Telex: A Semantic Platform for Cooperative Application Development

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

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Telex: A Semantic Platform for Cooperative Application Development*

Lamia Benmouffok,[a] Jean-Michel Busca,[a] Joan Manuel Marquès,[b]
Marc Shapiro,[a] Pierre Sutra,[a] Georgios Tsoukalas[c]

[a] INRIA Paris-Rocquencourt & LIP6; [b] UOC, Barcelona, Spain; [c] Ntl. Technical U. of Athens, Greece

Abstract

Developing write-sharing applications is challenging. Remote and offline data sharing are increasingly important. We propose a generic platform called Telex to ease development and to provide guarantees. Telex is driven by application semantics. Telex takes care of replication and persistence, drives application progress, and ensures that replicas eventually agree on a correct, common state. We show by example how application design proceeds from high-level application invariants to application-provided parameters that guides Telex. The main data structure of Telex is a large, replicated, highly dynamic graph; we discuss the engineering trade-offs for such a graph and our solutions. Finally, we report an experimental evaluation of Telex based on a cooperative calendar application and on benchmarks.

1. Introduction

Remote and offline data sharing are increasingly important. Examples include a shared wiki, cooperative offline editing with Google Gears, enterprise platforms such as Notes or Groove, collaborative code repositories CVS or SVN, and so on.

In distributed systems, access to shared data is a performance and availability bottleneck. To deal with failures, latency, and large scale, a common approach is *optimistic replication (OR)*. OR decouples data access from network access: it allows a processor to access a local replica without synchronising. A site makes progress, executing uncommitted actions, even while others are slow or unavailable. Local execution is tentative and actions may roll back later. An OR system propagates updates lazily, and ensures consistency by a global a posteriori agreement on a same state [1].

Implementing a collaborative application is complex. The programmer needs to manage asynchronous communication between collaborators, data replication, conflict detection and resolution, and eventual agreement. Besides, end-users should be able to make sens of the collaborative work: (i) they should not be overwhelmed or confused by remote updates and roll-back, (ii) conflict detection should be relevant, (iii) and users should be able to express preferences for conflict resolution. Additionally, the application needs to be responsive, and to provide garanties about the data persistence and convergence.

To solve these issues, current cooperative application developer uses ad-hoc approaches that do not guarantee correctness in the general case, and leave much manual work to the user.

In contrast, we propose an open platform called Telex that helps developers to build collaborative applications faster. Telex supports an optimistic replication model for sharing stateful data in a decentralised way over a large-scale network. Telex eases application development by taking care of the common data-related requirements of these systems, such as data replication and consistency issues. Based on a principled approach, Telex guaranties that replicas never violate safety and converge eventually. The Telex platform is designed for robustness, flexibility and performance.

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

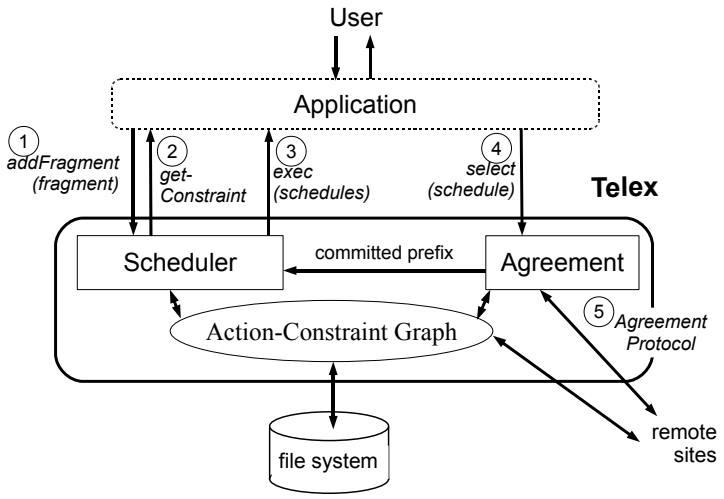

Figure 1: Telex site architecture and execution model

This paper reports on the design and implementation of Telex, and on our experience with several applications. Its key contributions are the following: (1) Guidelines for designing cooperative applications, to make the best use of a semantic platform (Section 3). (2) Addressing the engineering challenges of a large, replicated, mutable, concurrently-accessed graph data structure (Section 4). (3) Assessment of Telex based on application developer experience and on benchmarks (Section 5).

