Project no. 034567

Grid4All

Specific Targeted Research Project (STREP)
Thematic Priority 2: Information Society Technologies

D 1.3 Specifications and Models for the Actions-Constraints Framework

Due date of deliverable: January, 2009.

Actual submission date: January, 2009.

Start date of project: 1 June 2006 Duration: 30 months

Organisation name of lead contractor for this deliverable: INRIA Regal

Revision: 2

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level

PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contents

Abbreviations and acronyms used in this document

Abbrev./acronym	Def.	Description
ACF		Action-Constraint Framework
Grid4All		The FP6 STREP project that aims to democratise access to the Grid, through the use of peer-to-peer technologies. The co-ordinator is France Télécom; the other partners are a SME from Spain, and research labs from Greece, France, Spain, and Sweden.

Grid4All list of participants

Role	Part. #	Participant name	Part. short name	Country
CO	1	France Telecom	FT	FR
CR	2	Institut National de Recherche en Informatique en Automatique	INRIA	FR
CR	3	The Royal Institute of technology	KTH	SWE
CR	4	Swedish Institute of Computer Science	SICS	SWE
CR	5	Institute of Communication and Computer Systems	ICCS	GR
CR	6	University of Piraeus Research Center	UPRC	GR
CR	7	Universitat Politècnica de Catalunya	UPC	ES
CR	8	ANTARES Produccion & Distribution S.L.	ANTARES	ES

1 Introduction

Maintaining consistency in a distributed system is a difficult and challenging task. The classical approach is state machine replication (aka. Consensus [8]). In a system implementing Consensus, every site replicates the same set of data items. When a client wants to access a shared data item, it proposes an operation to the system. Consensus ensures that all replicas execute the same set of proposed operations in the same order.

With state machine replication, every site must replicate everything. This schema does not scale beyond a few tens of sites, and is not consequently appropriate for grids. Moreover Consensus does not take into account the semantics of operations: if two proposed operations are commuting, Consensus still pays to order them.

The Action-Constraint Framework [9]) leverages both partial replication and operation semantics. ACF clarifies the understanding of consistency, makes it easier to compare protocols, and helps with the design of new solutions. In ACF every site maintains a graph, where nodes are operations accessing shared data, and edges represent semantic links between them. The complexity of the consistency problem is related to the shape of the graph. A consistency protocol is a particular solution to a graph problem.

ACF was introduced by Shapiro et al. as a model to understand partial replication issues in failure-free distributed systems [9]. In the context of Grid4All, we (i) extended ACF to the crash-recovery model, (ii) applied it to database replication [15, 17], (iii) implemented it inside the Telex semantic store [3], and (iv) used it both to design distributed applications [5], and concurrency control protocols [11].

This paper presents the ACF model, its implementation in Telex, and illustrates how to use it. Deliverable D3.4 describes in detail our Telex platform, and the applications we developed on top of it.

We structure the rest of this document as follows: Section 2 presents ACF in details. Section 3 exposes our implementation of ACF in Telex. Section 4 is a tutorial on how applications use ACF through an example of shared text editor.

2 **The Action-Constraint Framework**

The Action-Constraint Framework helps to write and implement a specification of distributed system. ACF eases to determine the correctness and liveness conditions to build a consistent system despite concurrent access to shared data.

2.1 Actions, constraints and multilogs

We postulate a universal set of *actions* $\mathbb{A}$. A *constraint* is a relation over $\mathbb{A} \times \mathbb{A}$. Five constraints are of particular interest in our framework: $+$, $-$, $\rightarrow$, $\lhd$ and $\nparallel$; respectively pronounced "commit," "abort," "not after," "enables" and "non-commuting." The constraints $+$ and $-$ are unary relations over $\mathbb{A}$, $\rightarrow$, $\lhd$ and $\nparallel$ are binary. Their semantics will be explained shortly.

Our central structure in ACF is the *multilog*. A multilog is a sextuple $(K, +, -, \rightarrow, \lhd, \nparallel)$ where K is a set of actions, and $+$, $-$, $\rightarrow$, $\lhd$, $\nparallel$ are constraints over K.

We define union, intersection, difference, etc., between multilogs as component-wise operations. For instance, let $M = (K, +, -, \rightarrow, \lhd, \nparallel)$ and $M' = (K', +', -', \rightarrow', \lhd', \nparallel')$ be two multilogs. Then, $M \cup M' = (K \cup K', + \cup +', - \cup -', \rightarrow \cup \rightarrow', \lhd \cup \lhd', \nparallel \cup \nparallel')$. By abuse of notation, we also use the union operator to add an element to a single component, which should be clear from the context. For instance, $M \cup \{\alpha\}$ adds α to the K component, i.e., $M \cup \{\alpha\} = (K \cup \{\alpha\}, +, -, \rightarrow, \lhd, \nparallel)$. Similarly $M \cup \{\alpha \rightarrow \beta\}$ adds $\{\alpha, \beta\}$ to K, and the pair (α, β) to the $\rightarrow$ component. The notation $\alpha^- \in M$, or just (when clear from the context) α^-, is a shorthand for "(α, α) is in the $-$ component of M".

2.1.1 Schedules of multilogs and classes of schedules

Let E be a subset of $\mathbb{A}$. We call schedule a couple $S = (E, <_S)$ where $<_S$ is a strict total order over E. We note $\mathbb{S}$ the universal set of schedules over $\mathbb{A}$.

Given a multilog $M = (K, +, -, \rightarrow, \lhd, \nparallel)$, we say that $S = (E, <_S)$, with $E \subseteq K$, is a schedule of M, iff :

$$\begin{array}{l}
\forall \alpha, \beta \in K \\
\alpha^- \in M \Rightarrow \alpha \notin E \\
\alpha^+ \in M \Rightarrow \alpha \in E \\
\alpha \lhd \beta \in M \Rightarrow (\beta \in E \Rightarrow \alpha \in E) \\
\alpha \rightarrow \beta \in M \Rightarrow (\alpha, \beta \in E \Rightarrow \alpha <_s \beta)
\end{array}$$

We note $\Sigma(M)$ the set of schedules of M. The constraints: $\lhd$, $-$, $+$ and $\rightarrow$, restrict which schedules may appear in $\Sigma(M)$.

