diverse applications, sharing various data, and run by concurrent users. This paper focus on 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 order for collaborative users to contribute to the common task or coordinate, they must be able to update their replicas of the shared information. Furthermore, in mobile environments, mobile users need to access shared data during disconnection periods or to face slow or expensive networks. Thus local replicas may diverge and need to be *reconciled*. This is not trivial however because of conflicts.

Most existing reconcilers use syntactic mechanisms such as timestamps and drop actions to avoid conflicts. For instance in Figure 1, User 1 is requesting a reservation for room A and also for either B or C. A bit later User 2 requests either A or B. Reconciling in timestamp order reserves A and B for User 1, and User 2 cannot be satisfied. If instead the reconciler ignores timestamps but understands the meaning of “or” it can accomodate both users.

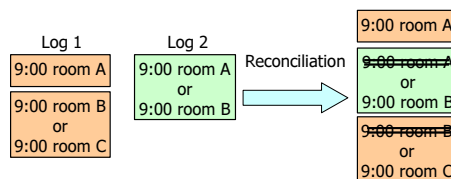


Fig. 1. Syntactic scheduling spuriously fails on this example

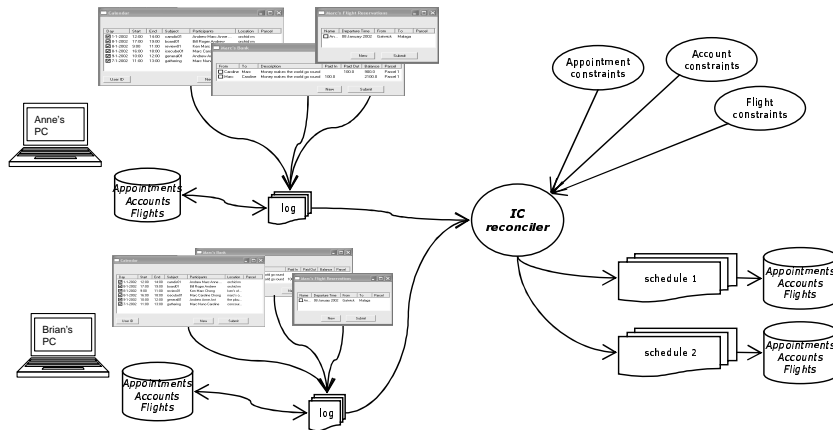


Fig. 2. IceCube system structure

The IceCube reconciler constitutes a general-purpose middleware that is flexibly parameterised by application semantics. It works seamlessly across users, applications and objects that need not be aware of each other’s existence.

A model IceCube environment is sketched in Figure 2. It shows two computers (Anne’s and Brian’s PCs), each with its own replicas of shared data (Appointments, Accounts, and Flights). An application can *tentatively* update a local replica [13]. Tentative updates are logged.

IceCube will combine the concurrent logs into sequential executions called *schedules*. In contrast to the inflexible schedulers of previous systems, IceCube obeys the application semantics, expressed by way of so-called *constraints*. Constraints constitute a general and powerful API for applications to express precisely their dependencies and invariants. By viewing scheduling as an optimisation problem, IceCube also avoids dropping actions unnecessarily.

Benchmarks show that IceCube reconciles in reasonable time and scales nicely to large logs, thanks to the following contributions. (i) Our *static* constraints, which incur no runtime cost, are sufficiently expressive for a wide spectrum of applications. (ii) The engine decomposes large inputs into independent sub-problems.

The IceCube approach may appear hard to use, but in our experience it is practical. We report on a number of useful applications that we have coded. IceCube simplifies application development. Furthermore, multiple applications and object types will reconcile consistently and seamlessly.

Table 1 summarises the different phases and the interactions between applications and IceCube, to which we will refer in the rest of this paper.

This paper is organised as follows. We present a usage scenario in Section 2. Section 3 discusses the basic data and action abstractions. Section 4 presents our central abstraction of constraints. The scheduler’s algorithms are explained in detail in Section 5. We evaluate performance and quality of IceCube in Section 7.