2. Telex overview

Telex is a generic platform to ease development of collaborative applications. Telex supports optimistic sharing over a large-scale network of computers or *sites*. Telex allows the application programmer to concentrate on core functionality, delegating distribution, replication, persistence and consistency issues to Telex. The Telex programming API enables the integration of the application semantics in the process. Telex implements a classical approach for optimistic replication: updates are logged, propagated and eventually re-executed at each user's site [1].

Programming an application using Telex API requires understanding the following basic concepts. We call *document* any shared mutable data managed by an application. A document may be, for instance, a user's calendar or a text document. Telex replicates a document at the sites where it is used.

Telex drives application progress guided by application-provided parameters: (i) *actions*, represent atomic operations performed by the application, (ii) *constraints*, represent semantic relations between two actions. For instance, an action inserting a line in a text ?le and another updating the same line, relate by a *causal* constraint. A set of actions and constraints constitutes an *Action-Constraint Graph* (ACG), where actions are the nodes of the graph, and constraints the edges and arcs. A document state results form the execution of a sequence of actions, called a *schedule*. A schedule must satisfy the constraints of the ACG.

Since applications execute optimistically, local document state remains tentative until all actions are received and conflicts resolved by the *Agreement Protocol*, committing a *committed prefix*, and possibly leading the application to roll-back. Agreement is a generalisation of database commitment. Agreement is asynchronous, in the background, it should not stop replicas from diverging past what is being agreed.

We now present the Telex life-cycle at a high level of abstraction, referring to Figure 1. We provide more detail later in the paper.

1. The application opens a document using Telex open primitive. Telex imports a copy of the corresponding document.
2. A user utters a command to the application. The application computes the corresponding actions. Actions can be augmented with constraints, explained in more details in the example hereafter. The application entrust Telex with a set of actions and constraints by calling the *addFragment*[2] primitive of Telex (Step 1 in Figure 1) . Telex logs them in the ACG, stores them persistently and propagates them to each site replicating the document.
3. To verify whether an action received from a remote site causes a conflict, Telex up-calls the application's *getConstraint* interface with a pair on actions to check, (step 2 in figure 1). This upcall is repeated for every pair of actions that might conflict.
4. Telex iteratively computes an initial state and a schedule that: *(i)* includes the committed prefix (explained shortly), *(ii)* combines actions received so far, and *(iii)* satisfies all the constraints. In the presence of conflicts, multiple alternative schedules are possible.

   Telex instructs the application to execute this schedule by up-calling the application's *exec* interface, (Step 3 in Figure 1). This may be repeated several times (at the application's request) for alternate schedules.
5. If the user or the application is satisfied with the resulting state, it calls Telex's *select* interface (Step 4 in Figure 1). This encourages Telex to extend the same schedule in the future, rather than consider an alternative, and to propose this schedule as a candidate for agreement.
6. Telex sites iteratively agree on a monotonously-extending committed prefix. The committed prefix is guaranteed to be compatible with the schedules that the applications *select*ed.

3. Collaborative application design

This section examines the design principles for collaborative applications using Telex. We illustrate with the example of a cooperative calendar application, Sakura. We will assess our experience designing applications in Section 5.1.

3.1. The Sakura application description

The Sakura application lets users collaboratively make decisions such as managing calendars and scheduling joint meetings. Any user may share his calendar and meetings with others, create a meeting, invite users to a meeting, change its time, or cancel it.

Sakura maintains a strong invariant: no double-booking. In contrast to common calendar systems such as Doodle[3], Sakura over Telex ensures consistency even when a user is tentatively engaged in several meetings, i.e., the agreement protocol will not commit conflicting meetings. Calendars may be connected by common meetings in unpredictable ways. The classical approach would store all users' calendars in a single database to ensure consistency. Instead, for scalability and privacy reasons, Sakura maintains a calendar document per user, replicated at a site only if the local user needs it. Such partial replication is challenging, but not application responsibility.

