Decentralised Commitment for Optimistic Semantic Replication

Pierre Sutra, Joao Barreto, Marc Shapiro

Decentralised Commitment for Optimistic Semantic Replication*

Pierre Sutra[1], João Barreto[2], and Marc Shapiro[1]

[1] Université Paris VI and INRIA Rocquencourt, France
[2] INESC-ID and Instituto Superior Técnico, Lisbon, Portugal

Abstract. We study large-scale distributed cooperative systems that use optimistic replication. We represent a system as a graph of actions (operations) connected by edges that reify semantic constraints between actions. Constraint types include conflict, execution order, dependence, and atomicity. The local state is some schedule that conforms to the constraints; because of conflicts, client state is only tentative. For consistency, site schedules should converge; we designed a decentralised, asynchronous commitment protocol. Each client 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.

1 Introduction

In a large-scale cooperative system, access to shared data is a performance and availability bottleneck. One solution is optimistic replication (OR), where a process may read or update its local replica without synchronising with remote sites [17]. OR decouples data access from network access.

In OR, each site makes progress independently, even while others are slow, currently disconnected, or currently working in isolated mode. OR is well suited to peer-to-peer systems and to devices with occasional connectivity.

Some limited knowledge of semantics provides a lot of extra power and flexibility. Therefore, we model the system as a graph, called a multilog, where each vertex represents an action (i.e., an operation proposed by some client), and an edge is a semantic relation between vertices, called a constraint. Our constraints include conflict, ordered execution, causal dependence, and atomicity. Each site has its own multilog, which contains actions submitted by the local client, and their constraints, as well as those received from other sites. The current state is some execution schedule that contains actions from the site's multilog, arranged to conform with its constraints. For instance, when actions are antagonistic, at least one must abort; an action that depends on an aborted action

* This research is funded in part by the European project Grid4All, the French project Respire and by FCT grant SFRH/BD/13859, Portugal.

R. Meersman and Z. Tari et al. (Eds.): OTM 2007, Part I, LNCS 4803, pp. 318–335, 2007.

must abort too; non-commutative actions should be scheduled in the same order everywhere, etc. The site may choose any conforming schedule, e.g., one that minimises aborts, or one that reflects user preferences.

For consistency, sites should agree on a common, stable and correct schedule. We call this agreement *commitment*. Some cooperative OR systems never commit, such as Roam [16] or Draw-Together [6]. Previous work on commitment for semantic OR such as Bayou [20] or IceCube [15] centralises the agreement at a central site. Other work decentralises commitment (e.g., Paxos consensus [11]) but ignores semantics. It is difficult to reconcile semantics and decentralisation. One possible approach would use Paxos to compute a total order, and abort any actions for which this order would violate a constraint. However this approach aborts actions unnecessarily. Furthermore, the arbitrary total order may be very different from what users expect.

A better approach is to order only non-commuting pairs of actions, to abort only when actions are antagonistic, to minimise dependent aborts, and to remain close to user expectations. We propose an efficient, decentralised protocol that uses semantic information for this purpose. Participating sites make and exchange proposals asynchronously; our algorithm decomposes each one into semantically-meaningful candidates; it runs elections between comparable candidates. A candidate that collects a majority or a plurality wins its election. Voting ensures that the common schedule is similar to the tentative schedules, minimising user surprise. Our protocol orders only non-commuting actions and minimises unnecessary aborts.

This paper makes several contributions:

- Our algorithm combines a number of known techniques in a novel manner.
- We identify the concept of a semantically-meaningful unit for election (which we call a candidate).
- We propose an efficient commitment protocol system that is both decentralised and semantic-oriented, and that has weak communication requirements.
- We show how to minimise user surprise, the committed schedule being similar to local tentative schedules.
- We prove that the protocol is safe 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 this paper follows. Section 2 introduces our system model and our vocabulary. Section 3 discusses an abstraction of classical OR approaches that is later re-used in our algorithm. Section 4 specifies client behaviour. Our commitment protocol is specified in Section 5. Section 6 provides a proof outline and adresses message cost. We compare with related work in Section 7. In conclusion, Section 8 discusses our results and future work.

2 System Model

Following the ACF model [18], an OR system is an asynchronous distributed system of *n sites* $i, j, \ldots \in \mathcal{J}$. A site that crashes eventually recovers with its identity and persistent memory intact (but may miss some messages in the interval). Clients propose actions (deterministic operations) noted $\alpha, \beta, \ldots \in A$. An action might request, for instance, "Debit 100 euros from bank account number 12345."

