An asynchronous, decentralised commitment protocol for semantic optimistic replication

Pierre Sutra, Marc Shapiro, Joao Barreto

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An asynchronous, decentralised commitment protocol for semantic optimistic replication

Pierre Sutra Marc Shapiro
Université Paris VI and INRIA Rocquencourt, France
— João Barreto
INESC-ID and Instituto Superior Técnico, Lisbon, Portugal

N° ????

Decembre 2006

Thème COM

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

An asynchronous, decentralised commitment protocol for semantic optimistic replication

Pierre Sutra* Marc Shapiro
Université Paris VI and INRIA Rocquencourt, France
, João Barreto
INESC-ID and Instituto Superior Técnico, Lisbon, Portugal

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° ???? — Decembre 2006 — 21 pages

Abstract: We study eventual consistency in an asynchronous system with optimistic data replication. A site executes actions submitted by the local client, and remote actions as they are received. This state is only tentative, because semantic constraints such as conflicts, dependence, or atomicity may cause it to roll back some of its state and compute a new state. The system should be eventually consistent, i.e., (i) each local schedule be correct and stabilise eventually, and (ii) the schedules at each site eventually converge. We propose a decentralised, asynchronous commitment protocol that ensures this. Each site proposes a set of schedules to all other sites. A proposal can be decomposed into one or more semantically-meaningful units, called candidates. A candidate wins when it receives a majority or a plurality of the votes in its election, leaving room for missing votes. The protocol is fully asynchronous: each site executes its tentative schedule independently, and determines locally when a candidate has won an election. The protocol is safe in the presence of non-byzantine faults. It supports a rich repertoire of semantic relations, viz., it resolves conflicts, it guarantees sound executions with respect to dependence or atomicity, and it orders non-commuting pairs of actions, but not necessarily commuting ones. We describe the protocol in detail and prove it safe.

Key-words: data replication, optimistic replication, semantic replication, commitment, voting protocols.

* LIP6, 104, ave. du Président Kennedy, 75016 Paris, France; mailto:pierre.sutra@lip6.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Un protocole de validation pour la réplication optimiste dans les systèmes répartis sémantiquement riches

Résumé : Nous examinons à travers ce document la cohérence dans les systèmes répartis répliquant des données de manière optimiste. Le paradigme de la réplication optimiste est que les sites composant le système réparti peuvent ré-éxecuter les requêtes des clients (actions) si la sémantique liant les actions le nécessite. Dans de tels systèmes le critère de cohérence est que les sites convergent à terme vers des exécutions équivalentes. Afin d'assurer cette convergence, un protocole de validation est nécessaire. C'est l'objet de cette étude. Notre protocole procède par éléctions successives sur des ensembles d'actions exécutées de manière optimiste par le système. La sémantique prise en compte dans ce protocole est suffisament riche pour exprimer des notions telles que la non-commutativité, le conflit ou encore la causalité entre les actions. Nous prouvons que notre protocole est sûr, et ce en dépit des éventuelles pannes franches pouvant survenir sur les sites.

Mots-clés : réplication optimiste, validation, protocoles de vote

1 Introduction

Access to shared data is a performance and availability bottleneck. This problem will only get worse as more mutable data is shared remotely, and as the gap between processing speeds and memory/ network latency continues to widen. One possible solution is to use optimistic replication (OR), where a process may read or update its local replica without synchronising with remote sites [13]. OR decouples data access from network access.

In OR, each site makes progress independently, even while others are slow or unavailable. Sites exchange updates lazily and asynchronously. OR insulates users from network disruption, and may improve network utilisation by batching. OR supports mobile computers with slow, expensive or intermittent network connections, and wide-area networks with high and variable latencies. OR is especially useful for loosely-coupled co-operative work, where each user works on a separate copy, and synchronises only occasionally with co-workers.

We model an OR system as a network of disjoint sites. A client submits actions for execution to the local site; the site occasionally exchanges actions with other sites and replays remote actions locally. A conflict occurs when a remote client submits actions that would violate application semantics when replayed against the local state. When this happens, one or the other site (or both) must roll back its state to an earlier one, and execute actions according to a different schedule. However, the system should ensure *eventual consistency*, i.e., schedules should eventually stabilise, and stable schedules at all sites should agree. Agreeing on a stable schedule is what we call *commitment*.

In order to resolve conflicts, and more generally to adapt to application requirements, the system should be aware of application semantics. To this effect, we parameterise system behaviour by *constraints*. A constraint reifies an invariant that is directly related to the scheduling of actions, for example dependence (one action may execute only if another has), atomic grouping (all actions or none in the group execute), non-commutativity (all stable schedules should execute them in the same order), or antagonism (if one action executes, some others may not). The set of constraint types is small, and cannot be claimed to support all possible semantics, but it is formalised [15], and experience shows that it is sufficient for a large class of applications [12].

The design trade-offs for commitment algorithms are different in OR systems and in classical ones. Generally speaking, previous commitment algorithms are inefficient when semantics are considered. For instance many compute a total order, even though only non-commuting pairs of actions need to be serialised. In the presence of conflicts, they often abort more actions than necessary. Classical systems commit one action at a time.

In contrast, a semantic OR commitment algorithm may batch its decisions, which allows it to look ahead at conflicts and dependencies, in order to minimise aborts [12].[1] Because commitment only impacts the stable state, it can occur in the background, messages can be batched, and minimising latency is less important.

[1] For instance, suppose commitment has to choose between aborting actions α and β. If no actions depend on α but a large number depend on β, it is better to abort α.

Unfortunately, previous semantic OR systems [12, 17] generally delegate commitment to a single primary site. We propose instead to decentralise commitment, in order to avoid performance and fault-tolerance bottlenecks.

We show that commitment should not consider a single action at a time, but instead should examine semantically-significant units. For instance, if two actions conflict, how one is scheduled impacts the other and vice-versa; therefore, the unit of commitment must encompass both actions.

