HAL
open science

Telex: Principled System Support for Write-Sharing in Collaborative Applications

Lamia Benmouffok, Jean-Michel Busca, Joan Manuel Marquès, Marc Shapiro, Pierre Sutra, Georgios Tsoukalas

To cite this version:

Lamia Benmouffok, Jean-Michel Busca, Joan Manuel Marquès, Marc Shapiro, Pierre Sutra, et al.. Telex: Principled System Support for Write-Sharing in Collaborative Applications. [Research Report] RR-6546, INRIA. 2008, pp.31. inria-00281329v2

HAL Id: inria-00281329

https://inria.hal.science/inria-00281329v2

Submitted on 5 Jun 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Telex: Principled System Support for Write-Sharing in Collaborative Applications

Lamia Benmouffok — Jean-Michel Busca — Joan Manuel Marquès — Marc Shapiro — Pierre Sutra — Georgios Tsoukalas

N° 6546

9 May 2008

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6546--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche PARIS - ROCQUENCOURT

Telex: Principled System Support for Write-Sharing in Collaborative Applications*

Lamia Benmouffok[†‡] , Jean-Michel Busca[†‡] , Joan Manuel Marquès[§‡] , Marc Shapiro[†‡] , Pierre Sutra[†‡] , Georgios Tsoukalas[¶]

Thème COM — Systèmes communicants

Équipe-Projet Regal

Rapport de recherche n° 6546 — 9 May 2008 — 28 pages

Abstract: The Telex system is designed for sharing mutable data in a distributed environment, particularly for collaborative applications. Users operate on their local, persistent replica of shared documents; they can work disconnected and suffer no network latency. The Telex approach to detect and correct conflicts is application independent, based on an action-constraint graph (ACG) that summarises the concurrency semantics of applications. The ACG is stored efficiently in a *multilog* structure that eliminates contention and is optimised for locality. Telex supports multiple applications and multi-document updates. The Telex system clearly separates system logic (which includes replication, views, undo, security, consistency, conflicts, and commitment) from application logic. An example application is a shared calendar for managing multi-user meetings; the system detects meeting conflicts and resolves them consistently.

Key-words: No keywords

* This research is supported in part by Respire (ANR, France, respire.lip6.fr), Grid4All (FP6, EU, www.grid4all.eu) and grant JC2007-00213 (Spain).

† INRIA, Paris-Rocquencourt, France

‡ LIP6, Paris, France

§ Universitat Oberta de Catalunya, Barcelona, Spain

¶ National Technical University of Athens, Greece

Centre de recherche INRIA Paris – Rocquencourt

Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex

Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Telex : un système de partage en écriture pour les applications collaboratives, basé sur un modèle formel

Résumé : Le système Telex est conçu pour le partage des données modifiables dans un environnement réparti, principalement pour des applications collaboratives. Les utilisateurs opèrent sur une copie locale et persistante des documents qu'ils partagent ils peuvent travailler en mode déconnecté, et ne sont pas ralentis par la latence du réseau. Telex utilise une approche indépendante de l'application pour détecter et corriger les conflits, qui se base sur un graphe actions-contraintes (ACG) qui résume la sémantique de concurrence des applications. L'ACG est stocké de façon efficace dans une structure dite *multi-journal* qui élimine la contention et est optimisée pour la localité. Des applications différentes s'exécutent sur Telex, qui permet de mettre à jour plusieurs documents de façon coordonnée. Telex sépare proprement la logique système (ce qui inclut la réplication, les vues, le «undo», la sécurité, la cohérence, les conflits, et la finalisation) de la logique applicative. Un exemple d'application est un calendrier partagé, pour gérer des réunions multi-utilisateur le système détecte les conflits de réunion et les résout de façon cohérente.

Mots-clés : Pas de motclef

1 Introduction

The Telex system provides novel solutions for write-sharing data in co-operative and disconnected work settings.

Existing approaches have severe limitations. For instance state machine replication [5] imposes high latency and does not support disconnected operation. The popular last-writer-wins algorithm [11] does not ensure any high-level correctness guarantees.[1] In contrast, Telex is based on a principled approach that combines flexibility and correctness, and cleanly separates application logic from system logic.

Application logic transmits to Telex actions (operations) and constraints (concurrency invariants), and applies execution schedules transmitted by Telex. In return, Telex takes care of: replication, consistency, storage and access control; collecting, transmitting and persisting operations; detecting conflicts and computing high-quality conflict-free schedules; forward execution and rollback; checkpointing; commitment; and access control. Telex supports multi-document updates and cross-application scenarios out of the box.

Telex is based on a principled approach, the Action-Constraint Graph (ACG) [12]. We designed the *multilog* data structure to store ACG-based documents in a distributed file system. Multilogs eliminate write contention and promote locality.

We developed a number of demonstration applications above Telex. For instance, a shared calendar application lets people organise their agenda collaboratively, arranging private events and group meetings. Telex detects meeting conflicts and proposes possible solutions.

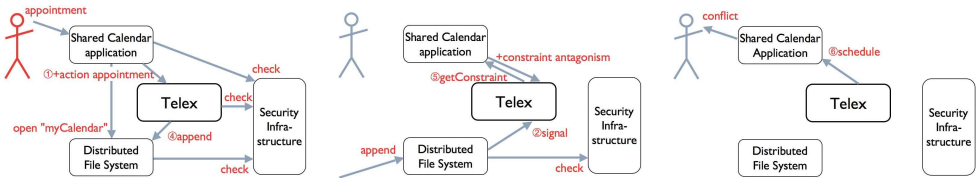
The contributions of this paper include: a novel approach to shared data replication that is application independent yet application-aware, the ACG; the practical engineering of an ACG system, in particular the document and multilog structures; design examples and lessons learned for ACG-based applications; and some benchmarks and performance measurements.

This paper proceeds as follows. Section 2 is an overview. Section 3 explains the data structures that Telex uses. Section 4 documents the Telex architecture and implementation. In Section 5, we present some example applications. Section 6 evaluates the Telex performance. We reflect on lessons learned in Section 7. Section 8 compares Telex with related work. Finally, Section 9 concludes.

2 Telex overview

We give an overview of the Telex system from three complementary points of view.

[1] Section 8 analyses the state of the art in detail.



a. App. reifies user op. as actions & constraints

b. Remote action rcvd.: upcall for conflict constraints

c. Compute schedule(s), execute, display

Figure 1: Telex interactions. (The circled numbers refer to Figure 5)

2.1 User/application perspective

Telex supports *participants*, i.e., users working at disjoint *sites*, which may be widely distributed. An authorised participant may replicate a shared *document* on his site.

A site operates optimistically [11]: it applies local *actions* (operations), sends them to other sites, and eventually *replays* the actions it receives. Hence, applications are not slowed down by remote synchronisation, network issues, or by remote failures.

A participant may work either connected or disconnected from others. Thus, each participant has his own *view* of the current state of the shared document. Documents and views persist across log-out/log-in and restarts. However, a view is only tentative and may have to roll back.

Telex, not applications, takes care of hard issues such as conflict detection, reconciliation, and consistency. However, since a conflict is the violation of some application invariant, Telex is parameterised by application-specific concurrency invariants called *constraints*. A constraint relates two actions, either of the same or distinct documents. Hence, Telex maintains consistency between documents.