In contrast, $\nparallel$ divides $\Sigma(M)$ into equivalence classes of schedules. Let $M = (K, +, -, \rightarrow, \lhd, \nparallel)$ be a multilog. Two schedules $S = (E, <_S)$ and $S' = (E', <_{S'})$ of $\Sigma(M)$ are *equivalent* according to $\nparallel$, $S \sim S'$, iff:

$$\forall \alpha, \beta \in K, \quad \left\{ \begin{array}{l} \alpha \in E \Leftrightarrow \alpha \in E' \\ \alpha \nparallel \beta \in M \Rightarrow (\alpha <_S \beta \Leftrightarrow \alpha <_{S'} \beta) \end{array} \right.$$

We note $\Sigma(M)_{/\sim}$ the quotient set of $\Sigma(M)$ by $\sim$, and $|\Sigma(M)_{/\sim}|$ the number of equivalence classes of schedules induced by $\sim$.

The following constraints or combinations of constraints are particularly useful for defining application semantics. Let $M = (K, +, -, \rightarrow, \lhd, \nparallel)$ be a multilog and α, β two actions of K. A cycle of $\rightarrow$-constraints in M (e.g., $\alpha \overset{\leftarrow}{\rightarrow} \beta = \alpha \rightarrow \beta \wedge \beta \rightarrow \alpha$) represents *antagonism*, i.e for any schedule S of $\Sigma(M)$, either α is in S, or β is in S, or neither of them; the conjunction $\beta \overset{\lhd}{\rightarrow} \alpha = \beta \rightarrow \alpha \wedge \beta \lhd \alpha$ means α *depends causally* on β; and an $\lhd$-cycle such as $\alpha \overset{\lhd}{\rhd} \beta = \alpha \lhd \beta \wedge \beta \lhd \alpha$ expresses an atomic grouping. Finally $\alpha \nparallel \beta$ means that α and β do not commute: if α and β are ACID transactions, it models isolation (the I of ACID). Writing a specification of a concurrent application in ACF is bottom-top. It requires to produce the constraints between operations from the invariants over the data items. Section 4 illustrates how to proceed to design a shared text editor.

2.1.2 Particular subsets of multilogs and concept of soundness

The following subsets of K are of particular interest for the study of consistency.

- *Committed* actions appear in every schedule of M. This set is the subset of K that satisfies:

$$Committed(M) = \{\alpha \,|\, \alpha^{+} \vee \exists \beta \in Committed(M), \alpha \lhd \beta\}$$

- *Aborted* actions never appear in a schedule of M. $Aborted(M)$ is the subset of K that satisfies:

$$Aborted(M) = \left\{ \alpha \,\middle|\, \begin{array}{l} \exists \beta_1, \ldots, \beta_{m \geq 0} \in Committed(M), \alpha \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha \\ \vee\ \exists \beta \in Aborted(M), \beta \lhd \alpha \\ \vee\ \alpha^{-} \end{array} \right\}$$

- *Serialized* actions are either aborted, or ordered with respect to all non-commuting constraints against non-aborted actions:

$$Serialized(M) \overset{\triangle}{=} \left\{ \alpha \,\middle|\, \forall \beta \in K, \alpha \nparallel \beta \Rightarrow \begin{pmatrix} \alpha \rightarrow \beta \vee \beta \rightarrow \alpha \\ \vee \alpha \in Aborted(M) \\ \vee \beta \in Aborted(M) \end{pmatrix} \right\}$$

- *Decided* actions are either aborted, or both committed and serialized:

$$Decided(M) \overset{\triangle}{=} Aborted(M) \cup (Committed(M) \cap Serialized(M))$$

A multilog M is *sound* iff $Committed(M) \cap Aborted(M) = \varnothing$. Observe that $\Sigma(M) \neq \varnothing$ implies M sound. A multilog M is *decided* iff $Decided(M) = K$. or equivalently iff M is sound and $|\Sigma(M)_{/\sim}| = 1$.

2.2 Formalizing consistency in replicated systems

We consider an asynchronous distributed system of n sites, connected through fair-lossy links [2]. The failure model is fail-stop. A global clock $t \in \mathcal{T}$ ticks at every step of any site, but site do not access to it.

We assume that some shared data items: $Items$, are partially-replicated across sites. Given a data item x, we note $replicas(x)$ the set of sites that replicates x. Initially, at $t = 0$, x is in the same state at every site in $replicas(x)$. Given an action α, we note $item(\alpha)$ the data item on which α applies. The granularity of a data item is unspecified, it can be a single bits, a memory page, or even a whole relational database.

A site contains two processes: an application process called the *client*, and a single *consistency agent* (or just *agent* hereafter). Clients receive and execute user actions accessing $Items$. Agents ensure the consistency of the system by executing a certain protocol.

2.2.1 Site-multilogs and site-schedules

Each site i contains a *site-multilog* $M_i(t) = (K_i(t), +_i(t), -_i(t), \rightarrow_i (t)), \lhd_i (t), \nparallel_i (t)$ and a *site-schedule* $S_i(t)$. At any point in time t, these two abstractions define entirely the state of site i.

- $M_i(t)$ is the local knowledge that i has at time t of the set of actions and of the semantics linking them.

  Initially, every site-multilog equals $(\varnothing,\varnothing,\varnothing,\varnothing,\varnothing,\varnothing)$.

  Site-multilogs grow monotonically over time, as clients and agents add new actions and constraints. The following rule captures this monotonic growth:

$$\forall i, \forall t \in \mathcal{T}, \exists M, M_i(t+1) = M_i(t) \cup M$$

- $S_i(t) \in \Sigma(M_i(t))$ represents the state of the subset of $Items$ that i replicates at time t. The choice of $S_i(t)$ is arbitrary when $\Sigma(M_i(t))_{/\sim}$ contains multiple equivalence classes. If $S_i(t-1)$ is not a prefix of $S_i(t)$, it represents a roll-back.

Agents and clients both access the site-schedule and the site-multilog. We suppose our clock fine-grain enough, that between t and $t+1$, at a site only the client, or the agent, access the site-multilog and the site-schedule.