A *multilog* is a quadruple $M = (K, \rightarrow, \lhd, \nparallel)$, representing a graph where the vertices K are actions, and $\rightarrow$, $\lhd$ and $\nparallel$ (pronounced NotAfter, Enables and NonCommuting respectively) are three sets of edges called *constraints*. We will explain their semantics shortly.[1]

We identify a state with a *schedule* S, a sequence of distinct actions ordered by $<_S$ executed from the common initial state INIT. The following safety condition defines semantics of NotAfter and Enables in relation to schedules. We define $\Sigma(M)$, the set of schedules S that are *sound* with respect to multilog M, as follows:

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

Constraints represent scheduling relations between actions: NotAfter is a (non-transitive) ordering relation and Enables is right-to-left (non-transitive) implication.[2]

Constraints represent semantic relations between actions. For instance, consider a database system (more precisely, a serialisable database that transmits transactions by value, such as DBSM [13]). Assume shared variables x, y, z are initially zero. Two concurrent transactions $T_1 = r(x)0; w(z)1$ and $T_2 = w(x)2$ are related by $T_1 \rightarrow T_2$, since T_1 read a value that precedes T_2's write.[3] T_1 and $T_3 = r(z)0; w(x)3$ are antagonistic, i.e., one or the other (or both) must abort, as each is NotAfter the other. In the execution $T_1; T_4$ where $T_4 = r(z)1$, the latter transaction depends causally on the former, i.e., they may run only in that order, and T_4 aborts if T_1 aborts; we write $T_1 \rightarrow T_4 \wedge T_1 \lhd T_4$. As another example, Section 6.4 discusses how to encode the semantics of database transactions with constraints.

Non-commutativity imposes a liveness obligation: the system must put a NotAfter between non-commuting actions, or abort one of them. (Therefore, non-commutativity does not appear in the above safety condition.) The system also has the obligation to resolve antagonisms by aborting actions.

For instance, transactions T_1 and $T_5 = r(y)0$ commute if x, y and z are independent. In a database system that commits operations (as opposed to commiting values), transactions $T_6 =$"Credit 66 euros to Account 12345" and $T_7 =$"Credit 77 euros to Account 12345" commute since addition is a commutative operation, but T_6 and $T_8 =$"Debit 88 euros from Account 12345" do not, if bank accounts are not allowed to become negative. We write $T_6 \nparallel T_8$.

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

[2] A constraint is a relation in $A \times A$. By abuse of notation, for some relation $\mathcal{R}$, we write equivalently $(\alpha \mathcal{R} \beta) \in M$ or $\alpha \mathcal{R} \beta$ or $(\alpha, \beta) \in \mathcal{R}$. $\nparallel$ is symmetric and $\lhd$ is reflexive. They do not have any further special properties; in particular, $\rightarrow$ and $\lhd$ are not transitive, are not orders, and may be cyclic.

[3] $r(x)n$ stands for a read of x returning value n, and $w(x)n$ writes the value n into x.

Order, antagonism and non-commutativity are collectively called conflicts.[4]

Clients submit actions to their local site; sites exchange actions and constraints asynchronously. The current knowledge of Site i at time t is the distinguished *site-multilog* $M_i(t)$. Initially, $M_i(0) = (\{\text{INIT}\}, \varnothing, \varnothing, \varnothing)$, and it grows over time, as we will explain later. A site's current state is the *site-schedule* $S_i(t)$, which is some (arbitrary) schedule $\in \Sigma(M_i(t))$.

An action executes tentatively only, because of conflicts and related issues. However, an action might have sufficient constraints that its execution is stable. We distinguish the following interesting subsets of actions relative to M.

- *Guaranteed* actions appear in every schedule of $\Sigma(M)$. Formally, $Guar(M)$ is the smallest subset of K satisfying: $\text{INIT} \in Guar(M) \wedge ((\alpha \in Guar(M) \wedge \beta \lhd \alpha) \Rightarrow \beta \in Guar(M))$.
- *Dead* actions never appear in a schedule of $\Sigma(M)$. $Dead(M)$ is the smallest subset of A satisfying: $((\alpha_1, \ldots, \alpha_{m\geq 0} \in Guar(M)) \wedge (\beta \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow \beta) \Rightarrow \beta \in Dead(M)) \wedge ((\alpha \in Dead(M) \wedge \alpha \lhd \beta) \Rightarrow \beta \in Dead(M))$.
- *Serialised* actions are either dead or ordered with respect to all non-commuting constraints. $Serialised(M) \stackrel{\text{def}}{=} \{\alpha \in K | \forall \beta \in K,\ \ \alpha \not\parallel \beta \Rightarrow \alpha \rightarrow \beta \vee \beta \rightarrow \alpha \vee \beta \in Dead(M) \vee \alpha \in Dead(M)\}$.
- *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 Not-After or Enables are themselves stable: $Stable(M) \stackrel{\text{def}}{=} Dead(M) \cup \{\alpha \in Guar(M) \cap Serialised(M) | \forall \beta \in A \quad (\beta \rightarrow \alpha \vee \beta \lhd \alpha) \Rightarrow \beta \in Stable(M)\}$.