Figure 1 illustrates the control structure of Telex with a Shared Calendar (SC) application.[2] In this example, the participant creates an appointment, which conflicts (double booking) with one created remotely. In Figure 1.a, the participant performs the appointment operation. The SC application logs the corresponding actions and constraints to the local Telex dæmon (+action appointment). In Figure 1.b, when the site receives a remote action (signal), it compares it to the concurrent actions. If Telex suspects a conflict, it calls up to the application (getConstraint), which replies with precise information (+constraint antagonism). Finally, as in Figure 1.c, Telex periodically sends *schedules* to the application, for execution and/or rollback. The application

[2] Elements of the figure not discussed here will be explained in later sections.

Name	Notation	Semantics
NotAfter	$A \rightarrow B$	A is never after B in any schedule
Enables	$A \lhd B$	B in a schedule implies A in same schedule
NonCommuting	$A \not\bowtie B$	Must agree on $A \rightarrow B$ or $B \rightarrow A$ (conflict)
Atomic	$A \overset{\lhd}{\rhd} B$	All or nothing
Causal	$A \overset{\lhd}{\rightarrow} B$	B depends causally on A
Antagonism	$A \overset{\leftarrow}{\rightarrow} B$	A and B never both in same schedule (conflict)

Table 1: Constraints

computes and displays the corresponding views, in this example with a conflict indication (conflict).[3]

2.2 Formal perspective: actions and constraints

Telex is based on a formal model, the Action-Constraint Graph (ACG) [12]. The ACG is a labelled graph whose nodes are the actions and edges are the constraints. The current view of a site is the result of executing a *sound schedule*, i.e., an ordering of actions currently known at that site, that obeys the safety constraints *NotAfter* and *Enables*. In effect, the ACG represents the set of all legal views.

Table 1 presents briefly the constraints supported by Telex; for full details please refer to the relevant publications [12]. The first three are primitive, the last three are combinations of the primitives.[4]

These represent important classes of concurrency invariants. While they can approximate the true application semantics only grossly, we have found that they are sufficiently expressive for reconciliation purposes in several kinds of applications [9, 13].

Formally, eventual consistency requires that all schedules be sound, that they have a common stable sound prefix, that every action eventually be either aborted or in the prefix, and that non-commuting actions that are in the prefix be ordered.[5] The latter two items imply a global consensus between sites. We call this consensus the *commitment protocol*. In Telex, commitment is optimistic, i.e., it occurs in the background, not in the critical path of applications.

2.3 Engineering perspective: multi-logs and commitment

The design of Telex is motivated by some major requirements and challenges: (i) Persist and replicate the ACG. (ii) Provide strong guarantees above a dis-

[3] For the purpose of this paper, document state, view and schedule are synonymous. "View" emphasises that the state is local and is not unique; "schedule" emphasises that it is computed by some ordering of available actions.

[4] *Atomic* does not ensure transactional isolation; an isolation constraint will be added in the future. Currently, to achieve isolation, the user must manually group operations into a single action.

[5] Mutually-commuting actions may run in any relative order.

tributed file system with only best-effort consistency. (iii) Integrate documents into the file system, with reasonable overhead and scalability. (iv) Provide access control, without violating consistency. (v) Remove old ACG entries from storage. (vi) Decentralised, peer-to-peer design, with support for casual disconnected operation.

A document is a named entity in the file system. For locality, a document stores only the portion of the ACG consisting of the actions operating on the document, and their constraints.

Telex documents coexist with ordinary files and directories in the file system. Using one or the other is up to the application.

Telex relies on external mechanisms to store and replicate documents, and to propagate changes to remote sites. To avoid file system bottlenecks and consistency issues, each participant writes to a distinct append-only *log* within a document. To enable incremental garbage collection, the log is broken down into successive chunk files. This structure is called *multilog*.

A log is a succession of actions and constraints in no particular order. We optimise for the expected common case, where constraints are inside the same log; inter-log constraints within the same document are slightly more expensive. Inter-document constraints are assumed to be relatively rare and are more costly.

Because of network delays and disconnections, and because of filtering and access control (explained later), at any point in time, different participants may observe different ACGs. However, each participant's view is consistent, because it results from a sound schedule. Thus, if some action A is not in a view, and A *Enables* B, then B is also not in that view.

The current view can be recorded in a *snapshot*. Snapshots name a view, speed up the computation of later views, and help with garbage collection.

A decentralised, background commitment protocol ensures that the common prefix of schedules makes progress. Each participant can vote for a schedule according, for instance, to user preference. Voting is decentralised and peer-to-peer.

Committed log records may be deleted. However it may be advantageous to retain them for auditing, recovery or selective undo (to be explained later).

3 Data structures

3.1 Document storage

Telex stores its documents in file systems with standard, best-effort consistency guarantees. The storage design obeys some specific requirements. Documents should be seamlessly integrated above a standard POSIX interface, with reasonable performance and scalability. They should co-exist with classical files and

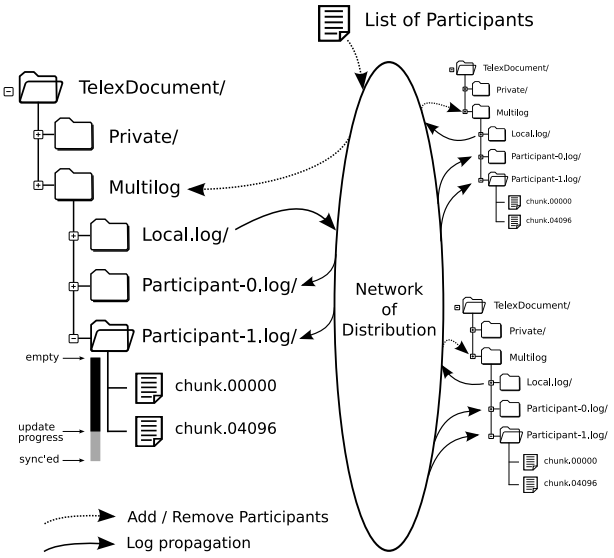


Figure 2: Storage of Telex document

directories. Participants must be able to work normally while disconnected. The system should scales well with the number of collaborating participants. Finally, Participants' data must be secured even when shared.

We implemented multilogs above the federative peer-to-peer file system VOFS [1]. VOFS provides global access to files with best-effort consistency. It supports disconnected operations via persistent replication, and notifications for file modifications on distributed files. A complete description of VOFS is outside of the scope of this paper; here we focus on specific features related to Telex integration.

3.1.1 Multilog Design

As illustrated in Figure 2, a Telex document is a structured directory of files. Applications and Telex may store document-specific data within the document, such as filters and snapshots. These data are local to a participant; only the multilog needs to be replicated.

A multilog is itself structured as a directory that contains an append-only log per participant. Actions and constraints created by an application are appended to that participant's log. Each participant's log is replicated at the other participants' sites; VOFS propagates the updates to the network. As each log has a single writer, is append-only, and local to a document, this avoids write contention and scalability issues.

