Application-independent reconciliation for nomadic applications

Marc Shapiro, Antony Rowstron and Anne-Marie Kermarrec
Microsoft Research Ltd.
1 Guildhall St., Cambridge CB2 3NH, United Kingdom
marc.shapiro@acm.org

Abstract

We describe attempts to build an application-independent model to support reconciliation of diverged replicas of shared objects. While replicas are disconnected from one another, actions on the shared objects are recorded in a log. An action is composed of a precondition, an operation and a postcondition. When reconnecting, the system attempts to reconcile the divergent replicas, in several phases. A symbolic phase merges the separate logs, creating one or more schedules, such that preconditions from one log remain true despite the postconditions introduced by the other. Then, a simulation phase checks the possible outcomes by actually applying the schedules to scratch copies of the shared objects. Finally, a selection phase allows users or applications to select one of the candidate schedules.

Our approach supports sharing general objects, where one update may reference multiple objects (not just a single file or database). Compared to previous work on log-based reconciliation, our logs capture more semantic information and provide the application with more powerful and finer control over the outcome of reconciliation.

1 Introduction

In mobile computing there is often the need for disconnected operation. The user works on a local replica of shared data; updates cause replicas to diverge, and later they must be reconciled. Cooperative applications are particularly affected because multiple users may simultaneously update the same data with no mutual synchronisation.

There is no fully automatic solution to reconciliation, since it depends strongly on application semantics and on users' intents. However we are designing a middleware platform to take some of the burden away from applications. We extend the Bayou [6] model of logs that store application-specific information but are reconciled according to an application-independent protocol. Our model provides greater opportunity for the application to express its semantics and to influence the outcome of the reconciliation process. We believe this will reduce the load on the application programmer. Our reconciliation algorithm takes into consideration a larger set of possible schedules, thus providing more flexibility.

The management of shared data occurs in several distinct phases. In the *disconnected* phase, the user (through an application) reads and writes local replicas of shared objects.[1] User actions are logged in a log with a graph structure. A node of this graph contains an operation, along with assertions describing its meaning. When two devices meet, they exchange their logs (starting from a previous common checkpoint). A *symbolic* phase generates a new log, composing the two submitted logs under the constraints imposed by the assertions, of which multiple possible schedules can be deduced. A *simulation* phase computes the possible outcomes from this schedule, and a *selection* phase lets the application decide between them. The rest of this paper focuses on the symbolic phase.

This paper proceeds as follows. Section 2 compares this with previous work. Section 3 discusses some experiments that brought us to our current design, which is explained in detail in Section 4. In Section 5 an example shows the power of this new approach. Finally Section 6 concludes with open issues.

2 Background

Lotus Notes [3] is well-known as providing support for collaborative computing while disconnected. It

[1] It is assumed that the local application is correct, in the sense that it takes the local replicas from a consistent state to another consistent state, i.e., updates respect some integrity constraints.

detects and resolves update conflicts. However, it only supports a fixed set of applications. Its resolution policies are hard wired; for instance, an update/delete conflict always gives priority to the delete operation.

The CVS [2] is a source code versioning tool. Developers manage their own replicas, and a repository holds the master copy of each file. Coherence is managed by developers themselves, so when they wish to synchronise with the repository, they retrieve the master from the repository and attempt to integrate their modifications. CVS detects conflicts (the same part of the file has been updated by two different users) but then requires the user to solve them.

Bayou [6] provides an application-independent reconciliation protocol for a single replicated database supporting mobile users. Replicas eventually converge towards the same value. Each write operation is logged and timestamped. During reconciliation, the timestamp is used to generate an order in which the updates are applied to the replicated databases. Bayou allows an application to choose within a catalogue of coherence constraints, called session guarantees, and to apply dependency checks, called preconditions, at reconciliation time. A precondition is a procedure that the application stores in the log. However, this is not sufficient to capture application semantics.

Jrep [1] manages the replication of arbitrary Java objects for asynchronous collaborative applications. Jrep is based on a log of write operations (create, delete, update). Multiple updates on a single object are coalesced into a single equivalent operation. Lamport clocks [4] are used at reconciliation time to provide a causally-consistent merging of the logs. Conflicts on a same object are detected and solved according to a application-defined strategy. However, the same strategy is applied for all shared objects of an application. Inter-object conflicts are detected but their resolution requires the assistance of the application.

