Eventual consistency and commitment in semantically-rich systems

Pierre Sutra
Université Paris VI Pierre et Marie Curie, Paris, and INRIA Rocquencourt, France

João Barreto
INESC-ID and Instituto Superior Técnico, Lisbon, Portugal

Marc Shapiro
INRIA Rocquencourt and LIP6, Paris, France

6 May 2006

Abstract

We study data replication and eventual consistency in an optimistic system that allows temporary divergence. We present a decentralised quorum commitment protocol that ensures eventual consistency in the presence of non-byzantine faults. It supports a rich repertoire of application semantics, viz., executions are sound with respect to dependence and atomicity, non-commuting and conflicting operations are entirely decided, and commuting operations are not necessarily totally ordered. We describe the protocol in detail and prove it safe.

Contact author: Pierre Sutra, LIP6, 8 rue du Capitaine Scott, 75015 Paris, France. mailto:pierre.sutra@lip6.fr. Telephone +33 1 4427 4363. Fax: +33 1 4427 7495.

This paper is submitted for regular or brief presentation. It contains 4410 words, not counting the bibliography nor the appendices. It is eligible for the "Best Student Paper" award. Both Pierre Sutra and João Barreto are full-time students.

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) [14]. OR decouples data access from network access: it allows a processor to access a local replica without synchronising. A processor make progress, executing uncommitted actions, even while others are slow or unavailable. Local execution is tentative and actions may roll back later. An OR system propagates updates lazily, and ensures consistency by a global *a posteriori* agreement on the set and order of actions, a process that we call *commitment*.[1]

We are especially interested in applications with rich semantics. For instance, atomicity, dependence and commutativity are central to database correctness. Also, cooperative work scenarios experience actions that conflict according to the application semantics [1, 13, 17].[2]

Many previous studies of replication use the state-machine approach [7], requiring a global, total, arbitrary order for executing all actions. This approach is inadequate when semantics are important. For instance, not all actions may execute, as the system enforces mutual exclusion between conflicting actions. An action that depends on another must not execute until its dependent does. An atomicity relation implies that the actions must either all execute, or none. Moreover, a total order is stronger than necessary if some actions commute.

Our approach has been to reify semantic relations as constraints that restrict the legal execution schedules. Constraints may express non-commutativity, order, implication, conflict, dependence, or atomicity. The system has an obligation to resolve conflicts and non-commutativity relations, to satisfy the constraints, and to eventually execute equivalent schedules at all sites.

This paper makes the following contributions:

- We study scheduling and commitment within a formal framework for systems with rich semantics. Schedules must be *sound* with respect to semantic constraints. Liveness imposes to *decide* with respect to conflict and non-commutativity. Safety imposes that decisions made by different agents are compatible with one another.

[1] The more traditional pessimistic replication, where propagation and commitment occur *a priori*, is a special case of OR.

[2] Conflict and non-commutativity are two different things; we depart from the database tradition of using the two words as synonyms.

- We propose the first decentralised commitment protocol for an OR environment with rich semantics. It satisfies the correctness conditions. Different sites vote asynchronously and continuously. It uses quorums to make progress even no proposal has a majority.

The outline of the paper is the following. Section 2 introduces our system model and our vocabulary. We give our quorum-based commitment protocol in Section 3. We compare with related work in Section 4. Section 5 concludes by recapitulating and discussing our results and with suggestions of future work. The bibliography is followed by Appendix A, containing a subsidiary algorithm.

2 System model

We consider an asynchronous distributed system of n distributed *sites* $i, j, \ldots$. A site contains two processes: an application process called the *client*, and a single *consistency agent* (or just *agent* hereafter), the focus of this paper. Sites are fail-stop. They communicate through reliable FIFO channels. Disconnection is a normal occurence: a site may be unable to communicate for extended periods. We assume a global clock that ticks at every step of any process, but processes do not have access to it.

Shared data is replicated across all sites. We identify the state of a replica with the *schedule* of actions executed at its site since the initial state INIT.

We use the Action-Constraint Framework (ACF) to model our system [15, 16]. Due to space limitations, our outline of ACF will be very terse.