Propagation of a log through the network is asynchronous, i.e., a log replica may contain only a prefix of its source, as indicated by the "sync" bar in the

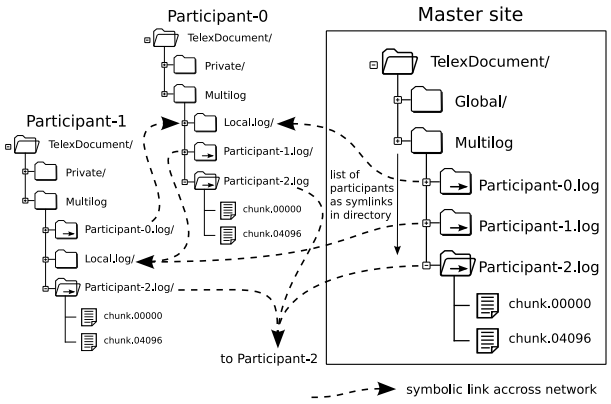


Figure 3: Implementation of multilogs over VOFS.

figure. Telex instances monitor the logs for new updates. Eventually, all actions and constraints are known to all participants.

As time passes, an action eventually becomes committed and is not needed any more. To enable removing such old records, a log is itself structured as a directory of chunk files. When the size of the current chunk reaches a threshold, a new one is created. The name of a chunk file includes a sequence number, making it convenient to read chunks in order, and to selectively delete chunks. A chunk may be deleted when all the actions it contains are committed and there is a later materialised snapshot. This is, however, a policy decision; a site may decide instead to retain old chunks for auditing or recovery.

3.1.2 Multilogs on VOFS

A document is stored by the Telex dæmon in the file system as a directory. The internal structure of this directory is not meaningful to users, and is intended to be hidden by the user interface (much like the "bundles" of MacOS).

In our deployed multilogs so far, we have used a centralised setup at a primary master site, containing the authoritative version of all the logs in a document. Participants' sites cache the logs persistently, making them available for disconnected operation. The master site is a single point of failure and a scalability bottleneck.

In the future, we plan to use a peer-to-peer configuration, using accross-network symbolic links that VOFS provides. Here, each participant hosts the authoritative version of his own log on his own site, as in Figure 3. As before, participants cache remote logs persistently. The master site serves only to list all the logs using symbolic links. Any other method of distributing the list could be used.

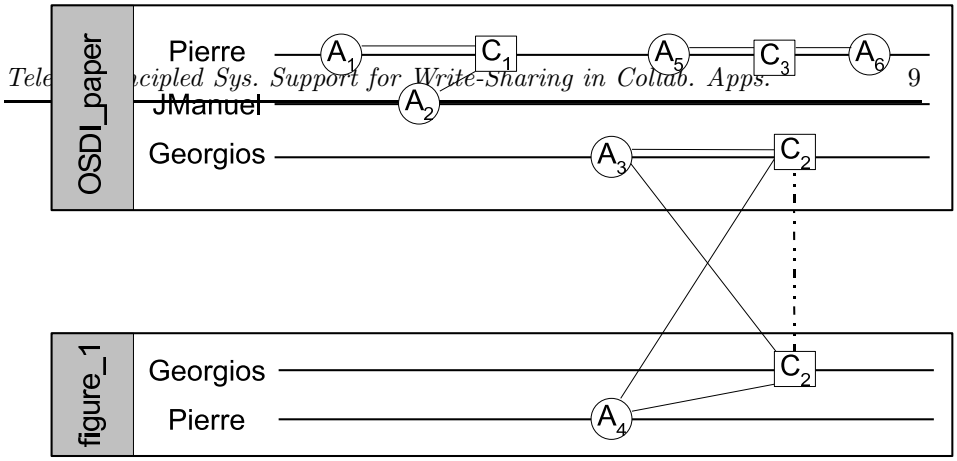


Figure 4: Two multilogs with their logs; note constraints within log, within document, and between documents

3.1.3 The Multilog Toolkit

VOFS is optimised for multilogs, which improves the user experience. However, multilogs can be implemented above any ordinary distributed file system. We provide a toolkit implementation of multilogs, as a set of simple programs and dæmons, providing simple and efficient multilog management and access above an ordinary file system.

The implementation follows closely the design of Figure 2. More details are available in Section 6.

3.2 Action and Constraint

An action represents an application operation. It is described by several attributes, of which some are known to Telex and other are application-specific. Among the former, the most important is a list of *action keys*. An action key indicates the document subset that this action targets; if two actions have a common key, this indicates suspicion that the actions conflict (see Section 4.2.1 for more detail). An action belongs to only one document. It is uniquely identified by the triple ⟨*document, issuer, timestamp*⟩. Telex logs an action in the log of the participant who issues it.

A constraint reifies a semantic relation between two actions. It is defined by its type (*NonCommuting*, *NotAfter* or *Enables*) and by the two actions it binds. A constraint is uniquely identified by the triple ⟨*type, action1, action2*⟩. Telex logs a constraint in the log of the participant who issues it.

Most often, a constraint binds two actions of the same document, whether issued by the same participant or not. Such a constraint is called an *intra-document* constraint. However, a constraint may bind actions of two distinct documents. Such a constraint is called a *cross-document* constraint. It is then logged in *both* documents.

A constraint C references an action A by using one of the three following forms: *(timestamp)* if A is issued by the same participant as C and belongs to the same document, *(issuer, timestamp)* if A belongs to the same document as C and *(docId, issuer, timestamp)* otherwise. In the latter form, *docId* is the id of the document that action A belongs to.

Figure 4 shows an example of the two types of constraint. Constraint C_1 is an intra-document constraint: it binds actions A_1 and A_2 of document *OSDI_paper*. Constraint C_1 is issued by *Pierre* and thus it is logged in *Pierre*'s log of *OSDI_paper*. On the other hand, constraint C_2 is a cross-document constraint: it binds action A_3 of document *OSDI_paper* and action A_4 of document *figure_1*. Constraint C_2 is issued by *Georgios* and thus it is logged in *Georgios*'s log of both *OSDI_paper* and *figure_1*.

3.3 Views

A desirable feature of replication in collaborative work is to enable different participants to have their own view of a shared document. For instance a participant working on a given section of a shared document may temporarily ignore updates to the same section by other participants. Telex allows the participant to select a particular view of a document by means of *action filters*. A filter defines which actions of the ACG Telex must exclude when computing sound schedules. When applying a filter, Telex also exclude all actions that filtered actions enable. This ensures that the view computed by filtering is always sound, i.e., document invariants are not violated.

A participant defines a filter by specifying its *name* and one or more filtering *criteria* involving any attribute of an action. The participant may define several filters on a document and dynamically add and remove them. Telex saves currently-defined filters as part of the persistent state of a document.

Note that a filter may target a specific action of a document. By adding and removing the filter, user may thus selectively undo and redo the corresponding action in his view of the document. (To undo an action persistently, the participant must *abort* it. By convention, this is expressed by marking the action as antagonistic with itself.)

Filters also provide a means to permanently exclude the operations of a participant who turns out to be malicious, as in the Ivy file system [6]. Contrary to Ivy, Telex filters maintain correctness, by excluding all actions that depends on the malicious participant's actions.