3 Experimental work

In order to understand the design space better for log-based reconciliation, we first developed a simple prototype system. Some of the issues we wished to evaluate were what information the log needs to capture, and what approaches to ordering the merged logs can be used. In this section, we outline some of our experiences, and what impact these have had on our current design.

This initial prototype uses a log similar to Bayou. Each log record represents an update to a shared replicated object. A record is composed of a precondition, an operation and a failure handler. All three are opaque Java procedures, i.e., their semantics are not known to the system. If the precondition evaluates to true the update can be applied, otherwise it cannot be performed and the error handler is called.

Reconciliation combines two logs to produce a merged log. The merged log is replayed from a common initial state of the shared objects, to yield a new common state. We investigated different strategies for log merging. Initially, we investigated using well-defined orderings of the logs, such as concatenation, or using time to order.

3.1 Ordering the logs

For a given set of updates to shared information, there will be one or more orderings of these updates that minimise the number of those that can not be applied. However, this ordering may not be the generated by concatenation of the logs, or the time ordering of the records within logs. Let us consider an example. For instance, consider two shared variables x and y, and logs A $\{write(x); delete(y)\}$ and B $\{write(y); delete(x)\}$. Assuming that *write* has a precondition that the shared variable exists, then these two logs need to be interleaved as $\{write(x); write(y); delete(x); delete(y)\}$ or, some permutation of this such that the *write* operations are performed before the *delete* operations, such as $\{write(x); delete(x); write(y); delete(y)\}$. This means that all the updates are performed, and there is no conflict between any of them.

In contrast, an approach that concatenates logs A and log B would yield a log where one of the *write* operations cannot be performed, because the variable being accessed has been removed. An approach based on real time or Lamport clocks [4] gives unpredictable results. Vector clocks [5] capture the true causal ordering, but are not adapted to systems with large or unknown numbers of replicas.

We conclude that the simple approaches to ordering the logs miss potential orderings of the log that reduce conflicts. However, if we allow log merging to consider arbitrary orderings of the records, searching for the best one suffers combinatorial explosion. Brute-force algorithms are untractable.

3.2 Capturing semantics

The second conclusion is the need to capture the application expectations regarding the outcome of the reconciliation phase. In the last example, a more satisfactory schedule might be $\{write(x); write(y)\}$ (omitting the deletes). The user should be able to specify which outcome is the correct one. In order to allow the application to control the behaviour of the reconciliation phase, more semantic information is required than can be captured with an opaque precondition.

In summary, our main conclusions are that: (i) simple orderings often fail unnecessarily, (ii) the application should be able to control the outcome of the reconciliation process, and (iii) the potential combinatorial explosion needs to be controlled.

4 System model

The system model considers several phases. In the *disconnected* phase applications at a site record their actions in that site's log. The other phases concern reconnection and reconciliation. The *symbolic phase* combines two logs symbolically, detecting constraints between separate logs, which prune the schedule space. The *simulation phase* goes through the possible schedules from the combined log, computing their actual outcome. Finally, in the *selection phase*, the application (and ultimately the human user) chooses among the outcomes remaining from the symbolic phase: removing schedules whose results are deemed unsatisfactory; editing the logs and resubmitting them to reconciliation; or selecting one of the outcomes as definitive.

It is possible to specify a policy to prioritise choices. It is applied during both the symbolic and simulation phase. It specifies criteria to reduce the number of possible outcomes, potentially reducing the search space. Example policies might be "I want schedules which maximise the number of records in a log", or "I prefer schedules that maximise the number of entries made by Antony over Marc".

In what follows, we focus exclusively on the symbolic phase.

4.1 Schedules

Our log is a graph structure that can be traversed according to one or more schedules. A schedule is a program for bringing the shared objects from their initial state (nominally, at disconnection time) to their final state (nominally, just before reconnection). Orderings between records are scheduling constraints. The initial and final states are assumed correct.