The main contribution of this paper is a decentralised and asynchronous commitment protocol for semantic OR systems. It builds upon existing, primary-based semantic algorithms. Several instances of such algorithms exchange proposals and vote on each other's proposal. Each proposal decomposes into semantically-significant granules, called candidates. A candidate wins its election when it receives more votes than any opponent, leaving room for votes not yet received. It may win either by majority or by a simple plurality. Sites communicate by asynchronous messages. As soon as a site has received a sufficient number of proposals and votes, it is capable of determining locally which candidate wins its election. This protocol ensures that tentative schedules at each site eventually stabilise. We prove that the protocol is safe, i.e., local stable schedules are mutually equivalent, even in the presence of non-byzantine faults. The protocol is live as long as a sufficient number of votes are received.

The outline of the paper is the following. Section 2 introduces our system model and our vocabulary. Section 3 links between classical approaches and ours. We give our commitment protocol in Section 4. Section 5 provides a proof outline and discusses message cost. We compare with related work in Section 6. In conclusion, Section 7 discusses our results and future work.

2 System model and Terminology

We consider an asynchronous distributed system of *n sites* $i, j, \ldots \in \mathcal{I}$. Sites are reliable. They communicate through fair-lossy channels. We assume a global clock $t \in \mathcal{T}$ that ticks at every step of any process, but processes do not have access to it.

A site executes an application thread called the *client*, a *proposer* process that makes proposals, and an *acceptor* thread where agreement takes place. Sites communicate through asynchronous messages. Together, the set of proposers and acceptors at all sites execute the commitment protocol described herein. We formally define some site i as the tuple $(M_i(t), S_i(t), c_i, p_i, a_i)$ where c_i (resp. p_i, a_i) is the client (resp. the proposer, the acceptor) of the site. The $M_i(t)$ and $S_i(t)$ elements respectively denote the site-multilog and site-schedule described later.[2]

2.1 The Action-Constraint Framework

We use the Action-Constraint Framework (ACF) to model our system [14, 15]. The rest of this section describes our model along with a terse introduction to ACF.

[2] When there is no ambiguity, we drop the word client, proposer and acceptor, and just say site.

Shared data is replicated across all sites. We do not represent data directly; instead we identify its state at some site with a *schedule* of *actions S* (or simply a schedule), defined as a sequence of actions ordered by $<_S$, where any action appears at most once, executed at that site since the common initial state INIT.

An *action* is a request to execute some logical operation. We assume actions to be unique and distinguishable from one another. A *constraint* represents a scheduling invariant between actions. For instance, consider that user Alice has a meeting planned with Bob and needs to buy a ticket to attend it. This may involve actions "debit my bank account by 100€" and "buy ticket to Paris next Monday at 10:00." It is useful to add the semantic information that the goal of the debit (action α) is to pay for the ticket (bought by action β). In other words, β depends causally on α, noted $\alpha{\rightarrow}\beta \wedge \alpha{\lhd}\beta$.[3] Given this constraint, the following executions are sound (i.e., legal): just α, or $\alpha;\beta$. However the execution β, with α absent, is unsound.

We consider two kinds of conflicts.[4] If executing two actions α and β in different orders gives different results, we call this *non-commutativity*, noted $\alpha{\nparallel}\beta$. If no execution order could satisfy the invariants of two (or more) actions, we call this *antagonism*, noted $\alpha{\rightarrow}\beta \wedge \beta{\rightarrow}\alpha$.

To resolve an antagonism conflict, it is necessary to remove one or the other action (or both) from legal schedules; removed actions are said *dead* or killed. Killing an action also resolves a non-commutativity conflict, but a better approach is to *serialise* the actions, i.e., to ensure that stable schedules execute the two actions in the same order.

Note that in the database literature, the word conflict usually designates what we call non-commutativity, whereas in the CSCW (Computer-Supported Cooperative Work) community, conflict usually means antagonism.

2.1.1 Multilogs and constraints

Our central data structure is the *multilog*. Let A be the set of all *actions*, noted $\alpha, \beta, \ldots$. A multilog is a quadruple $M = (K, \rightarrow, \lhd, \nparallel)$, where $K \subseteq A$, and $\rightarrow$, $\lhd$ and $\nparallel$ are sets of *constraints* (relations over $A \times A$), respectively called NotAfter, Enables and NonCommuting.[5] Relation $\nparallel$ is symmetric. Relations $\rightarrow$ and $\lhd$ do not have any particular properties.

[3] Our notations will be explained shortly.

[4] Some authors suggest to remove conflicts by transforming the actions [16, 18]. We assume that, if such transformations are possible, they have already been applied.

[5] Multilog union, inclusion, difference, etc., are defined as component-wise union, inclusion difference, etc., respectively. For instance if $M = (K, \rightarrow, \lhd, \nparallel)$ $M' = (K', \rightarrow', \lhd', \nparallel')$ their union is $M \cup M' = (K \cup K', \rightarrow \cup \rightarrow', \lhd \cup \lhd', \nparallel \cup \nparallel')$.

2.1.2 Soundness and equivalence

A schedule S is *sound* with respect to multilog M if:

$$S \in \Sigma(M) \stackrel{\text{def}}{=} \forall \alpha, \beta \in A, \begin{cases} \text{INIT} \in S \\ \alpha \in S \wedge \alpha \neq \text{INIT} \Rightarrow \text{INIT} <_S \alpha \\ \alpha \in S \Rightarrow \alpha \in K \\ \alpha \rightarrow \beta \Rightarrow \neg(\beta <_S \alpha) \\ \alpha \lhd \beta \Rightarrow (\beta \in S \Rightarrow \alpha \in S) \end{cases}$$

where $\Sigma(M)$ is the set of schedules that are sound with respect to M. $\Sigma(M)$ grows as K grows, and shrinks as $\rightarrow$ or $\lhd$ grow. Multilog M is said *sound* if $\Sigma(M) \neq \varnothing$. Any subset of a sound multilog is sound; conversely, any superset of an unsound multilog is unsound.

Relations $\rightarrow$ and $\lhd$ restrict which schedules are legal. In contrast, $\nparallel$ defines an equivalence relation between schedules, where S and S' are *equivalent* iff they contain the same actions, and non-commuting actions are ordered in the same direction.

