Exploiting Our Computational Surroundings for Better Mobile Collaboration [*]

João Barreto [†] Paulo Ferreira
Inesc-ID/Technical University of Lisbon, Portugal
[joao.barreto,paulo.ferreira]@inesc-id.pt

Marc Shapiro [‡]
INRIA Rocquencourt/LIP6, Paris, France
http://www-sor.inria.fr/~shapiro/

Abstract

Mobile collaborative environments, being naturally loosely-coupled, call for optimistic replication solutions in order to attain the requirement of decentralized highly available access to data. However, such connectivity assumptions are also a decisive hindrance to the ability of optimistic replication protocols to rapidly guarantee consistency among the set of loosely-coupled replicas.

This paper proposes the extension of conventional optimistic replication protocols to exploit the presence of extraneous nodes surrounding the group of replica nodes in an increasingly ubiquitous computational universe. In particular, we show that using such extra nodes as temporary carriers of lightweight consistency meta-data may significantly improve the efficiency of a replicated system; notably, it reduces commitment delay and conflicts, and allows more network-efficient propagation of updates. We support such a statement with experimental results obtained from a simulated environment.

1 Introduction

Optimistic replication is a fundamental technique for supporting concurrent work practices and for tolerating failures in low-quality communication links [5, 18]. As collaboration through mobile and other loosely-coupled networks becomes popular (e.g. by using asynchronous groupware applications, or distributed file or database systems, and, in the future, collaborative wikis), the importance of this technique increases.

Optimistic replication enables access to a replica without *a priori* synchronization with the other replicas. Epidemic *anti-entropy* (AE) sessions [6] between pairs of replicas of the same object eventually bring any short-term divergence in their values to a consistent state. The final step of this process ensures that updates are durably either executed in a consistent order or aborted everywhere. This step is called *commitment*. Ideally, commitment should arrive rapidly. Nevertheless, the very aspects that motivate optimistic replication, i.e. poor and intermittent connectivity, are a crucial obstacle to efficient commitment.

A number of solutions for the update commitment problem have been proposed. Most [5, 7, 16, 9, 1] share a common property: they exclusively exploit the interactions between the nodes that hold a replica of a given logical object; i.e. assume the presence of a connected path between such nodes for AE to occur. However, in mobile environments, alternative opportunities of interaction are possible if one exploits the presence of other nodes, which we designate *non-replicas*.

For instance, assume that mobile workers A and B, both holding a replica of a shared object, are rarely mutually accessible. Consider that they have a common acquainted mobile user, X, who is not sharing the object, but is regularly in contact with either A and B; or, alternatively, consider a common public place, such as a train station or a restaurant, equipped with some stationary non-replica, Y, where A and B go on a daily basis, yet at different times.

As our current real world reaches closer to Weiser's vision of ubiquitous computing [21], the set of non-replicas surrounding any particular group of replica nodes becomes denser, and the above situations should happen more frequently. This paper advocates the exploitation of non-replicas such as X and Y to improve replica consistency.

Some proposed solutions support off-line AE [16, 8, 14], which enables a first solution: to make non-replicas carry enough information to emulate AE between A and B. However, AE includes propagating some representation of replica modifications. This entails relatively high memory requirements on non-replicas when replicated objects are sufficiently large (in state-transfer systems [18]), or writes are frequent (in operation-transfer systems [18]). In these cases, such requirements may be prohibitive for a significant proportion of non-replicas, especially in memory-constrained ubiquitous environments; thus, the presence of such potentially numerous non-replicas would still be neglected for the purpose of maintaining consistency.

The contribution of this paper is a novel solution where memory-constrained non-replicas carry only

[*]Work partially funded by project UbiRep POSI/CHS/47832/2002.
[†]Funded by FCT grant SFRH/BD/13859 (Portugal).
[‡]Funded by projects Grid4All (EU) and Respire (France).

lightweight consistency meta-data, excluding state-transfer or operation-transfer data. This way, the system exploits a substantially larger universe of non-replicas surrounding it. We designate such a piece of meta-data a *consistency packet*. We formally present a generic extension of consistency protocols that supports such an exchange of consistency packets. It is based on the decoupling of the agreement phase from the remaining phases of update commitment. Further, it is generic enough to be applied to representative state-of-the-art systems.

Most importantly, we show that the simple exchange of consistency packets, as enabled by non-replicas, may accelerate commitment, avoid conflicts, and allow more network-efficient update propagation. We evaluate it by simulation; we conclude that the predicted improvements become significant for sufficiently high non-replica densities. We believe that such densities are realistic in ubiquitous computing environments.

The paper is organized as follows. Section 2 proposes the extension of abstract consistency protocols in order to incorporate non-replicas carrying consistency packets, and discusses the achievable improvements. Section 3 addresses consistency packet management. Section 4 supports our claims with experimental results. Section 5 describes related work, while Section 6 concludes.