Tentative operation App → IceCube	Initialise reconciliation IceCube → App	Compute schedule IceCube → App	Commit IceCube → App
Tentative execution Record action in log Set log constraint	Collect object constraints	Take checkpoint Execute action Compensate action Return to checkpoint	Commit actions

Table 1. Summary of interface between applications and IceCube

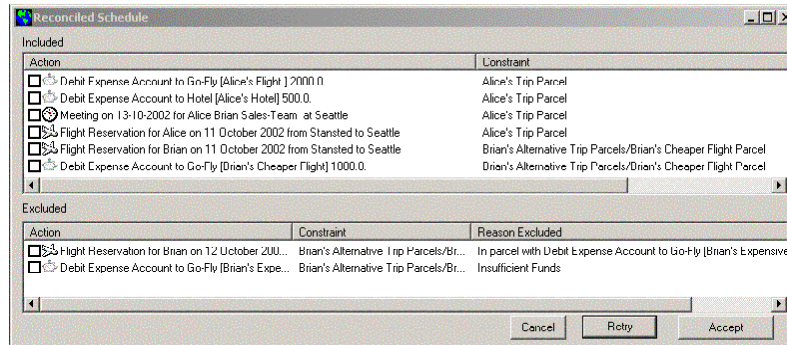


Fig. 3. Possible reconciliation for travel scenario

Section 8 discusses related work, and Section 9 summarises conclusions and lessons learned.

2 A multi-application, multi-data scenario

A simple scenario will give a feel of how IceCube operates. Anne and Brian are planning a business trip to meet some colleagues. While away from the network, Anne uses a calendar application to schedule a meeting, a travel application to reserve a flight to get there, and an account manager to pay for the flight. Anne groups her three actions into an atomic “parcel” (to be described shortly), even though they span separate applications.

Anne’s machine being disconnected, the local database replicas are necessarily partial and possibly stale. Disconnected updates are only tentative, against the local replicas. However, they will become effective after connecting and reconciling; for instance, her payments will be executed, when, and if, the flights are committed.

In the meantime, Brian wants the system to help him choose between two different possibilities, a convenient but expensive flight on 12 October, and a cheaper but less convenient one on 11 October. He gives a higher value to the former to indicate his preference. The two choices constitute an “alternatives” construct, each of which contains a “parcel” combining reservation and payment.

Other users are similarly updating the databases concurrently. When Anne and Brian reconcile, they may experience conflicts such as double bookings,

insufficient funds, or a full flight. Figure 3 presents an output from the reconciler: on top, a suggested non-conflicting schedule; the bottom lists the actions dropped from the schedule with explanations why dropped. In this case, Brian could not pay for the expensive flight, presumably because in the meantime a colleague has spent some of the travel account. The booking action in the same parcel is therefore dropped too. Instead, the cheaper flight alternative is scheduled. If Anne presses “Accept” at this point, the updates are committed to the underlying databases, i.e., the meeting added to the shared calendar, the flights are reserved, and costs are debited. Alternatively she may press “Retry” to ask IceCube to propose a new schedule based on the same inputs, or “Cancel” to exit and reconcile later.

3 Shared data and actions

Here we describe the replicated data and action classes; this information will be useful when we come to understanding constraints.

Replicated data is of class `ReplicatedState`, where applications provide implementations for the abstract `checkpoint` and `returnToCheckpoint` methods. This is so that the IceCube engine can undo tentative actions.

An action is an operation that is provided by some application; in effect, a method pointer or a closure. Some important interfaces of actions are in Figures 4, 5 and 6. The application programmer provides implementations for these interfaces, which are invoked by IceCube during reconciliation.

```
interface ActionExecution {
    // test precondition; no side effects
    boolean precondition (ReplicatedState);
    // update state; return postcondition + undo info
    boolean execute (ReplicatedState, UndoData);
    // undo; return false if can not undo
    boolean compensate (ReplicatedState, UndoData);
}
```

Fig. 4. Action execution interface

The `ActionExecution` interface of Figure 4 is used to perform an action. IceCube first executes its `preCondition` method, then `execute`. The former (which may have no side effects) tests whether the action is valid in the current state. The latter updates the state. If either returns false, this indicates a dynamic constraint (defined shortly) has been violated.