2.1 Multilogs and schedules

Our central data structure is the *multilog*. Let A be the infinite set of possible *actions*, noted $\alpha, \beta, \ldots$. A multilog is a quadruple $M = (K, \rightarrow, \lhd, \between)$,[3] where $K \subseteq A$, and $\rightarrow$, $\lhd$ and $\between$ are sets of *constraints* (relations over $A \times A$). $\alpha \rightarrow \beta$ (read "α Before β") imposes an ordering on schedules, $\alpha \lhd \beta$ ("β MustHave α") is implication over schedules, and $\alpha \between \beta$ means "α is NonCommuting with β." They may combined: a Before cycle (e.g., $\alpha \rightarrow \beta \wedge \beta \rightarrow \alpha$) represents a *conflict*; the conjunction $\alpha \rightarrow \beta \wedge \alpha \lhd \beta$ means β *depends* causally on α;[4] a MustHave cycle such as

[3] Multilog union, inclusion and difference are defined as component-wise union, inclusion and difference, respectively.

[4] The classical happens-before relation [7] can be encoded as dependence, but this is often stronger than necessary.

$\alpha \lhd \beta \wedge \beta \lhd \alpha$ is *atomicity*.[5]

A schedule S (a sequence of actions ordered by $<_S$) is *sound* with respect to a multilog M iff:

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

The set of schedules that are sound with respect to M is noted $\Sigma(M)$. $\Sigma(M)$ grows as K grows, and shrinks as $\rightarrow$ or $\lhd$ grow.

Two schedules, S and S', are equivalent (noted $S \equiv S'$) if they differ only by swapping adjacent commuting actions: $S \equiv S' \stackrel{\text{def}}{=} (\alpha \in S \Leftrightarrow \alpha \in S') \wedge (\forall \alpha, \beta \in S,\ \alpha \nvdash \beta \Rightarrow (\alpha <_S \beta \Leftrightarrow \alpha <_{S'} \beta))$.

The following subsets of actions are of particular interest. They grow as M grows:

- *Guaranteed* actions appear in every schedule of $\Sigma(M)$. $Guar(M) \stackrel{\text{def}}{=} \{\text{INIT}\} \cup \{\alpha \in A | \exists \beta \in Guar(M),\ \alpha \lhd \beta\}$
- *Dead* actions never appear in a schedule of $\Sigma(M)$. $Dead(M) \stackrel{\text{def}}{=} \{\alpha \in A | \exists \beta_1, \ldots, \beta_m \in Guar(M),\ \alpha \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha\} \cup \{\alpha \in A | \exists \beta \in Dead(M),\ \beta \lhd \alpha\}$
- *Serialised* actions are ordered with respect to all non-commuting actions that are not dead. $Serialised(M) \stackrel{\text{def}}{=} \{\alpha \in A | \forall \beta \in A,\ \alpha \nvdash \beta \Rightarrow \alpha \rightarrow \beta \vee \beta \rightarrow \alpha \vee \beta \in Dead(M)\}$
- *Decided* actions are either dead, or both guaranteed and serialised. $Decided(M) \stackrel{\text{def}}{=} Dead(M) \cup (Guar(M) \cap Serialised(M))$.
- *Stable* actions are decided, and all actions that precede them by Before are themselves stable: $Stable(M) \stackrel{\text{def}}{=} Dead(M) \cup \{\alpha \in Guar(M) | \forall \beta \in A,\ \beta \rightarrow \alpha \Rightarrow \beta \in Stable(M)\}$.

To force an action α to be dead (resp. guaranteed) it suffices to add the constraint $\alpha \rightarrow \alpha$ (resp. $\alpha \lhd \text{INIT}$) to M.

Multilog M is said *sound* iff $\Sigma(M) \neq \emptyset$, or equivalently, iff $Dead(M) \cap Guar(M) = \emptyset$.

[5] In this paper we do not consider the isolation property of transactions, which is future work.