3.2. Using constraints

The first step is to specify the application with regard to the optimistic execution model. The design must specify: *(i)* the data or documents that the application manages. *(ii)* the prototype of actions performed on the documents *(iii)* and a set of rule defining relations between actions expressed in term of Telex supported constraints, as we illustrate shortly.

Telex supports a predefined set of constraints, presented in Table 1, which the programmer can combine to express a rich semantics. The first three are primitive, the last three are combinations of the primitives.[4] We refer the reader interested in more information to a previous publication [5].

[2] Named thus because it adds a piece to the existing ACG.

[3] www.doodle.com

[4] *Atomic* does not ensure transactional isolation; an isolation constraint will be added in the future. Currently, to achieve isolation,

Name	Notation	Meaning
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	$A \between B$	N/A
Atomic	$A \overset{\lhd}{\rhd} B$	A and B both execute or neither does
Causal	$A \overset{\lhd}{\rightarrow} B$	B executes after A, and only if A succeeds
Antagonism	$A \overset{\leftarrow}{\rightarrow} B$	Conflict: A and B never both in same schedule

Table 1: Constraint types. (A, B are arbitrary actions)

To discuss how to design for replication in more detail, consider a developer who has a rough design for a sequential version of an application, its data structure, invariants and operations. What are the steps to make the application run well in an optimistic model?

3.2.1. Actions and sequential constraints

Let us first consider sequential execution, ignoring concurrency and conflicts for now. An application submits to Telex actions, as well as causality and atomicity constraints. Dependence translates to a *Causal* constraint; for instance, if action B reads some value used by A, the application should add a fragment containing both action B and constraint $B \overset{\lhd}{\rightarrow} A$. (A is not in the fragment if it is already known.) Similarly, to ensure action composition (either both A and B execute or neither does), the application adds a fragment containing A, B and $A \overset{\lhd}{\rhd} B$.

3.2.2. Concurrency and constraints

When Telex suspects conflicts between concurrent actions A and B, Telex upcalls the application using the *getConstraint (A,B)* interface. It is the programmer responsibility to implement in the *getConstraint* method, the rules to compute the correct constraints between actions. We discuss what those rules should be.

Actions commute if either relative execution order yields the same final state. *getConstraint* indicates commutativity by returning nothing.

The two conflict responses are *Antagonism* and *NonCommuting*. Actions are antagonistic when they cannot both execute together; this should be obvious based on action types and their arguments. A Sakura illustrative rule is: "concurrent actions inviting the same user to different meetings on the same time are *antagonistic*"; Sakura checks actions' arguments and returns $A \overset{\leftarrow}{\rightarrow} B$.

In all other cases where two actions are incompatible, *getConstraint* should return $A \between B$. A Sakura exemple is "concurrent *setDate* actions performed on the same meeting *do not commute*".

Later, the scheduler will pick an arbitrary execution order, for instance A; B. As this might violate an application invariant, the application must check its invariants at run time. An example would be ensuring that no user has more than 10 meetings in any given week.

When a run-time check fails, the corresponding schedule is unsafe. The application may request an alternative schedule from the scheduler. It must not select an unsafe schedule as an agreement proposal (i.e. a suggested schedule for the agreement) , to ensure that agreed schedules are safe.

3.2.3. Constraints and application design

Well-chosen constraints are critical to performance and user experience. Commutativity is the most favorable case, as it bypasses the expensive mechanisms. Antagonism is next: it *Antagonism* is resolved by the scheduler. Non-commutativity is by far the worst, as a *NonCommuting* conflict is deferred to the application at run time.

---

the user must manually group operations into a single action.