The `compensate` method rolls back an update, provided undo information that was returned by the corresponding `execute`.

Not illustrated in the figure, an action has an integer *value*, 1 by default; this conveys user preferences between actions. The remaining interfaces will be detailed in later sections.

4 Constraints

A *constraint* is a correctness condition on schedules. Constraints are our central abstraction for conflict detection and scheduling. A *static* constraint relates two actions unconditionally. An example is the conflict between two appointment

requests for Anne at 10:00 in different places. A *dynamic* constraint consists of the success or failure of a single action, depending on the current state. For instance, overdraft of the expense account constitutes a dynamic constraint.

No schedule generated by the engine ever violates a static constraint, therefore static constraints incur no run-time cost. In contrast, dynamic constraints are expensive, as a violation may cause unlimited roll-back. Fortunately, our technique of dividing the work into sub-problems (to be explained later) usually limits roll-backs to a small number of actions.

4.1 Primitive static constraints

The static constraints are built upon two primitives, Before, noted \rightarrow , and MustHave, noted \triangleright . Any schedule s must satisfy the following correctness conditions:

1. For all actions $\alpha, \beta \in s$, if $\alpha \rightarrow \beta$ then α comes before β in the schedule (although not necessarily immediately before),
2. For any $\alpha \in s$, every action β such that $\alpha \triangleright \beta$ is also in s (but not necessarily in that order nor contiguously).

Although the Before relation might be cyclic, a correct schedule may not contain a cycle of Before. The scheduler breaks any existing cycles by dropping actions appropriately. This search for an optimal acyclic sub-graph in a non-binary cyclic graph makes reconciliation an NP-hard problem [4].

4.2 Log constraints

A *log constraint* is a static constraint between actions of the same log. User and application use log constraints to make their intents explicit. The `addLogConstraint` API assigns log constraints to an action.

In the current prototype, we have found the following three log-constraint types useful, which are built by composing the primitives. Constraint `predSucc`(α, β) establishes that action β executes only after α has succeeded (causal ordering). For instance, say a user tentatively updates a file, then copies the new version. To maintain correct behaviour in the reconciliation schedule, the application records a `predSucc` constraint between the write and the copy. `predSucc`(α, β) is equivalent to $\alpha \rightarrow \beta \wedge \beta \triangleright \alpha$.

The `parcel` log-constraint is an atomic (all-or-nothing) grouping. Either all of its actions execute successfully, or none. `parcel`(α, β) is equivalent to $\alpha \triangleright \beta \wedge \beta \triangleright \alpha$.³ For instance a user might copy two whole directory trees inside a third directory as a parcel. If any of the individual copies would fail (e.g., for lack of space, or because the user doesn't have the necessary access rights) then none of the copies is included in the reconciled schedule.

³ Unlike a traditional transaction, a `parcel` does not ensure isolation. Its actions may run in any order, possibly interleaved with other actions (unless otherwise constrained).

The alternative log constraint provides choice of at most one action in a set. $\text{alternative}(\alpha, \beta)$ translates to $\alpha \rightarrow \beta \wedge \beta \rightarrow \alpha$, i.e., a cycle that the scheduler breaks by excluding either α or β (or both) from the schedule. An example is submitting an appointment request to a calendar application, when the meeting can take place at (say) either 10:00 or 11:00. Users use alternative constraints to provide the scheduler with a fallback in case of a conflict.

4.3 Object constraints

An *object constraint* is a semantic relation between concurrent actions.

```
interface ActionObjectConstraint {
    // test whether this and other action conflict
    boolean mutuallyExclusive (Action other);
    // Favorable ordering of actions
    int bestOrder (Action other);
}
```

Fig. 5. Object constraint interface

return true if both actions cannot be in the same schedule; and `bestOrder` should return true to indicate a preference for scheduling this action before the other.