2 Architecture

In short, the goal of our architecture is to complement a system of replicas with non-replicas acting as carriers of consistency packets. Replicas generate consistency packets reflecting their state. A consistency packet should be of a lower order of magnitude than the information normally exchanged in AE. As consistency packets are delivered to non-replicas through ad-hoc encounters, and eventually to replicas, they should contribute to the progress of replica consistency. This requires extending the consistency protocol so that it becomes possible to make *some* progress by exchange of a *sufficiently small* subset of the information that is handled in AE.

Conceptually, AE exchanges information for two basic functions, update propagation and update commitment. The latter ensures that updates are consistently committed at all replicas, by (i) proposing a tentative update schedule; (ii) detecting and resolving conflicts; and (iii) agreeing on a schedule to be consistently committed.[1] We propose that consistency packets contain only the meta-data necessary for the agreement step (iii). This follows from the observation that (a) we may complete the agreement on a schedule independently of the remaining phases; and, (b) if decoupled from such phases, agreement is possible by the exchange of some lightweight meta-representation of schedules instead of the updates themselves (plus some algorithm-specific control data). This information comprises a consistency packet. In most systems, such a consistency packet will be sufficiently small for our intentions, as the next sections explain.

2.1 System Model and Terminology

Our system model consists of a set of logical objects, replicated at multiple network nodes. We designate a node as a *replica node* of o, or simply *replica* of o, if it replicates object o; and a *non-replica node* of o, or simply *non-replica* of o, otherwise. Whenever clear, we omit the reference to the object.

We assume the system to be asynchronous and nodes fail-stop. Network partitions may also occur. A process running in some replica node of object o may access the local replica of o without synchronizing a priori with the other replicas. In general, an access is tentative: a read may be stale and a write may later abort. In the case of writes, we say an *update* is generated.

We designate a totally ordered set of updates as a *schedule*. Moreover, each update u in a schedule is marked either as *executed* (denoted $\underline{u}$) or *aborted* (denoted $\overline{u}$). Given a schedule s, we denote by $\underline{s}$ the sub-schedule of s containing only executed updates; i.e. $\underline{s} = s \backslash \{u : \overline{u} \in s\}$.

An update u has two components: op_u, and $precond_u$. The first, op_u, completely specifies the operation. The second, $precond_u$, is a precondition defined by application semantics that determines whether u may be correctly included at a given position of a given schedule. A schedule s such that $\underline{s} = \underline{u}_1; ..; \underline{u}_n$ is said to be *sound* if, for any $\underline{u}_i \in \underline{s}$, $precond_{u_i}(\underline{u}_1; ..; \underline{u}_{i-1})$ is true.

Each replica r of o maintains a local schedule, sch_r, of all updates that it has received (either from a local process or from a remote replica). The current value of r is given by the ordered execution of $\underline{sch_r}$.

The consistency protocol should be designed to ensure that the possibly divergent replica values will eventually converge to a consistent one. We consider the following criterion of *eventual consistency* [18]: (a) for every replica r, sch_r is sound; (b) at any moment, for each replica r, there is a prefix of sch_r, denoted $allCommitted_r$, that is equivalent [2] to a prefix of sch_i, for every other replica i; (c) $allCommitted_r$ grows monotonically by suffixing over time; and (d) for every update u, generated at any replica, and for every replica r, after a sufficient finite number of AE sessions, either $\underline{u}$ or $\overline{u}$ is included in $allCommitted_r$.

We use the terms *stable* and *committed* distinctively to characterize updates. Although in most protocols both terms are synonyms, the distinction becomes necessary when consistency packets are considered. Update $\underline{u}$ (or $\overline{u}$)

[1]Note that some protocols may omit some of these steps.

[2]According to some equivalence relation, defined by the protocol.

is stable at a replica r once, based on the local state of r, the protocol guarantees that, after a sufficient finite number of AE sessions, $\underline{u}$ (respectively, $\overline{u}$) will *eventually* be included in $allCommitted_r$. Further, $\underline{u}$ is $committed$ at r when (i) $\underline{u}$ is stable at r, and (ii) $\underline{u} \in sch_r$; hence, u may be executed and produce a value that is guaranteed to never be rolled-back. Given some schedule s, $stable_r(s)$ denotes the largest prefix of s containing only updates stable at r. Finally, an update that is not yet stable in r is said to be $tentative$ in r.

We consider the class of protocols that (i) satisfy eventual consistency, as defined above, (ii) are able to explicitly characterize, at each moment, each update $\underline{u}$ (or $\overline{u}$) in sch_r, as either tentative, stable, or committed. We believe these requirements are desirable for most collaborative applications, as they allow applications to access a strongly consistent view, in complement to the tentative view [16]. They are verified in a number of representative systems such as Bayou [16], IceCube[10], Coda [12], CVS [3], Deno [9] or VVWV [1].