2.2.2 Eventual consistency

A distributed system is a tuple $\mathcal{S}_n = ((M_1, S_1), \ldots, (M_n, S_n))$ of n sites. A run r of $\mathcal{S}_n$ is an array of n rows. Each row i represents the evolution over time of (M_i, S_i) starting at $t = 0$. Considering a run r, we say that a site i is *correct* in r iff $r[i]$ is infinite (noted $i \in correct(r)$); otherwise we say that i is *crashed* in r.

Definition 1 (Eventual Consistency). *A system $\mathcal{S}_n$ is* eventually consistent *in a run r iff it satisfies the following correctness conditions:*

- *Eventual Decision:*

$$\forall i \in correct(r), \forall t \in \mathcal{T}, \forall \alpha \in K_i(t), \exists t' \in \mathcal{T}, \alpha \in Decided(M_i(t'))$$

- *Mergeability:*

$$\Sigma\Big(\bigcup_{\substack{i \in [\![1,n]\!] \\ t \in \mathcal{T}}} M_i(t)\Big) \neq \varnothing$$

- *Eventual Propagation of Actions*

$$\begin{aligned}&\forall t \in \mathcal{T}, \forall x \in Items, \forall \alpha \in \mathbb{A}, item(\alpha) = x,\\ &\forall i, j \in replicas(x) \cap correct(r),\\ &\alpha \in M_i(t) \Rightarrow \exists t' \geq t, \alpha \in M_j(t')\end{aligned}$$

- *Eventual Propagation of Constraints*

$$\begin{aligned}&\forall t \in \mathcal{T}, \forall \alpha, \beta \in \mathbb{A},\\ &\forall i, j \in \mathcal{S}_n, \alpha, \beta \in M_i(t) \cap M_j(t),\\ &(\alpha, \beta) \in M_i(t) \Rightarrow \exists t' \geq t, \alpha \in M_j(t')\end{aligned}$$

Roughly speaking, Eventual Decision and eventual Propagation of Actions and Constraints ensure that the system makes progress. Mergeability ensures that the system is globally sound, i.e., no decision ever puts it in an error state.

An algorithm that ensures eventual consistency, i.e. reconciles divergent replicas. is a concurrency control algorithm or a commitment protocol.

2.3 Contribution

The model we described previously brings some minor enhancements to the original work on ACF [9]. Notably, we introduced crashes in the definition of eventual consistency.

During Grid4All, our work on ACF mainly focused on understanding and designing commitment protocols. In particular we modeled the DataBase State Machine approach [7] and Holiday et al. [16] in ACF. These protocols are the state of the art in database replication. Thanks to this work, we were able to propose two new concurrency control algorithms for database replication:

- An improvement over DBSM [16]. Our protocol reduces the number of aborted transactions while preserving the advantage of DBSM: fault-resilience and read one / write all replicas.
- A decentralised algorithm [18] to committing transactions in a partially-replicated database systems. Previous protocols either reexecute transactions entirely and/or compute a total order of transactions. In contrast, this algorithm applies update values, and generate a partial order between mutually conflicting transactions only. Transactions execute faster, and distributed databases commit in small committees. Both effects contribute to preserve scalability as the number of databases and transactions increase. This algorithm is live and safe in spite of faults.

We have also designed in ACF a voting algorithm to reconcile distributed collaborative systems [13]. This algorithm works as follows. Each site makes a proposal, reflecting its tentative and/or preferred schedules. Our protocol distributes the proposals, which it decomposes into semantically-meaningful units called candidates, and runs an election between comparable candidates. A candidate wins when it receives a majority or a plurality. The protocol is fully asynchronous: each site executes its tentative schedule independently, and determines locally when a candidate has won an election. The committed schedule is as close as possible to the preferences expressed by clients. We further detail this protocol in Section 4.

3 Implementation of ACF in Telex

This section describes our in-memory implementation of ACF in Telex. This work was done during Grid4All. For further details about Telex, the reader is referred to deliverable D3.4.

3.1 Implementation of the Multilog concept

Our implementation of ACF in Telex use JGraphT [1], an open-source library for graph manipulation in Java.

We represent a multilog that is not a site-multilog with the Fragment class. A Fragment is a set of constraints, and a set of actions. We use this representation when

- Sites exchange actions and constraints.
- A client or an agent adds new actions and constraints to the the site-multilog.

A site-multilog is an instance of the GraphComponent class. This class is the core of our implementation of the multilog abstraction The GraphComponent class defines methods to compute (i) committed, aborted and serialized actions in M, (ii) a set of constraints D such that $M \cup D$ is a decided multilog and (iii) the sound schedules of M. Computing (ii) and (iii) both use the procedure $decide()$ that we detail now.

3.2 Deciding a multilog

$Decide()$ takes as input a multilog $M = (K, +, -, \rightarrow, \lhd, \nparallel)$, and outputs a multilog $M' = (K', +', -', \rightarrow', \lhd', \nparallel')$ such that :

1. $Decide()$ adds only *decisions*, i.e.:
   - $Decide()$ does not add actions: $K' = K$.
   - $+' \supseteq +$
   - $-' \supseteq -$
   - $\alpha \rightarrow' \beta \Rightarrow \alpha \rightarrow \beta \vee \alpha \nparallel \beta$
   - $\nparallel' = \nparallel$
2. Multilog M' is decided.
3. If M is sound, then M' is sound.

Our implementation follows the general guidelines proposed by Shapiro and Krishna [10]. We decompose decision into three parts: serialization, conflict-breaking and validation: serialization orders any non-commuting pairs of actions, conflict-breaking aborts at least one transaction in every $\rightarrow$-cycle, and validation commits the remaining set of non-aborted actions: Algorithm 1.

We serialize the non-commuting actions computing the relation $\rightarrow'$ (lines 2-4). $\rightarrow'$ is a linear extension of the order $\rightarrow$ over the set of non-serialized actions SER.