To *decide* an action α relative to a multilog M, means to add constraints to the M, such that $\alpha \in Decided(M)$. In particular, to guarantee α, we add $\alpha \lhd \text{INIT}$ to the multilog, and to kill α, we add $\alpha \rightarrow \alpha$; to serialise non-commuting actions α and β, we add either $\alpha \rightarrow \beta$, $\beta \rightarrow \alpha$, $\alpha \rightarrow \alpha$, or $\beta \rightarrow \beta$.

Multilog M is said sound iff $\Sigma(M) \neq \varnothing$, or equivalently, iff $Dead(M) \cap Guar(M) = \varnothing$. An unsound multilog is definitely broken, i.e., no possible schedule can satisfy all the constraints, not even the empty schedule.

Referring to the standard database terminology, a committed action is one that is both stable and guaranteed, and aborted is the same as dead.

The standard correctness condition in OR systems is Eventual Consistency: if clients stop submitting, eventually all sites reach the same state. We extend this definition by not requiring that clients stop, by requiring that all states be correct, and by demanding decision.

Definition 1. *Eventual Consistency. An OR system is eventually consistent iff it satisfies all the following conditions:*

- *Local soundness (safety): Every site-schedule is sound:* $\forall i,t \quad S_i(t) \in \Sigma(M_i(t))$

[4] Some authors suggest to remove conflicts by transforming the actions [19]. We assume that, if such transformations are possible, they have already been applied.

α	$<$	β	$<$	γ	Decision	
		β	$\nparallel$	γ	(Serialise)	$\beta \rightarrow \gamma$
guar.	$\leftarrow$	β			(Kill β)	$\beta \rightarrow \beta$
dead	$\lhd$	β			(β is dead)	
		β	$\rhd$	γ	(Kill β)	$\beta \rightarrow \beta$
β not dead by above rules					(Guarantee β)	$\beta \lhd \text{INIT}$

Fig. 1. $\mathcal{A}_{\text{Conservative}(<)}$: Applying semantic constraints to a given total order

– *Mergeability (safety): The union of all the site-multilogs over time is sound:*

$$\Sigma(\bigcup_{i,t} M_i(t)) \neq \varnothing$$

– *Eventual propagation (liveness):* $\forall i, j \in \mathcal{I} \quad \forall t \quad \exists t' : M_i(t) \subseteq M_j(t')$
– *Eventual decision (liveness): Every submitted action is eventually decided:*

$$\forall \alpha \in A \quad \forall i \in \mathcal{I} \quad \forall t \quad \exists t' : K_i(t) \subseteq \mathit{Decided}(M_i(t'))$$

We assume some form of epidemic communication to fulfill Eventual Propagation. A commitment algorithm aims to fulfill the obligations of Eventual Decision. Of course, it must also satisfy the safety requirements.

3 Classical or Commitment Algorithms

Our proposal builds upon existing commitment algorithms for OR systems. Generally, these either are centralised or do not take constraints into account. We note $\mathcal{A}(M)$ some algorithm that offers decisions based on multilog M ; with no loss of generality, we focus on the outcome of $\mathcal{A}$ at a single site. Assuming M is sound, and noting the result $M' = \mathcal{A}(M)$, $\mathcal{A}$ must satisfy these requirements:

– $\mathcal{A}$ extends its input: $M \subseteq M'$.
– $\mathcal{A}$ may not add actions: $K' = K$.
– $\mathcal{A}$ may add constraints, which are restricted to decisions:

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

– M' is sound.
– M' is stable: $\mathit{Stable}(M') = K$.

$\mathcal{A}$ could be any algorithm satisfying the requirements.

One possible algorithm, $\mathcal{A}_{\text{Conservative}(<)}$, first orders actions, then kills actions for which the order is unsafe. It proceeds as follows (see Figure 1). Let $<$ be a total order of actions and M a sound multilog. The algorithm decides one action at time,

varying over all actions, left to right; call the current action β. Consider actions α and γ such that $\alpha < \beta < \gamma$: α has already been decided, and γ has not. If $\beta \nparallel \gamma$, then serialise them in schedule order. If $\beta \rightarrow \alpha$, and α is guaranteed, kill β, because the schedule and the constraint are incompatible. If $\gamma \lhd \beta$, conservatively kill β, because it is not known whether γ can be guaranteed. By definition, if $\alpha \lhd \beta$ and α is dead, then β is dead. If β is not dead by any of the above rules, then decide β guaranteed (by adding $\beta \lhd \text{INIT}$ to the multilog). The resulting $\Sigma(\mathcal{A}_{\text{Conservative}(<)}(M))$ contains a unique schedule.