If M contains a NotAfter cycle such as $\alpha \rightarrow \beta \wedge \beta \rightarrow \alpha$, then no sound schedule may contain both α and β. Therefore, NotAfter cycles represent antagonism. The degenerate cycle $\alpha \rightarrow \alpha$ causes α to be dead. The conjunction $\alpha \rightarrow \beta \wedge \alpha \lhd \beta$ means that β cannot execute unless α has executed previously; β causally depends upon α. An Enables cycle such as $\alpha \lhd \beta \wedge \beta \lhd \alpha$ encodes *atomicity*: in any sound schedule, either both α and β are present, or neither is. (In this paper, to encode the isolation property of transactions, the whole transaction is represented as a single action.)

2.1.3 Site-multilogs and site-schedules

Each site i has a distinguished *site-multilog* $M_i(t) = (K_i(t), \rightarrow_i(t), \lhd_i(t), \nparallel_i(t))$. It contains i's local knowledge of the distributed state at time t. Initially, $M_i(0) = (\{\text{INIT}\}, \varnothing, \varnothing, \varnothing)$. It grows over time, as we explain shortly. Associated with the site-multilog, each site has a *site-schedule* $S_i(t) \in \Sigma(M_i(t))$. We identify the current state of site i with (the equivalence class of) site-schedule $S_i(t)$.

By design, the choice of site-schedule within $\Sigma(M_i(t))$ is non-deterministic, in order to account for a wide range of implementations. In particular, unless the constraints in M dictate otherwise, the site-schedule at time $t+1$ does not necessarily extend that at time t; this represents a roll-back.

2.2 Client Behaviour and client interaction

An application performs tentative operations by submitting actions and constraints to its local site-multilog, which the site-schedule will (hopefully) include.

We abstract the details of applications, by postulating that clients have access to a multilog $\mathcal{M} = (\varnothing, \rightarrow_{\mathcal{M}}, \lhd_{\mathcal{M}}, \nparallel_{\mathcal{M}})$, such that $\mathcal{M}' = \mathcal{M} \cup (A, \varnothing, \varnothing, \varnothing)$ is sound. $\mathcal{M}$ contains all application constraints. We postulate that as the client submits actions L to the site-multilog, function

Algorithm 1 *ClientActionsConstraints*(L)

Require: $L \subseteq A$

1: $K_i := K_i \cup L$
2: **for** all $\alpha \rightarrow_{\mathcal{M}} \beta$ such that $\alpha \in K_i \wedge \beta \in K_i$ **do**
3: $\quad \rightarrow_i := \rightarrow_i \cup \{(\alpha, \beta)\}$
4: **for** all $\alpha \lhd_{\mathcal{M}} \beta$ such that $\alpha \in K_i \wedge \beta \in K_i$ **do**
5: $\quad \lhd_i := \lhd_i \cup \{(\alpha, \beta)\}$
6: **for** all $\alpha \nparallel_{\mathcal{M}} \beta$ such that $\alpha \in K_i \wedge \beta \in K_i$ **do**
7: $\quad \nparallel_i := \nparallel_i \cup \{(\alpha, \beta)\}$

ClientActionsConstraints (Algorithm 1) adds constraints with respect to actions that the site already knows.[6]

To illustrate, consider the previous example of Alice's meeting with Bob. Assume that Alice and Bob run some distributed application for shared project and time management, which is supported by an OR system. Alice and Bob access site 1 and site 2 respectively. Both may read and update their local replicas. Accordingly, clients c_1 and c_2 add new actions (access to shared data) along with their constraints to M_1 and M_2 (according to Algorithm 1), respectively. Alice's actions are α, a request to debit money from her account, and β, buying a ticket to meet Bob. Semantically, β depends on α; hence, $\mathcal{M}$ contains $\alpha \rightarrow_{\mathcal{M}} \beta \wedge \alpha \lhd_{\mathcal{M}} \beta$. Alice calls *ClientActionsConstraints*$(\{\alpha\})$ to add action α to M_1, and, some time later, similarly for β. At this point, Algorithm 1 adds the constraints $\alpha \rightarrow_1 \beta$ and $\alpha \lhd_1 \beta$ taken from $\mathcal{M}$.

2.3 Multilog Propagation

Every site occasionally sends a copy of its site-multilog to other sites, which the receiver merges into its own site-multilog. By this so-called epidemic communication [2, 4, 17], every site eventually receives all actions and constraints submitted at any site. When site i receives a remote multilog M, it executes function *ReceiveAndCompare* (Algorithm 2), which first merges what it received into the local site-multilog. Then it adds any client conflict (non-commutativity or antagonism) relations that may exist between previously-known actions and the received actions. Note that no Enables relations may appear here.

To simplify exposition, we will assume here that communication is all-or-nothing: if communication succeeds, the receiver receives the full state of the sender's multilog. The protocol remains correct under weaker, FIFO-like assumptions.

Recall the example of Alice and Bob. Suppose that, concurrently with Alice's activity, Bob added action γ, meaning "cancel the meeting," to M_2. Action γ is antagonistic with action β (whereby Alice buys the ticket to attend the meeting); hence, $\beta \rightarrow_{\mathcal{M}} \gamma \wedge \gamma \rightarrow_{\mathcal{M}} \beta$. Sometimes later, site 2 sends its site-multilog to site 1; when site 1 receives it, it runs Algorithm 2. Client c_1 notices the antagonism

[6] In the algorithms, we leave the current time t implicit. Statements in curly brackets {like this} are comments.

Algorithm 2 *ReceiveAndCompare*(M)

Require: $M = (K, \rightarrow, \lhd, \nparallel)$ is a site-multilog received from a remote site
  $M_i := M_i \cup M$
  for all $\alpha \rightarrow_{\mathcal{M}} \beta$ such that $\alpha \in K_i \wedge \beta \in K_i$ **do**
    $\rightarrow_i := \rightarrow_i \cup \{(\alpha, \beta)\}$
  for all $\alpha \nparallel_{\mathcal{M}} \beta$ such that $\alpha \in K_i \wedge \beta \in K_i$ **do**
    $\nparallel_i := \nparallel_i \cup \{(\alpha, \beta)\}$

and adds constraint $\beta \rightarrow_1 \gamma \wedge \gamma \rightarrow_1 \beta$ to M_1. Thereafter, site-schedules at site 1 may include either β or γ, but not both.

