Proceedings of the 5^{th} MiNEMA Workshop

Middleware for Network Eccentric and Mobile Applications

11-12 September 2007, Magdeburg, Germany

We thank the European Science Foundation (ESF) for funding and supporting the 5^{th} MiNEMA Workshop in Magdeburg

Semantic Middleware for Designing Collaborative Applications in Mobile Environment

Lamia Benmouffok
Université Pierre et Marie Curie
Laboratoire d'Informatique de Paris 6
104, avenue du Président Kennedy
75016 Paris - France
Email: lamia.benmouffok@lip6.fr

Jean-Michel Busca
INRIA - Rocquencourt
Laboratoire d'Informatique de Paris 6
104, avenue du Président Kennedy
75016 Paris - France
Email: jean-michel.busca@inria.fr

Marc Shapiro
INRIA - Rocquencourt
Laboratoire d'Informatique de Paris 6
104, avenue du Président Kennedy
75016 Paris - France
Email: marc.shapiro@acm.org

***Abstract*— The Telex middleware facilitates the design of collaborative applications in a mobile environment. It provides optimistic replication, tentative execution and disconnected work. It solves conflicts based on semantic information provided by applications. We study in particular a Shared Calendar (SC) application, whereby mobile users can create and manage meetings in a collection of shared calendars. The application provides Telex with objects representing (1) meeting creation and modification operations (actions), (2) dependence or conflict information between actions (constraints). When a conflict occurs, Telex proposes solutions to users.**

The advantage of this approach is a clean separation of concerns. The SC application writer concentrates on application logic, whereas Telex takes care of replication, consistency, conflicts, and commitment across all applications.

I. Introduction

Designing collaborative applications raises the key problem of ensuring the consistency of shared mutable data. This problem is even more difficult in a mobile environment due to its decentralized nature and to the volatility of participants.

The Telex middleware facilitates the design of collaborative applications by taking care of complex application-independent aspects, such as replication, conflict detection and repair, and ensuring eventual commitment. It supports optimistic replication [1], which decouples data access from network access. Telex allows an application to access a local replica without synchronizing with peer sites. The application makes progress, executing uncommitted operations, even while peers are disconnected. Telex propagates updates lazily and ensures consistency by a global *a posteriori* agreement on the set and order of operations. Local execution is tentative; due to conflicts, some operations may roll back later.

Unlike previous optimistic replication systems, Telex takes the semantic of the collaboration into account, building on the Action Constraint Formalism (ACF) [2]. A Telex application represents its shared data as a set of actions (representing application operations submitted by users), and a set of constraints between these actions, expressing their concurrency semantics. Telex uses this semantic information to accuraty detect conflicts and to propose solutions.

We designed a Shared Calendar (SC) application to demonstrate how Telex facilitates the design of collaborative applications in mobile environment. A SC application aims to help people organizing their agenda in a collaborative way. It allows people to create and manage private events as well as group meetings, scheduled on a collection of online calendars. The design of SC application illustrates the benefit of using Telex middleware; it also illustrates some limitations of Telex.

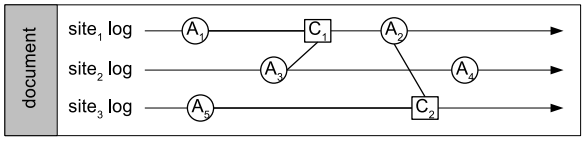

Fig. 1. Shared Document Representation

The remainder of this paper is organized as follows. Section II briefly describes the ACF. Section III presents the architecture of Telex and its operation. Section IV describes the Shared Calendar application built on Telex. Section V concludes.

II. Action Constraint Formalism

In ACF, an *action* represents an application operation and a *constraint* defines a scheduling invariant between two actions. The ACF defines three elementary constraints expressing commutativity, order and dependency relations. `(A non-commuting B)` states that executing A before B does not yield the same result as executing B before A. `(A not-after B)` indicates that A must not execute after B. `(A enables B)` means that B can execute if and only if A also executes.

Elementary constraints can be combined to express richer semantic relations, encompassing data semantics, application semantics and user intents. Thus, the cycle `((A not-after B) and (B not-after A))` states that A and B are *antagonistic*, i.e. an execution cannot contain both actions. `((A not-after B) and (A enables B))` expresses the fact that B *causally depends* on A. The cycle `((A enables B) and (B enables A))` indicates that A and B must be executed *atomically*.