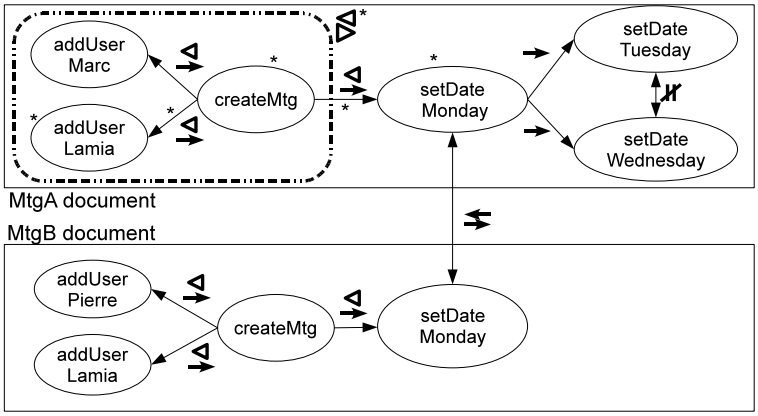

Figure 2: Sakura example ACG. Stared actions and constraints refer to section 5.3

A simple way to decrease conflicts is to break out each piece of mutable information into a separate, fine-grain document. Sometimes, operations logically commute but conflict in the implementation; it may be necessary to redesign the internal data structure in this case [6].

Run-time checking of application invariants is required for safety, but is not related to concurrency, since applications only ever execute sequential schedules. Indeed, the corresponding sequential application would include the same checks.[5]

Conversely, conflict constraints are not strictly necessary for safety. In the absence of conflict information, Telex remains correct but the heuristic will be ineffective, and in case of conflict, the scheduler will not explore alternatives, as in Bayou [7]. Constraints in Telex avoid such dead-ends.

3.3. An example

Let us consider the execution scenario illustrated in Figure 2. Marc invites Lamia to meeting *MtgA* on Monday, by logging four actions: *createMtg* creates the meeting, *addUser* invites a user (one for Marc, another for Lamia), and *setDate* sets the time. The latter three actions depend on the first one, as captured by the $\overset{\triangleleft}{\rightarrow}$ constraint.

Suppose now that Marc wants to create *MtgA* only if both users can attend. Using the $\overset{\triangleleft}{\triangleright}$ constraint, either all actions succeed, or none executes.

Now consider that Pierre concurrently invites Lamia to *MtgB* on Monday. The *getConstraint* upcall to Sakura returns an $\overset{\leftarrow}{\rightarrow}$ constraint between the two antagonistic actions. This ensures that agreement will commit at most one of the Monday invitations.

Finally, Lamia wishes to move *MtgA* to Tuesday. This should override the previous Monday date by running after it, but does not depend on it succeeding or not. Therefore the application sets the weaker constraint *MtgA.setDate (Monday)* $\rightarrow$ *MtgA.setDate (Tuesday)*.

For an example of non-commuting concurrent actions, consider Lamia setting *MtgA* to Tuesday, while Marc concurrently sets it to Wednesday. To ensure that all sites observe the same outcome, Sakura returns *MtgA.setDate (Tuesday)* $\curlywedge$ *MtgA.setDate (Wednesday)* to *getConstraint*.

4. Engineering a dynamic, replicated graph

The ACG is dynamic graph accessed by concurrent threads, and that must be efficiently stored and transmitted over the network. This raises engineering issues of general interest.

[5] Databases make a similar assumption: this the "C" of the famous ACID properties that define database transactions.

4.1. Graph definitions

An ACG is a graph that can only grow, although obsolete parts can be garbage-collected. The ACG nodes are (opaque) actions. Its arcs and edges are constraints, labeled by constraint type ($\rightarrow$, $\lhd$ or $\nr$).

A node may have a colour; the available colours are white and black.

A *sound cut* (or just *cut* when there is no ambiguity) is a conflict-free sub-graph, i.e., formally: (i) every node is either white or black; (ii) In the white sub-graph, nodes connected by $\nr$ edges and $\rightarrow$ arcs are ordered, and this order is consistent with $\rightarrow$, (iii) if white node B is in the cut and A $\lhd$ B, then A is also in the cut and also white.