2.4 Commitment and Consistency

Epidemic communication ensures that all site-multilogs eventually receive all information, but site-schedules might still differ between sites. For instance, in our previous example, site 1 might execute $S_i(t) = \text{INIT}; \alpha; \beta$, whereas site 2 may run $S_j(t) = \text{INIT}; \gamma$. To ensure consistency, we need global agreement on the set and order actions; this process is called *commitment*. We will now define precisely what we mean by consistency and commitment in terms of multilogs.

The following subsets of actions are of particular interest.

- *Guaranteed* actions appear in every schedule of $\Sigma(M)$. Formally, $Guar(M)$ is the smallest subset of K containing $\{\text{INIT}\} \cup \{\alpha \in A | \exists \beta \in Guar(M) : \alpha \lhd \beta\}$
- *Dead* actions never appear in a schedule of $\Sigma(M)$. $Dead(M)$ is the smallest subset of A containing $\{\alpha \in A | \exists m \geq 0, \beta_1, \ldots, \beta_m \in Guar(M) : \alpha \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha\} \cup \{\alpha \in A | \exists \beta \in Dead(M) : \beta \lhd \alpha\}$
- *Serialised* actions are those that are ordered with respect to all non-commuting actions that are not dead. $Serialised(M) \stackrel{\text{def}}{=} \{\alpha \in A | \forall \beta \in A,\ \alpha \nparallel \beta \Rightarrow \alpha \rightarrow \beta \vee \beta \rightarrow \alpha \vee \beta \in Dead(M)\}$
- *Decided* actions are either dead, or both guaranteed and serialised. $Decided(M) \stackrel{\text{def}}{=} Dead(M) \cup (Guar(M) \cap Serialised(M))$
- *Stable* (i.e., *durable*) actions are decided, and all actions that precede them by NotAfter are themselves stable: $Stable(M) \stackrel{\text{def}}{=} Dead(M) \cup \{\alpha \in Guar(M) | \forall \beta \in A,\ \beta \rightarrow \alpha \Rightarrow \beta \in Stable(M)\}$.

Recall that multilog M is said sound iff $\Sigma(M) \neq \varnothing$. Equivalently, M is sound iff $Dead(M) \cap Guar(M) = \varnothing$. If all actions in a multilog are decided, they are also stable: $Decided(M) = K \Rightarrow Stable(M) = K$.

An action that is both stable and guaranteed is called committed in the standard database terminology, whereas a dead action is called aborted. We do not use this vocabulary because we distinguish between guaranteed, decided and stable.

It is the role of proposers and acceptors to decide actions, by means of a *commitment protocol*. Acceptor a_i makes some action γ guaranteed (resp. dead, resp. ordered before non-commuting δ) by adding constraint $\gamma \lhd \text{INIT}$ (resp. $\gamma \rightarrow \gamma$, resp. $\gamma \rightarrow \delta$) into M_i.

The commitment protocol must ensure that the decisions taken at each site are consistent across the whole system. The standard OR concept of *eventual consistency* is captured by the following formal definition [14].

Definition 1 (Eventual Consistency). *An OR system is eventually consistent in a run r iff it satisfies the following correctness conditions:*

- *Local soundness (safety): Every site-schedule is sound.* $\forall i,t,\ S_i(t) \in \Sigma(M_i(t))$
- *Mergeability (safety): The union of all the site-multilogs along the run is sound.* $\Sigma(\bigcup_{i,t} M_i(t)) \neq \varnothing$
- *Eventual decision (liveness): Any action known at some site is eventually decided at every site.* $\forall t,i,j,\ \forall \alpha \in K_i(t),\ \exists t',\ \alpha \in \mathit{Decided}(M_j(t'))$

Local soundness means that every execution satisfies the known constraints. Eventual decision ensures that every action eventually becomes stable (durable) at correct sites. Mergeability ensures that local decisions do not eventually make the distributed system unsound.

We return to our example of Alice and Bob. Assuming users add no more actions, eventually all site-multilogs become $(\{\text{INIT}, \alpha, \beta\}, \{\alpha \rightarrow \beta, \alpha \rightarrow \gamma, \gamma \rightarrow \alpha\}, \{\alpha \lhd \beta\}, \varnothing)$. In this state, actions remain tentative; at time t, site 1 might execute $\text{INIT}; \alpha$, site 2 $\text{INIT}; \alpha; \beta$, and just INIT at $t+1$. A commitment protocol ensures that α and β eventually stabilise, and that both Alice and Bob learn the same outcome. It might, for instance, guarantee both α and β, hence aborting action γ. Acceptor a_1 would add $\beta \lhd \text{INIT}$ to M_1, which eventually propagates to M_2. This makes α and β guaranteed, decided and stable, and γ dead, in all site-multilogs. Inevitably, all site-schedules will eventually be $\text{INIT}; \alpha; \beta$.

3 Classical OR commitment algorithms

We can abstract a number of previous commitment algorithms for OR systems as an algorithm, noted $\mathcal{A}(M)$, that offers decisions based on multilog M. $\mathcal{A}$ is assumed to run at a single site (although it may be possible to run several instances at different sites). Noting the result $M' = \mathcal{A}(M)$, $\mathcal{A}$ must satisfy these requirements:

- $\mathcal{A}$ adds constraints; they represent decisions:

$$\begin{array}{rcl} \alpha \rightarrow' \beta & \Rightarrow & \alpha \rightarrow \beta \vee \alpha \nparallel \beta \vee \beta = \alpha \\ \alpha \lhd' \beta & \Rightarrow & \alpha \lhd \beta \vee (\beta = \text{INIT}) \\ \nparallel' & = & \nparallel \end{array}$$

- The algorithm does not add actions: $K = K'$.

- If M is sound, then M' is sound.
- If invoked sufficiently often, $\mathcal{A}$ eventually decides: For any non-decreasing series of sound multilogs $M^0 \subseteq M^1 \subseteq \ldots \subseteq M^k \subseteq \ldots \; \forall i, \alpha \in K^i, \; \exists j : \alpha \in \mathit{Decided}(\mathcal{A}(M^j))$.