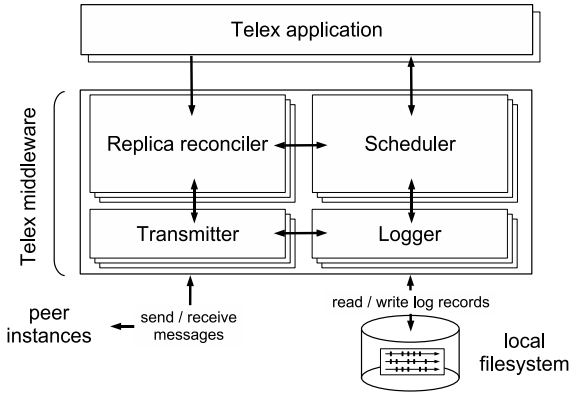

Fig. 2. Telex Architecture

III. Telex Middleware

A *document* is the basic sharing unit, and Telex represents its current state as a graph of submitted actions connected by constraints. As shown in Figure 1, it implements the history as a set of per-site logs, or *multi-log*. Furthermore, Telex allows applications to define *cross-document* constraints.

Figure 2 shows the overall architecture of a Telex instance running at some site. This instance supports several applications that together use its services. The Telex middleware is composed of two main modules — the scheduler and the replica reconciler — layered on top of two auxiliary modules — the transmitter and the logger. For each open document, Telex creates one instance of each module, which maintains the execution context of the document.

The transmitter and the logger are responsible for maintaining a replica of the document's multi-log at the local site. To this end, they implement an epidemic replication protocol which ensures that multi-log updates are eventually propagated to all *participating sites*, i.e. sites that collaboratively edits the document, either at the same time, earlier or later. The scheduler and the replica reconciler are described next.

A. Scheduler

The scheduler maintains an action-constraint graph that represents the state of the document known locally. Actions and constraints are added to the graph either by: the application, when local user updates the document, (ii) the logger, when it receives an update issued by a remote user, (iii) the replica reconciler, when it commits a schedule (see below).

Based on the action-constraint graph, the scheduler periodically computes *sound schedules*, i.e. sequences of actions that comply with constraints, and proposes them to the application for execution. In case some actions conflict, i.e. they do not commute or they are antagonistic, several schedules exist, each representing a particular solution to the conflict. The scheduler computes them one by one, upon application request, until one or more satisfaying schedules are found.

Actions submitted concurrently at different sites may turn out to be conflicting. Therefore, whenever a new action is addded to the graph, Telex checks whether constraints exist against concurent actions. Conceptually, Telex calls the application for every pair $\langle newaction, existingaction \rangle$ and adds to the graph the constraints that the application returns, if any. In order to optimize this CPU-intensive check, the application tags each action it submits with a set of numeric keys, one for each object that the action operates upon. Actions conflict only if their key sets intersect. Telex checks for this condition before calling the application, thus saving a significant number of unnecessary calls. False positives cause only a performance loss.

B. Replica Reconciler

Participating sites may generate different sound schedules from the same action-constraint graph. The role of the replica reconciler is to make sites agree on a common schedule to apply and thus achieve (eventual) mutual consistency. The agreed-upon schedule is said to be *committed*.

The replica reconciler implements a decentralized asynchronous commitment protocol based on voting. Periodically or on user request, each site proposes and votes for one or more schedules generated by the scheduler. Local user may specify the schedule(s) of his choice, if any. Votes are sent to participating sites, and a schedule that receives a majority or a plurality of votes is committed. The committed schedule is then materialized as a set of constraints added to the action-constraint graph.

The commitment protocol is fully asynchronous. It runs in the background and each instance determines locally when a schedule has won an election. Meanwhile, the scheduler keeps proposing (tentative) sound schedules to the application. In addition, the protocol may run only on a subset of participating sites that are know to be stable. The voting process is automated and does not require user intervention. The detailed protocol can be found in Sutra et al. [3].

IV. Shared Calendar Application

The Shared Calendar (SC) application design demonstrates how Telex facilitates the design of collaborative applications. The SC application provides users a way to manage their activities collaboratively. Each user has his own calendar, which he shares with the other authorised users.

To create a meeting with a group, a SC user creates a "meeting object" and shares it with the invitees. He notifies invitees by creating an action on their respective calendar.

When one receives an invitation he can accept it or decline it. If he accepts it, he can collaborate to hold the meeting: he can invite other users, and modify the meeting time, and location. For that purpose he creates actions on the corresponding meeting object, concurrently with other invitees. Consequently conflicts may appear. As we are in an optimistic replicated environment, those actions are tentative until committed. In case of antagonism, some of them are aborted.