It should be clear that this approach is safe but tends to kill actions unnecessarily.

The Bayou system [20] applies $\mathcal{A}_{\text{Conservative}(<)}$, where $<$ 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 (LWW) approach [7], 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).

The decisions computed by the above systems are mostly arbitrary. A better way would be to minimise aborts, or to follow user preferences, or both. This was the approach of the IceCube system [15]. $\mathcal{A}_{\text{IceCube}}$ is an optimization algorithm that minimises 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. The system suggests a number of possible decisions to the user, who states his preference.

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

To decentralise decision, one approach might be to determine a global total order $<$, using a decentralised consensus algorithm such as Paxos [11], and apply $\mathcal{A}_{\text{Conservative}(<)}$. As above, this order is arbitrary and $\mathcal{A}_{\text{Conservative}(<)}$ tends to kill unnecessary. Instead, our algorithm allows each site to propose decisions that minimises aborts and follows local client preferences, and to reach consensus on these proposals in a decentralised manner. This is the subject of the rest of this paper.

4 Client Operation

We now begin the discussion of our algorithm. We start with a specification of client behaviour.

4.1 Client Behaviour and Client Interaction

An application performs tentative operations by submitting actions and constraints to its local site-multilog; they will eventually propagate to all sites.

We abstract application semantics by postulating that clients have access to a sound multilog containing all the semantic constraints: $\mathcal{M} = (A, \rightarrow_{\mathcal{M}}, \lhd_{\mathcal{M}}, \nparallel_{\mathcal{M}})$. For an example $\mathcal{M}$, see Section 6.4.

Algorithm 1. *ClientActionsConstraints*(L)

Require: $L \subseteq A$

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

As the client submits actions L to the site-multilog, function *ClientActionsConstraints* (Algorithm 1) adds constraints with respect to actions that the site already knows.[5]

To illustrate, consider Alice and Bob working together. Alice uses their shared calendar at Site 1, and Bob at Site 2. Planning a meeting with Bob in Paris, Alice submits two actions: $\alpha =$"Buy train ticket to Paris next Monday at 10:00" and $\beta =$"Attend meeting". As β depends causally on α, $\mathcal{M}$ contains $\alpha \rightarrow_{\mathcal{M}} \beta \wedge \alpha \lhd_{\mathcal{M}} \beta$. Alice calls *ClientActionsConstraints*$(\{\alpha\})$ to add action α to site-multilog M_1, and, some time later, similarly for β. At this point, Algorithm 1 adds the constraints $\alpha \rightarrow \beta$ and $\alpha \lhd \beta$ taken from $\mathcal{M}$.

4.2 Multilog Propagation

When a client adds new actions L into a site-multilog, L and the constraints computed by *ClientActionsConstraints*, form a multilog that is sent to remote sites. Upon reception, receivers merge this multilog into their own site-multilog. By this so-called epidemic communication [3], every site eventually receive all actions and constraints submitted at any site.

When Site i receives a multilog M, it executes function *ReceiveAndCompare* (Algorithm 2), which first merges what it received into the local site-multilog. Then, if any conflicts exist between previously-known actions and the received ones, it adds the corresponding constraints to the site-multilog.[6]

Let us return to Alice and Bob. Suppose that Bob now adds action γ, meaning "Cancel the meeting," to M_2. Action γ is antagonistic with action β; hence, $\beta \rightarrow_{\mathcal{M}} \gamma \wedge \gamma \rightarrow_{\mathcal{M}} \beta$. Some time later, Site 2 sends its site-multilog to Site 1; when Site 1 receives it, it runs Algorithm 2, notices the antagonism, and adds constraint $\beta \rightarrow \gamma \wedge \gamma \rightarrow \beta$ to M_1. Thereafter, site-schedules at Site 1 may include either β or γ, but not both.

[5] In the pseudo-code, we leave the current time t implicit. A double-slash and sans-serif font indicates a comment, as in // This is a comment.

[6] *ClientActionsConstraints* provides constraints between successive actions submitted at the same site. These consist typically of dependence and atomicity constraints. In contrast, *ReceiveAndCompare* computes constraints between independently-submitted actions.

Algorithm 2. *ReceiveAndCompare*(M)

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

5 A Decentralised Commitment Protocol

Epidemic communication ensures that all site-multilogs eventually receive all information, but site-schedules might still differ between sites.