In the symbolic phase, the system combines two logs. The combined log contains all the records of the original two, connected by composition operators (to be presented in Section 4.3). Its schedules are compatible with the original schedules and satisfies inter-log dependencies, as will be explained in Section 4.4. A schedule of the combined log is a program that can be executed (at either site) to bring the shared objects, from their common initial state, to a final state that incorporates the updates made independently at each site.

The symbolic phase succeeds if one or more satisfactory schedules can be found. A conflict between the two logs may cause it to fail. In this case the system presents the application with an explanation of the conflict. The application may then edit the input logs to remove the conflict and submit to reconciliation again.

4.2 Actions

A log is composed of a set of records connected by a dependency graph, whose operators will be defined in Section 4.3.

Each log record describes an action performed by the user while disconnected. A log record is composed of:

Precondition: an assertion about the expected state of the objects before executing the operation.

Operation (with arguments and results): a subprogram that accesses the shared objects in some way.

Postcondition: an assertion about the effect on the state of the objects after the method has finished executing.

Assertions are written in a first-order logic language. Symbolic assertions are evaluated symbolically against a model of the system; the code is unused. In the simulation phase, the assertions are checked against, and the code is executed on, scratch copies of the actual shared objects. In contrast to previous systems, our assertions are not opaque procedures, but instead provide input to the symbolic phase.

Some examples of assertions are $x < 5$ or *after*(a), where a is another action. In a calendar program, *Marc.free*(*27-mar-2000-11:00*) asserts that the 11:00 slot on 27 March 2000 is free in Marc's calendar.

4.3 Dependencies

A log is an acyclic directed graph. Nodes represent actions. An edge represents a constraint: sequentiality (noted "$\prec$" hereafter), independence (noted "$\oplus$") or choice (noted "$\Box$"). If $a \prec b$ then a appears before b in any schedule. If $a \oplus b$ then both a and b must appear in any schedule, but there is no ordering constraint between them. If $a \Box b$ then any schedule must contain either a or b.

If some action b in a log depends on the results or side effects of action a of the same log, assertions provide the way to express this dependency. For instance

b might contain precondition *after*(a); or a could contain postcondition $x = 10$ and b precondition $x \geq 5$ (assuming no intervening postcondition changes x).

Although two logs represent work done independently, dependencies may appear between them. For instance if log A contains an action a with precondition $x < 10$ and log B contains an action b with postcondition $x > 20$, then in the combined log b may not be scheduled before a (assuming no other actions modify x).

The clause *last*() in a precondition forces that action to occur last in any schedule. This can force a particular outcome by asserting it in the postcondition. For instance, suppose a user writes a cheque for £50 on a shared bank account x. Other users of the account should not cause the cheque to bounce, i.e., he wants to ensure x remains positive despite what other users do. The following log will do the trick:

```
Log X {
    ...
    action cheque-action {
        pre:  ∃x0 : x = x0 ∧ x0 > 50
        op:   write-cheque(x, 50)
        post: x = x0 − 50
    }
    ...
    action no-overdraw {
        pre:  last()
        op:   no-op
        post: x ≥ 0
    }
}
```

A log may contain multiple *last*() actions connected by the "$\oplus$" operator.

4.4 Reconciliation

We now examine the algorithm for combining logs. Two logs A and B are independent if:

- No precondition of one contradicts a postcondition of the other, and
- No *post*condition of one contradicts a postcondition of the other.

If independent, the combined log is $A \oplus B$. If they are not independent, then possibly one may be scheduled before the other (e.g. $A \prec B$ if A's postconditions are compatible with B's preconditions). Otherwise maybe A and B can be interleaved (e.g. $a_1 \prec b_1 \prec ... \prec a_n \prec b_m$). To avoid considering all the possible combinations of nodes from A and B, we repeat the test for independence to successively finer granularities (binary search) down to individual actions. In the expected common case (few conflicts) it should converge rapidly, but in the worse case, when many nodes of A conflict with many nodes of B this process suffers combinatorial explosion.

4.5 Commitment

A disconnected update is tentative, as it may be found later that it cannot be applied due to a conflict. An application can commit an action by marking it as *definitive*, meaning that any future schedule must contain that action. For instance, a user presented with a set of possible outcomes in the selection phase might decide to commit one of them, by marking the corresponding actions definitive. An action that cannot be undone (such as the physical issuing of a check) is also marked as definitive. Furthermore the ordering between definitive actions is itself definitive. In a definitive action, assertion *last*() evaluates to true (i.e., does not influence future scheduling decisions).