A *sound schedule* (or just schedule) is a totally-ordered sound cut, i.e., one possible execution of the cut. All schedules that order a given cut are equivalent, since they differ only by the order of commuting actions.

Soundness is strongly related to consistency, as we explain now. (1) In a Telex schedule, white actions execute, and black ones do not. (2) The constraint A $\lhd$ B captures the application requirement that B implies A. In a sound cut, if B is white, then A is also white; conversely if A is black, B is too. This is obviously equivalent. (3) Constraint A $\rightarrow$ B requires that if A and B both execute, they execute in order. By definition, the ordering of a cut satisfies this requirement. (4) The conflict constraint A $\overleftrightarrow{\rightarrow}$ B forbids executing both A and B in the same schedule; more generally, cycles in the $\rightarrow$ graph are forbidden. Since, in a cut, white nodes are ordered, there can be no white cycles; therefore, at least one node in every cycle must be black.

Here, we do not discuss conflict constraint A $\nr$ B, since it is only a liveness requirement.

A *snapshot* is some data labeled with a cut. In Telex, we store on disk occasional snapshots of application state, to speed up rollback and replay.

Since replicas of the ACG may differ, we sometimes need to know whether the local ACG contains sufficient information; this is captured by the concept of closure. We say a node A is *enables-closed* if all the predecessors of A by $\lhd$ arcs, known at any site, are in the local ACG, and transitively their predecessors. A is said *order-closed* if all the predecessors of A by $\rightarrow$ arcs, and all its neighbours by $\nr$ edges, are in the ACG, and transitively their predecessors and neighbours.

4.2. In-memory representation

Conceptually, the ACG contains all actions and constraints in the system. In practice we only instantiate particular sub-graphs. At some site, only the actions and constraints of currently-active documents need to be in memory. In the common case, documents are independent, and we instantiate disconnected sub-graphs, one per document. Occasionally, however, there are arcs or edges between documents, and their sub-graphs are merged.

The ACG is accessed by several concurrent threads. For instance, applications, loggers and agreement add fragments to the ACG. The scheduler must observe a consistent state of the ACG to compute sound cuts, and the agreement module uses it to check the soundness of proposals. In our initial implementation, both the scheduler and agreement worked in the background after atomically making a full copy of the current in-memory ACG. However those repeated copies had prohibitive cost.

Our current implementation simply locks the graph. Despite the reduced concurrency, memory footprint and performance have improved tremendously. This approach works well enough for our current applications and benchmarks. We are planning more concurrent implementation for the future.

Another performance improvement is provided by caching a compact version of the ACG, sufficient for checking and generating proposals. We omit the details as they are somewhat technical. Later, we discuss breaking the ACG into independent connected components also improve performance considerably. Performance benchmarks appear in Section 5.

4.3. External representation

The ACG is replicated over the network and stored on disk. We discuss the challenges of efficiency and scalability, providing strong guarantees over a best-effort substrate, supporting disconnected operation, and enabling garbage collection. Performance measurements are deferred to Section 5.2.

Updating shared information on disk is problematic. If multiple processes write to the same file, they suffer synchronisation effects (write contention). We designed a storage and communication format that avoids this overhead, called a *multilog*. Multilogs generalise the concept of an append-only log. Each document forms a separate multilog, containing the actions and constraints relative to that document. The ACG is the (conceptual) union of all documents.

The multilog is composed of a set of logs, one per user. A user appends fragments to his designated log, at his local site, thus avoiding write contention. The system replicates every log to the other sites. Reading a document requires checking the logs of that document only.

There are four types of log records: actions, constraints, fragment-begin and fragment-end. The order of action records within a log is not significant (any significant ordering is indicated by a $\rightarrow$ constraint), and incomplete fragments are ignored.

A constraint record may be in-log (the common case, implemented the most efficiently), in-document (slightly more expensive) or cross-document (assumed rare). For instance, application uses a cross document constraint to ensure that a text file contains the date of a meeting, the user groups the two corresponding actions (setting the date and writing the file) with $\overset{\triangleleft}{\triangleright}$. This ensures that if the meeting request does not go through, the file will not be written, and vice-versa.