2.1.1 Base Protocol Abstraction

We start by introducing a simple abstraction of a generic protocol providing eventual consistency. We are concerned with the phases comprising the update commitment component (recalling the functional decomposition in Section 2); namely, (i) scheduling, (ii) conflict detection and resolution, and (iii) agreement. We abstract such phases by the functions $constructSch$ (representing phases i and ii combined), and $proposeSch$ and $getAgreedSch$ (phase iii). For the sake of generalization, we leave the update propagation component unspecified. All these phases make progress by information exchanged during AE.

Algorithm 1 Update Commitment at replica r

```
1:  loop
2:      if new updates $\{u_1, .., u_m\}$ exist then
3:          $sch_r \leftarrow constructSch(sch_r, \{u_1, .., u_m\})$;
4:          $proposeSch(sch_r)$;
5:      end if
6:  end loop
7:  ||
8:  loop
9:      if $sch_r \neq stable_r(sch_r)$ then
10:         $sch' \leftarrow getAgreedSch(sch_r)$;
11:         $sch_r \leftarrow constructSch(sch', sch_r \setminus sch')$;
12:     end if
13: end loop
```

Function $constructSch$ incrementally constructs a schedule at a given replica r as new updates are generated or received, and is specified as follows. Given an original schedule, s, and a set of new updates, U, as input, $constructSch(s, U)$ returns a sound schedule s' that (a) contains each update $u \in s \cup U$ (either as $\underline{u}$ or $\overline{u}$), and (b) leaves the original stable prefix unchanged (i.e., $stable_r(s') = stable_r(s)$).

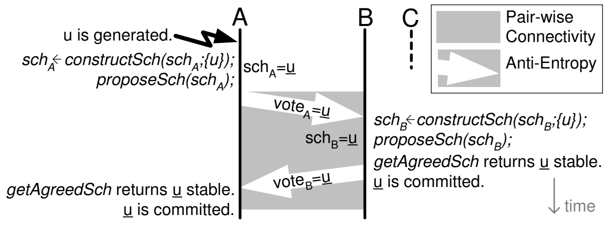
A
B
C
Pair-wise Connectivity
Anti-Entropy
u is generated.
$sch_A \leftarrow constructSch(sch_A;\{u\})$;
$proposeSch(sch_A)$;
$sch_A = \underline{u}$
$vote_A = \underline{u}$
$sch_B = \underline{u}$
$vote_B = \underline{u}$
$sch_B \leftarrow constructSch(sch_B;\{u\})$;
$proposeSch(sch_B)$;
getAgreedSch returns $\underline{u}$ stable.
$\underline{u}$ is committed.
getAgreedSch returns $\underline{u}$ stable.
$\underline{u}$ is committed.
time

Figure 1. Example of update commitment.

As sets of new updates, $\{u_1, .., u_m\}$, are either generated at a replica r, or received at r by AE from sch_p of another replica p, they are inserted into sch_r by $constructSch$, in some order and state (i.e., $\underline{u_i}$ or $\overline{u_i}$) that keeps sch_r sound. As new schedules sch_r are incrementally built, they may then be proposed for agreement. This is described in lines 3 and 4 of Algorithm 1.

At each replica r, the agreement algorithm incrementally decides, locally at each replica r, which updates in sch_r have become stable (at r), and at which order and state. Such a decision is output by $getAgreedSch$. Given an original schedule, s, as input, $getAgreedSch(s)$ returns, as soon as a new agreement is reached at r, a sound schedule, s', (a) that is completely stable at r (i.e. $stable_r(s') = s'$), and (b) that suffixes the stable prefix of the original one (i.e. $stable_r(s)$ is a prefix of s'. $getAgreedSch$ is called iteratively at a replica r as long as there are still tentative updates in sch_r (line 10 of Algorithm 1).

Figure 1 gives an example of update commitment with the above abstraction, in a partitioned 3-replica system. It assumes a voting algorithm for agreement that requires two replicas to vote for a tentative schedule to decide it as stable. When replicas A and B become accessible, two AE sessions enable the commitment of update u.

2.2 Incorporating Consistency Packets into the Protocol

The above abstraction defines the agreement phase by functions ($proposeSch$ and $getAgreedSch$) that manipulate updates included in schedules, plus some complementary control information specific to each particular algorithm. [3]

However, if we decouple the agreement phase from the remaining phases, we observe that proposed agreement algorithms (e.g. Bayou's primary-commit [16] or VVWV's weighted voting [1] agreement algorithms) do not actually require access to all the information an update holds;

[3]For instance, in weighted voting agreement (e.g. [1]), the weights and the replica identifier associated with each vote.

namely, they are not concerned with the op_u and $precond_u$ components of updates in a schedule. Instead, they may simply deal with some meta-data representation that identifies the (i) updates, (ii) their order, and (iii) their state (i.e. executed or aborted) in each schedule that is proposed or agreed. We call such a representation of a schedule s a *meta-schedule*, and denote it by *meta(s)*.