Pre-condition	Transition
true	$K_i := K_i \cup \{\alpha\}$
$\beta \notin Q$	$\rightarrow_i := \rightarrow_i \cup \{\alpha \rightarrow \beta\}$
$\beta \notin Q$	$\lhd_i := \lhd_i \cup \{\alpha \lhd \beta\}$
$\alpha, \beta \notin Q$	$\nvdash_i := \nvdash_i \cup \{\alpha \nvdash \beta\}$

(a) Client transitions

Precondition	Transition
$\alpha \rightarrow_j \beta$	$\rightarrow_i := \rightarrow_i \cup \{\alpha \rightarrow \beta\}$
$\alpha \lhd_j \beta$	$\lhd_i := \lhd_i \cup \{\alpha \lhd \beta\}$
$\alpha \nvdash_j \beta$	$\nvdash_i := \nvdash_i \cup \{\alpha \nvdash \beta\}$
$\beta \in K_j \wedge (\alpha \rightarrow_j \beta \Rightarrow \alpha \rightarrow_i \beta)$ $\wedge (\alpha \lhd_j \beta \Rightarrow \alpha \lhd_i \beta)$ $\wedge (\alpha \nvdash_j \beta \Rightarrow \alpha \nvdash_i \beta)$	$K_i := K_i \cup \{\beta\}$

(b) Communication transitions: receiving from site j

Precondition	Transition	Meaning
$\alpha \in Q \cap K_i \wedge \mathit{Mergeable}$	$\rightarrow_i := \rightarrow_i \cup \{\alpha \rightarrow \alpha\}$	Make dead
$\alpha, \beta \in Q \cap K_i \wedge \mathit{Mergeable}$	$\rightarrow_i := \rightarrow_i \cup \{\alpha \rightarrow \beta\}$	Serialise
$\alpha \in Q \cap K_i \wedge \mathit{Mergeable}$	$\lhd_i := \lhd_i \cup \{\alpha \lhd \text{INIT}\}$	Make guaranteed

(c) Agent transitions

Table 1: Legal transitions for site-multilog M_i, at site i

2.2 Eventual Consistency

Each site i has a distinguished *site-multilog* $M_i(t) = (K_i(t), \rightarrow_i (t), \lhd_i(t), \nvdash_i (t))$ that contains the local knowledge that i has of the state of the distributed system at time t. Initially, the site-multilog is $M_i(0) = (\text{INIT}, \emptyset, \emptyset, \emptyset)$. It grows over time, as explained shortly. The local state is given by the current *site-schedule*, $S_i(t) \in \Sigma(M_i(t))$.[6]

Our formal definition of consistency is the following; it has been proved equivalent to Eventual Consistency [15]:

Definition (Consistency)**.** *A system is consistent iff it satisfies the following correctness conditions:*

- *Local soundness:* $\forall i, t,\ S_i(t) \in \Sigma(M_i(t))$
- *Mergeability:* $\forall i, i', i'', \ldots, t, t', t'', \ldots,\ \Sigma(M_i(t) \cup M_{i'}(t') \cup M_{i''}(t'') \cup \ldots) \neq \emptyset$
- *Eventual decision:* $\forall i, j, t,\ \forall \alpha \in K_i(t),\ \exists t',\ \alpha \in \mathit{Decided}(M_j(t'))$

Eventual decision ensures that the system makes progress, local soundness that every state is semantically correct, and mergeability that sites do not make incompatible decisions.

[6] If $|\Sigma(M_i(t))| > 1$ the choice of $S_i(t)$ is arbitrary. If $S_i(t-1)$ is not a prefix of $S_i(t)$, this represents a roll-back.

2.3 Evolution of the site-multilog

The site-multilog grows over time, by the following types of events:

1. The client puts actions into K_i reflect user activity. The actions may now be executed tentatively, and are candidates for commitment.
2. The client puts constraints into M_i to reflect application semantics. Some constraints are typically added immediately; for instance when an action depends on an earlier one, or when several actions form an atomic grouping. Others cannot be known upfront, and the client can add them later only, if and when certain other actions appear in K_i. This is the case for dependence by later actions, and for conflict and non-commutativity constraints.
3. The site receives site-multilogs from other sites (anti-entropy). In the absence of faults, we assume *eventual propagation*, i.e., $\forall i, j, t,\ \exists t',\ M_i(t) \subseteq M_j(t')$.
4. The commitment protocol adds constraints to decide actions.