A. Use case

Figure 3 shows an example concurrent execution of the calendar application. Users Jean-Michel, Lamia and Marc use

a Shared Calendar application to plan meetings between colleagues. Jean-Michel, Lamia and Marc are working separately and communicate only via the application.

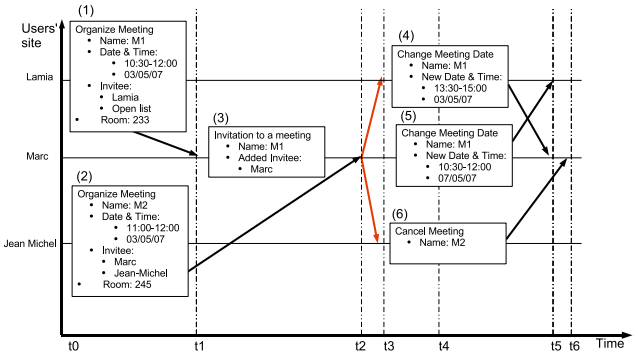

Fig. 3. Execution scenario the Shared Calendar application.

Jean-Michel organizes meeting *M2* on 3 May between 11:00 and 12:00. He allocates Room 245 for that purpose. He requires the presence of Marc and himself. This is illustrated by Action 2 in Figure 3.

Lamia organizes meeting M1 on the same day between 10:30 and 12:00. She allocates Room 233. She will attend the meeting, and allows other people to invite themselves, with actions 1 in Figure 3.

The application only needs to provide the above actions to Telex. Telex propagates them eventually to all interested sites (in this example, to Marc's site) even if some users are offline. Suppose that, at some point in time $t1$, Marc has received Lamia's actions, but not yet Jean-Michel's. This may happen, for instance, if Jean-Michel is working offline. Marc is interested in M1 and invites himself to that meeting (action 3 in the figure). Later, at $t2$ Marc knows Jean-Michel's actions.

As M1 overlaps, with M2 a conflict is detected at time $t3$. For this to happen, the SC application arranges that the corresponding actions' key sets overlap. Therefore, Telex upcalls SC, which returns an antagonism constraint, as explained elsewhere.

To avoid this conflict, Lamia shifts the start time of M1 to 13:30 (action 4). Concurrently ($t4$), Marc also sets the date of M1 to the 7 May (action 5), and Jean-Michel cancels M2 (action 6).

At $t5$ Lamia and Marc have received their concurrent modifications of meeting M1. Obviously M1 is scheduled at different times on Lamia's and Marc's calendar. SC provides the non-commute constraint, which causes Telex to order them the same way at all sites, after the commitment phase.

B. SC design using Telex

The application expresses its semantics by defining appropriate constraints between actions. The code for computing actions and constraints is part of the application; at run time SC outputs appropriate action and constraint instances to Telex, and Telex propagates them to the appropriate replicas. Thus application semantics is cleanly separated from the difficult systems task of ensuring consistency.

In more detail, the shared calendar application supports the following actions:

- *createEvent (meetingId)*: Create some event, for instance a meeting.
- *setInfo (time, meetingId)*: Modify the schedule of an event.
- *invite/addUser (userId, meetingId)*: Invite a person to an existing event.
- *allocate (roomId, meetingId)*: Allocate a room for an event.
- *cancelInvitation/cancelUser (userId, meetingId)*: Cancel an invitation.
- *cancelAllocation (roomId, meetingId)*: cancel a room allocation for a meeting.
- *Cancel (meetingId)*: Cancel a meeting.

Recall that this application supports optimistic replication. Each user of this application can generate one of the previous actions and execute it locally. However the execution remains tentative until an agreement phase reaches a consensus whether actions are committed, or aborted, or reordered.

In the use case scenario, Jean-Michel generates actions : A = *createEvent (M2)*, B = *allocates (245, M2)*, C = *addUser(Jean-Michel, M2)*, D = *addUser(Marc, M2)*. It groups them atomically with an enables cycle: `((A enables B) and (B enables C) and (C enables D) and (D enables A))`. He also generates E = *setInfo (11:00-12:00 03/05/07, M2)* and the constraint `A enables E`. Telex propagates these actions and constraints to Marc's site, as well as Lamia's actions concerning meeting M1.

Telex checks each pair of actions for their keys. Telex upcalls the application for possible constraints only two actions have a same key. SC computes its keys as follows. Each discrete 30-minute time slot has a unique identifier. For action *setInfo (description, time, meetingId)* where *time* is a set of slots $\{S_i, i = 1 \ldots n\}$ the generated keys are:

- The hash of *meetingId*,
- The hash of each slot identifier S_i.