Object constraints express static concurrency semantics, similarly to Schwartz [14] or Weihl [18]. For instance, creating a file and creating a directory with the same name is `mutuallyExclusive`.

Another example: an account manager indicates a `bestOrder` preference to schedule credits before debits. Like `alternative`, the constraint `mutuallyExclusive(α, β)` translates to $\alpha \rightarrow \beta \wedge \beta \rightarrow \alpha$. `bestOrder(α, β)` is equivalent to $\alpha \rightarrow \beta$.

As an example, Table 2 summarises the object and dynamic constraints of the account manager, and a calendar application with actions to add or remove an appointment.

4.4 Explicit commutativity

Two actions that are not explicitly related by \rightarrow can run in any order. However it is useful to further differentiate pairs of commuting actions that have no mutual side effects. This is important in several places; for instance when rolling back an action, later actions that commute with it do not need to be rolled back.

The `ActionEnhancement` methods of Figure 6 provide explicit commutativity information; they have priority over `mutuallyExclusive` and `bestOrder`.

A *domain* is an opaque hash characterising a set of objects read or written by an action. `GetDomain` returns any number of domains. Actions with no common

In preparation for reconciliation, and for every pair of concurrent actions, IceCube calls the methods in the `ActionObjectConstraint` interface of Figure 5 to collect its object constraints. The application developer provides implementations for these methods: `mutuallyExclusive` should

Account	credit/credit	debit/debit	debit/credit
Different accounts	$\neg\text{overlap}$	$\neg\text{overlap}$	$\neg\text{overlap}$
Same account	commute	commute	bestOrder
Dynamic constraint: no overdraft			
Calendar	add/add	remove/remove	remove/add
Other user, time	$\neg\text{overlap}$	$\neg\text{overlap}$	$\neg\text{overlap}$
Same user & time	Mut.Excl.	commute	bestOrder
Dynamic constraint: no double-booking			

Table 2. Account and Calendar constraints

domain are commutative. For instance the domain of an account manager might be a hash of the account number.

If two actions have a common domain, IceCube calls their `overlap` method to test whether they overlap; if not, they are commutative. In our accounting example, `overlap` tests, first if the other action is also a accounting action (because domains are not guaranteed unique), then whether it operates on the same branch and account number.

```
interface ActionEnhancements {
    // domain identifiers
    long[] getDomain ();
    // do this and other action (same domain) overlap
    boolean overlap (Action other);
    // do this and other (overlapping) action commute
    boolean commute (Action other);
}
```

Fig. 6. *Action commutativity interface*

When two actions overlap, method `commute` tests whether they commute semantically; if yes they are commutative. For instance, two credits to the same account overlap but commute.

4.5 Dynamic constraints

To check a dynamic constraint the system must execute the action against the current state. Both `preCondition` and `execute` return a boolean value; if either returns false a dynamic constraint has been violated. A `preCondition` is not allowed to have side effects in order to avoid the cost of rolling back.

These methods can test arbitrary predicates that cannot easily be expressed with static constraints. The typical example is to check that the balance of an account is sufficient before executing a debit action.

It could be argued that dynamic constraints subsume static ones. While it is true that \rightarrow could be checked dynamically, this would be orders of magnitude more expensive than our static scheduling algorithm. Furthermore \triangleright cannot be captured with a dynamic check, which can only look at already-executed actions, not future ones.

5 Reconciliation scheduler

As the scheduling problem is NP-hard, IceCube explores the space of possible schedules heuristically.

We now present more detail of the heuristics and optimisations. Benchmarks presented in Section 7 show that the algorithm is efficient, scales well and closely approximates the true optimum.

5.1 Partitioning into sub-problems

For efficiency, we first the search space such that the combined complexity is much smaller than the original problem. IceCube partitions the actions into disjoint *sub-problems*, such that: actions in any sub-problem commute with actions in all other sub-problems, and there are no static constraints connecting actions

from different sub-problems. Actions from different sub-problems may be scheduled in arbitrary order, and executing or rolling back an action belonging to some sub-problem does not affect actions of another sub-problem.