$\mathcal{A}$ could be any algorithm satisfying the requirements. Here is one possibility, noted $\mathcal{A}_{\text{Conservative}}(<)$. Assume some arbitrary total order of actions $<$. A schedule executing in this order can be made sound, with respect to some multilog M, by the following procedure. If $\alpha < \beta$ and $\alpha \nparallel \beta$, then $\mathcal{A}_{\text{Conservative}}(<)$ decides $\alpha \rightarrow \beta$. It decides α dead (add $\alpha \rightarrow \alpha$ to M) if either: $\alpha < \beta$ but $\beta \rightarrow \alpha$ (because otherwise they would execute in the wrong order), or $\alpha < \beta$ but $\beta \lhd \alpha$ (because it is not known whether β can be guaranteed). Otherwise, it decides α guaranteed (add $\alpha \lhd \text{INIT}$ to the multilog). It should be clear that in general, this approach, while safe, will tend to kill more actions than necessary, unless the total order $<$ is computed with knowledge of the constraints.

In the Bayou system [17], $<$ is the order in which actions are received at a single primary site. An action aborts if it fails an application-specific precondition, which we reify as a $\rightarrow$ constraint. In the Last-Writer-Wins approach [5], an action (completely overwriting some datum) is stamped with the time it is submitted. Two actions that modify the same datum are related by $\rightarrow$ in timestamp order. Sites execute actions in arbitrary order and apply $\mathcal{A}_{\text{Conservative}}(<)$. Consequently, a datum has the state of the most recent write (in timestamp order).

Previously, in the IceCube project [12] we proposed a different approach. $\mathcal{A}_{\text{IceCube}}$ is an optimization algorithm that minimizes the number of dead actions in $\mathcal{A}_{\text{IceCube}}(M)$. It does so by heuristically comparing all possible sound schedules that can be generated from the current site-multilog.

Except for LWW, which is deterministic, the above algorithms centralise commitment at a primary site.

4 A decentralised commitment protocol

To decentralise decision, one approach might be to determine a global total order $<$, using a consensus algorithm such as Paxos [9], and apply $\mathcal{A}_{\text{Conservative}}(<)$. However, this tends to kill more actions than necessary; we would rather base our solution on a batching and optimising algorithm such as IceCube.

A key observation is that eventual consistency is equivalent to the following property [15]: The site-multilogs of all sites share a common prefix of stable actions, which grows to include every action eventually. Commitment serves to agree on an extension of this prefix. Since clients continue to make progress beyond this prefix, the commitment protocol can run asynchronously in the background.

In our protocol, different sites run instances of $\mathcal{A}$ to make proposals. It achieves agreement between proposals via decentralised election. This works even if $\mathcal{A}$ is non-deterministic, or if sites use different $\mathcal{A}$ algorithms.

Algorithm 3 Algorithm at site i

Require: M_i: local site-multilog
Require: $proposals_i[n]$: array of proposals, indexed by site; a proposal is a multilog
1: $M_i := (\{\text{INIT}\}, \varnothing, \varnothing, \varnothing)$
2: $proposals_i := [((\{\text{INIT}\}, \varnothing, \varnothing, \varnothing), 0), \ldots, ((\{\text{INIT}\}, \varnothing, \varnothing, \varnothing), 0)]$
3: **loop** {Client submits}
4: Choose $L \subseteq A$ {Submit actions}
5: *ClientActionsConstraints*(L) {Compute local constraints}
6: ||
7: **loop** {Compute current local state}
8: Choose $S_i \in \Sigma(M_i)$
9: Execute S_i
10: ||
11: **loop** {Proposer}
12: *UpdateProposal* {Suppress redundant parts}
13: $proposals_i[i] := \mathcal{A}(M_i \cup proposals_i[i])$ {New proposal, keeping previous}
14: Increment $proposals_i[i].ts$
15: ||
16: **loop** {Acceptor}
17: *Elect*
18: ||
19: **loop** {Epidemic transmission}
20: Choose $j \neq i$;
21: Send copy of M_i, $proposals_i$ to j
22: ||
23: **loop** {Epidemic reception}
24: Receive multilog M, proposals P from some site $j \neq i$
25: *ReceiveAndCompare*(M)
26: *MergeProposals*(P)

4.1 Variables and notation

In what follows, i represents the current site, and j,k range over $\mathcal{I} \setminus \{i\}$.

Each proposer has a fixed *weight*, such that $\sum_{k \in \mathcal{I}} weight_k = 1$. In practice, we expect only a small number of sites to have non-zero weights (in the limit one site might have weight 1, this is a primary site as in Section 3), but the safety of our protocol does not depend on how weights are allocated.

Each site stores the most recent proposal received from each proposer in array $proposals_i$, of size n (the number of sites). To keep track of proposals, each entry $proposals_i[k]$ carries a logical timestamp, noted $proposals_i[k].ts$.

Algorithm 4 *UpdateProposal*

1: Let $P = (K_P, \rightarrow_P, \lhd_P, \not\Vdash_P) = proposals_i[i]$
2: $K_P := K_P \setminus Decided(M_i)$
3: $\rightarrow_P := \{(\alpha,\beta) \in \rightarrow_P \mid \alpha \in K_P \vee \beta \in K_P\}$
4: $\lhd_P := \{(\alpha,\beta) \in \lhd_P \mid \alpha \in K_P \vee \beta \in K_P\}$
5: $\not\Vdash_P := \varnothing$

Each site performs Algorithm 3. First it initialises the site-multilog and proposals data structures, then it consists of a number of parallel iterative threads, detailed in the next sections. Within a thread, an iteration is atomic. Iterations are separated by arbitrary amounts of time.

4.2 Client, local state, proposer

The first thread (lines 3–5) constitutes one half of the client. An application submits tentative operations to its local site-multilog, which the site-schedule will (hopefully) execute in the second thread. Constraints relating new actions to previous ones are included at this stage by function *ClientActionsConstraints* (defined in Algorithm 1). The other half of the client is function *ReceiveAndCompare* (Algorithm 2) invoked in the last thread (line 25).

The second thread (lines 7–9) computes the current tentative state by executing some sound site-schedule. It is possible that the current schedule does not linearly extend the previous one; this can be implemented as a roll-back followed by forward execution.

The third thread (11–14) computes proposals by invoking $\mathcal{A}$. A proposal extends the current site-multilog with proposed decisions. A proposer may not retract a proposal that was already received by some other site. According to the definition of $\mathcal{A}$ (Section 3), argument $M_i \cup proposals_i[i]$ ensures that these two conditions are satisfied. However, once a candidate has either won or lost an election, it becomes redundant; *UpdateProposal* removes it from the proposal (Algorithm 4).