Breaking $\rightarrow$-cycles minimizing the number of actions aborted is stated as follows:

Definition 2. *Consider a weighted graph $G = (V, E)$ where:*

Algorithm 1 $Decide(M)$

```
1: {Serialization}
2: let $SER := K \setminus Serialized(M)$
3: choose $\rightarrow'$ such that:
     $\rightarrow \subseteq \rightarrow'$
     $\wedge\ \forall \alpha, \beta \in SER,\ ((\alpha, \beta) \in \rightarrow') \vee ((\alpha, \beta) \in \rightarrow')$
4: $\rightarrow := \rightarrow'$
5: {Cycle breaking}
6: $M := breakCycles(M)$
7: {Validation}
8: for all $T \notin Decided(M)$ do
9:    $M := M \cup \{\alpha^{+}\}$
10: end for
```

- *Each node in V is an action α in $K \setminus Committed(M)$ weighted by k, where k equals one plus the number of distinct predecessors by $\lhd$ that α has in M (i.e. the actions that α enables).*
- *For $(v, v') \in V$, a directed edge going from v to v' exists in E, iff $v \rightarrow v'$ is in M.*

Conflict breaking is the problem of finding the minimum feedback vertex set of G.

This problem is an NP-complete optimization problem, and the literature upon this subject is important [4]. In Telex we implement 3 solutions:

- The IceCube heuristic [6], that requires quadratic time. This heuristic is incremental.
- An heuristic that stores the predecessors of each actions, killing an action when it is the predecessor of itself. This heuristic is incremental, and requires linear time.
- An exhaustive solver that requires exponential time, and is not incremental. We use this solver for collaborative distributed applications where there is few actions (typically the shared calendar application).

At the end of the serialization process and the conflict breaking, remaining non-aborted actions are committed: lines 8 and 9.

3.3 Caching features and split mode

The GraphComponentCache class caches the set of aborted actions, the set of committed actions, and a reduced NotAfter graph.

The reduced NotAfter graph accelerates $decide()$. It contains all actions in $K \setminus (Aborted(M) \cup Committed(M))$. An edge (α, β) in this graph means that either $\alpha \rightarrow \beta$, or it exists a set of committed actions $\{\gamma_1, \ldots, \gamma_m\}$ such that $\alpha \rightarrow \gamma_1, \ldots \gamma_m \rightarrow \beta$.

We refresh the cache incrementally each time we add a new action/constraint.

Telex also implements a split mode for multilogs. Telex splits a multilog into independent subgraphs, or *connected components*, and compute decide over each component.

Figure 1 shows the time to execute $decide()$ according to different modes. In this synthetic benchmark, we repeatedly add a fragment of 100 actions, up to 45,000, to a site-multilog, and compute a single proposal. The top-most curve, labeled “nocache,” uses our standard implementation.

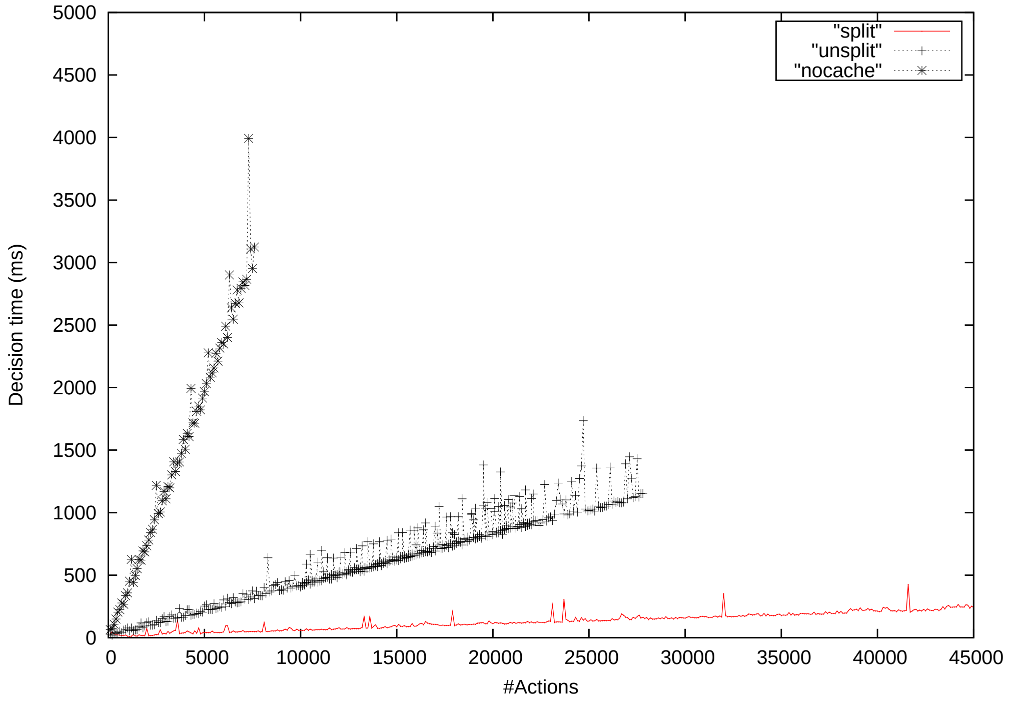


Figure 1: Incremental schedule computation efficiency

The middle curve, "unsplit," shows the effect of the cache. The lowest curve, "split," additionally breaks the multilog into connected components. In all cases, the computation time increases linearly with the number of actions; the cache and connected components improve performance considerably.

4 Using ACF

The work we describe along this section, was published during the Grid4All project [3, 5].

4.1 General considerations

ACF is independent of a particular application. Each application defines its own action types, and parametrises the system with constraints between them.

To discuss how to design an application using ACF, 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?

4.1.1 Actions and sequential constraints

Let us first consider sequential execution, ignoring concurrency and conflicts for now. An application puts actions, as well as causality and atomicity constraints, in its local site-multilog. Dependence translates to a causal constraint; for instance, if action α reads some value used by β, the application should add a constraint $\alpha \overset{\lhd}{\rightarrow} \beta$. Similarly, to ensure action composition (either both α and β execute or neither does), the application adds a fragment containing α, β and $\alpha \overset{\lhd}{\underset{\rhd}{}} \beta$.

4.1.2 Concurrency and constraints

Operations commute if either relative execution order yields the same final state.