We designate the necessary information sent from one replica to another for the sole purpose of agreement as a *consistency packet*. It encapsulates meta-schedules as well as complementary control information of the agreement algorithm. Hence, given a particular agreement algorithm, implemented by the $proposeSch$ and $getAgreedSch$ functions, one may directly obtain their meta-data versions. We designate them as $m\text{-}proposeSch$ and $m\text{-}getAgreedSch$. Their input and output, respectively, are meta-schedules, instead of schedules. Whereas agreement by $proposeSch$ and $getAgreedSch$ progresses by AE, agreement by $m\text{-}proposeSch$ and $m\text{-}getAgreedSch$ evolves by exchange of consistency packets; these may be exchanged normally in the course of AE, or propagated individually through non-replicas.

A trivial meta-schedule consists of a list of update identifiers, id_{u_i}, ordered according to the corresponding schedule, and marked either as *executed* (denoted $\underline{id_{u_i}}$) or *aborted* (denoted $\overline{id_{u_i}}$). More efficient protocol-specific meta-schedules are also possible in some cases; the important case of version vectors is described in Section 2.4.

A meta-schedule is smaller in size than a schedule, as it no longer includes the op_u and $precond_u$ components of scheduled updates. For most systems, it will be of a lower order of magnitude in size than the corresponding schedule. We are interested in such systems. As a particular example, in the case of state-transfer systems [18] of reasonably sized objects, the size of schedules is dominated by the size of object state data (the op_u component); this component is absent in meta-schedules. A second example is that of operation-transfer systems [18] with frequent updates; meta-schedule representations that do not depend on the number of updates (e.g. version vectors) will be significantly smaller in size than long schedules.

Section 2.2.1 describes how to extend regular protocols in order to incorporate meta-agreement by exchange of consistency packets. Section 2.3 then shows that such an extension may entail important improvements on commitment delay, conflict rate and update propagation efficiency of the protocol.

2.2.1 Protocol Extension

So far, we have a generic protocol that (i) fits into the abstraction defined in Section 2.1.1, relying on $proposeSch$ and $getAgreed$, which manipulate and exchange schedules; but (ii) whose agreement algorithm may also work with meta-schedules, encapsulated in consistency packets (i.e. $m\text{-}proposeSch$ and $m\text{-}getAgreedSch$ exist).

We need to extend the protocol in order to use $m\text{-}proposeSch$ and $m\text{-}getAgreedSch$ instead of $proposeSch$ and $getAgreedSch$, respectively. This section describes such an extension by proposing adaptation functions, $proposeSchEx$ and $getAgreedSchEx$ with a similar interface to $proposeSch$ and $getAgreedSch$, respectively. They rely on $m\text{-}proposeSch$ and $m\text{-}getAgreedSch$. Hence, the original protocol may be extended by transparently replacing $proposeSch$ and $getAgreedSch$ by $proposeSchEx$ and $getAgreedSch$, respectively. The adaptation functions require the maintenance of a meta-schedule, $m\text{-}sch_r$, at each replica r. It represents the most recent stable meta-schedule that r knows so far.

Algorithm 2 $proposeSchEx(schedule\ s)$
```
1: $m\text{-}proposeSch(meta(s))$;
```

Algorithm 3 schedule commitOneLocalUpdate()
```
1: {Let $stable_r(sch_r) = u_1; ..; u_k$, and
   $stable_r(m\text{-}sch_r) = id_{u_1}; ..; id_{u_k}; ..; id_{u_m}$ $(k \leq m)$.}
2: while $m = k$ or $(m > k$ and $u_{k+1} \notin sch_r)$ do
3:     Wait for the first to complete:
4:     $sch_r$ has grown with new updates received;
5:     ||
6:     $m\text{-}sch_r \leftarrow m\text{-}getAgreedSch()$;
7: end while
8: if $\underline{id_{u_{k+1}}} \in m\text{-}sch_r$ then
9:     $s' \leftarrow stable_r(sch_r); \underline{u_{k+1}}$;
10: else
11:    $s' \leftarrow stable_r(sch_r); \overline{u_{k+1}}$;
12: end if
13: mark $u_{k+1}$ as stable in $s'$;
14: return $s'$;
```

The adaptation functions convert schedules (manipulated by the original protocol) into meta-schedules (handled by the meta-data agreement versions), and vice-versa. Function $proposeSchEx$, presented in Algorithm 2, performs the first conversion; by definition, this is trivial. On the other hand, function $m\text{-}getAgreedSch$, described in Algorithm 3, converts meta-schedules into schedules. This conversion requires synchronization, since the updates identified by a meta-schedule may not yet be locally available in order to compose the corresponding schedule. In that case, the conversion waits for subsequent AE sessions until having received each needed update (Alg. 3, line 4).