3.4 Snapshot

A snapshot records some view of the document. To define a snapshot, a participant specifies its *name* and the *schedule* of actions whose execution yields

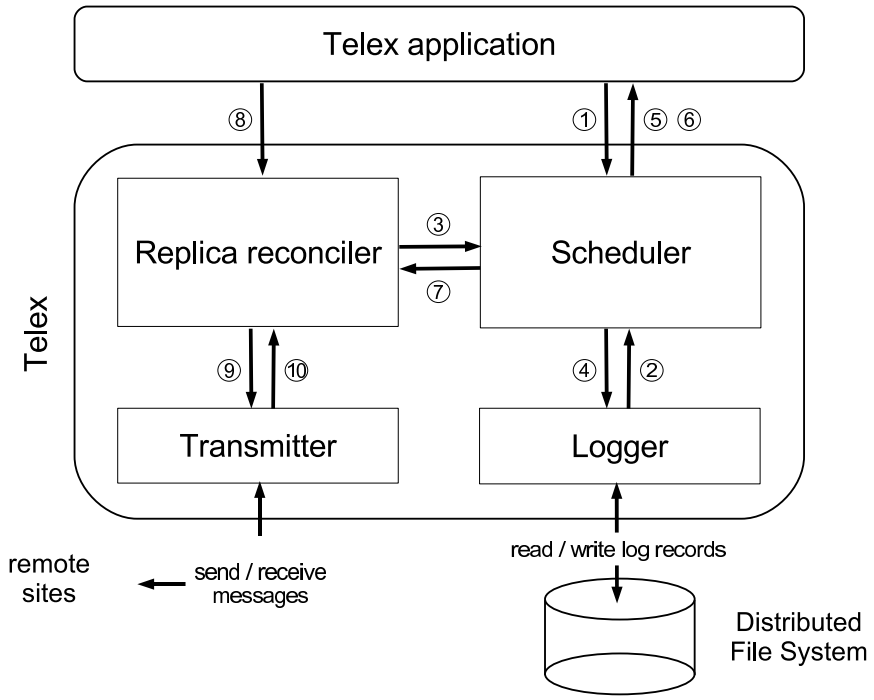


Figure 5: Telex architecture

the state being recorded. In addition, the application may provide the corresponding binary state of the document. In this case, the snapshot is said *materialised*. Materialised snapshots speed up the computation of a view and are used as garbage collection points.

The participant may define any number of snapshots of interest to him, and later remove those that are no longer useful. Telex saves the set of currently-defined snapshots as part the persistent state of the document.

4 Telex architecture and operation

Figure 5 is a detailed view of Figure 1 which shows the overall architecture of Telex. An instance of Telex runs at each site and communicates with remote sites.

On top of the figure are the applications using the services of Telex. Several such applications may run concurrently at the same site. In the middle of the figure is the Telex system. It is composed of two main modules — the scheduler and the replica reconciler — layered on top of two auxiliary modules — the transmitter and the logger. Arrows in the figure represent invocation paths between Telex modules and to/from applications.

Each application may open one or more documents. For each open document, Telex creates one instance of each module, which maintains the execution context of the document. The only exception is when documents are bound by cross-document constraints, as described in section 4.2.3. In this case, the bound documents share the same instance of the replica reconciler and the scheduler.

We describe next the interaction between a Telex instance and the outside world and then detail the operation of the main modules.

4.1 Interactions

Telex-application interactions involve exchanging pieces of AC graphs (sets of actions and constraints downwards, sets of schedules upwards). The interaction cycle is as follows. The participant acts upon the application, which translates his request into one or more actions and constraints and passes them to Telex. In return, Telex computes a sound schedule from the set of locally-known actions and constraints and hands the schedule to the application. The application executes the schedule and presents the resulting state to the participant. If some actions conflict, then several sounds schedules exist, each corresponding to a possible solution to the conflict. The application presents the resulting states to the participant so that he can select the solution he prefers.

Telex sites exchange actions and constraints through multilogs, and communicate with each other in the commitment protocol. The logger module logs the actions and constraints submitted by the local participant in the participant's log. In return, the VOFS notifies the logger when remote participant's log are updated. The transmitter determines the set of peer sites and provides an Atomic Multicast service among peer sites (arrows #9 and #10).

4.2 Scheduler

The role of the scheduler is twofold. First, it maintains the in-memory ACG that represents the state of the document at the local site. Second, it periodically computes sets of sound schedules from the ACG and proposes them to the application for execution. Actions and/or constraints are added to the graph either by:

- The application (Figure 5, arrow #1), when the local participant updates the document.
- The logger (arrow #2), when it receives an update issued by a remote participant.
- The replica reconciler (arrow #3), when it commits a schedule.

The scheduler passes locally-submitted actions and constraints to the logger (arrow #4) to log them on persistent storage.

4.2.1 Cross-site constraint generation

Actions logged independently by two participants may conflict; for instance in the shared calendar application, a same user could be added to two parallel meetings. Telex ensures that conflicts are reified by constraints as follows. When a site receives a new action, it compares it against already-known, concurrent actions of the same document. If they have a common key, then Telex invokes the corresponding application's *getConstraint* upcall. If the actions really conflict, the application responds by logging an appropriate constraint (arrow #5 in Figure 1.b or Figure 5).

Action keys are opaque to Telex, which tests them for equality only. Action keys serve as a compact, but approximate, representation of the document subset that the action uses or updates. Typically, an action key hashes the identifier of a parameter of the action. Multiple keys have "or" semantics (Telex upcalls *getConstraint* if a key of one action equals any key of the other). To implement "and" semantics (for instance, to get an upcall only if two given objects are involved) the application hashes the XOR of their identifiers into a single key. An action with no keys conflicts with no other.

If two unrelated actions happens to have equal action keys, no harm is done, other than a loss of performance.

4.2.2 Schedule generation

A large number of sound schedules exist for any given ACG in the general case. It is therefore not feasible to compute all sound schedules beforehand and present them to the application. Besides, the application may be interested only in a few or even just one schedule. For these reasons, Telex generates sound schedules dynamically, upon application request (this is not shown in Figure 5). The application may thus iterate through the proposed schedules and stops when one or more appropriate schedules are found.

Telex generates the best schedules first, where the quality metric is the number of actions included (implying fewer actions aborted). Optimal scheduling is NP-complete, therefore Telex runs a heuristic inspired by IceCube [9]. Secondary goals of the heuristic are to give preference to actions of the local participant in the case of a conflict, and to avoid returning a schedule equivalent to one returned previously.

4.2.3 Bound documents

Two documents are said bound if there exists a constraint between an action of one and an action of the other, and either action (or both) is not committed. For instance, if a participant wishes to update two documents atomically, he sets an *Enables* constraint in each direction between the updates.

The actions of a document may not be scheduled independently from those of the documents it is bound to. Scheduling is optimised for the common case of non-bound documents, but we provide special processing for this particular case. Note that bound documents may be handled by distinct applications.

Telex processes bound document by merging them into a single *shared* ACG in order to compute *global* schedules over all actions and constraints. Each global schedule generally contains actions from all bound documents. Thus, in order to execute a global schedule, Telex first projects the schedule on each document and passes each resulting sub-schedule to the relevant application. The projection operation simply consists in retaining only those actions that belong to the target document while preserving their order. Telex assigns the same identifier to the sub-schedules deriving from the same global schedule. This way, the participant can identify matching sub-schedules on each bound document.

4.3 Replica reconciler

Each Telex site proposes a set of constraints, a *proposal*, to remote sites. A proposal contains decision to commit, abort or serialise actions. These proposals may differ, due to asynchronous communication, filtering, differing local information, or user preference. The *replica reconciler* is in charge of *commitment*, i.e., reaching agreement on a common schedule prefix. Commitment occurs in the background, not within the critical path of applications. The committed proposal appears as a prefix of the local schedules.

We propose a plug-in replica reconciler architecture, providing different strategies according to needs. A reconciler has four (asynchronous) phases.