Two types of conflict exist in ACF: the antagonism ($\alpha \overset{\leftarrow}{\underset{\rightarrow}{}} \beta$), and the non-commutativity ($\alpha \nparallel \beta$
Actions are antagonistic when they cannot both execute together; this should be obvious based on action types and their arguments.

In all other cases, the actions are non-commuting $\alpha \nparallel \beta$. As an ordering between α and β might violate an application invariant, the application must check its invariants at run time.

4.1.3 Constraints and application design

Well-chosen constraints are critical to performance and user experience. Commutativity is the most favorable case.

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.

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.[1]

We now further illustrate how to design using ACF with a concrete example: a collaborative text editor.

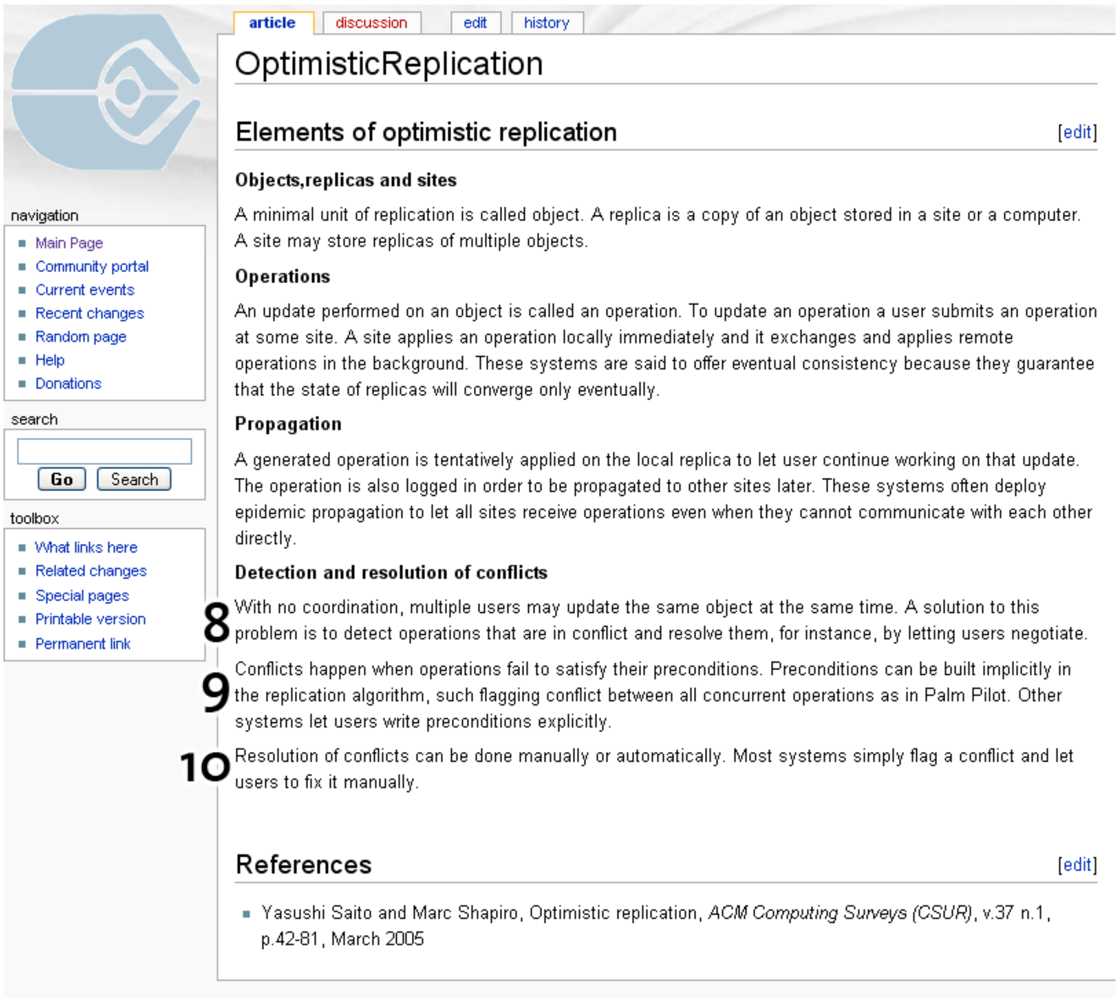
4.2 Designing a collaborative text editor in ACF

Consider three users collaboratively writing a wiki page about optimistic replication. Suppose the three users concurrently edit the section *"Detection and resolution of conflicts"*, of the wiki page whose initial state is illustrated in Figure 2.

Further suppose the three users perform the operations : User 1 inserts a new line as the 10th line, User 2 updates the 9th line, User 3 deletes the lines 8 to 10. The modifications of each user are shown in the same figure, Figure 3.

Afterwards, all three users try to publish their versions of the document by exchanging their modifications. Eventually all replicas should converge.

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



OptimisticReplication

Elements of optimistic replication [edit]

Objects,replicas and sites

A minimal unit of replication is called object. A replica is a copy of an object stored in a site or a computer. A site may store replicas of multiple objects.

Operations

An update performed on an object is called an operation. To update an operation a user submits an operation at some site. A site applies an operation locally immediately and it exchanges and applies remote operations in the background. These systems are said to offer eventual consistency because they guarantee that the state of replicas will converge only eventually.

Propagation

A generated operation is tentatively applied on the local replica to let user continue working on that update. The operation is also logged in order to be propagated to other sites later. These systems often deploy epidemic propagation to let all sites receive operations even when they cannot communicate with each other directly.

Detection and resolution of conflicts

8 With no coordination, multiple users may update the same object at the same time. A solution to this problem is to detect operations that are in conflict and resolve them, for instance, by letting users negotiate.