2.3 Benefits

The exchange of consistency packets via non-replicas adds an additional possibility of agreement progress. Figure 2 illustrates that by complementing the example of the

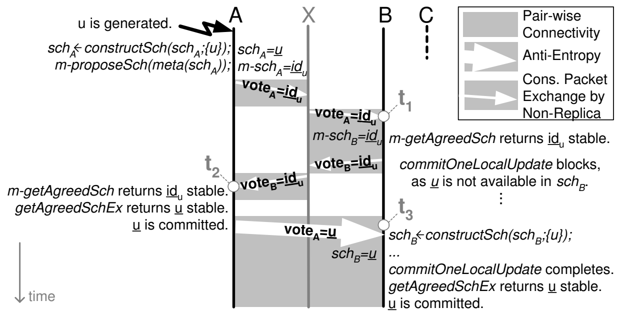
u is generated.
A
X
B
C
Pair-wise Connectivity
Anti-Entropy
Cons. Packet Exchange by Non-Replica
sch_A ← constructSch(sch_A;{u});
m-proposeSch(meta(sch_A));
sch_A=u
m-sch_A=id_u
vote_A=id_u
t1
vote_A=id_u
m-sch_B=id_u
m-getAgreedSch returns id_u stable.
vote_B=id_u
commitOneLocalUpdate blocks, as u is not available in sch_B.
t2
vote_B=id_u
m-getAgreedSch returns id_u stable.
getAgreedSchEx returns u stable.
u is committed.
t3
vote_A=u
sch_B ← constructSch(sch_B;{u});
...
sch_B=u
commitOneLocalUpdate completes.
getAgreedSchEx returns u stable.
u is committed.
time

Figure 2. Example of update commitment with exchange of consistency packets.

same 3-replica system as in Figure 1 with a passer-by non-replica, X. Instead of votes containing schedules, replicas A and B exchange consistency packets with X representing votes as meta-schedules. Despite being inaccessible for an initial period, A and B are able to exchange consistency packets through X. Consequently, agreement on a meta-schedule corresponding to $\underline{u}$ arrives before A and B could perform AE.

The *direct* effect of agreement acceleration does not, however, seem to be of great value to applications of the replicated system. However, it entails more interesting *indirect* improvements to the overall efficiency of the protocol. These are described as follows.

A. Commitment Acceleration. When agreement on a meta-schedule completes, the corresponding updates may be immediately committed if already available in sch_r (in a tentative form). Hence, agreement acceleration means commitment acceleration in this case. An example is that of replica A of Figure 2 (see t_2) after having received a consistency packet from B through X. Since update u is already available in sch_A, its actual commitment at A is also accelerated, even without AE.

B. Conflict Prevention. Accelerated agreement may cause the meta-schedule of a replica r to reference updates not yet included in sch_r. This denotes the existence of concurrent updates, whose awareness at r is anticipated by consistency packets. It implies that, if new updates are generated at r, there is a higher conflict probability than if no concurrent updates were known to exist. Applications may react to it by postponing planned writes to a time where the missing concurrent updates are received, hence reducing the conflict probability. An example is that of replica B in Figure 2, in moment t_1, after receiving a consistency packet from A denoting the existence of update u.

C. Reduction of *Hidden Conflicts*. Besides conflicts caused between updates generated concurrently at distinct replicas, conflicts may also arise when one update is generated *after* another one has already been locally received. We designate this a *hidden conflict*.

To illustrate, let u_1 and u_2 be two updates received at replica r that are conflicting so that a sound schedule may only either include $\underline{u_1}$ and $\overline{u_2}$, or $\overline{u_1}$ and $\underline{u_2}$. Assume that sch_r includes $\underline{u_1}$ and $\overline{u_2}$, both still tentative. In this case, the tentative value of r, which applications access, will reflect the outcome of $\underline{u_1}$ but not of $\overline{u_2}$. Now, consider that an application issues an update u_3, added tentatively as $\underline{u_3}$ to sch_r. Given the natural assumption that issued updates are not conflicting with the value of the replica upon which they are issued, u_3 will be compatible with u_1; however, it may (or may not) conflict with u_2. Let us consider the case where such a conflict holds. Finally, assume that the agreement algorithm chooses a stable schedule with $\overline{u_1}$ and $\underline{u_2}$ (contrarily to the previous tentative schedule), and thus both commit at r. Accordingly, u_3 will abort ($\overline{u_3}$) in sch_r, due to a hidden conflict with committed update $\underline{u_2}$.

Clearly, the agreement delay of a tentative schedule determines the time window during which hidden conflicts may arise. Hence, an accelerated agreement shortens such a window and, therefore, contributes to reducing the ratio of aborted updates.

D. Efficient Update Propagation. Agreement acceleration increases the frequency of updates that are already stable at the receiving replica, r, prior to being propagated to it. In particular, let u be one such update; i.e. $u \notin sch_r$ and $id_u \in m\text{-}sch_r$. Since u already has its final order with sch_r agreed, the following optimizations may take place before propagating u from some other replica to r by AE:

1. If $\overline{id_u} \in m\text{-}sch_r$, u needs not be propagated;
2. If $\underline{id_u} \in m\text{-}sch_r$, $precond_u$ may be left empty, as it is no longer needed (in t_3 of Figure 2, this is the case of update u before being propagated by AE from A to B);
3. Redundant or self-cancelling updates in a batch of consecutive (according to $m\text{-}sch_r$) stable updates need not be propagated.