A document is replicated on all the sites that require it, and is cached on disk. The persistent cache supports disconnected operation: a disconnected writer logs to its local disk, and sends the new records to remote replicas when it reconnects. Similarly, a disconnected site reads from its cached logs, and pulls missing records when reconnecting.

We integrated multilogs into a best-effort peer-to-peer file system, VOFS [8]. On disk, a document is structured as a directory, containing meta-data files (e.g., snapshots) and a sub-directory of logs. Each log is itself a directory of log chunks. Writing a record appends to the current chunk, until a maximum size is reached, at which point a new chunk is created.

A chunk whose entire contents is decided (committed) may be garbage collected locally if there is a later snapshot of the state.

5. Assessment

The Telex platform is 33,000 lines of commented Java code, of which 7,000 for scheduling and 2,000 for the agreement module.

5.1. Application experience

A number of applications were developed both by the Telex designers (INRIA/LIP6), and by partners in the Grid4All project.

We ported STMBench7, a concurrent transactional benchmark, initially designed for transactional memories [9]. It has a rich data structure and different operations. Among well-known benchmarks, STMBench7 was chosen because it stresses Telex with a high load of concurrent reads and updates. It also serves as an illustration of the use of Telex on a complex data structure. It contains 12,700 lines of commented Java code (LOC): approximately 7,700 for STMBench7 proper, and 5,000 to for interfacing to Telex.

A replicated dictionary data type was implemented as a building block for other applications, such as a shared address book. It took only a few days to deal with with distribution and reconciliation, i.e., to express the concurrency semantics as actions and constraints. It contains 1,460 LOC, of which 43% of which are related Telex.

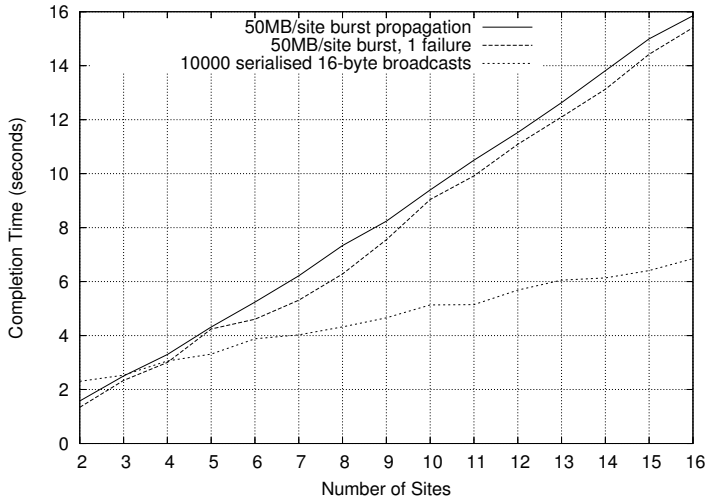

Figure 3: Completion times for multilog access experiments

The Sakura calendar application, described earlier, is interesting as a representative of decision-making applications. It is a command-line application, containing 3,400 LOC, of which approximately 1,000 for interfacing with Telex. Adding a GUI would increase the first figure.

The Decentralised Collaborative Environment (DCE) is a collaboration workspace enabling users to share sub-workspace, file histories and discussion forums. It uses Telex to manage workspace meta-data; file versions are stored in the VOFS file system. Thanks to Telex, it is decentralised, and can be used in disconnected mode.

The developers of all these examples found that Telex eases their job. Telex takes care of replication and reconciliation. The developer may concentrate application functionality.

5.2. Multilog benchmarks

In this section we assess the performance of our multilog on-disk data structure. To factor out any Telex-specific overhead, we developed a separate multilog toolkit that accesses multilogs and replicates them over the network. We deployed it on a cluster of high-end computers, interconnected by Gigabit Ethernet, using a memory-only file system. The experiments saturate the cluster's networking capacity.