The goal of an agent is to decide actions, while maintaining mergeability. However, if clients had complete freedom, mergeability violations might still occur. Therefore, we specify transition rules in Table 1 with an explicit synchronisation between clients and agents, the *onus* set Q of actions for which agents assume responsibility. A client may add a constraint to the site-multilog only if it satisfies certain pre-conditions with respect to Q; for instance adding $\alpha \rightarrow \beta$ is not allowed if $\beta \in Q$. Conversely, an agent may decide α only if $\alpha \in Q$.

Once in Q, an action remains in Q. Initially, Q contains only INIT. Hereafter we assume that site i maintains a safe approximation Q_i of Q; we discuss a possible implementation in Appendix A.

Table 1 lists the legal transitions for clients, communication, and agents respectively. To simplify notation we leave out the explict time parameter and use assignments.

A client is free to add new actions, and to add dependence on earlier actions that are already in Q. The communication rules support asynchronous, epidemic communication, while ensuring that an agent receives the full set of an action's constraints before deciding on that action.

In the agent transition rules, *Mergeable* means "new state satisfies mergeability." This is the main difficulty in designing a commitment protocol.

3 A quorum protocol for eventual consistency

To satisfy Consistency (Section 2.2), a commitment protocol aims to decide every action, i.e., to make it either dead, or both guaranteed and serialised. To this effect, an agent i may propose transitions from Table 1(c) if they are sound according to its local knowledge, the site-multilog M_i.

Agents together vote on each proposal. A proposal wins and is added to the site-multilog if approved by a majority or a plurality of agents. As no agent may vote for incompatible transitions, this ensures that M_i remains sound. This, together with the client and communication transition rules, ensures mergeability.

Rather than a single proposal at a time, agents vote continously on all available proposals. Any stable prefix of proposals that gathers a majority or a plurality is elected.

3.1 Proposing

We postulate a local algorithm $\mathcal{A}(M,R)$ for proposing decisions given a multilog M, and an onus set R. $\mathcal{A}$ may decide only actions in R. The initial proposal of site i will be $\mathcal{A}(M_i,Q_i)$.

The formal specification of $\mathcal{A}$ is the following. Let $M=(K,\rightarrow,\lhd,\nVdash)$ be a sound multilog ($\Sigma(M)\neq\emptyset$) and R a set of actions, such that $\mathit{Decided}(M)\subseteq R$. $\mathcal{A}(M,R)$ computes a multilog $M'=(K,\rightarrow',\lhd',\nVdash')$ such that:

1. M' is sound: $\Sigma(M')\neq\emptyset$.
2. M' extends M: $\rightarrow\subseteq\rightarrow'\wedge\lhd\subseteq\lhd'$.
3. The decided actions are in R: $\mathit{Decided}(M')\subseteq R$.

Note that $\mathcal{A}$ is not assumed deterministic. Two agents with identical site-multilogs might still offer differing proposals.

A discussion of $\mathcal{A}$ is out of scope of this article. We note simply that $\mathcal{A}$ exists: for instance, making all actions dead trivially satisfies the specification. In the IceCube project [6, 12] we proposed an optimisation algorithm that minimises the number of dead actions in M'.

3.2 Variables and notation

Each agent has a fixed *weight* for its votes, such that $\sum_i weight_i = 1$. A proposal that gathers more than $1/2$ weight wins the election (majority); it also wins if it gathers more weight than any incompatible proposal (plurality). Two multilogs are compatible if their union is sound: $compatible(M, M') \stackrel{\text{def}}{=} \Sigma(M \cup M') \neq \emptyset$.

An agent implicitly votes for its own proposal. It also votes for other proposals that are compatible with its own. Each agent stores the set of proposals received so far from all sites in an array $proposals_i[n]$. Each entry is of type multilog and is the result of some execution of $\mathcal{A}$. Its initial value is $[(\text{INIT}, \emptyset, \emptyset, \emptyset), \ldots, (\text{INIT}, \emptyset, \emptyset, \emptyset)]$. Moreover each agent stores an array $counters_i[n]$ of sequence numbers, one for each proposal. Its initial value is $[0, \ldots, 0]$.