Partitioning occurs in three stages. First, actions are partitioned according by domain identifier (see Section 4.4), with a complexity linear in the number of actions. Then, each such domain again subdivided into sets of commuting actions (thanks to the interface of Figure 6), for a complexity quadratic in domain size. Finally, any of the resulting sets that are connected by a static constraint or have an action in common are joined together, for a complexity proportional to the number of actions. For space reasons, we omit further detail; interested readers are referred to our technical report [11].

5.2 Heuristic search

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

An exhaustive search has exponential complexity. In contrast, our heuristics have only quadratic cost (in sub-problem size), and have results virtually indistinguishable from exhaustive search. This is confirmed by our benchmarks in Section 7.

Iteratively, the scheduler heuristically selects the best (as defined in the next paragraph) action α from a set of candidates. If executing α violates a dynamic constraint, the scheduler undoes its execution and removes it from the candidate list. Otherwise it adds α to the current schedule, and removes from the candidates any action that conflicts statically with α . This algorithm guarantees that schedules are heuristically optimal and satisfy the constraints.

Given some partial schedule s , each candidate action α is assigned a merit that estimates the benefit of adding α to s , measured by the number of other actions can be scheduled after α . After experimenting, we have found that the following heuristic to be the most effective. The merit of α is:

1. Inversely proportional to the total value of actions β that could have been scheduled only before α , i.e., such that $\beta \notin s \wedge \beta \rightarrow \alpha$.
2. Inversely proportional to the total value of alternatives to α .
3. Inversely proportional to the total value of actions mutually exclusive with α .
4. Proportional to the total value of actions β that can only come after α , i.e., such that $\beta \notin s \wedge \alpha \rightarrow \beta$.

The above factors are listed in decreasing order of importance.

The merit also takes dynamic constraints into account. When a dynamic constraint is violated, we want to avoid violating it again. It is not known precisely which action(s) caused the constraint to fail, but it can only be an action of the same sub-problem. Therefore we decrease the merit of the current action and of all actions that precede it in the same sub-problem.


```

scheduleOne (state, summary, goodActions) =
  schedule := []
  value := 0
  actions := goodActions
  WHILE actions <> {} DO
    nextAction := selectActionByMerit (actions, schedule, summary)
    precondition := nextAction.preCondition (state)
    IF precondition = FALSE
      THEN // pre-condition false
        // abort 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
        // pre-condition succeeded; now execute
        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 false: roll back
            toExclude := MustHaveMe (nextAction)
            SIGNAL dynamicFailure (goodActions \ toExclude)
  RETURN { state, schedule, value }

```

Fig. 7. *Selecting and executing a single schedule*

The merit estimator executes in constant time.

Our scheduling algorithm, displayed in pseudo-code in Figure 7, selects, with randomisation, some action among those with highest merit, executes it, and adds it to the schedule if execution succeeds.

If executing candidate action α violated a dynamic constraint, it will not be scheduled, but the scheduler must roll back any side effects α may have had.⁴ Furthermore, if the scheduler previously executed actions β such that $\beta \triangleright \alpha$ (for instance, α and β are part of a parcel), then β is removed from the schedule and its side effects rolled back as well.

When adding some action α to the schedule, the engine drops from future consideration any action β that (if scheduled) would violate a static constraint against α . This consists of the sets $\text{MustHaveMe}(\alpha) = \{\beta \mid \beta \triangleright \alpha\}$ and $\text{OnlyBefore}(\alpha, s) = \{\beta \mid \beta \rightarrow \alpha \wedge \beta \notin s\}$.

The scheduler calls `scheduleOne` repeatedly and remembers the highest-value schedule. It terminates when some application-specific selection criterion is satis-

⁴ Selected actions are executed immediately, in order to reduce the amount of roll-back in case of dynamic constraint violation.

fied — 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 (a sub-problem). Readers interested in the full algorithm and justification of the complexity estimate are referred to our technical report [11].