4.3 Election

The fourth thread (16–17) conducts elections. Several elections may be taking place at any point in time. An acceptor is capable of determining locally the outcome of elections. A proposal can be decomposed into a set of eligible candidates.

4.3.1 Eligible candidates

A candidate cannot be just any subset of a proposal. Consider for instance a proposal $P = (\{\text{INIT}, \alpha, \gamma\}, \{\alpha \rightarrow \gamma, \gamma \rightarrow \alpha, \alpha \rightarrow \alpha\}, \{\gamma$
and a candidate X constructed upon P. If X could contain γ and not α, then we might guarantee γ without killing α, which would be incorrect. Capturing this intuition, X must be a *well-formed prefix* of P:

Definition 2 (Well-formed prefix). *Let $M' = (K', \rightarrow', \lhd', \nparallel')$ and $M = (K, \rightarrow, \lhd, \nparallel)$ be two multilogs. M' is a* well-formed prefix *of M, noted $M' \overset{wf}{\sqsubset} M$, if (i) it is a subset of M, (ii) it is stable, (iii) it is left-closed for its actions, and (iv) it is closed for its constraints.*

$$M' \overset{wf}{\sqsubset} M \overset{\text{def}}{=} \begin{cases} M' \subseteq M \\ K' = \mathit{Stable}(M') \\ \forall \alpha, \beta \in A,\ \beta \in K' \Rightarrow \begin{cases} \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow' \beta \\ \alpha \lhd \beta \Rightarrow \alpha \lhd' \beta \\ \alpha \nparallel \beta \Rightarrow \alpha \nparallel' \beta \end{cases} \\ \forall \alpha, \beta \in A,\ (\alpha \rightarrow' \beta \vee \alpha \lhd' \beta \vee \alpha \nparallel' \beta) \Rightarrow \alpha, \beta \in K' \end{cases}$$

Well-formedness ensures that if a $\rightarrow$ or $\lhd$ cycle is present in M, then M' either includes the whole cycle or none of its actions. Unfortunately, because of concurrency and asynchronous communication, it is possible that some sites know of a $\rightarrow$ cycle and not others. Therefore we also require the following property:

Definition 3 (Eligible). *An action is* eligible *in set L if all its predecessors by client NotAfter and NonCommuting relations are in L. A multilog M is eligible if all actions in K are eligible in K: $\mathit{eligible}(M) \overset{\text{def}}{=} \forall \alpha, \beta \in A,\ \beta \in K \wedge (\alpha \rightarrow_{\mathcal{M}} \beta \vee \alpha \nparallel_{\mathcal{M}} \beta) \Rightarrow \alpha \in K$*

To compute eligibility precisely would require local access to the distributed state, which is impossible. Therefore acceptors must compute a safe approximation (i.e., false negatives are allowed) of eligibility. Here is an example possible approximation algorithm evaluated at site i. Consider some action α, submitted at site j, and known in K_i. If all actions submitted before or concurrently[7] with α have been received at site i, then all those actions have gone through either *ClientActionsConstraints* or *ReceiveAndCompare*; hence α is eligible.

It is possible to compute better approximations under some conditions; for instance if it is known that $\rightarrow$ and $\nparallel$ relations are acyclic in $\mathcal{M}$, then all candidates are eligible.

4.3.2 Computation of votes

We define a vote as a pair $(\mathit{weight}, \mathit{siteId})$. The comparison operator for votes breaks ties by comparing site identifiers: $(w,i) > (w',i') \overset{\text{def}}{=} w > w' \vee (w = w' \wedge i > i')$. Therefore, votes add up as follows: $(w,i) + (w',i') \overset{\text{def}}{=} (w + w', \max(i,i'))$. Candidates are *compatible* if their union is sound: $\mathit{compatible}(M,M') \overset{\text{def}}{=} \Sigma(M \cup M') \neq \varnothing$. The votes of compatible candidates add up; $\mathit{tally}(X)$ com-

[7] In the sense of the happens-before relation [8].

Algorithm 5 *Elect*

1: Let X be a multilog such that:

$$\begin{array}{ll} & \exists k \in \mathcal{J} : X \stackrel{wf}{\sqsubset} proposals_i[k] \\ \wedge & X \not\subseteq M_i \\ \wedge & eligible(X) \\ \wedge & tally(X) > \max_{B \in opponents(X)} (tally(B)) + cotally(X) \end{array}$$

2: **if** such an X exists **then**
3: Choose such an X
4: $M_i := M_i \cup X$

putes the total vote for some candidate X:

$$tally(X) \stackrel{\text{def}}{=} \sum_{k:X \stackrel{wf}{\sqsubset} proposals_i[k]} (weight_k, k)$$

An election pits some candidate against *comparable* candidates from all other sites. Two multilogs are comparable if they contain the same set of actions: $comparable(M, M') \stackrel{\text{def}}{=} K = K'$. The direct opponents of candidate X in some election are comparable candidates that are not compatible with X: $opponents(X) \stackrel{\text{def}}{=} \{B | \exists k : B \stackrel{wf}{\sqsubset} proposals_i[k] \ \wedge (comparable(B, X) \wedge \neg compatible(X, B))\}$. However, we must also count missing votes, i.e., the weights of sites whose proposals do not yet include all actions in X. Function $cotally(X)$ adds these up:

$$cotally(X) \stackrel{\text{def}}{=} \sum_{k:K_X \not\subseteq K_{proposals_i[k]}} (weight_k, k)$$

Algorithm 5 depicts the election algorithm. A candidate is a well-formed prefix of some proposal. We ignore already-elected candidates and we only consider eligible ones. A candidate wins its election if its tally is greater than the tally of any direct opponent, plus its cotally. Note that as proposals make progress, cotally tends towards 0, therefore some candidate is eventually elected. We merge the winner into the site-multilog.

4.3.3 Epidemic communication

The last two threads (lines 19–26) exchange multilogs and proposals between sites. Function *ReceiveAndCompare* (defined in Algorithm 2, Section 2.3) compares actions newly received to already-known ones, in order to compute non-commutativity and antagonism constraints. In Algorithm 6 a receiver updates its own set of proposals with any more recent ones.