Given a multilog $M = (K, \rightarrow, \lhd, \Vdash)$, multilog $M' = (K', \rightarrow', \lhd', \Vdash')$ is a *stable prefix* of M iff:

$$M' \ll M \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} & K' = Stable(M') \\ \wedge & \forall \beta \in K',\ (\alpha \in K \wedge \alpha \rightarrow \beta) \Leftrightarrow (\alpha \in K' \wedge \alpha \rightarrow' \beta) \\ \wedge & \forall \beta \in K',\ (\alpha \in K \wedge \alpha \lhd \beta) \Leftrightarrow (\alpha \in K' \wedge \alpha \lhd' \beta) \\ \wedge & \forall (\alpha, \beta) \in \rightarrow',\ \alpha, \beta \in K' \\ \wedge & \forall (\alpha, \beta) \in \lhd',\ \alpha, \beta \in K' \end{array} \right.$$

Every action in M' is stable in both M and M', and every schedule in $\Sigma(M)$ has a prefix that is in $\Sigma(M')$.

In what follows, i represents an agent executing some algorithm, and j, k range over the set of all agents $\neq i$.

3.3 Algorithms

Each agent repeatedly does the following (Algorithm 1): compute a proposal, elect, and update information according to the outcome; concurrently, send and receive proposals to and from remote sites.

3.3.1 Proposing a multilog for election

$\mathcal{A}$ computes proposals. For every new proposal, we increment i's proposal sequence number. We give the pseudo-code as Algorithm 2.

Algorithm 1 Agent Algorithm at site i

```
1: loop
2:    Propose()
3:    Elect()
4:    UpdateProposal(proposals_i[i], M_i)
5: end loop
6: ||
7: loop
8:    Choose j ≠ i; send (proposals_i, counters_i) to j
9: end loop
10: ||
11: loop
12:    For some agent j ≠ i: ReceiveProposals(j, P, C)
13: end loop
```

Algorithm 2 $Propose()$

```
1: proposals_i[i] := A(proposals_i[i] ∪ M_i, Q_i)
2: increment counters_i[i].
```

3.3.2 Receiving proposals

When agent i receives a set of proposals from j (Algorithm 3), it updates its own set with any more recent ones. If a received proposal is compatible with its own, then it merges the two, in order to accelerate elections.

Algorithm 3 $ReceiveProposals(j, P, C)$

```
1: for all k do
2:    if counters_i[k] < C[k] then
3:       proposals_i[k] := P[k]
4:       if compatible(proposals_i[i], proposals_i[k]) then
5:          proposals_i[i] := proposals_i[i] ∪ proposals_i[k]
6:       end if
7:    end if
8: end for
```

3.3.3 Election

An election considers all possible stable prefixes of the proposals, as specified in Algorithm 4. Each agent independently is capable of determining if a stable prefix wins the election. It wins if it garners a majority, or if it has a greater weight than any incompatible prefix (a plurality). The plurality algorithm is specified later.

Function $tally_i(P)$ counts the total weight of agents that vote for some stable prefix P.

$$tally_i(P) \stackrel{\text{def}}{=} \sum_k (weight_k : P \ll proposals_i[k])$$

Algorithm 4 Electing a proposal: $Elect()$

choose P such that $(tally_i(P) > 1/2) \vee (Plurality(P))$
$M_i := M_i \cup P$

Algorithm 5 specifies plurality vote. In the case where there are several mutually incompatible proposals, none of which reaches a majority, we can still make progress by electing the one with the highest tally. We compare site identifiers to break ties.

We note $potential_i(P)$ the total weight of agents whose proposal is compatible with P, i.e., they could possibly vote for P:

$$potential_i(P) \stackrel{\text{def}}{=} \sum_k (weight_i[k] : compatible(P, proposals_i[k]))$$

Algorithm 5 $Plurality(P)$

if $\forall\, j \neq i,\ \neg compatible(P, proposals_i[j]) \vee P \ll proposals_i[j]$ **then**
  if $\forall\, j \neq i : \neg compatible(P, proposals_i[j]),$
    $\forall B \ll proposals_i[j] : \neg compatible(B, P),$
      $potential_i(B) < tally_i(P) \vee$
        $(potential_i(B) = tally_i(P)$
        $\wedge \inf(k : compatible(B, proposals_i[j])) < \inf(k : P \ll proposals_i[k]))$
  then
    return true
  end if
  return false
end if

The plurality algorithm enables elections to progress even when there is no majority, by computing the scores (tally vs. potential) of all incompatible proposals. The one with the highest score wins. However if some site is not communicating, then the computation might block.