For instance, let us return to Alice and Bob. Assuming users add no more actions, eventually all site-multilogs become $(\{\text{INIT}, \alpha, \beta, \gamma\}, \{\alpha \rightarrow \beta, \beta \rightarrow \gamma, \gamma \rightarrow \beta\}, \{\alpha \lhd \beta\}, \varnothing)$. In this state, actions remain tentative; at time t, Site 1 might execute $S_1(t) = \text{INIT}; \alpha; \beta$, Site 2 $S_2(t) = \text{INIT}; \alpha; \gamma$, and just INIT at $t+1$. A commitment protocol ensures that α, β and γ eventually stabilise, and that both Alice and Bob learn the same outcome. For instance, the protocol might add $\beta \lhd \text{INIT}$ to M_1, which guarantees β, thereby both guaranteeing α and killing γ. α, β and γ are now decided and stable at Site 1. M_1 eventually propagates to other sites; and inevitably, all site-schedules eventually start with $\text{INIT}; \alpha; \beta$, and γ is dead everywhere.

5.1 Overview

Our key insight is that eventual consistency is equivalent to the property that the site-multilogs of all sites share a common *well-formed prefix* (defined hereafter) of stable actions, which grows to include every action eventually. Commitment serves to agree on an extension of this prefix. As clients continue to make optimistic 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; a proposal being a tentative well-formed prefix of its site-multilog. Sites agree via a decentralised election. This works even if $\mathcal{A}$ is non-deterministic, or if sites use different $\mathcal{A}$ algorithms. We recommend IceCube [15] but any algorithm satisfying the requirements of Section 3 is suitable.

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

We distinguish two roles at each site, proposers and acceptors. Each proposer has a fixed *weight*, such that $\sum_{k \in \mathcal{J}} weight_k = 1$. 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. To simplify exposition, weights are distributed ahead of time and do not change; it is relatively straightforward to extend the current algorithm, allowing weights to vary between successive elections.

An acceptor at some site computes the outcome of an election, and inserts the corresponding decision constraints into the local site-multilog.

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 $proposals_i[k].ts$. Timestamping ensures the liveness of the election process despite since links between nodes are not necessarily FIFO.

Algorithm 3. Algorithm at Site i

Declare: M_i: local site-multilog
Declare: $proposals_i[n]$: array of proposals, indexed by site; a proposal is a multilog

```
 1: M_i := ({INIT}, ∅, ∅, ∅)
 2: proposals_i := [(({INIT}, ∅, ∅, ∅), 0), ..., (({INIT}, ∅, ∅, ∅), 0)]
 3: loop // Epidemic transmission
 4:       Choose j ≠ i;
 5:       Send copy of M_i and proposals_i to j
 6: ||
 7: loop // Epidemic reception
 8:       Receive multilog M and proposals P from some site j ≠ i
 9:       ReceiveAndCompare(M) // Compute conflict constraints
10:       MergeProposals(P)
11: ||
12: loop // Client submits
13:       Choose L ⊆ A
14:       ClientActionsConstraints(L) // Submit actions, compute local constraints
15: ||
16: loop // Compute current local state
17:       Choose S_i ∈ Σ(M_i)
18:       Execute S_i
19: ||
20: loop // Proposer
21:       UpdateProposal // Suppress redundant parts
22:       proposals_i[i] := A(M_i ∪ proposals_i[i]) // New proposal, keeping previous
23:       Increment proposals_i[i].ts
24: ||
25: loop // Acceptor
26:       Elect
```

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.

5.2 Epidemic Communication

The first two threads (lines 3–10) exchange multilogs and proposals between sites. Function *ReceiveAndCompare* (defined in Algorithm 2, Section 4.2) compares actions

Algorithm 4. *UpdateProposal*

1: Let $P = (K_P, \rightarrow_P, \lhd_P, \not\parallel_P) = proposals_i[i]$
2: $K_P := K_P \setminus Decided(M_i)$
3: $\rightarrow_P := \rightarrow_P \cap K_P \times K_P$
4: $\lhd_P := \lhd_P \cap K_P \times K_P$
5: $\not\parallel_P := \varnothing$
6: $proposals_i[i] := P$

newly received to already-known ones, in order to compute conflict constraints. In Algorithm 6 a receiver updates its own set of proposals with any more recent ones.

5.3 Client, Local State, Proposer

The third thread (lines 12–14) 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 fourth 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 second thread (line 9).

The fourth thread (lines 16–18) computes the current tentative state by executing some sound site-schedule.

The fifth thread (20–23) 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. Passing argument $M_i \cup proposals_i[i]$ to $\mathcal{A}$ 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).

The last thread is described in the next section.

5.4 Election

The last thread (25–26) 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.