2.4 Version Vector Meta-Schedules

A particularly efficient meta-schedule representation is a version vector [11]. This representation is applicable in systems with the following two properties. First, commitment respects the happens-before relation [13] in the sense that, if update u_1 happens-before update u_2, then the inclusion of $\underline{u_2}$ in a schedule sch_r implies that $\underline{u_1}$ also appears in sch_r, in a previous position than $\underline{u_2}$. Second, propagation of updates ensures the *prefix property* [16]: if a replica r holds an update u_x that was initially generated at another replica x, then r will also have received all other updates held by x (in sch_x) prior to u_x. For example, Bayou [16], Coda [12], and VVWV [1] satisfy the above properties.

Let each update u in sch_r be time-stamped by a version vector, vv_u, that expresses the happens-before relation between updates. When the above properties hold, a schedule s, such that $\underline{s} = \underline{u_1}; ..; \underline{u_n}$, may be identified by a version vector, vv_s, that is the component-wise maximum of each vv_{u_1},..,vv_{u_n} [11]. Provided that, for all $u \in s$, vv_u is available, vv_s represents a meta-schedule. In fact, from vv_s one can tell that whether, for a given update u, (a) $\underline{id_u}$ is in the meta-schedule (when $vv_u \geq vv_s$); or (b) $\overline{id_u}$ is in the meta-schedule (when vv_u and vv_s are concurrent); or (c) id_u is not in the meta-schedule (otherwise). Further, the total order of the update identifiers within the meta-schedule is any that respects the partial order defined by the happens-before relation. However, if a replica r receives vv_s from a remote replica, r will not always hold vv_{u_i} for every update u_i in the meta-schedule; instead, r has only vv_{u_i} for each update $u_i \in sch_r$. Hence, vv_s may only identify, at each moment, meta-schedules corresponding to updates currently in sch_r.

However, such a partial representation is sufficient for the Algorithms presented in Section 2.2.1, since they are not concerned with updates identified in meta-schedules until the updates are in sch_r.

3 Consistency Packet Management

Resilience to malicious non-replicas is achieved by having replicas encrypt and digitally sign consistency packets using a secret *object key*. The object key is associated with the corresponding logical object, and is exclusively known by its rightful replicas. This ensures the privacy and integrity of consistency packets.

The object key also securely identifies the logical object of a consistency packet, as follows. Each consistency packet includes a secure hash of the corresponding object key. A replica may verify that a consistency packet corresponds to a logical object of interest by hashing the object's key that is owned by the replica and comparing the result with the hash contained in the consistency packet. Since such a hash is included in the digitally signed contents of the consistency packet, the integrity of the consistency packet identification is also ensured.

A second issue is buffer management at non-replicas. Non-replicas have limited buffers, which means that, due to contention of consistency packets (from different logical objects and even from the same object), some may have to be discarded. Two techniques may reduce this occurrence. Firstly, an appropriate erasure method such as those studied in [19] may avoid obsolete consistency packets consuming network bandwidth and buffer space. Secondly, a substitution method may prevent older consistency packets from being stored and propagated simultaneously with more recent ones belonging to the same object. One possibility is to use version vectors to detect obsolete consistency packets.

4 Evaluation

The evaluation used a C# implementation of the VVWV protocol [1], extended as proposed in Section 2, in a simulator.[4] The simulator includes a collection of mobile nodes, randomly distributed by a set of network partitions. A node may be a replica or a non-replica; the latter may volunteer to carry consistency packets, in which case it is designated a non-replica carrier (*NRC*). We do not evaluate buffer management issues, as that is out of the scope of our main contribution in this paper. Instead, we assume that only a single object is replicated, and thus there is no contention of consistency packets of distinct objects at the buffers of NRCs.

Time is divided into logical time steps. At each time step, each node A: (1) migrates to a random network partition with a given *mobility probability*; (2) if A is replica or NRC, then A randomly selects a node, B, in its current partition, and A either performs AE from B (if A and B are replicas), or receives a consistency packet from B (if B and/or A are NRCs); finally, (3) if A is a replica, it generates an update with a given update probability (*updprob*).

We evaluate the incorporation of NRCs against two variants: NRCs do not exist (*NoNRCs*); and NRCs carry enough information to emulate complete offline AE [16, 8, 14] (*FullNRCs*). We simulated all variants in the same conditions and with the same randomization seeds. Each experiment lasted for 2000 logical time steps, and considered 45 nodes, among which 5 were replicas. The results assume 5 partitions, and 20% mobility probability. The variation of these parameters did not invalidate our conclusions.