Back to the use case scenario. At time $t2$ Marc receives actions F = *setInfo (" " ,10:30-12:00 03/05/07, M2)* and G = *setInfo (11:00-12:00 03/05/07, M2)*. Keys of the two setInfo match as both have a key that is a hash of the {11:00-12:00 03/05/07} slot. Therefore, Telex asks SC for the corresponding constraints. SC returns an antagonisism constraint `((F not-after G) and (G not-after F))`, which causes either F or G, or both, to eventually abort.

Telex suffers from some usability issues, because reifying application semantics into actions and constraints is not very intuitive. Furthermore, constraints are hard to validate.

To facilitate the use of Telex to develop collaborative applications, we propose a generic methodology for principled designs. A rule for using Telex is to make any shared mutable information a Telex-managed object. For instance, a meeting is an implicit information inherent to each invitees' calendar.

This information is shared between all invitees, and is mutable as any invitee can collaborate to modify the meeting information. The number of invitees is also dynamic. Thus the easiest way to manage a meeting is to make it an explicit Telex object, with corresponding actions and semantics. Figure 4 shows the state of Marc's site at time $t2$.

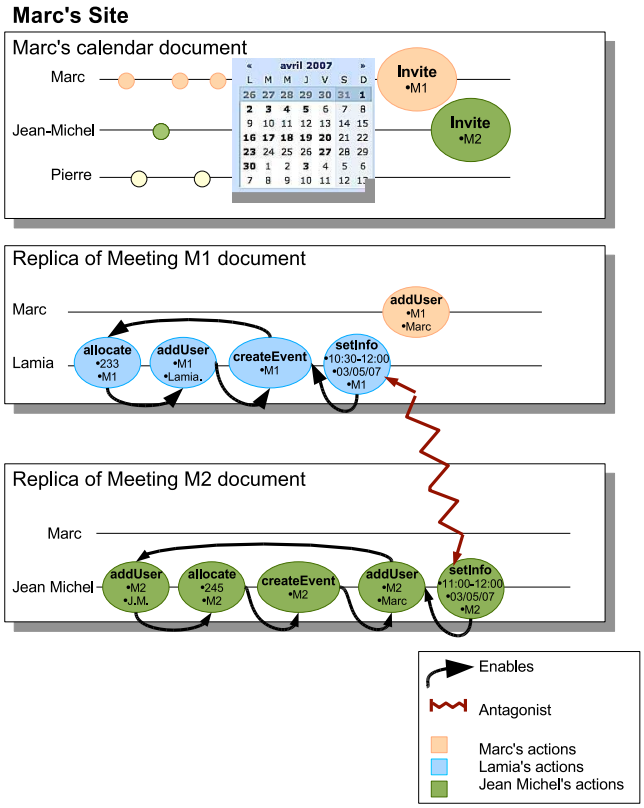

Fig. 4. Marc Site at $t2$.

The Telex scheduler insures that Marc will not be scheduled for $M1$ and $M2$ at the same time (Antagonistic actions). The reconciliation phase ensures eventual consistency.

V. Conclusion and Future Work

The Telex middleware facilitates the development of collaborative applications in mobile environment. The main contribution of Telex is that the reconciliation between replicas is application independent as Telex is semantically rich. We presented a design of a Shared Calendar application to demonstrate Telex benefits and highlight Telex usability limitations. To facilitate the use of Telex to develop collaborative applications, we propose a generic methodology for more principled designs. However this approach also suffers from limitations. Reifying application semantics into actions and constraints is not very intuitive. Furthermore, constraints are computed in advance, without knowledge of the actual state. We suggest two complementary approaches - A compiler should generate actions and constraints from a high-level specification - A checker should verify that all action-constraint combinations verify the application invariants.

References

[1] Y. Saito and M. Shapiro, "Optimistic replication," *Computing Surveys*, vol. 37, no. 1, pp. 42–81, Mar. 2005, http://doi.acm.org/10.1145/1057977.1057980.

[2] M. Shapiro, K. Bhargavan, and N. Krishna, "A constraint-based formalism for consistency in replicated systems," in *Proc. 8th Int. Conf. on Principles of Dist. Sys. (OPODIS)*, ser. lncs, no. 3544, Grenoble, France, dec 2004, pp. 331–345, http://www-sor.inria.fr/ shapiro/papers/opodis2004-final-2004-10-30.pdf.

[3] P. Sutra, J. Barreto, and M. Shapiro, "An asynchronous, decentralised commitment protocol for semantic optimistic replication," INRIA, Research Report 6069, 12 2006. [Online]. Available: https://hal.inria.fr/inria-00120734