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

Definition 2. *Well-formed prefix. Let $M = (K, \rightarrow, \lhd, \not\parallel)$ and $M' = (K', \rightarrow', \lhd', \not\parallel')$ 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.*

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

A well-formed prefix is a semantically-meaningful unit of proposal. For instance, if a $\rightarrow$ or $\lhd$ cycle is present in M , every well-formed prefix 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; or more embarassingly, that sites know only parts of a cycle. Therefore we also require the following property:

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

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. For instance, in the database example, a sufficient condition for transaction T to be eligible at Site i is that all transactions submitted (at any site) concurrently with T are also known at Site i. Indeed, all such transactions have gone through either *ClientActionsConstraints* or *ReceiveAndCompare*; hence according to Table 1, T is eligible.

Computation of votes. We define a vote as a pair $(weight, siteId)$. 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: $compatible(M,M') \overset{\text{def}}{=} \Sigma(M \cup M') \neq \varnothing$. The votes of compatible candidates add up; $tally(X)$ computes the total vote for some candidate X:

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

An election pits some candidate against *comparable* candidates from all other sites. Two multilogs are comparable if they contain the same set of actions: $comparable(M,M') \overset{\text{def}}{=} K = K'$. The direct opponents of candidate X in some election are comparable candidates that X does not prefix:

$$opponents(X) \overset{\text{def}}{=} \{B | \exists k : B \overset{wf}{\sqsubset} proposals_i[k] \quad \wedge comparable(B,X) \wedge X \overset{wf}{\not\sqsubset} B)\}$$

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:

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

Algorithm 5. *Elect*

1: Let X be a multilog such that:

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

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

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: $proposals_i[k].ts := P[k].ts$

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 are received, cotally tends towards 0, therefore some candidate is eventually elected. We merge the winner into the site-multilog.

5.5 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 $(\{\text{INIT}, \alpha, \beta\}, \{\alpha \to \beta, \alpha \to \gamma, \gamma \to \alpha\}, \{\alpha \lhd \beta\}, \varnothing)$. Now Alice (Site 1) proposes to guarantee α and β, and to kill γ: $proposals_1[1] = M_1 \cup \{\beta \lhd \text{INIT}\}$. In the 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 $proposals_1[1]$; now suppose that X is eligible as all sites have voted on K_X; $tally(X) = \frac{2}{3}$ 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.

6 Discussion

6.1 Safety Proof Outline

Section 1 states our safety property, the conjunction of mergeability and local soundness. Clearly Algorithm 3 satisfies local soundness; see lines 16–18. We now outline a proof of mergeability.

We say that candidate X is *elected* in a run r at time t, if some acceptor i executes Algorithm 5 in r at t, and elects a candidate Y such that $X \overset{wf}{\sqsubset} Y$. Given a run r of Algorithm 3, we note $Elected(r,t)$ the set of candidates elected in r up to time t (inclusive), 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).

Suppose, by contradiction, that during run r, this set is unsound. As $\mathcal{M}$is sound, by $\mathcal{A}$ candidates are sound. Consequently there must exist an unsound set of candidates $C \subseteq Elected(r)$. Let us now consider the following property:

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

As candidates are eligible, there must exist two candidates X and X' in C such that: (i) X and X' are non-compatible, and (ii) X and X' are minimal.

We define the following notation. Let i (resp. i') be the acceptor that elects X (resp. X') in r. t is the time where i elects X in r (resp. t' for X' on i'). For a proposer k, t_k (resp. t'_k) is the time at which it sent $proposals_i[k](t)$ to i (resp. $proposals_{i'}[k](t')$ to i'). Q (resp. Q') is the set of proposers that vote for X at t on i (resp. for X' at t' on i'); formally $Q = \{k | X \overset{wf}{\sqsubset} proposals_i[k](t)\}$ and $Q' = \{k | X' \overset{wf}{\sqsubset} proposals_{i'}[k](t')\}$.

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 i' elects X' at t', at that time on Site i':

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

Equation 1 defines 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 $X \overset{wf}{\sqsubset} proposals_{i'}[k](t')$.

If now $t_k > t'_k$, then as $tally(X')$, $opponents(X')$ and $cotally(X')$ define a partition of $\mathcal{J}$, either:

1. k has not yet voted on $K_{X'}$ at t' on i' and its weight is counted in $cotally(X')$.
2. Or, if its vote already includes $K_{X'}$, it is counted in $opponents(X')$ as X is the first candidate non-compatible with X' elected in r, $X \overset{wf}{\sqsubset} proposals_i[k](t)$, and $\neg\, compatible(X,X')$.