1. Each sites compute a proposal, according to its local view, for instance based on the user's preferences (arrow #8 in Figure 5).
2. The transmitter atomic multicasts proposals to set of sites directly concerned (arrow #9) by the agreement (in case of bound documents more than one replica group may be concerned). Atomic multicast maintains liveness in presence of faults and network lags.
3. The transmitter forwards proposals it receives up to the replica reconciler (arrow #10).
4. According to the commitment algorithm (described next) the reconciler chooses a winning proposal, and logs it (arrows #3 and #4).

Currently we propose two commitment algorithms. (i) A first-in first-out algorithm for applications such as a distributed database. At each site the FIFO algorithm proposes to minimise the number of dead actions according to its local view. When a site delivers a new proposal, the FIFO algorithm checks the soundness of the proposal according to the previous winning proposals (arrows #8 and #7). If the decision is sound, the reconciler adds it to the ACG, if not the decision is discarded.

(ii) A voting algorithm that takes into account local preferences. A proposal is a vote spanning one or multiple actions over one or more documents. A proposal is broken into sub-ACGs with specific properties, called candidates. Candidates containing the same actions challenge each other. A candidate may be elected only if its set of actions is transitively closed in the union of all the ACGs across sites. This protocol is described in detail in a separate publication [15].

4.4 Access control

The Telex design includes access control at increasingly fine-grain levels, using a security framework (whose description is out of scope of this document). This is indicated by the three arrows marked check in Figure 1. (i) Access control at file granularity ensures that a single participant writes a given log, and that only authorised users can read a log. (ii) The Telex dæmon checks whether a user is allowed to access an individual log record.[6] (iii) Applications may enforce further control. For instance, in the SC application, a user might observe the times that another user is busy, but not be allowed to see the other details of his meetings. As explained in Section 2.3, access control does not violate consistency.

5 Applications

To provide insight on the issues involved in using the Telex system, this section presents some of our example applications. We will return to the lessons learned in a later section.

5.1 Simple Replicated Dictionary

We start with a simple example. Our Simple Replicated Dictionary Application (SRDA) manages shared dictionaries. SRDA is intended as a building block for applications such as a shared address book. Users can operate on a dictionary in either connected or disconnected mode. Telex guarantees that, in spite of node arrivals, departures or failures, all instances of a given dictionary converge.

A document contains tuples of the form $\langle tupleID, attribute_1, attribute_2, \ldots \rangle$, for any number of attributes. Each attribute is a $\langle name, value \rangle$ pair. SRDA provides these operations:

- $insert(tupleID, attrs)$: inserts a new entry, with identifier *tupleID* and attributes *attrs*, into the dictionary document.
- $modify(tupleID, attrs)$: modifies attributes for the given *tupleID*.

[6] This is not yet implemented in the current version.

insert	$\forall$ previous $rem_i.TID$: $rem_i.TID \rightarrow$ current $ins.TID$
remove	$ins.TID \overset{\lhd}{\rightarrow}$ current $rem.TID$
modify	$ins.TID \overset{\lhd}{\rightarrow}$ current $mod.TID$ $\forall$ previous $mod_i.TID.attr_j$: $mod_i \rightarrow$ current mod

Table 2: Sequential execution constraints (Notation: *ins* = *insert*, *mod* = *modify*, *rem* = *remove*, *attr* = *attribute*, *TID* = *tupleID*)

- *remove*(*tupleID*): deletes the tuple corresponding to the given *tupleID*.
- *read*(*tupleID*): returns the attributes corresponding to the given *tupleID*.

In the first operation, the *tupleID* must be previously unused or removed; for all the others, a tuple identified by *tupleID* must already exist. The modify operation assigns the listed attributes if they already exist for the tuple, otherwise it adds them.

Insert, modify and remove operations translate to a Telex action. Because Telex does not yet support isolated multi-operation transactions, we manage write dependencies in the write operations, as explained shortly. Read operations are treated as local.

5.1.1 Sequential constraints

Table 2 summarises the sequential semantics of SRDA. SRDA logs these constraints at the same time as it logs the right-hand action of the constraint.

In the Telex design, the application should log causal dependence only when the second action truly depends on the first. Hence, a *modify* action, or a *remove*, is causally dependent on the *insert* that created the tuple. Thus, if the *insert* aborts or fails, the dependent *modify* and *remove* actions will be discarded from any sound schedule. Furthermore, we treat every write operation as a read-compute-write transaction.

In order to ensure read-your-writes session guarantees [16], we set *NotAfter* constraints between *insert*, *modify* and *remove* actions in the same user session, even between different dictionary documents.

Finally, to ensure the correct scheduling of a *remove* followed by an *insert* with the same tuple identifier, we make all previous *remove* with the same tuple-id *NotAfter* the current *insert*. The SRDA application logs the above constraints in the multilog, at the same time as it logs the right-hand action.

The SRDA application logs the above constraints in the multilog, at the same time as it logs the right-hand action.

	ins_2	mod_2
ins_1	$ins_1.TID = ins_2.TID$ $\Rightarrow ins_1 \nparallel ins_2$	–
mod_1	impossible	$mod_1.TID = mod_2.TID \wedge$ $attrs_1.TIDs \cap attrs_2.TIDs \neq \emptyset$ $\Rightarrow mod_1 \nparallel mod_2$

Table 3: SRDA *getConstraint*

5.1.2 Concurrency constraints

Since it is illegal to insert the same identifier twice, two concurrent *insert* actions that refer to the same identifier are *NonCommuting*. Otherwise, concurrent inserts commute. Similarly, two concurrent *modify* operations with the same identifier and overlapping attributes are also *NonCommuting*.

Those constraints are added by the application when Telex invokes its *getConstraint* method. They are summarised in Table 3, where *NonCommuting* is noted $\nparallel$. In order to ensure that Telex upcalls the *getConstraint* method as needed, *insert* and *modify* actions have an action key, computed as a hash of the *tupleID*.

5.2 Shared Calendar

Our Shared Calendar (SC) application is representative of collaborative decision-making applications. SC illustrates the advantages of Telex for semantically-rich collaborative applications.

SC helps people organise private events and group meetings collaboratively, possibly in disconnected and asynchronous mode. Contrary to existing calendar applications, SC detects conflicts (such as double booking), proposes solutions, and ensures agreement and eventual consistency.

This would be difficult to achieve without Telex support. Application logic (i.e., maintaining the data structures and identifying constraints) is well separated from the system logic, i.e., persistence, replication, conflict detection and resolution, commitment, etc.

5.2.1 SC logic

Each user or location has an associated *calendar* document. Each *event* (e.g., a meeting) is a separate document. A calendar may be read or updated by other users, who can (if so authorised) create or manage events, invite people to an event, or identify conflicts and free time.

We use the following notations. An event e is unique, has a name $e.name$, and a date$e.date$, and is materialised by a Telex document $e.dox$.