In this work we do not make the policy decision as to who has the authority to mark an action as definitive. This might a particular authorised user (as in Lotus Notes) or a particular site (as in Bayou). Whatever policy is chosen, the possibility exists that two conflicting actions are independently marked as definitive. Although this is a serious error, we do not attempt to provide a solution in our framework, because we consider this is an unfortunate but inherent characteristic of disconnected work.

4.6 Parcels and transactions

The parcel construct links an set of actions together indivisibly. Consider a reservation application, where a user reserves transportation to a city, a hotel, and a car rental. If any of the three fail, the trip cannot take place. A parcel captures this all-or-nothing property. A parcel is composed of a *begin*(X) operation (where X is an arbitrary name), a set of actions with precondition *parcel*(X), and an *end*(X) operation. The following properties hold for a parcel: (i) a *parcel*(X) action a is scheduled after the *begin*(X) and before the *end*(X) (i.e., $\mathit{begin} \prec a \wedge a \prec \text{end}$); (ii) in any schedule, either all actions a_i of the parcel appear, or none (i.e., $(\mathit{begin} \oplus a_1 \oplus a_2 \oplus \ldots \oplus \mathit{end}) \Box \mathit{nil}$); (iii) if any action in a parcel is marked as definitive then all must be.

Note that parcels are strictly more powerful than traditional ACID transactions. Parcels provide the A property (all-or-nothing), the definitive mark the D property (durability). According to Footnote 1 the C property (consistency) is assumed. To provide the I property (isolation) a number of approaches are possible. We can emulate traditional serialisability by implementing locks as assertions. For every value x

read and not modified by an action, the action has the precondition $\exists x_0 : x = x_0$ and postcondition $x = x_0$. For every value y written by an action, the action has asserts the postcondition $y = \textit{value-written-to-y}$. If an action uses a value such as x_0, read or written in a previous action, then its precondition would contain the clause $x = x_0$.

5 A nomadic shared text editor

In this section, an example is presented that is representative of real cooperative working, and that is poorly supported by traditional merge procedures. We especially want to demonstrate that the postconditions used in our approach allow the capture of the application semantics.

Consider two disconnected users A and B editing a shared MS Word document. A replaces all occurrences of "red" with "white". Meanwhile, user B inserts the same word "red" in the document. There are two reasonable outcomes of reconciling these actions: (B-Red) insert B's "red" in the text, or (B-White) replace B's "red" with "white". CVS implements the B-Red semantics. Bayou would unpredictably choose B-Red or B-White depending on the timing of the operations by the users. Instead we want the outcome to be predictable and application-selectable. Assuming that the word "red" in the original document designates some concept that is now denoted by "white" and that B is ignorant of A's actions, then B-White is the correct outcome. However if B really means something different, then B-Red is desirable.

Assume that A's log is composed of a search-and-replace operation and log B contains a single action "insert red". The two possible outcomes, B-red or B-white, are forced by the use of different postconditions in the *last*() actions.

Assume *find*(s) returns the set of locations of string s in the document. The respective logs of A and B would be:

Log A {

  action A_1 {
    pre: $\exists s_0 :$ *find*(`"red"`) $= s_0$
    op: *search-&-replace*(`"red"`,`"white"`)
    post: *find*(`"white"`) $\subset s_0$
  }

}

Log B {

  action B_1 {
    pre: true
    op: *insert-at*(`"red"`,l)
    post: *looking-at*(`"red"`,l)
  }

}

Merging the two logs with no further constraints will yield unpredictable results (either B-Red or B-White).

If user B wants to enforce B-Red semantics, putting the following additional action at the end of his log ensures that his insertion is scheduled after the replacements:

action B-red {
  pre: *last*()
  op: no-op
  post: *looking-at*(`"red"`,l)
}

The postcondition ensures that the `"red"` inserted by B will survive action A. The only possible interleaving is thus $A_1 \prec B_1$.