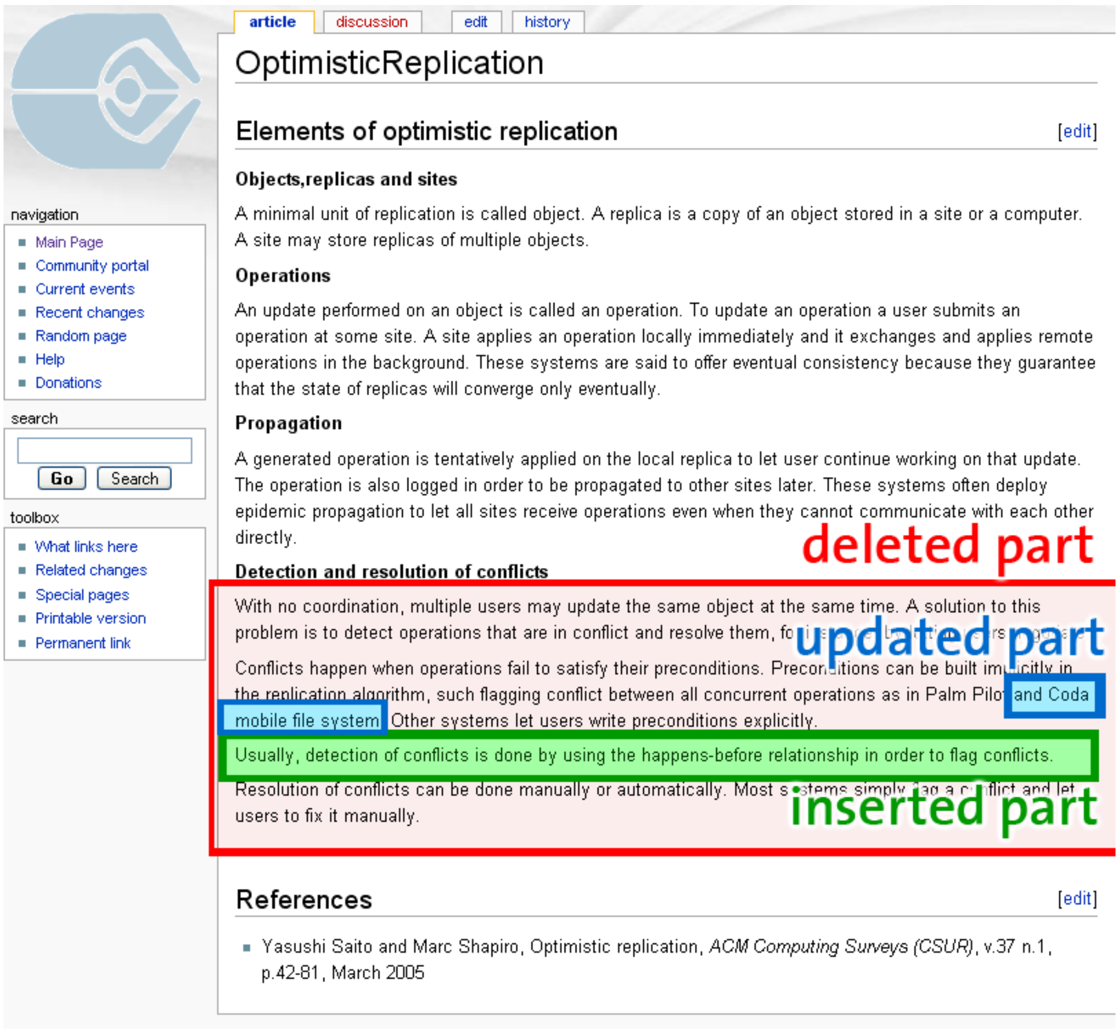
9 Conflicts happen when operations fail to satisfy their preconditions. Preconditions can be built implicitly in the replication algorithm, such flagging conflict between all concurrent operations as in Palm Pilot. Other systems let users write preconditions explicitly.

10 Resolution of conflicts can be done manually or automatically. Most systems simply flag a conflict and let users to fix it manually.

References [edit]

- Yasushi Saito and Marc Shapiro, Optimistic replication, *ACM Computing Surveys (CSUR)*, v.37 n.1, p.42-81, March 2005

Figure 2: Initial state of the section *“Detection and resolution of conflicts”*



OptimisticReplication

Elements of optimistic replication [edit]

Objects,replicas and sites

A minimal unit of replication is called object. A replica is a copy of an object stored in a site or a computer. A site may store replicas of multiple objects.

Operations

An update performed on an object is called an operation. To update an operation a user submits an operation at some site. A site applies an operation locally immediately and it exchanges and applies remote operations in the background. These systems are said to offer eventual consistency because they guarantee that the state of replicas will converge only eventually.

Propagation

A generated operation is tentatively applied on the local replica to let user continue working on that update. The operation is also logged in order to be propagated to other sites later. These systems often deploy epidemic propagation to let all sites receive operations even when they cannot communicate with each other directly.

Detection and resolution of conflicts

With no coordination, multiple users may update the same object at the same time. A solution to this problem is to detect operations that are in conflict and resolve them, fo [illegible]

Conflicts happen when operations fail to satisfy their preconditions. Preconditions can be built implicitly in the replication algorithm, such flagging conflict between all concurrent operations as in Palm Pilot and Coda mobile file system. Other systems let users write preconditions explicitly.

Usually, detection of conflicts is done by using the happens-before relationship in order to flag conflicts.

Resolution of conflicts can be done manually or automatically. Most systems simply flag a conflict and let users to fix it manually.

References [edit]

- Yasushi Saito and Marc Shapiro, Optimistic replication, *ACM Computing Surveys (CSUR)*, v.37 n.1, p.42-81, March 2005

Figure 3: Overview of changes performed by users.

To ensure this property, a simple technique is the "Thomas Write Rule" approach [19], also called "Last-Writer Wins" in replicated file systems. The successive versions of a file are timestamped or numbered; the version with the highest number is retained, and other versions are thrown away. The drawback of course is that concurrent updates are lost.

Instead, the literature on computer-supported co-operative work says that "user intention" should be preserved. Unfortunately, it is difficult to characterise user intention formally. The action-constraint framework maintains an explicit representation of semantic relations between operations, e.g., conflicts, to express users' intention and try to combine them optimally.

We model a collaborative text editing as a totally ordered set of lines: $D = (L, <_D)$, and each line in L is uniquely identified. Users update D using the following actions:

- $create(c,k,l)$: create a line with content c between lines k and l; return its unique identifier.
- $delete(l)$: hide line l.
- $update(l,c)$: replace the contents of line l with c.