3.3.4 Updating current proposal

Once an agent has made a proposal, he may not retract it until it either wins or loses the election. Once a proposal has lost, it is incompatible with the new site-multilog. Therefore it must be removed, but with care: only the parts that are unsound are removed; the other parts remain pending. In the update algorithm, Algorithm 6, variable Z contains the set of "zombies," the actions over which the proposed decisions were incompatible. These are the actions that are both dead according to the losing proposal and guaranteed according to the winning one, or vice-versa.

Algorithm 6 *UpdateProposal*(M, M')

1: **if** $\Sigma(M \cup M') = \emptyset$ **then**
2: Note that $M = (K, \rightarrow, \lhd, \#)$.
3: Let $Z = \mathit{Guar}(M \cup M') \cap \mathit{Dead}(M \cup M')$.
4: $\rightarrow := \rightarrow \setminus \{(\alpha, \beta) | \alpha \in Z \vee \beta \in Z\}$
5: $\lhd := \lhd \setminus \{(\alpha, \beta) | \alpha \in Z \vee \beta \in Z\}$
6: **end if**

3.4 Correctness

Our protocol is safe. It is live in the absence of faults, but may block when some sites are not communicating.

For lack of space we omit the full proof. Briefly, by the properties of majorities and pluralities, no two incompatible proposals can be elected. By the properties of stable prefixes, every election contains the result of previous elections as a prefix. This ensures that mergeability is preserved. Assuming that communication is live, and that $\mathcal{A}$ is live, the protocol makes progress because every new proposal eventually either wins or loses, thanks to plurality voting.

4 Discussion and Related work

Keleher's Deno system [5] 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 generalises Deno's voting procedure, enabling continuous voting [2].

The only relation between actions considered by both Deno and VVWV is Lamport's happens-before relation [7]; concurrent actions are assumed to conflict. Happens-before captures potential causality; however an event may happen-before another even if they are not truely dependent. This paper further generalises VVWV by considering semantic constraints. In particular, we distinguish causal dependence from happens-before. Using the separate constraints $\rightarrow$ and $\lhd$ we can encode more relations. We allow non-conflicting concurrent actions to co-exist in a schedule.

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 in a single primary replica.

IceCube [6] introduced the idea of reifying semantics with constraints. The IceCube algorithm aims to produce optimal proposals, minimising the number of dead actions.

Lamport introduced *state-machine replication* [7], 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. We generalise to schedule equivalence, and we allow a partial ordering when actions commute. This allows greater implementation freedom.

The well-known Paxos distributed protocol [8] computes a total order. Generalized Paxos [9] and Generic Broadcast [11] take commutativity relations into account and compute a partial order. They do not consider other semantic relations. Both Generalized Paxos [9] 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.

5 Conclusion, discussion 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 rely on

a centralised point of commitment. They typically impose a total order, which is stronger than necessary.

In contrast, we propose the first decentralised commitment protocol for semantically-rich systems. Our approach is to reify semantic relations as constraints. The scheduling behaviour of the system is restricted by the constraints. Thus an atomic group of actions is either entirely guaranteed or entirely dead. A dependent action may execute only after those it depends on. If actions conflict, they may not all become guaranteed, and the system must choose at least one and make it dead. If actions are non-commuting, the system must choose an execution order. Our formal definition of consistency puts an obligation on the system to resolve conflicts and non-commuting relations, and to eventually execute equivalent schedules at all sites.

Our protocol ensures eventual consistency in the presence of non-byzantine faults, as long as communication is live. It uses quorums to avoid any centralisation bottleneck. It uses plurality voting to make progress even when an election does not reach a majority. It proceeds with elections continuously and incrementally, instead of successive rounds. However it might block while a site is disconnected for a long time or fails.

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 behaviours. 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.