Similarly, for user A to enforce B-White semantics a *last*() action will do the trick. A's log should contain:

action B-white {
  pre: *last*()
  op: no-op
  post: *find*(`"red"`) $= \emptyset$
}

The postcondition of action "B-white" ensures that no `"red"` occurrence survives action A. The only way to respect this postcondition is to schedule $B \prec A$.

If both A and B add these *last* actions into their logs, their postconditions contradict each other and they cannot be both scheduled – there is an unresolvable conflict. This is a true conflict between the users' intents, not an artificial limitation of the reconciliation engine.

Our approach allows logs to be merged according to logical constraints, not only in time order. Consider now the following example. Log A is composed of two independent actions:

- Operation A_1: *search-&-replace*(`"red"`,`"white"`)
- Operation A_2: *search-&-replace*(`"pink"`,`"green"`)

Log B is composed of two independent actions as well:

- Operation B_1: *insert-at*(`"pink"`,l_1)
- Operation B_2: *insert-at*(`"red"`,l_2)

Assume that the user A has the same expectations as in the first example and requires B-white semantics whereas the user B expects his insertion of `"pink"` to appear in the reconciled state. The outcome of the reconciliation should contain the `"pink"` inserted by

B but not his `"red"`. Thus, A_2 should be scheduled before B_1 and B_2 before A_1.

To enforce this scheduling, A and B should add the following *last*() actions into their logs:

action no-red {
  pre: *last*()
  op: no-op
  post: *find*(`"red"`) $= \emptyset$
}

action pink-inserted {
  pre: *last*()
  op: no-op
  post: *looking-at*(`"pink"`,l_1)
}

This will yield the correct merged log $(A_2 \prec B_1) \oplus (B_2 \prec A_1)$. Thus our approach supports useful schedules not allowed by approaches relying on a time ordering.

6 Conclusion

Working while disconnected is making a bet on the future, so it is not surprising that reconciliation is a difficult problem. The intended result of reconciliation depends on subtle details of application semantics and on users' intents. Our goal is to ease the burden for applications, putting as much as possible on the system instead, but providing for application semantics and user policies. An application expresses its semantics and user intents by attaching appropriate assertions to each action. Our logs are richer and more expressive than logs previously used for reconciliation.

To alleviate combinatorial explosion of the search space we came up with the three-phase approach.

This work is still at an initial phase and we don't claim to have solved all problems. Combinatorial explosion is an issue; however our binary-search algorithm for combining logs has complexity combinatorial in the number of conflicts (assumed low), not in the number of nodes. Furthermore the search can be stopped at any level. If the number of possibilities is too large for the simulation phase a ranking based on some policy could help. To keep the number of conflicts low, we can suggest making informed decisions (conflict avoidance) and reconciling often.

One problem not addressed here is that even if each user's actions are correct, their combined activity might not be. This might be addressed by checking global integrity constraints on the shared objects at the end of every schedule.

One big problem is getting applications to enter faithful and meaningful records in the log. Another is the symbolic modelling of the state space for the symbolic phase. Some of the logic is application-independent (e.g., that whatever P, $\neg P$ contradicts assertion P). However some is application-specific (for instance, in Section 5, that *looking-at*(`"red"`,l) contradicts *looking-at*(`"white"`,l)). Although extremely powerful, such a general application-specific logic is probably too complex for the average application programmer. We are therefore now focusing on a more restricted application-specific logic language, related to Schwartz's compatibility matrix [7].

Acknowledgments

Thanks to Tony Hoare and Ralph Becket for several illuminating discussions.

References

[1] Olivier Dedieu. *Réplication optimiste pour les applications collaboratives asynchrones.* PhD thesis, University of Marne-la-Vallée, To appear, fourth quarter 2000. http://www-sor.inria.fr/~dedieu/.

[2] P. Cederqvist et al. Version management with CVS, 1992.

[3] L. Kawell Jr., S. Beckhart, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *2nd. Conf. on Comp.-Supported Coop. Work*, Portland OR (USA), September 1988.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] Friedmann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[6] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. Symp. on Operating Systems Principles (SOSP-16)*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS. http://www.parc.xerox.com/csl/projects/bayou/.

[7] P. M. Schwartz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.