From these reasonnings (if $t_k < t'_k$ and if $t'_k < t_k$), and Equation 1, we derive:

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

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

1. $X' \overset{wf}{\sqsubset} proposals_i[k](t)$
2. or k has not yet voted on $X.K$ on i at t.

The reasoning here is similar to $k \in Q$: we use the minimality of X and X', the fact that they are non-compatible, and that X is the first candidate non-compatible with X' elected in r.

From the above, it follows that:

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

Now, combining equations 2 and 3, we conclude that, at site i at time t:

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

X cannot be elected on i at t. Contradiction.

6.2 Message Cost

Interestingly, the message cost of our protocol varies with application semantics, along two dimensions.

First, the *degree of semantic complexity*, i.e., the complexity of the client constraint graph $\mathcal{M}$, influences the number of votes required. To illustrate, consider an application where all actions are mutually independent, i.e., $\mathcal{M}$ contains no constraints. Then, all actions commute with one another, and no action never needs to be killed. Every candidate is trivially eligible, and trivially compatible with all other candidates.

Second, call *degree of optimism d* the size of a batch, i.e., the number of actions that a site may execute tentatively before requiring commitment. This measures both that replicas relax consistency and that clients propose to the same replica, concurrent commutative actions. It takes a chain of $\frac{n}{2}$ messages to construct a majority. A candidates may contain up to d actions. Therefore, the amortised message cost to commit an action is $\frac{n}{2} \times \frac{1}{d}$.

A more detailed evaluation of message cost is left for future work.

6.3 Implementation Considerations

Our pseudo-code was written for clarity, not efficiency. Many optimisations are possible. For instance, a site i does not need to send the whole $proposals_i$ [i]. When sending to j, it suffices to send the difference $proposals_i[i] \setminus proposals_i[j]$.

Conceptually, a multilog grows without bound. However, a stable action, and all its constraints, can safely be deleted.

Table 1. $\mathcal{M}_{\text{SER-DB-after}}$: Constraints for a serialisable database that transmits after-values

	$T \prec T'$	$T \parallel T'$	$T' \prec T$
$RS(T) \cap WS(T') \neq \varnothing$	$T \rightarrow T'$	$T \rightarrow T'$	$T' \rightarrow T \wedge T' \lhd T$
$WS(T) \cap WS(T') \neq \varnothing$	$T \rightarrow T'$	$T \not\parallel T'$	$T' \rightarrow T$

Conceptually, our algorithm executes all actions everywhere. A practical implementation only needs to achieve an equivalent state; in particular actions that do not have side-effects do not have to be replayed. For instance, in a database application, read operations do not to be replayed.[7]

6.4 Example Application

We illustrate the application of our algorithm to a replicated database. The semantic constraints between two transactions depend on several factors: (i) Whether the transactions are related by happens-before or are concurrent. (ii) Whether their read- and write-sets intersect or not. (iii) What consistency criterion is being enforced (for instance, constraints differ between serializability and snapshot isolation [2]). (iv) How, after executing a transaction on some initial site, the system replicates its effects at a remote site: by replaying the transaction, or by applying the after-values computed at the initial site.

Table 1 exhibits semantic constraints between transactions, where (a) the system replicates a transaction by writing its after-values, and (b) transactions are strictly serialisable.[8] Supporting a different semantics, e.g., (a') replaying actions, or (b') SI, requires only some small changes to the table.

7 Related Work

In previous OR systems, commitment was often either centralised at a primary site [15,20] or oblivious of semantics [7,17]. It is very difficult to combine decentralisation with semantics.

Our election algorithm is inspired by Keleher's Deno system [8], 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 [10]; all actions are assumed be mutually non-commuting. Happens-before captures potential causality; however an event may happen-before another even

[7] Formally, we need to generalise the equivalence relation between schedules, which currently is based only on $\not\parallel$ [18]. The definition of consistency now becomes that every pair of sites eventually converges to schedules that are equivalent according to the new relation.

[8] $T \prec T'$ denotes T happens-before T [10]. $T \parallel T'$ denotes concurrency, i.e., neither $T \prec T'$, nor $T' \prec T$. $RS(T)$ and $WS(T)$ denote T's read set and write set respectively.

if they are not truly dependent. This paper further generalizes VVWV by considering semantic constraints.

Holliday et al. depict a family of epidemic algorithms to ensure serializability in replicated datbase systems [5]. The three algorithms consider that concurrent conflicting transactions are antagonistic. Two of them abort concurrent conflicting transactions, and the last one (quorum-based) can only commit one transactions among a set of concurrent conflicting ones. Our algorithm consider that concurrent conflicting transactions are not necessarily antagonistic, it tries to optimize the number of committed transactions, computing a best-effort proposal , and electing them with plurality.