6 Calendar application

To give a flavour of practical usage, we describe the calendar application in some detail (other applications are presented in more detail elsewhere [11]). It is an appointment database shared by multiple users. User commands may **request** a meeting, possibly proposing several possible times, and **cancel** a previous request. A user command tentatively updates the database and logs the corresponding actions.

Database-level actions **add** or **remove** a single appointment. The user-level **request** command is mapped onto an **alternative** containing a set of **add** actions; similarly for **cancel**. Each such action contains the time, duration, participants and location of the proposed appointment.

Figure 8 contains some relevant code for **add** and **remove**. Object constraint and commutativity methods do the following:

1. **getDomain**: A shared calendar constitutes a domain.
2. **overlap**: Two actions (of the same calendar) overlap when either time and location or time and participants intersect.
3. **commute**: Overlapping **remove** actions commute with one another, since removing a meeting twice has the same effect as once.
4. **mutuallyExclusive**: if two **add** actions overlap, they also are mutually exclusive.
5. **removes** should preferably execute before **adds** to increase the probability that **adds** can be accommodated.

The code for **add** and **remove** actions implement the action execution methods from Figure 4:

1. **preCondition**: `add.preCondition` checks that the appointment doesn't double-book anything currently in the database.⁵ `remove.preCondition` returns true.
2. The **execute** method updates the database and saves undo information.
3. The **compensate** method rolls back a previous **execute** using the undo information.

Besides the **add** and **remove**, it was necessary to create a calendar **ReplicatedState** (**CalendarState**) to keep the calendar information (i.e., the scheduled appointments). Finally, a GUI interface allows users to access and modify the shared calendar.

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 The static constraints appear to make this check unnecessary, but it remains necessary in some corner cases.

```

class AddAction extends AbstractAction {
    long []domainId;
    MeetingInfo meeting;

    public boolean preCondition( ReplicatedState s0) {
        CalendaState s = (CalendarState)s0;
        return s.roomFree( meeting.location, meeting.time) &&
            s.peopleFree( meeting.participants, meeting.time);
    }
    public boolean execute( ReplicatedState s, UndoData info) {
        boolean result = ((CalendarState)s).insert( meeting);
        info.set( new Boolean( result));
        return result;
    }
    public boolean compensate( ReplicatedState s, UndoData info) {
        if( ! ((Boolean)info.get()).booleanValue()) // nothing to undo
            return true;
        return ((CalendarState)s).remove( meeting) != null;
    }
    public long []getDomain() {
        return domainId;
    }
    public boolean overlap( Action otherAction) {
        if( otherAction instanceof AddAction)
            return meeting.overlaps( ((AddAction)otherAction).meeting);
        else
            return meeting.overlaps( ((RemoveAction)otherAction).meeting);
    }
    public boolean commute( Action otherAction) {
        return false;
    }
    public boolean mutuallyExclusive( Action otherAction) {
        return otherAction instanceof AddAction &&
            meeting.overlaps( ((AddAction)otherAction).meeting);
    }
    public int bestOrder( Action otherAction) {
        if( otherAction instanceof RemoveAction &&
            meeting.overlaps( ((RemoveAction)otherAction).meeting))
            return ActionConstants.OTHER_FIRST;
        else
            return ActionConstants.ANY_ORDER;
    }
}

class RemoveAction extends AbstractAction {
    long []domainId;
    MeetingInfo meeting;

    public boolean preCondition( ReplicatedState s0) {
        return true; //removing a non-existent meeting is not as error
    }
    public boolean execute( ReplicatedState s, UndoData info) {
        MeetingInfo oldMeeting = s.remove( meeting);
        info.set( oldMeeting);
        return true;
    }
    public boolean compensate( ReplicatedState s0, UndoData info) {
        MeetingInfo oldMeeting = (Meeting)info.get();
        if( meeting != null)
            return ((CalendarState)s0).insert( oldMeeting);
        return true;
    }
    public boolean commute( Action otherAction) {
        return otherAction instanceof RemoveAction;
    }
    public boolean mutuallyExclusive( Action otherAction) {
        return false;
    }
}

```

Fig. 8. Calendar actions (simplified).