A first experiment considered a contention-free situation, with *updprob*=0.2%. Figure 3 shows how the agreement and commitment delays (measured at each replica from the update issue time) vary with an increasing number of NRCs. As expected from Benefit A (refer to Section 2.3), NRCs accelerate not only agreement, but also commitment. Most importantly, commitment acceleration becomes substantial as NRCs become sufficiently numerous; with 40 NRCs, commitment delay is reduced by 48% and 24%, respectively for the first and last replicas to commit.

One perhaps surprising observation is that, for a sufficient number of NRCs, their commitment acceleration is comparable to, or better than, that achieved with FullNRCs (though with a lower number of FullNRCs). For example, 30 NRCs ensure better commitment delays than 20 FullNRCs, for all but the last replica to commit. This suggests that the commitment acceleration that is attainable with a set of resource-rich FullNRCs may still be achieved with a larger set of resource-constrained NRCs. This is especially interesting in ubiquitous computing environments, where the latter are the norm.

[4]The implementation and complete results are available at `http://www.gsd.inesc-id.pt/~jpbarreto/nonreplicas`.

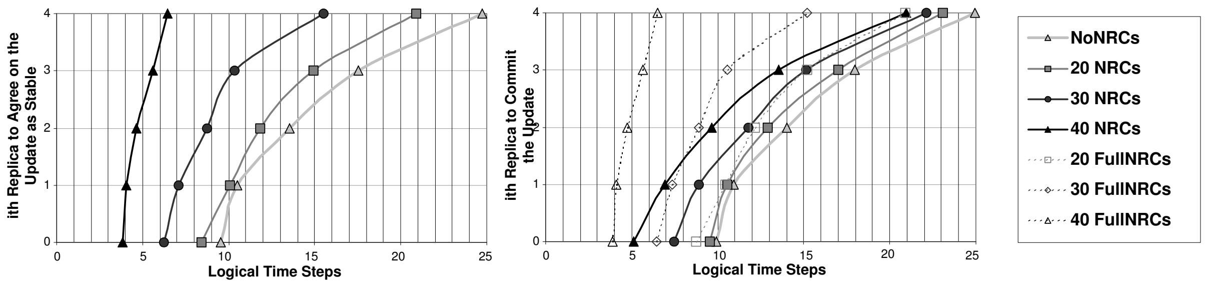

Figure 3. Agreement and commitment delays, no update contention (*updprob*=0.2%).

The above results suggest that cases where updates have already been received in a tentative form, and their commitment is waiting for agreement to complete, are significant. However, the inverse situation (i.e. agreement before update propagation) prevails (71% of updates are already stable when propagated with no NRC); it becomes more frequent as NRCs are considered: 73% with 20 NRCs, 79% with 30 NRCs, 84% with 40 NRCs. This results in more efficient update propagation (benefit D); our generic evaluation does not quantify such gains, as they are is application-specific.

A second experiment considered a scenario of high update contention, yielding frequent conflicts (*updprob* of 2%); this allowed us to measure benefits B and C. Benefit B was exploited as follows: when simulation step 3 decides to generate an update at replica r, r defers its generation until the schedule corresponding to $m\text{-}sch_r$ is available.

Starting at 20 NRCs, there is an important decrease on the frequency of aborted updates due to conflicts (8%, 40%, 73% with 20, 30, 40 NRCs, resp.), as Figure 4 depicts. For most prevented abortions (i.e., committed executed updates that are aborted in *NoNRC*), we were able to correlate them to Benefits B or C. Benefit B becomes dominant as the postponed updates grows with the number of NRCs. An important observation, however, is that hidden conflicts are significant (16% of conflicts in *NoNRC* are hidden conflicts), despite undocumented in optimistic replication literature, to the best of our knowledge; and Benefit C largely prevents them (more than 50% avoided with 30 and 40 NRCs).

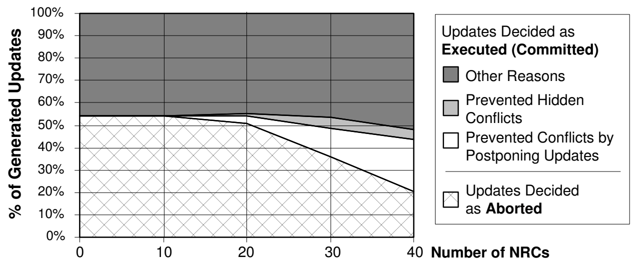

Figure 4. State of committed updates, high update contention (*updprob*=2%).

5 Related Work

The exploitation of non-replica nodes is already permitted by systems such as Bayou [16], Rumor [8], or Footloose [14]. However, they require non-replica nodes to carry complete anti-entropy information, instead of meta-data; the implicit memory requirements will thus be prohibitive for a significant number of nodes, especially in resource-constrained mobile environments. Nevertheless, their approach may be transparently combined with ours.