5.2.1. Broadcast throughput

In our first experiment, N sites share a multilog. Each site logs a burst of $50 \cdot 10^6$ bytes, then reads from all other sites. The solid line in Figure 3 shows the results. Concurrent writes are completely independent. This shows that a multilog is accessed at a high rates with minimal overhead, limited only by the total system bandwidth.

To demonstrate fault resilience, we repeat the experiment, one site becoming unreachable after approximately 1s, and the others continue, reading from $N - 1$ sites (dashed line in the figure). For $4 \leq N \leq 16$, the average incoming + outgoing traffic is 90MB/s at each site, computed by taking the total traffic per site ($2 \cdot (N-1) \cdot 50 \cdot 10^6$ bytes) and dividing by the completion time.

This experiment over a cluster is not representative of Internet environments. In future work, we will make use of existing high-volume transmission protocols such as SplitStream [10].

5.2.2. Broadcast latency

To evaluate the latency of broadcasting a log record, each of N sites logs a 16-byte record and then waits for all other sites' entries. This is repeated 10,000 times. The dashed line in Figure 3 shows that performance scales linearly with the number of sites, limited only by system bandwith.

5.3. Schedule computation

In this section, we discuss schedule computation. Its performance depends on the number of actions and constrains, and the constraints pattern.

The experiment uses a Sakura benchmark. It creates multiple independent meetings, with no conflict between meetings. We invite one user to every meeting. A meeting corresponds to 3 actions and 5 constraints, as described in Section 3.3.

This execution pattern shows how Telex behaves when only one solution is possible. Our experiments shows that for 10000 meetings, i.e 30000 actions and 50000 constraints, Telex takes approximately 82 milliseconds to compute the first solution, and 93 milliseconds to notice that there are no alternative solutions.

6. Related work

Optimistic replication [1] has been widely used, e.g., in replicated file systems [11] and for collaborative work [7]. Such systems generally do not ensure any high-level correctness. For instance, the widely-used "last-writer-wins" (LWW) protocol loses updates when conflicts occur, and does not maintain consistency across objects.

Coda-style application-specific resolvers [12] give applications full control over conflicts. However, they require developers to have a deep understanding of distributed systems issues. Telex provides high-level, application-independent reconciliation parametrised by constraints.

The prototypical sharing tool is a shared text editor, such as a version-control system or a Wiki. These require manual repair when updates overlap, a labour-intensive task. The decentralised version-control system DARCS uses operational transformation [13] to commute edits. This is beneficial as it avoids all concurrency control. Telex makes use of the commutativity property, and supports any mix of commutative and non-commutative actions.

Bayou is a general-purpose system designed for disconnected operation [7]. It supports application-defined operations, optimistic execution, and checks invariants at run-time. Bayou commits at a central site, in the background, in arbitrary order, thus suffering spurious aborts. Telex borrow many of these ideas; additionally Telex has constraints to guide reconciliation, and decentralised agreement.

Constraints and heuristic scheduling were introduced in IceCube [14, 15]. IceCube runs on a primary site. Joyce is a distributed IceCube system [16]. Joyce has a single application (a text editor) and was never deployed at scale. This paper examines application design guidelines and engineering challenges that were not considered in Joyce.

The Ivy peer-to-peer file system [17] 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 user. 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.

7. Conclusion

In this paper, we presented the Telex platform for sharing mutable data in a decentralised way over a large-scale network. Telex supports an optimistic application model, and takes over system issues such as replication, persistence, disconnected operation, and reconciliation. Telex eases application development, by separating system from application functionality, and by setting clear goals for the developer thanks to its constraint interface. Telex is based on a principled approach, actions and constraints. They guide Telex away from conflicts and help make reconciliation a high-level, guided activity. Applications specify the consistency they require, and the system enforces it.

One contribution to this paper is some design guidelines for optimistic application developpers.

The engineering challenges caused us to design some original data structures, the ACG and the multilog. Independently of Telex, we argue that graphs and multilogs are of general interest in a collaborative environment. For instance, on-disk multilogs decouple reads and writes, avoid contention, encourage locality, and allow efficient linear access.