7 Measurements and evaluation

This section reports on experiments that evaluate the quality, efficiency and scalability of IceCube reconciliation. Our two benchmarks are the calendar application, described previously, and an application described by Fages [3].

The calendar inputs are based on traces from actual Outlook calendars. These were artificially scaled up in size, and were modified 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 sub-problems. The average number of **add** alternatives per **request** is two.

In each sub-problem, 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 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 done offline.)

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.

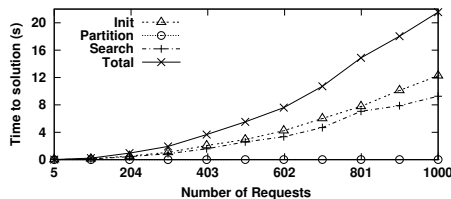


Fig. 9. *Decomposition of reconciliation time (single sub-problem).*

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 negligible because the **add** code is extremely simple.

7.1 Single sub-problem

To evaluate the core heuristics of Figure 7, we isolate the effects of partitioning into sub-problems with a first set of inputs that gives birth to a single sub-problem.

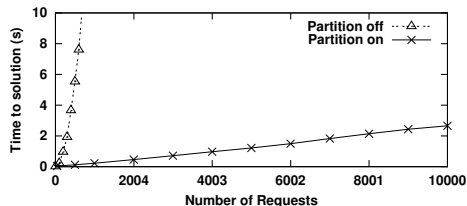


Fig. 10. Performance improvement with partitioning

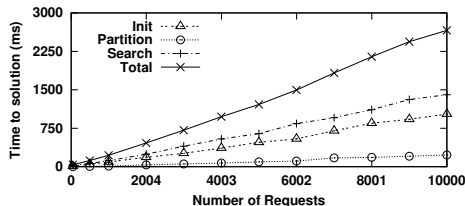


Fig. 11. Decomposition of reconciliation time with partitioning.

Figure 9 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 partitioning algorithm (although the experiment is rigged to generate a single sub-problem, the partitioning algorithm still runs). “Search” is the time to create and execute schedules. “Total” is the total execution time. As expected, partitioning 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 adds; each add in one request conflicts with a single add of another request. A log of x requests 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 sub-problems

We now show the results when it is possible to partition the actions into sub-problems. 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 adds can be partitioned alone; 25% of the remaining adds are in sub-problems with two actions; and so on. Thus, as problem size increases, the size of the largest sub-problem increases slightly, as one would expect in real life. For instance, when the logs contain 1,000 actions,

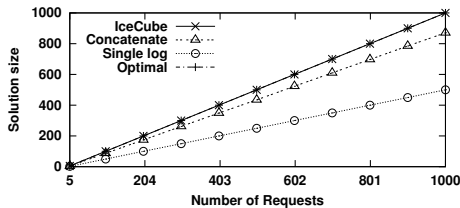


Fig. 12. *Syntactic vs. semantic schedule quality.*

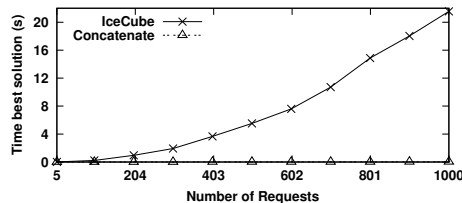


Fig. 13. *Syntactic vs. semantic performance.*

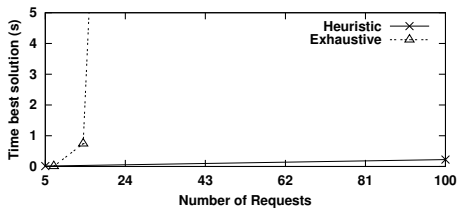


Fig. 14. *Exhaustive vs. heuristic search performance.*

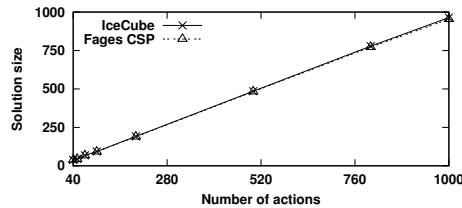


Fig. 15. *IceCube vs. Fages schedule quality (Fages' benchmarks).*

the largest sub-problem contains the adds from 12 Requests, and 18 when the logs total 10,000. The number of sub-problems 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 partitioning is in use or not.