Volatile witnesses [15] consist of non-replica nodes that carry lightweight meta-data to replace replicas partitioned from the main partition; its goal is to increase the availability of pessimistic write-all read-one protocols. We address the radically different domain of optimistic replication.

The principle underlying our contribution is the separation of control from data, which is not new in optimistic replication. In Coda [12], servers send invalidations to clients holding replicas of an updated object. Update propagation to invalidated replicas is only performed subsequently. If we regard invalidations as meta-schedules, these systems correspond to a limited application of the principle in a client-server context. They achieve only a subset of our benefits, namely more scalable and efficient update propagation and prevent conflicts (Benefits D and B, resp.).

Fluid Replication [4] extends the principle to reconciliations between the clients generating updates and the server. A client sends meta-schedules represented by *Least Common Ancestors* [4]; in the absence of conflicts, the server confirms that the corresponding new updates are stable. The actual update propagation is deferred. Fluid Replication achieves benefits similar to ours. However, it is only suitable for client-server systems, while our work is topology-independent, and does not consider non-replicas.

The PRACTI [2] optimistic replication approach allows invalidations to be exchanged in arbitrary network topologies. Nevertheless, their notion of eventual consistency is weaker than ours, as they do not have an explicit agreement phase; that is, PRACTI is not able to tell if, at a given moment in time, a particular update is already committed. Therefore, PRACTI's invalidations only serve conflict prevention (Benefit B) by restricting access to invalidated replicas. Further, none of the above mentioned solutions is applicable to rich-semantic systems.

The Pangaea [17] file system floods small messages containing timestamps of updates (called harbingers) before propagating the latter. Their purpose is different than ours: harbingers eliminate redundant update deliveries, and prevent conflicts in the sense of benefit B. The Roma personal meta-data service [20] separates consistency meta-data management from data itself. Such a management is for user presentation only, and is not integrated into the underlying replication protocol. Moreover, Roma is centralized and designed for personal, not collaborative, scenarios.

6 Concluding Remarks

We formally present a generic extension of optimistic replication protocols that decouples the agreement phase from the remaining phases of AE. Hence, it enables consistency to evolve, to some extent, by exchange of meta-data that is a small subset of the information exchanged in regular AE interactions. Consequently, potentially memory-constrained non-replica nodes surrounding the system of mobile replicas may be exploited as carriers of such meta-data. Based on simulation, we show that the exploitation of non-replica nodes reduces commitment delay and update abortions, and enables more network-efficient update propagation. Such improvements become substantial as the density of non-replicas grows to what we believe to be expectable in ubiquitous computing environments.

Non-replicas constitute a low-bandwidth transport channel. Due to decoupled agreement, such a channel may be used for dissemination of consistency packets, whereas it is prohibitive for AE. Similarly, our contribution is also applicable to other instances of transport channels with limited bandwidth or high monetary or energy costs, for example.

References

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

[2] N Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX NSDI*, May 2006.

[3] Per Cederqvist et al. Version management with CVS. `http://www.cvshome.org/docs/manual/`, 1993.

[4] Landon P. Cox and Brian D. Noble. Fast reconciliations in fluid replication. In *Proc. of ICDCS*, pages 449–458, 2001.

[5] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.

[6] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of PODC'87*, pages 1–12, 1987.

[7] Richard Golding. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical Report UCSC-CRL-93-09, UC Santa Cruz, 1993.

[8] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER Workshops*, pages 254–265, 1998.

[9] P. Keleher. Decentralized replicated-object protocols. In *Proc. of PODC'99*, 1999.

[10] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proc. of PODC'01*, 2001.

[11] D.S. Parker; G.J. Popek; G. Rudisin; A. Stoughton; B.J. Walker; E. Walton; J.M. Chow; D. Edwards; S. Kiser and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering SE-9*, (3):240–7, 1983.

[12] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proc. of 13th ACM SOSP*, pages 213–25, 1991.

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

[14] Justin Mazzola Paluska, David Saff, Tom Yeh, and Kathryn Chen. Footloose: A case for physical eventual consistency and selective conflict resolution. In *Proceedings of the 5th IEEE WMCSA*, pages 170–180, Monterey, CA, USA.

[15] Jehan-Francois Paris. A highly available replication control protocol using volatile witnesses. In *Proc. of ICDCS'94*, pages 536–543, 1994.

[16] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the 16th ACM SOSP*, 1997.

[17] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.

[18] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[19] Tara Small, Zygmunt J. Haas, Alejandro Purgue, and Kurt Fristup. *Handbook of Sensor Networks*, chapter 11, A Sensor Network for Biological Data Acquisition. CRC Press, 2004.

[20] Edward Swierk, Emre Kiciman, Nathan C. Williams, Takashi Fukushima, Hideki Yoshida, Vince Laviano, and Mary Baker. The roma personal metadata service. *Mob. Netw. Appl.*, 7(5):407–418, 2002.

[21] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265:94–104, 1991.