A user A creates an event e by creating the document $e.dox$, and by logging an *open-event* action in his own calendar and an *invite(A)* actions in $e.dox$. He

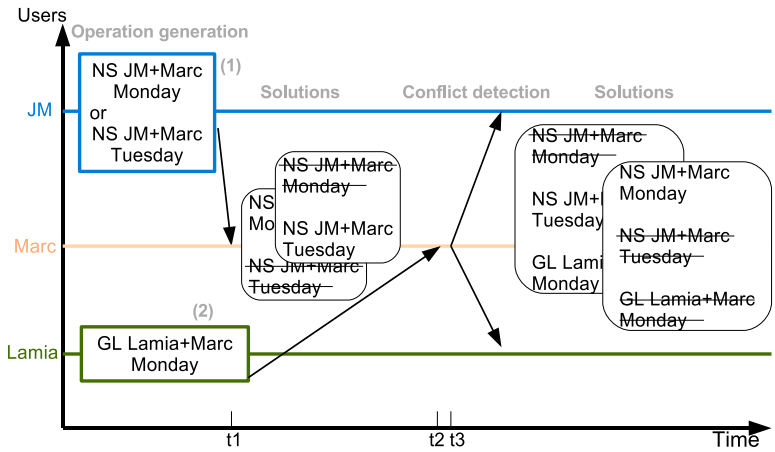


Figure 6: Execution scenario for the Shared Calendar application

also logs an *enable-event* action in *e.dox* that symbolises the creation of the event. This action is used to specify constraints on the event creation as shown next.

Later, user *A* may invite other users by logging an *open-event* action in his log within their calendars, and a corresponding *invite* action in *e.dox*.

Once a user has opened an event document, he may invite more users. He also can cancel the event or some user invitation by logging a *cancel-event* or a *cancel-invitation* action in *e.dox*.[7]

The action keys identify the event and its time-slots. Therefore, actions in the same calendar for the same event, or for different events at the same time, will have overlapping keys, causing Telex to invoke the *getConstraint* upcall interface of SC.

A calendar document action commutes with all other calendar document actions. Constraints between event document actions are similar to the SRDA constraints, where *enable-event*, *cancel-event* and *invite* (or *cancel-invitation*) are like like *insert*, *remove* and *modify* respectively.

To avoid double bookings, concurrent *invite* actions are antagonistic, if they concern the same user at the same time but different events.

5.2.2 Use case

Consider the scenario in Figure 6. Users Jean-Michel, Lamia and Marc are working separately and communicate only via the SC application.

[7] Currently it is not possible to collaboratively change the time of an event. This will require extensions to Telex to associate the time updates with some user invitation to detect a double booking, which is future work.

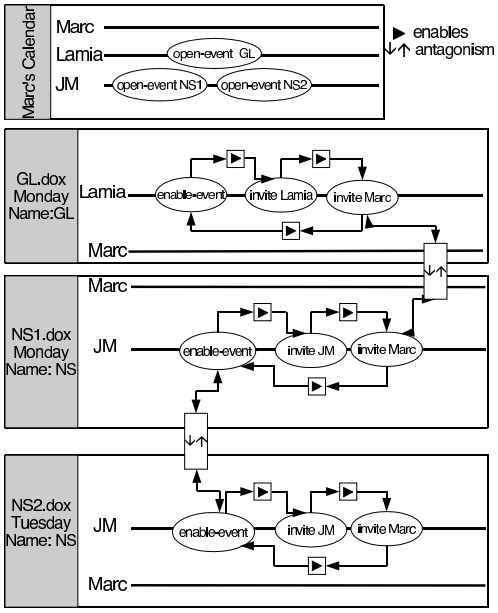


Figure 7: Marc's site at $t3$

Jean-Michel organises meeting Networking Seminar *NS* with Marc. He proposes two alternative dates, Monday and Tuesday (Operation 1 in the figure).

Lamia also organises a meeting Greek Lesson *GL* with Marc on Monday (Operation 2).

SC creates the event documents and logs the actions and constraints to Telex, as detailed in Figure 7, depicting the state of Marc's site at time $t3$.

Lamia's SC instance creates *GL.dox* document, imports Marc's calendar, and logs the following actions:

- On Marc's and Lamia's calendar: *open-event (e2)*.
- On *GL.dox*: $A = \textit{enable-event}, B = \textit{invite(Lamia)}, C = \textit{invite(Marc)}$. SC groups them atomically: $A \overset{\lhd}{\rhd} B \wedge B \overset{\lhd}{\rhd} C$.

To express the alternative Jean-Michel's SC instance transparently creates two events *NS1* and *NS2* with conflicting *enable-event* actions. For both events, SC generates similar actions as for the *GL* event.

Suppose that, at some point in time $t1$, Marc has received Jean-Michel's actions, but not yet Lamia's. This may happen, for instance, if Lamia is working offline. Telex computes the schedules corresponding to two possible solutions: (i) holding *NS* on Monday and aborting *NS* on Tuesday; or (ii) holding *NS* on Tuesday, and aborting *NS* on Monday. Since the former solution contains more actions, it will be proposed first.

Later, at $t2$ Marc knows Lamia's actions. Telex checks the keys of Lamia's actions with Jean-Michel's. $C = invite(Marc)$ on *GL.dox* and $E = invite(Marc)$ on *NS1.dox* both have a key representing the *Monday* slot. Therefore, Telex asks SC for the corresponding constraints. SC returns an antagonism constraint $C \leftrightarrows E$. This ensures that no view contains both C or E, and that one or the other (or both) eventually abort.

Finally, Telex offers the two possible solutions: (i) *NS* on Tuesday and *GL* on Monday, aborting *NS* on Monday; or (ii) *NS* on Monday, aborting *GL* on Monday and aborting *NS* on Tuesday.

Lamia is not invited to event *NS*, she may not read *NS1.dox* nor *NS2.dox*. Nevertheless, Telex ensures that she eventually gets notified of a conflict occurrence that may abort *GL*. The same goes for Jean-Michel. The reconciliation phase ensures that Marc, Lamia and Jean-Michel eventually see a consistent state for *GL* and *NS* events.

5.3 Shared wiki

For lack of space, we describe our Shared Wiki Application (SWA) only briefly.

Each wiki page is a separate document. Every user currently editing it has a log in the document. His site keeps a local replica of the wiki text, which the user modifies locally using a standard text editor. Every time the user saves, the SWA computes the difference from the previous version, and translates it into insert-line and delete-line actions. Modifying a line is interpreted as an atomic grouping of delete-line and insert-line.

The SWA uses the WOOTO operational transformation algorithm [7] to ensure that concurrent edit operations commute. A delete-line action depends causally on the action that inserted the line. Inserting a line between two other lines depends causally on the two corresponding line insert operations.

Since all concurrent operations inside a document commute, there will never be any conflicts. Therefore, edit actions carry no keys, and Telex never upcalls *getConstraint* to the SWA. Schedule computation is trivial, since all schedules that are compatible with causal dependence order are equivalent.

Existing wiki editors maintain the set of past versions of a page. Thanks to Telex, SWA can reconstruct any past version, and additionally maintains the relations between versions. In the future, we could extract more history information from the persistent multi-log, including page splits and merges, and copy-paste between pages.

From the perspective a single page, Telex serves mainly to reliably broadcast actions and replay them in causal order. One added value of Telex for SWA is the ability to perform multi-document updates, e.g., a global replace through all wiki pages consistently. Telex also enables multi-application scenarios, e.g., ensuring that a wiki page contain the details of a meeting agreed in the shared calendar application.