Algorithm 6 *MergeProposals*(P)

```
1: for all k do
2:     if proposals_i[k].ts < P[k].ts then
3:         proposals_i[k] := P[k]
4:         proposals_i[k].ts := P[k].ts
```

4.4 Example

We return to our example. Recall that, once Alice and Bob have submitted their actions, and site 1 and site 2 have exchanged site-multilogs, both site-multilogs are equal to $(\{\text{INIT}, \alpha, \beta\}, \{\alpha \rightarrow \beta, \alpha \rightarrow \gamma, \gamma \rightarrow \alpha\}, \{\alpha \lhd \beta\}, \varnothing)$. Now Alice (site 1) proposes to guarantee α and β, and to kill γ: $proposals_1[1] = M_1 \cup \{\beta \lhd \text{INIT}\}$. Meanwhile, Bob at site 2 proposes to guarantee γ and α, and to kill β: $proposals_2[2] = M_2 \cup \{\gamma \lhd \text{INIT}, \alpha \lhd \text{INIT}\}$. These proposals are incompatible; therefore that the commitment protocol will eventually agree on at most one of them.

Consider now a third site, site 3; assume that the three sites have equal weight $\frac{1}{3}$. Imagine that site 3 receives site 2's site-multilog and proposal, and sends its own proposal that is identical to site 1's. Sometime later, site 3 sends its proposal to site 1. At this point, site 1 has received all sites' proposals. Now site 1 might run an election, considering a candidate X equal to $proposals_1[1]$. X is indeed a well-formed prefix of $proposals_1[1]$; X is eligible; $tally(X) = \frac{2}{3}$ is greater than that of X's only opponent ($tally(proposals_1[2]) = \frac{1}{3}$); and $cotally(X) = 0$. Therefore, site 1 elects X and merges X into M_1. Any other site will either elect X (or some compatible candidate) or become aware of its election by epidemic transmission of M_1.

5 Discussion

5.1 Safety proof outline

Section 1 states our safety property, the conjunction of mergeability and local soundness. Clearly Algorithm 3 satisfies local soundness; see lines 7–9. We now outline a proof of mergeability.

We will say that candidate X is elected in a run r if at a time t and for some acceptor i, i executes at t Algorithm 5 electing a candidate Y such that $X \overset{wf}{\sqsubset} Y$. Moreover for a run r of Algorithm 3, we will note $Elected(r,t)$ the set of candidates elected in r up to t (included), and $Elected(r)$ the set of candidates elected during r. Observe that, since $\mathcal{M}'$ is sound, Algorithm 3 satisfies mergeability in a run r if and only if the acceptors elect a sound set of candidates during r ($\bigcup_{X \in Elected(r)} X$ is sound). Now suppose by contradiction that during a run r, this set is unsound.

In every run of Algorithm 3, candidates are well-formed and eligible, therefore $Elected(r)$ forms an unsound set of candidates, i.e., there are two elected candidates X and X' such that (i) X and X' are non-compatible, and (ii) X and X' are minimal. Minimality is defined as follows:

Definition 4 (minimality). *A multilog M is said minimal iff:* $\forall M' \subseteq M,\ M' \overset{wf}{\sqsubset} M \Rightarrow M' = M$.

Let us define some notation: i (resp. j) is the acceptor who elects X (resp. X') in r. t is the time where i elects X in r (resp. t' for X' on j). For a proposer k, t_k (resp. t'_k) is the time at which it sent $proposals_i[k](t)$ to i (resp. $proposals_j[k](t')$ to j). Q (resp. Q') is the set of proposers that vote for X at t on i (resp. for X' at t' on j); formally $Q = \{k | X \overset{wf}{\sqsubset} proposals_i[k](t)\}$ and $Q' = \{k | X' \overset{wf}{\sqsubset} proposals_j[k](t')\}$. Hereafter, and without loss of generality, we suppose that (i) $t' > t$, (ii) X is the first candidate non-compatible with X' elected in r, and (iii) $Elected(r, t'-1)$ is sound.

Since j elects X' at t', at that time on site j':

$$tally(X') > \max_{B \in opponents(X')}(tally(B)) + cotally(X') \tag{1}$$

Equation 1 yields an upper bound for $tally(X)$ on i at t, as follows. Consider some $k \in Q$. If $t_k < t'_k$ then from Algorithm 4, and the fact that $Elected(r, t'-1)$ is sound, we know that $X \overset{wf}{\sqsubset} proposals_j[k](t')$. If now $t_k > t'_k$, then either (i) k has not yet voted on $K_{X'}$ at t' on j and its weight is counted in $cotally(X')$, or (ii) its vote at t' on j already includes X.

The other cases are impossible: if k votes for X' or for an opponent of X' – that is not X – at t', since X and X' are not compatible, X' and X are minimal, and $Elected(r, t-1)$ is sound, k cannot vote for X at t.

Thus from Equation 1 we obtain:

$$tally_j(X')(t') > tally_i(X)(t) \tag{2}$$

where $tally_k(Z)(\tau)$ means the value of $tally(Z)$ computed at time τ on site k.

Now consider some $k \in Q'$. If $t_k > t'_k$ then X being the first candidate non-compatible with X' elected in r, from Algorithm 4, we have $X' \overset{wf}{\sqsubset} proposals_i[k](t)$. If $t_k < t'_k$, now either (i) $X' \overset{wf}{\sqsubset} proposals_i[k](t)$ or (ii) k has not yet voted on $X.K$ on i at t.

The reasoning here is similar: namely we use the minimality of X and X', the fact that they are non-compatible, and this time, that X is the first candidate non-compatible with X' elected in r.

From the above, it follows:

$$tally_j(X')(t') < tally_i(X')(t) + cotally_i(X)(t) \tag{3}$$

Now combining equations 2 and 3, we obtain on i at t,

$$tally(X) < \max_{B \in opponents(X)}(tally(B)) + cotally(X) \tag{4}$$

X cannot be elected on i at t. Contradiction.

5.2 Liveness proof outline

Consider by contradiction that there exists an execution r such that in r eventual decision is violated. Hence there is an action α that is never decided. Since sites are reliable, and links are fair-lossy, Algorithm 3 ensures that eventually every proposer decides on α.