A welcome addition to the application interface would be a multi-action transaction construct: currently, to achieve isolation, the user must manually group operations into a single action. Application checkpoint and restore currently clone the whole memory, a very expensive operation in Java; we plan to investigate copy-on-write and similar techniques. Concurrent access to the ACG could be improved. We need more experience with the agreement protocol to remove unnecessary overheads and scale up much further. As in IceCube, we should extract constraints automatically from application source code [18].

Telex is available at gforge.inria.fr/projects/telex2 under a BSD licence.

References

[1] Saito, Y., Shapiro, M.: Optimistic replication. Computing Surveys **37**(1) (March 2005) 42–81

[2] Benmouffok, L., Busca, J.M., Shapiro, M.: Semantic middleware for designing collaborative applications in mobile environment. In: Middleware for Network Eccentric and Mobile Apps. W. (MiNEMA), Magdeburg, Germany (September 2007) 58–61

[3] Krasner, G., Pope, S.: A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. Journal of Object Oriented Programming (1988)

[4] Sutra, P., Barreto, J., Shapiro, M.: An asynchronous, decentralised commitment protocol for semantic optimistic replication. Rapport de recherche 6069, Institut National de la Recherche en Informatique et Automatique, Rocquencourt, France (December 2006)

[5] Shapiro, M., Bhargavan, K., Krishna, N.: 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., Grenoble, France (December 2004) 331–345

[6] Preguça, N., Marquès, J.M., Shapiro, M., Letia, M.: A commutative replicated data type for cooperative editing. In: Int. Conf. on Distributed Comp. Sys. (ICDCS), Montéal, Canada (June 2009)

[7] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: 15th Symp. on Op. Sys. Principles (SOSP), Copper Mountain, CO, USA, ACM SIGOPS, ACM Press (December 1995) 172–182

[8] Chazapis, A., Tsoukalas, G., Verigakis, G., Kourtis, K., Sotiropoulos, A., Koziris, N.: Global-scale peer-to-peer file services with DFS. In: Int. Conf. on Grid Computing (GRID 2007). (2007) 251–258

[9] Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: a benchmark for software transactional memory. In: Euro. Conf. on Comp. Sys. (EuroSys). (2007) 315–324

[10] Castro, M., Druschel, P., Kermarrec, A.M., Nandi, A., Rowstron, A.I.T., Singh, A.: SplitStream: high-bandwidth multicast in cooperative environments. In: Symp. on Op. Sys. Principles (SOSP), Lake Bolton, NY, USA (October 2003) 298–313

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

[12] Kumar, P., Satyanarayanan, M.: Flexible and safe resolution of file conflicts. In: Usenix Tech. Conf., New Orleans, LA, USA (January 1995)

[13] Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. Trans. on Comp.-Human Interaction **5**(1) (March 1998) 63–108

[14] Kermarrec, A.M., Rowstron, A., Shapiro, M., Druschel, P.: The IceCube approach to the reconciliation of divergent replicas. In: Symp. on Principles of Dist. Comp. (PODC), Newport, RI, USA, ACM SIGACT-SIGOPS, ACM Press (August 2001)

[15] Preguiça, N., Shapiro, M., Matheson, C.: Semantics-based reconciliation for collaborative and mobile environments. In: Int. Conf. on Coop. Info. Sys. (CoopIS). Volume 2888 of Lecture Notes in Comp. Sc., Catania, Sicily, Italy, Springer-Verlag GmbH (November 2003) 38–55

[16] O'Brien, J., Shapiro, M.: An application framework for nomadic, collaborative applications. In: Int. Conf. on Dist. App. and Interop. Sys. (DAIS), Bologna, Italy, IFIP WG 6.1 (June 2006) 48–63

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

[18] Preguiça, N., Shapiro, M., Legatheaux Martins, J.: Automating semantics-based reconciliation for mobile transactions. In: CFSE'3 : conférence française sur les systèmes d'exploitation, La-Colle-sur-Loup, France (October 2003) 515–524