Figure 10 shows the time to find a max-solution, with partitioning 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 3s for a log size of 10,000, much larger than expected in practice, is quite reasonable even for interactive use. As the number of sub-problems grows almost linearly with the number of actions and the size of the largest sub-problem 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 11, shows that all components of the reconciliation time grow approximately linearly, as expected.

7.3 Comparisons

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

Most previous systems use a syntactic scheduling algorithm such as timestamp order or log concatenation. As these all suffer equally from false conflicts, Figure 12 compares IceCube against the simplest one, log concatenation.

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 13 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 partitioning, and $O(n)$ for the syntactic approach.

Figure 14 compares execution time of our heuristic with an exhaustive search algorithm [7]. 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 [3] 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 randomly generates (on average) $1.5 \times$ size Before constraints per node. Figure 15 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.

8 Related Work

Several systems use optimistic replication and implement some form of reconciliation for divergent replicas. Many older systems (e.g., Lotus Notes [5] and Coda [8]) 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 [19] and Deno [6]. Balasubramanian and Pierce [1] and Ramsey and Csirmaz [12] study file reconciliation from a semantics perspective. Operational Transformation techniques [15] 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 [13].

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 very hard to program [16]. 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. [9] 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 [10] 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 follows on from the work of Kermarrec et al. [7]. They were the first to distinguish static from dynamic constraints. However their engine only supports Before (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

Supporting collaboration in mobile computing environments requires that applications and data management system rely on optimistic replication. As uncoordinated contributions (updates) from several users may conflict, the reconciliation mechanism is an important piece in system that support collaborative activities in mobile computing environments.

In this paper, we have 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.

Our system has made it easier for developers to make their applications tolerant of tentative operation and to reconcile replicated data. Application developers need not develop their own reconciliation logic. The framework design keeps application logic largely independent from the distribution, replication, and reconciliation. For the latter, IceCube provides a general-purpose middleware that applies to a large spectrum of applications. On the other hand, application design for this environment remains a demanding intellectual task.

The source code for IceCube is available from <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. Nuno Preguiç is partially supported by FCT/MCT.

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. François Fages. A constraint programming approach to log-based reconciliation problems for nomadic applications. In *Proc. 6th Annual W. of the ERCIM Working Group on Constraints*, June 2001.
4. R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.
5. 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.

6. 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>.
7. Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *20th Symp. on Principles of Distr. Comp. (PODC)*, Newport RI (USA), August 2001. ACM SIGACT-SIGOPS. <http://research.microsoft.com/research/camdis/Publis/podc2001.pdf>.
8. Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Usenix Tech. Conf.*, New Orleans, LA, USA, January 1995. <http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/usenix95.pdf>.
9. E. Lippe and N. van Oosterom. Operation-based merging. *ACM SIGSOFT Software Engineering Notes*, 17(5):78–87, 1992.
10. Sirish Phatak and B. R. Badrinath. Transaction-centric reconciliation in disconnected databases. In *ACM MONET*, July 2000.
11. Nuno Preguiça, 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. http://research.microsoft.com/scripts/pubs/view.asp?TR_ID=MSR-TR-2002-52.
12. Norman Ramsey and Előd Csirmaz. An algebraic approach to file synchronization. In *9th Foundations of Softw. Eng.*, pages 175–185, Austria, September 2001.
13. 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>.
14. Peter M. Schwartz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.
15. 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/csw/289444/p59-sun/p59-sun.pdf>.
16. Douglas B. Terry, Marvin Theimer, Karin Petersen, and Mike Spreitzer. An examination of conflicts in a weakly-consistent, replicated application. Personal Communication, 2000.
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 *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO (USA), December 1995. ACM SIGOPS. <http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf>.
18. W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988. <http://www.computer.org/tc/tc1988/t1488abs.htm>.
19. 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>.