Reasoning from the fact that a minimal candidate cannot remain undecided (because plurality always happens), we prove that such a run is impossible.

5.3 Message cost

Interestingly, the message cost of our protocol varies with application semantics, along two dimensions. *Optimism degree* is the size of a batch, the number of actions that a site may execute optimistically before requiring commitment. *Semantic complexity degree*, which relates the complexity of the client constraint graph $\mathcal{M}$ with the number of votes required.

To illustrate how the cost varies with semantics, consider an application where there are no client constraints whatsoever. All actions commute with one another and no action never needs to be made dead. In such a case, every candidate is trivially eligible and trivially compatible with all other candidates. Assume the optimism degree is d, then the amortised message cost to commit an action is $\frac{n}{2} \times \frac{1}{d}$, since a chain of $\frac{n}{2}$ messages constructs a majority and (ii) candidates can contain up to d actions.

Conversely, an application where all action pairs are non-commuting and there are no antagonisms (hence no actions to be made dead) requires a total order; hence it will have to pay at least the cost of consensus.

A related issue is fault tolerance, which is also related to semantics. An application with no constraints whatsoever is trivially fault tolerant, since in this case eventual delivery of site-multilogs is sufficient. In general a small number of faults can be tolerated as long as cotally remains small enough to elect. Obviously, a site whose weight is zero can crash without impacting liveness of the system.

A precise evaluation of message cost and fault resilience is left for future work.

6 Related work

In previous OR systems, commitment was often either centralised at a primary site [12, 17] or oblivious of semantics [5, 13].

Our election algorithm is inspired by Keleher's Deno system [6], a pessimistic system, which performs a discrete sequence of elections. Keleher proposes plurality voting to ensure progress when none of multiple competing proposals gains a majority. The VVWV protocol of Barreto and Ferreira generalizes Deno's voting procedure, enabling continuous voting [1].

The only semantics supported by Deno or VVWV is to enforce Lamport's happens-before relation [8]; all actions are assumed be mutually non-commuting. Happens-before captures potential causality; however an event may happen-before another even if they are not truly dependent. This paper further generalizes VVWV by considering semantic constraints.

ESDS [3] is a decentralised replication protocol that supports some semantics. It allows users to create an arbitrary causal dependence graph between actions. ESDS eventually computes a global total order among actions, but also includes an optimisation for the case where some action pairs commute. ESDS does not consider atomicity or antagonism relations, nor does it consider dead actions.

Bayou [17] supports arbitrary application semantics. User-supplied code controls whether an action is committed or aborted. However the system imposes an arbitrary total execution order. Bayou centralises decision at a single primary replica.

IceCube [7] introduced the idea of reifying semantics with constraints. The IceCube algorithm computes optimal proposals, minimizing the number of dead actions. Like Bayou, commitment in IceCube is centralised at a primary. Compared to this article, IceCube supports a richer constraint vocabulary, which is useful for applications, but harder to reason about formally.

The Paxos distributed protocol [9] computes a total order. Such total order may be used to implement *state-machine replication* [8], whereby all sites execute exactly the same schedule. Such a total order over all actions is necessary only if all actions are mutually non-commuting. In Section 3 we showed how to add semantic constraints to a total order, but this kind of approach is clearly sub-optimal.

Generalized Paxos [10] and Generic Broadcast [11] take commutativity relations into account and compute a partial order. They do not consider any other semantic relations. Both Generalized Paxos [10] and our algorithm make progress when a majority is not reached, although through different means. Generalized Paxos starts a new election instance, whereas our algorithm waits for a plurality decision.

7 Conclusion and future work

The focus of our study is applications with rich semantics. Previous approaches to replication did not support a sufficiently rich repertoire of semantics, or relied on a centralized point of commitment. They often impose a total order, which is stronger than necessary.

In contrast, we propose a decentralized commitment protocol for semantically-rich systems. Our approach is to reify semantic relations as constraints, which restrict the scheduling behavior of the system. According to our formal definition of consistency, the system has an obligation to resolve conflicts, and to eventually execute equivalent stable schedules at all sites.

Our protocol is safe in the absence of Byzantine faults. It uses voting to avoid any centralization bottleneck. It uses plurality voting to make progress even when an election does not reach a majority.

It proceeds with elections continuously and incrementally, without any obvious rounds. However, as currently specified, it is not live in the presence of faults.

There is an interesting trade-off in the proposal/voting procedure. The system might decide frequently, in small increments, so that users quickly know whether their tentative actions are accepted or rejected. However this might be non-optimal as it may cut off interesting future behaviors. Or it may decide less frequently, and base its decisions on a large batch of tentative actions at a time. This imposes more uncertainty on users, but decisions may be closer to the optimum. We plan to study this trade-off in our future work.

References

[1] João Barreto and Paulo Ferreira. An efficient and fault-tolerant update commitment protocol for weakly connected replicas. In *Euro-Par*, pages 1059–1068, Lisbon, Portugal, September 2005. .

[2] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 1–12, Vancouver, BC, Canada, August 1987. Also appears Op. Sys. Review 22(1): 8-32 (1988).

[3] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(Special issue on Distributed Algorithms):113–156, 1999.

[4] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California Santa Cruz, Santa Cruz, CA, USA, December 1992. Tech. Report no. UCSC-CRL-92-52, .

[5] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976. .

[6] Peter J. Keleher. Decentralized replicated-object protocols. In *Symp. on Principles of Dist. Comp. (PODC)*, Atlanta, GA, USA, May 1999. .

[7] 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. .

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

[9] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. .

[10] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005. .

[11] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing Journal*, 15(2):97–107, 2002. .

[12] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Tenth 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. .

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

[14] Marc Shapiro and Karthik Bhargavan. The Actions-Constraints approach to replication: Definitions and proofs. Technical Report MSR-TR-2004-14, Microsoft Research, March 2004. .

[15] 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. .

[16] 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. .

[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)*, Copper Mountain CO, USA, December 1995. ACM SIGOPS. .

[18] Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Computer Supported Cooperative Work*, pages 171–180, Philadelphia, PA, USA, December 2000.

Contents

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

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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