We identify the state of a document with a *schedule*. Consider that $INIT$ is the schedule that leads to the state depicted in Figure 2, before users start editing the document. As a simplification, we identify line creation actions by the identifier of the created line. Thus, we note $l_8 = create($"With no ...", $-,-)$ the action that created line 8; similarly $l_9 = create($"Conflict happen ...", $-,-)$ for line 9, and $l_{10} = create($"Resolution ...", $-,-)$ for line 10. Since User 3 wants to atomically delete lines 8 to 10, his change is modeled as:

- A set of three actions: $d_1 = delete(l_8)$, $d_2 = delete(l_9)$, $d_3 = delete(l_{10})$.
- An $\lhd$-cycle: $d_1 \lhd d_2 \lhd d_3 \lhd d_1$. Therefore all three delete operations execute, or none of them does.

Action $u_1 = update($"Conflict happens ... and Coda ...",) is User 2's update. Notice that actions u_1 and d_1 conflict. Depending on the desired effect, the application may declare them, either to be non-commuting: $u_1 \nparallel d_1$, or antagonistic: $u_1 \overset{\leftarrow}{\rightarrow} d_1$ (shortcut for $u_1 \rightarrow d_1 \land d_1 \rightarrow u_1$). In what follows, we declare them to be antagonistic.

The system has the obligation to eventually resolve conflicts: the Eventual Consisntency property (see Section 2.2.2). In the case of non-commuting actions, it must either order them (by adding a NotAfter constraint), or abort one or the other or both. In the case of an antagonism, it can only abort. A commitment protocol ensures that sites agree on a final common state.

Designing an application in ACF is invariant-driven. Let us consider the following (informal) set of invariants:

1. Any two collaborators eventually observe any two visible lines in the same order.
2. If line l is created, eventually all collaborators see l, or none of them.
3. Given a line l, l has eventually the same content for all collaborators.

Tables 4 and 5 specify constraints that satisfy these invariants. ($k \leq_D k' \triangleq k <_D k' \lor k = k'$, $o' \prec o$ means that o happens-before o', $o' \parallel o$ means that o and o' are concurrent: $o' \nprec o \land o \nprec o'$, and $l = o'$ means that the identifier of l (a line) and that of o' (an action of creation) are equal.)

o \ o'	$create(c',k',l')$	$delete(l')$	$update(c',l')$
$create(c,k,l)$	$o' \to o$ $\left.\begin{array}{l} k = o' \\ \vee\, l = o' \end{array}\right\} \Rightarrow o' \lhd o$	$\varnothing$	$\varnothing$
$delete(l)$	$l = o' \Rightarrow o' \overset{\lhd}{\to} o$	$\varnothing$	$\varnothing$
$update(c,l)$	$l = o' \Rightarrow o' \overset{\lhd}{\to} o$	$\varnothing$	$(l = o')$ $\Rightarrow o' \to o$

Figure 4: Collaborative editing constraints for $o' \prec o$

o \ o'	$create(c',k',l')$	$delete(l')$	$update(c',l')$
$create(c,k,l)$	$\left.\begin{array}{l} k \leq_D k' <_D l \\ \vee\, k' <_D k <_D l' \end{array}\right\} \Rightarrow o \dashv\vdash o'$	$\varnothing$	$\varnothing$
$delete(l)$	$\varnothing$	$\varnothing$	$l = l'$ $\Rightarrow l \overset{\leftarrow}{\to} l'$
$update(c,l)$	$\varnothing$	$l = l'$ $\Rightarrow l \overset{\leftarrow}{\to} l'$	$(l = l')$ $\Rightarrow o' \dashv\vdash o$

Figure 5: Collaborative editing constraints for $o' \parallel o$

4.3 Agreeing when collaborating

The system sends the contents of site-multilogs, using background epidemic communication (to achieve Eventual Propagation of Actions and Constraints). Eventually, users become aware of the actions of their collaborators, and the possible conflicts. Thus every user eventually receives the insertion: $l_{11} = create($“Usually, the detection... ”$, l_9, l_{10})$ from User 1, the update: u_1 from User 2, and the actions d_1, d_2, d_3 with the parcel constraint from User 3. According to Table 5, every site eventually includes the following information in its multilog: the set of actions $\{l_{11}, u_1, d_1, d_2, d_3\}$, the parcel constraint: $d_1 \lhd d_2 \lhd d_3 \lhd d_1$, and the conflict: $u_1 \overset{\leftarrow}{\to} d_2$.

However, until operations commit, different users may view different states of the document. For instance if User 1 ignores User 3, his current view could be the following sound schedule: $S_1 = INIT.l_{11}.u_1$. Similarly User 3 might observe his own actions only: $S_3 = INIT.d_1.d_2.d_3$.

To attain Eventual Consistency we propose a voting protocol, whereby each site makes a proposal reflecting its tentative state and/or the user's preference [14].

Back to the example. Actions u_1 and d_1 are antagonistic (linked by a NotAfter cycle), and actions d_1, d_2 and d_3 are atomic (linked by an Enables cycle). Say User 1 and User 2 both vote to commit l_{11} and u_1; consistency obligates them to vote to abort d_1, d_2 and d_3. Note this proposal P, and note Q the proposal of User 3 to commit his own actions and to abort both l_{11} and u_1. Our protocol distributes proposals epidemically. Eventually all sites are aware of P and Q.

Our algorithm decomposes a proposal into semantically-meaningful units, called candidates. An election runs locally between competing candidates. A candidate C receives a number of votes, equal to the sum of the weights of the sites that voted for some proposal containing C. (If a single site has a weight of 100%, our algorithm degenerates to a primary approach.) Suppose that weights are uniformly distributed among three sites. Then candidate $C =$ "commit u_1" extracted from P has a weight of $\frac{2}{3}$, and candidate $C' =$ "abort u_1" extracted from Q has a weight of $\frac{1}{3}$.

A candidate cannot be just any subset of a proposal. It must also be consistent with existing constraints. For instance "abort d_1" is not a well-formed candidate, because of the atomicity constraint, but "abort d_1 and d_2 and d_3" is well-formed. During an election a candidate competes against comparable candidates. Two candidates are comparable if they contain the same set of actions. For instance candidates C and C' are comparable. A candidate wins when it receives a majority or a plurality. In our example, eventually every site aborts d_1, d_2 and d_3, and commits l_{11} and u_1.