Config Name	1x8M	8x8M	1x8L	8x8L
Writers	1	8	1	8
Log size (MB)	50	50	5	5
RX limiting	no	no	yes	yes
runtime (sec)	3.4	9.3	306.48	309.31
avg RX+TX (B/s)	102.9M	75.3M	228.4K	226.3K

Table 4: Representative results for shared multilogs with 1 and 8 writers, with and without limiting receiving traffic

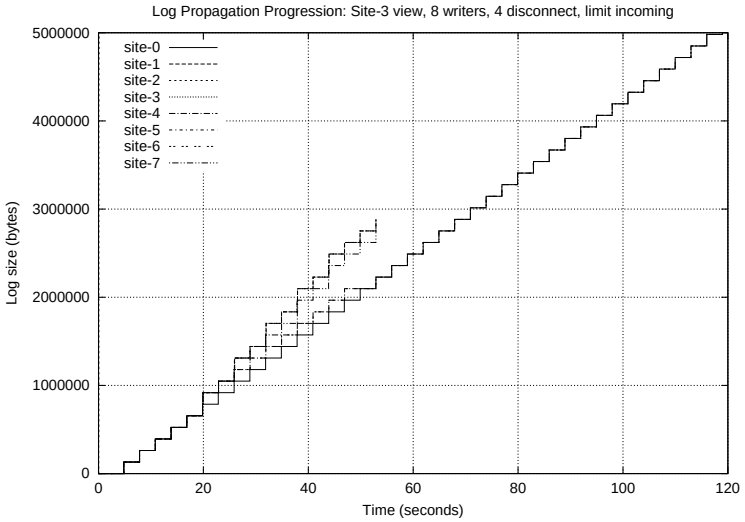


Figure 8: Multilog replication progression for 8 writers, throttled incoming traffic. 4 disconnect.

6 Performance evaluation

6.1 Multilog experiment

The multilog toolkit is a simple set of tools and dæmons that create, access and connect logs in multilogs. It is written in Python and uses TCP/IP for networking. It straightforwardly implements the design illustrated in Figure 2.

There are four main utilities in the toolkit. *LogServer* monitors a log and propagate updates. *LogClient* contacts a list of LogServers and locally replicates their logs. *LogTool* is a utility that can read or write a log. *MultilogD* is a simple dæmon that given a list of participants, combines the log-tools to implement a multilog.

6.1.1 Evaluation summary

The multilog structure decouples reads and writes and promotes mostly-linear access patterns. Therefore, the read/write performance of multilogs is dominated by the local filesystem and of the network stack. The purpose of this evaluation is to demonstrate this fact; the results are summarised in Table 4

Our performance goals are to scale to very large numbers of readers. The numbers of writers for a single document is expected to remain relatively small, on the order of tens of participants. This is typical for the internet society.

Efficient propagation from a small number of writers to a huge number of readers is possible in peer-to-peer networks, where recipients of data propagate them further. The net effect of such a solution is a high outgoing bandwidth and limited incoming bandwidth. In some of our experiments, we emulate this effect by severely limiting incoming traffic of participants while leaving outgoing traffic unlimited.

6.1.2 Detailed Results

The experimental setup involves one participant installed on each of 8 nodes interconnected with Gigabit Ethernet. The scenario is simple; Either one or all 8 participants begin to log a specific amount of data as fast as possible. At the same time, each participant reads his logs and records its replication progression over time. The writers and readers are implemented with LogTool instances, logs are served by LogServers and propagated updates are received and written to replicas by LogClients.

Table 4 lists representative results for running 1 and 8 concurrent writers both with and without limiting the incoming traffic. The average traffic is the sum of the incoming and outgoing traffic combined.

Our conclusion is that, when there is no limit in effect, multilog propagation performance is comparable to the maximum network bandwidth. When limits are in place, although overall bandwidth drops as expected, we observe that varying the number of writers between 1 and 8 has no effect. Furthermore, in all the experiments, disconnection of a participant does not disrupt the remaining ones, as illustrated in Figure 8.

6.2 Synthetic benchmarks

Sound schedules computation Telex computes sound schedules using the IceCube algorithm [9]. For a randomly generated graph containning 10000 actions and 20000 constraints, our algorithm computes a sound schedule in 200 ms. In running mode Telex uses incremental mode, and the computation is around a millisecond.

Reconciliation time We test the time to decide newly proposed actions. During this experience we compute a schedule every 100ms, and a proposal every 100ms. Each site submits 20 actions per second. The average time to commit an action using the FIFO algorithm (see Section 4.3) is 64ms.

6.3 STMBench

We run the STMBench7 benchmark [2], which emulates an application with a rich data structure and many different operations. We chose STMBench7 mainly because it demonstrates concurrency and conflicts. It also serves as an illustration of the use of Telex on a complex data structure.

STMBench7 was developed to exercise software transactional memories, based on the previous OO7 benchmark for object-oriented databases. STMbench7 builds an object graph with millions of objects and connected by numerous pointers. It contains 45 operations (21 read-only, 24 read-write) with various scope and complexity. We ported to Telex the read-write operations only. They all operate in a similar manner: traverse the data structure, reading one or many attributes of one or many objects, and modify an object.

An STMBench7 benchmark consists of two phases: creating a randomised object graph, and invoking operations. We measure only the second phase. There are four four main categories of operations:

- Long traversal: access large parts of the object graph, typically all "assemblies" and "atomic parts".
- Short traversals: access fewer objects, traversing the graph along a randomly chosen path.
- Short operations: choose a small number of objects, and perform an operation on these objects or in their neighbourhood.
- Structure modifications: randomly create or delete objects, or create or delete pointers between objects.

Each STMBench7 operations is mapped to a single action, hence will be isolated from concurrent operations.

Unexpectedly, in the original code, operations always commute, because the updates either swap two shared pointers, or add 1 modulo 2 to a shared integer. We therefore modified the benchmark so that, with some probability, updates either commute or do not commute.

Due to the large number of operations, we will not present a comprehensive list of constraints. Instead, we explain the rules we follow to define the constraints.

- Any modification to an object is causally dependent on the creation of the same object.
- Two actions that modify the same data are *NonCommuting*.

Number of sites	Time to benchmark (s)
1	20
2	21
3	21
4	21
5	21
6	21

Table 5: STMBench7 results

- If an action reads some data, and another action concurrent writes the same data, the former is *NotAfter* the latter. This ensures that, at all sites, the read will see the value before the write.

The results of the benchmark are shown in Table 5, executing the operations that modify data (not the structure). Performance is independent of the number of sites.

7 Lessons learned

Experience with applications and benchmarks has given us useful feedback, both regarding the implementation of Telex, as well as guidelines for application developers.

The current implementation of Telex suffers from excessive memory consumption. The ACG can quickly reach sizes of several tens of thousands of nodes, and is accessed concurrently by many threads. For instance, the scheduler parses the ACG at the same time as local and remote applications are modifying it. To avoid concurrency issues, the scheduler takes a full copy of the current ACG, which both consumes memory and is slow (in Java). Similarly, forward execution and rollback of applications involves copying their internal state, which can be very large. In both cases, an obvious solution (and future work) is to copy-on-write instead.

Translating application semantics into actions and constraints is a skill that takes time to acquire. We present some guidelines derived from our own experience. Note that these are not hard rules, and even may be conflicting.

The most important suggestion is to leverage commutativity as much as possible. As noted in the SWA, if all operations commute, consistency is trivial. The SWA example also shows that, sometimes, operations that appear non-commuting intuitively, can be designed or transformed to commute.

We learned that it is important to turn every piece of shared information into a separate document. In the initial design of SC, calendars were the only documents, and events were implicit in the calendars. This raised a number

of problems, because there was no obvious way to detect when a meeting conflict would impact another user indirectly. Separating out events as distinct documents solved this.