ESDS [4] 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 [20] 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 [9] 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 [11] computes a total order. Such total order may be used to implement *state-machine replication* [10], 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 combine semantic constraints with a total order, but this approach is clearly sub-optimal. Howover, Paxos remains live even if $f < \frac{n}{2}$ sites crash forever, whereas the other systems described here (including ours) block if a site crashes forever. We assume that a site stores its multilogs and its proposals in persistent memory, and that after a crash it with its identity and persistent store intact. This is a fairly reasonable assumption in a well-managed cooperative system. (For instance, each site might actually be implemented as a cluster on a LAN, with redundant storage, and strong consistency internally.)

Generalized Paxos [12] and Generic Broadcast [14] take commutativity relations into account and compute a partial order. They do not consider any other semantic relations. Both Generalized Paxos [12] 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.

8 Conclusion and Future Work

The focus of our study is cooperative 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, and live in the absence of crashes. It uses voting to avoid any centralization bottleneck, and to ensure that the result is similar to local proposals. It uses plurality voting to make progress even when an election does not reach a majority.

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 base its decisions on a large batch of tentative actions, deciding less frequently. 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.

Another future direction is partial replication. In such a system, a site receives only the actions relative to the objects it replicates (and their constraints). A site votes only on the actions it knows. Because constraints might relate actions known only by distinct sites, these sites must agree together; however we expect that global agreement is rarely necessary. By exploiting knowledge of semantic constraints, we hope to limit the scope of a commitment protocol to small-scale agreements, instead of a global consensus.

References

1. Barreto, J., Ferreira, P.: An efficient and fault-tolerant update commitment protocol for weakly connected replicas. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 1059–1068. Springer, Heidelberg (2005)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
3. Demers, A.J., Greene, D.H., Hauser, C., Irish, W., Larson, J.: Epidemic algorithms for replicated database maintenance. In: Symp. on Principles of Dist. Comp. (PODC), pp. 1–12, Vancouver, BC, Canada (August 1987). Also appears Op. Sys. Review 22(1), 8–32 (1988)
4. Fekete, A., Gupta, D., Luchangco, V., Lynch, N., Shvartsman, A.: Eventually-serializable data services. Theoretical Computer Science 220(Special issue on Distributed Algorithms), 113–156 (1999)
5. Holliday, J., Steinke, R., Agrawal, D., Abbadi, A.E.: Epidemic algorithms for replicated databases. IEEE Transactions on Knowledge and Data Engineering 15(5), 1218–1238 (2003)
6. Ignat, C.-L., Norrie, M.C.: Draw-Together: Graphical editor for collaborative drawing. In: CSCW. Int. Conf. on Computer-Supported Cooperative Work, Banff, Alberta, Canada, pp. 269–278 (November 2006)
7. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (January 1976)
8. Keleher, P.J.: Decentralized replicated-object protocols. In: Symp. on Principles of Dist. Comp. (PODC), Atlanta, GA, USA, pp. 143–151. ACM Press, New York (1999)
9. Kermarrec, A.-M., Rowstron, A., Shapiro, M., Druschel, P.: The IceCube approach to the reconciliation of divergent replicas. In: Symp. on Principles of Dist. Comp. (PODC), Newport, RI, USA ACM SIGACT-SIGOPS, ACM Press, New York (2001)

10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
11. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
12. Lamport, L.: Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research (March 2005)
13. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. J. of Dist. and Parallel Databases and Technology 14(1), 71–98 (2003)
14. Pedone, F., Schiper, A.: Handling message semantics with generic broadcast protocols. Distributed Computing Journal 15(2), 97–107 (2002)
15. Preguiça, N., Shapiro, M., Matheson, C.: Semantics-based reconciliation for collaborative and mobile environments. In: Meersman, R., Tari, Z., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 38–55. Springer, Heidelberg (2003)
16. Ratner, D., Reiher, P., Popek, G.: Roam: A scalable replication system for mobile computing. In: Int. W. on Database & Expert Systems Apps (DEXA), pp. 96–104. IEEE Comp. Society, Los Alamitos, CA, USA (1999)
17. Saito, Y., Shapiro, M.: Optimistic replication. Computing Surveys 37(1), 42–81 (2005)
18. Shapiro, M., Bhargavan, K.: The Actions-Constraints approach to replication: Definitions and proofs. Technical Report MSR-TR-2004-14, Microsoft Research (March 2004)
19. Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. Trans. on Comp.-Human Interaction 5(1), 63–108 (1998)
20. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: 15th Symp. on Op. Sys. Principles (SOSP), Copper Mountain, CO, USA, pp. 172–182 ACM SIGACT-SIGOPS, ACM Press, New York (1995)