4.4 Evaluation

Communication Complexity We consider that m sites execute an action concurrently. Sites exchange their proposals and their multilogs epidemically. In the best case, the communication cost is $4(m-1)$:

- Every site sends its actions to site i: $m-1$ messages.
- Site i computes the constraints, and sends its multilog M to all other sites: $m-1$ messages.
- When a site receives M, it computes a proposal and returns the result to i: $m-1$ messages.
- Site i receives all proposals, and sends them to other sites: $m-1$ messages.
- Each site decides locally.

This reduces to $2(m-1)$ if a single site holds 100% weight.

Time Complexity The complexity of the election algorithm is $O(m^3n^2)$ in the worst case [12].

Space complexity In ACF a site stores all the non-durable actions, be they either local or remote. However, durable actions and their constraints are eventually garbage-collected, and replaced by snapshots. A snapshot contains the state of the document, its size is proportional to l. If we assume that GC keeps a small and constant number of snapshots at each site, the space complexity is $O(lm)$.

The size of a proposal is eventually $O(n)$. As each site keeps tracks of all proposals, the space requirement for proposals is $O(nm)$. (Proposals are also eventually garbage-collected, but we ignore this effect in this evaluation.)

First Site Convergence Latency In the best case execution (the one depicted above), the number of asynchronous rounds to converge is 3. It reduces to 1 if a primary site holds all the weight.

Convergence Latency Once a site has elected a candidate locally, a single additional round ensures other sites are informed.

Semantic Expressiveness ACF supports arbitrary operation types. The ACF logic is parametrised by application semantics; see for instance Tables 4 and 5. A different application differs only by a different set of constraints. Conflict is an important concept in collaborative applications, and ACF supports it. ACF recognises two variants of conflict, non-commutativity and antagonism. ACF also allows users to group operations atomically. This is particularly useful in our example. For instance, if a user inserted a line between lines 8 and 9, it probably would not make sense to keep the inserted line without the lines around it. In our system, this will be flagged as an antagonism, and either the insert or the delete would fail (or both).

Determinism By design, our approach is not deterministic, since the outcome depends on the collaborators' votes. However, for a given set of votes, the system is deterministic.

5 Conclusion

In this paper, we presented the Action-Constraint Framework. ACF is a framework that helps describing consistency issues, and designing new solutions.

During Grid4All, we implemented ACF in our semantic store: Telex. Telex is a platform for sharing mutable data in a decentralised way over a large-scale network. Telex supports an optimistic application model based on ACF, and takes over system issues such as replication, persistence, disconnected operation, and reconciliation.

Reconciliation was our main focus during Grid4All. Using ACF, we designed new concurrency control protocols [16, 13, 18]. Our algorithms brings elegant solutions to the known-difficult problems of database replication and distributed collaboration.

We have also explored distributed collaborative design. We proposed a design methodology for building applications with ACF, and compared it to existing solutions in the field of shared text edition [3]. We used this methodology to construct applications on top of Telex [5].

References

[1] JGraphT: an open-source graph manipulation library in java.

[2] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609, Cornell University, Computer Science Department, 1996.

[3] Lamia Benmouffok, Jean-Michel Busca, Joan Manuel Marquès, Marc Shapiro, Pierre Sutra, and Georgios Tsoukalas. Telex: Principled system support for write-sharing in collaborative applications. Technical Report 6546, INRIA, May 2008.

[4] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[5] Claudia-Lavinia Ignat, Gérald Oster, Pascal Molli, Michèle Cart, Jean Ferrié, Anne-Marie Kermarrec, Pierre Sutra, Marc Shapiro, Lamia Benmouffok, Jean-Michel Busca, and Rachid

Guerraoui. A comparison of optimistic approaches to collaborative editing of Wiki pages. Number 3, White Plains, NY, USA, November 2007.

[6] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Symp. on Principles of Dist. Comp. (PODC)*, Newport RI, USA, August 2001. ACM SIGACT-SIGOPS. `http://research.microsoft.com/research/camdis/Publis/podc2001.pdf`.

[7] F Pedone, R Guerraoui, and A Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[8] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[9] Marc Shapiro, Karthikeyan Bhargavan, and Nishith Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. 8th Int. Conf. on Principles of Dist. Sys. (OPODIS)*, number 3544 in Springer-Verlag, pages 331–345, Grenoble, France, December 2004. `http://www-sor.inria.fr/~shapiro/papers/opodis2004-final-2004-10-30.pdf`.

[10] Marc Shapiro and Nishith Krishna. The three dimensions of data consistency. In *Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR)*, pages 54–58, CNAM, Paris, France, November 2005. `http://www-sor.inria.fr/~shapiro/papers/cdur2005.pdf`.

[11] Pierre Sutra, João Barreto, and Marc Shapiro. An Asynchronous, Decentralised Commitment Protocol for Semantic Optimistic Replication. Research Report 6069, INRIA, December 2006.

[12] Pierre Sutra, João Barreto, and Marc Shapiro. An asynchronous, decentralised commitment protocol for semantic optimistic replication. Technical report, December 2006.

[13] 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. `http://www-sor.inria.fr/~shapiro/papers/sutra-barreto-shapiro-coopis40.pdf`.

[14] Pierre Sutra, João Barreto, and Marc Shapiro. Decentralised Commitment for Optimistic Semantic Replication. In *Proc. of the International Conference on Cooperative Information Systems - CoopIS 2007*, Vilamoura, Algarve, Portugal, November 2007.

[15] Pierre Sutra and Marc Shapiro. Comparing Optimistic Database Replication Techniques. In *Bases de Données Avancées*, Marseille, France, October 2007.

[16] Pierre Sutra and Marc Shapiro. Comparing optimistic database replication techniques. In *Bases de Données Avancées*, Marseille, France, October 2007.

[17] Pierre Sutra and Marc Shapiro. Fault-tolerant partial replication in large-scale database systems. pages 404–413, Las Palmas de Gran Canaria, Spain, August 2008.

[18] Pierre Sutra and Marc Shapiro. Fault-tolerant partial replication in large-scale database systems, 2008.

[19] Robert H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.