It is important to distinguish the sequential constraints (mainly, *NotAfter* and *Causal*) from the concurrency constraints (conflicts). The former are logged with their right-hand action; the latter are logged in response to *getConstraint*. Concurrency constraints are derived from the application invariants. For instance, in SRDA, the sequential specification forbids two tuples with the same identifier; it follows that concurrent *insert*s with the same identifier are in *Antagonism*.

One lesson from STMBench7 is to reason about high-level operations rather than low-level ones, in order to deal with fewer combinations. Furthermore, it is sometimes the case where high-level operations commute (for instance, increment and decrement a shared integer) even though their low-level implementations (e.g., reads and writes) do not.

However, in some cases, it may be simpler to reason about a small number of low-level primitives when they may be combined into a large number of operations. Currently, this kind of approach is complicated by the lack of support for transactional isolation, which is future work.

Constraints are hard to validate. We suggest two complementary approaches for future work. A compiler could generate actions and constraints from a high-level specification, and a checker could verify that all action-constraint combinations verify the application invariants.

8 Related work

State-machine replication [5] is based a total order of operations. This ensures consistency and correctness, but requires consensus at each operation, in the critical path of the application. In contrast, Telex's optimistic approach performs consensus in batches, in the background.

Optimistic replication [11] has been widely used, e.g., in replicated file systems (for instance, Coda [3] or Roam [10]) and for collaborative work (e.g., Bayou [17]). In these systems, replicas eventually converge, but they generally do not ensure any high-level correctness. For instance, the widely-used "last-writer-wins" (LWW) loses updates when conflicts occur, and does not maintain consistency between objects. Our constraints additionally ensure that application invariants are preserved.

Many replicated systems transmit new values or deltas (the state-based model). The operation-based model used in Telex (i.e., the system stores, transmits and replays logs of operations) retains more useful information for reconciliation. This is especially advantageous when high-level operations logically commute despite reading and writing the same physical data, as in our SC and SWA applications.

The literature on computer-supported co-operative work is widely based on operational transformation (OT) [14]. OT ensuring commutativity between concurrent operations by modifying them at replay time. Combined with reliable causal-order broadcast, this ensures convergence with no further concurrency control, but unfortunately OT appears limited to very simple text-editing scenarios. Telex takes advantage of commutativity when it is available, and supports any mix of commutative and non-commutative operations.

Coda's application-specific resolvers [4] or Bayou [17] give applications full control over conflicts. However, this requires developers to have a deep understanding of distributed systems issues. Instead, Telex requires stylised concurrency constraints from applications and takes care of conflict resolution in an application-indendent manner.

Telex has many similarities with Bayou [17] and also many differences. Bayou is an operation-based system that provides commitment; the committed state is guaranteed correct. However, Bayou relies on a primary site for commitment and the committed schedule is unpredictable. Furthermore, the system offers no help for reconciliation.

Constraints were used for reconciliation in the IceCube [9] system. IceCube relies on a primary site for commitment. In Telex, each site runs an IceCube engine (or any alternative) to propose schedules, and the commitment protocol ensures consensus based on these proposals. IceCube supports a richer set of constraints and can extract them from the applications' source code [8].

The Ivy peer-to-peer file system [6] reconciles the current state of a file from single-writer, append-only logs. There are several differences between Ivy and Telex. Ivy is designed for connected operation. Ivy is state-based and reconciles using a per-byte LWW algorithm by default. Whereas Telex localises logs per document, in Ivy there is a single global log for all the updates of a given participant. Reading any file requires scanning all the logs in the system, which does not scale well, although this is offset somewhat by caching. Ivy has no commitment protocol, therefore a state may remain tentative indefinitely.

The Ivy authors suggest that malicious updates can be removed after the fact, by ignoring the corresponding log. However, since Ivy does not record constraints, it cannot reconstruct a correct state: for instance, an update by an innocent user that depends on a previous but malicious update cannot be removed.

9 Conclusion

We presented the Telex system for shared mutable documents in a distributed system. We presented our motivations, its formal principles, the engineering design and implementation, and a number of prototypical applications. We also provided some performance measurements.

Our two main innovations are our principled approach based on action-constraint graph, and the multilog structure. The former enables Telex to provide correctness guarantees while maintaining application concurrency invariants. It also allows a clear separation between the responsibilities of applications, and those of the system. Thanks to constraints, applications specify precisely the level of consistency that they need, and the system enforces that level efficiently, and no more.

Independently of the ACG, we argue that the multilog structure is better adapted to shared, mutable documents than ordinary files, especially in a collaborative environment. A file system may provide guarantees for directories, but generally only best-effort consistency for files. Furthermore, the design goals of a file system are likely to be different from the needs of actual applications.

The multilog structure decouples reads and writes, avoids contention, encourages locality, and allows efficient linear access. Software at a higher level interprets the logs to reconstruct the application state. In our case, this is Telex, but it could be the application directly. Multilogs do not impose any unnecessarily limitations.

Telex is open source software, available at gforge.inria.fr/projects/telex2.

Acknowledgments

We thank Abhishek Gupta, of Indian Institute of Technology Guwahati, for implementing multilogs during an internship at INRIA and for authoring the SWA application, and Zenon Perisé, of Universitat Oberta de Catalunya, for authoring the Collaborative Environment application.

References

[1] Antony Chazapis, Georgios Tsoukalas, Georgios Verigakis, Kornilios Kourtis, Aristidis Sotiropoulos, and Nectarios Koziris. Global-scale peer-to-peer file services with dfs. In *GRID*, pages 251–258, 2007.

[2] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a benchmark for software transactional memory. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pages 315–324, 2007.

[3] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)*, 10(5):3–25, February 1992.

[4] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Usenix Tech. Conf.*, New Orleans, LA, USA, January 1995.

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

[6] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, Boston, MA, USA, December 2002. Usenix.

[7] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, pages 259–268, Banff, Alberta, Canada, November 2006. ACM Press.

[8] Nuno Preguiça, Marc Shapiro, and J. Legatheaux Martins. Automating semantics-based reconciliation for mobile transactions. In *CFSE'3 : conférence française sur les systèmes d'exploitation*, pages 515–524, La-Colle-sur-Loup, France, October 2003.

[9] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Int. Conf. on Coop. Info. Sys. (CoopIS)*, volume 2888 of *Lecture Notes in Comp. Sc.*, pages 38–55, Catania, Sicily, Italy, November 2003. Springer-Verlag GmbH.

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

[11] Yasushi Saito and Marc Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, March 2005.

[12] Marc Shapiro, Karthikeyan Bhargavan, and Nishith Krishna. A constraint-based formalism for consistency in replicated systems. In *Int. Conf. on Principles of Dist. Sys. (OPODIS)*, number 3544 in Lecture Notes in Comp. Sc., pages 331–345, Grenoble, France, December 2004.

[13] Marc Shapiro, Nuno Preguiça, and James O'Brien. Rufis: mobile data sharing using a generic constraint-oriented reconciler. In *Conf. on Mobile Data Management*, pages 146–151, Berkeley, CA, USA, January 2004.

[14] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *Trans. on Comp.-Human Interaction*, 5(1):63–108, March 1998.

[15] Pierre Sutra, João Barreto, and Marc Shapiro. Decentralised commitment for optimistic semantic replication. In *Int. Conf. on Coop. Info. Sys. (CoopIS)*, Vilamoura, Algarve, Portugal, November 2007.

[16] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994.

[17] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399