Further future work is to incorporate a failure detector. Currently, if a number of sites fail permanently, it may become impossible to ever finish an election. We plan to leverage the existing literature on group membership to recompute weights after a failure [3]. Finally, we plan a more detailed comparison between our approach and Generalized Paxos [9].

References

[1] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Int. Conf. on Mobile Comp. and Netw. (MobiCom '98)*. ACM/IEEE, Oct. 1998.

[2] J. Barreto and P. Ferreira. An efficient and fault-tolerant update commitment protocol for weakly connected replicas. In *Euro-Par*, pages 1059–1068, Lisbon, Portugal, Sept. 2005.

[3] U. Cetintemel and P. Keleher. Light-weight currency management mechanisms in mobile and weakly-connected environments. *The Journal of Distributed and Parallel Databases (JDPD)*, 11:53–71, 2002.

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

[5] P. J. Keleher. Decentralized replicated-object protocols. In *18th Symp. on Princ. of Distr. Comp. (PODC)*, Atlanta, GA, USA, May 1999.

[6] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *20th Symp. on Principles of Dist. Comp. (PODC)*, Newport RI, USA, Aug. 2001. ACM SIGACT-SIGOPS.

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

[8] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[9] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, Mar. 2005.

[10] F. Mattern. Virtual time and global states of distributed systems. In *Int. W. on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

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

[12] N. Preguiça, M. Shapiro, and C. 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, Nov. 2003. Springer-Verlag.

[13] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. In *9th Foundations of Softw. Eng.*, pages 175–185, Austria, Sept. 2001.

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

[15] M. Shapiro and K. Bhargavan. The Actions-Constraints approach to replication: Definitions and proofs. Technical Report MSR-TR-2004-14, Microsoft Research, Mar. 2004.

[16] M. Shapiro, K. Bhargavan, and N. 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, Dec. 2004.

[17] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. 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, Dec. 1995. ACM SIGOPS.

A Implementing the onus set Q

This appendix discusses a possible implementation of the onus set Q synchronising clients and agents.

A.1 Client-agent synchronisation

For moving actions into Q, two approaches are possible: (1) Clients pass onus for α when they are finished with α's constraints (we will say α is *closed*). Agents just wait for clients. (2) Allow agents to forcibly seize onus. A client that would want to add a new constraint to α's would either have to give up or would put its site into an error state.

Alternative 1 is preferable as it allows the greatest application flexibility; however consistency makes no progress if a client is unavailable. Our choice is to use Alternative 1 whenever possible, and Alternative 2 when necessary, for instance if a client does not respond within a timeout. Thus we let $Q = Q^C \cup Q^A$ where Q^C is managed by clients and Q^A by agents.

Note that the specification of Q^A is very weak; the only restrictions are that Q initially contains INIT, and that once an action is in Q, it remains in Q. In particular, Q^A does not rely on a fault detector.

A.2 Implementing Q^C

Now we consider the implementation of Q^C. Set Q^C is global, hence is not implementable directly. We now suggest one possible implementation whereby clients together compute a safe approximation of Q^C using only local information, similarly to Golding [4].

Each site maintains a matrix clock MC_i, which it attaches to every outgoing message and updates every time it receives a message, according to the classical rules [4, 10]. Column $\mathrm{MC}_i[\bullet, i]$ is the vector clock of i, and $\mathrm{MC}_i[\bullet, j]$ is a safe approximation at i of the vector clock of site j.

Action α is closed once its conflict and non-commutativity constraint have been produced. This happens once some client has had the opportunity to compare α with all actions submitted concurrently (in the sense of happens-before [7]). When site i receives for the first time an action β that conflicts or non-commutes with $\alpha \in K_i$, it adds the corresponding constraints and increments its vector clock, before sending any anti-entropy messages.

If the vector clock of site j is greater than $\alpha.ts$ then site j has received α, and was able to compare α for conflict or non-commutes to any concurrent action β known only at site j. If $\alpha.\text{ts} \leq \text{MC}_i[\bullet, j]$ then, by FIFO communication and according to the communication transition rules, then site i has received both β and the relevant constraints. Thus site i may approximate Q^C by $\{\alpha \in A | \forall\, j,\ \alpha.\text{ts} \leq \text{MC}_i[\bullet, j]\}$