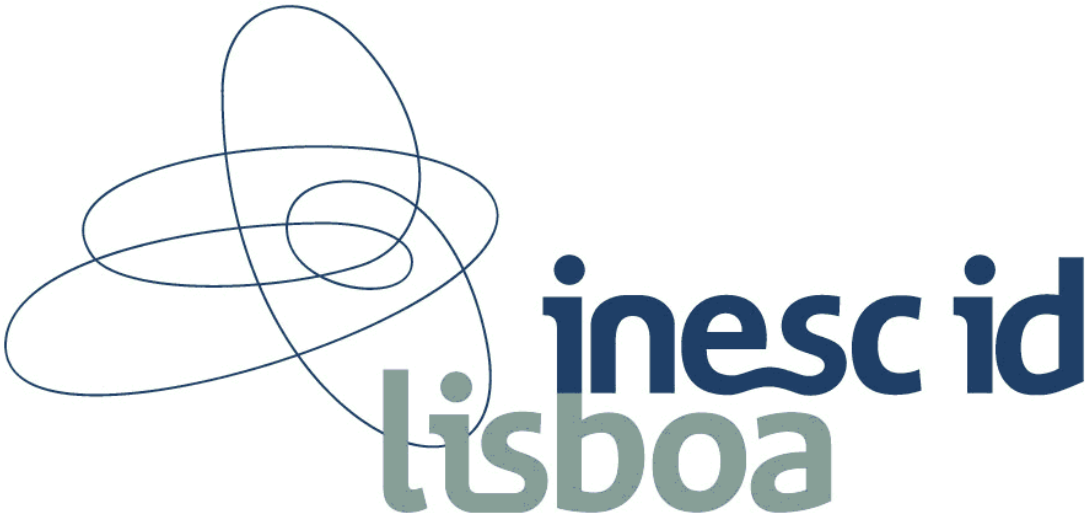

Technical Report RT/007/2004

Optimistic Consistency with Dynamic Version Vector Weighted Voting

João Barreto and Paulo Ferreira
INESC-ID/IST
Distributed Systems Group
Rua Alves Redol N 9, 1000-029 Lisboa
[joao.barreto,paulo.ferreira]@inesc-id.pt

October 2004

Abstract

Mobile and loosely-coupled environments call for decentralized optimistic replication protocols that provide highly available access to shared objects. A fundamental property of most optimistic protocols is to guarantee an eventual consensus on a commit order among the set of tentatively issued updates. In this paper we propose a replicated object protocol that employs a novel epidemic weighted voting algorithm based on version vectors for achieving such goal. An epidemic voting strategy eliminates the single point of failure of primary commit approaches, while not imposing the simultaneous accessibility of a plurality quorum.

Our protocol introduces a significant optimization over basic epidemic weighted voting solutions by allowing multiple-update candidates through the use of version vectors. As a result, it is able to commit multiple, causally related updates at a single distributed election round. Complementarily, we describe how dynamic version maintenance can be easily incorporated into the voting protocol in order to reduce version vector size and avoid the need for complete knowledge of group membership.

We demonstrate that our proposed algorithm is especially advantageous when considering realistic, non-uniform update models. We support such assumptions by presenting comparison results obtained from side-by-side execution of reference protocols in a simulated environment.

1 Introduction

Data replication is a fundamental mechanism for most distributed systems for a number of highly desirable properties. It enhances performance and scalability by enabling local data replicas to be accessed rather than a centralized physical data source, potentially located on a remote server. Replication improves fault tolerance if replicas are ensured to be kept consistent: should one replica be lost, the contents of any other consistent replica can be used to recover the lost data. Finally, it increases availability as applications may continue to access their local replicas even in case of failure or network inaccessibility of other replicas.

In particular, optimistic replication protocols [1] are of extreme importance in mobile and other loosely-coupled network environments. The nature of these environments calls for decentralized replication protocols that are able to provide highly available full access to shared objects. Such requirement is accomplished by optimistic replication strategies, which, in contrast to their pessimistic counterparts, enable updates to be issued at any one replica regardless of the availability of other replicas. The replication protocol is then responsible for disseminating such updates among the remaining replicas.

As a trade-off, the issue of consistency of an optimistically replicated system is problematic. Since replicas are allowed to be updated at any time and circumstance, updates may conflict if issued concurrently at distinct replicas. Some optimistic replication protocols ensure that, from such a possibly inconsistent *tentative* state, replicas evolve towards an eventual consistent *stable* state. For this end, a distributed consensus algorithm is executed so as to reach an agreement on a common order in which tentative updates should be committed. Such consistency approach can be regarded as a hybrid strategy: at the immediate update time, only weak consistency guarantees are provided upon the tentative replica values; on an eventual point in the future, however, a strongly consistent stable value is attained.

There are many scenarios where users and applications, in order to benefit from high availability, are willing to work with temporarily tentative data, provided that a commitment agreement regarding such data will eventually be reached. Consider, for instance, a worker carrying a laptop that becomes disconnected from his corporate fixed network file server after leaving his office and heading to a work meeting. If, by chance, he comes to think of some changes to a report that is currently replicated at his laptop cache, he might expect to be able to modify it at that moment, even if tentatively. Afterwards, he would return to his office and synchronize his laptop's replicas with the fixed file server. Provided that his team colleagues had not modified the shared report during his absence, his tentative updates would then be safely committed in the fixed file server.

If a pessimistic protocol was employed, the previous example would require the mobile user to obtain, in advance, a lock over the shared report. However, it is not always possible to correctly anticipate that a given replica will be requested for update. For instance, if all the team colleagues had similar probabilities of modifying the shared report, granting the token to one of them would possibly prevent progress to be made by other worker that unexpectedly decided to modify the file. Such unpleasant effect would have a particular impact in the case of extended disconnections of the lock holders, which is a usual occurrence in mobile networks.

Furthermore, consider that the mobile worker happens to meet the remaining of his team colleagues while away from the office who, in an ad-hoc fashion, establish a short term work group to review the shared report. If every worker is carrying some mobile device with a cached replica of the report, a collaborative activity might be carried if each worker is able to perform updates to its local replica and have such updates propagated to the remaining replicas. A set of causally related tentative updates will result from such activity. Hopefully, if no update is concurrently issued from outside the group, such tentative work will be eventually committed.

Hence, the high availability provided by an optimistic replication strategy is especially interesting in such scenarios as the previous ones. However, the usefulness of one such approach strongly depends on the ability of the underlying replication protocol to efficiently achieve a commitment decision concerning the tentatively issued data. Users are typically not inclined towards working on tentative data unless they trust the protocol to rapidly achieve a strongly consistent commitment decision regarding such data.

Aiming at such central objective, this paper proposes a novel optimistic replication protocol to accomplish efficient and highly available update commitment through the use of an epidemic weighted voting algorithm based on version vectors. The use of a voting approach eliminates the single point of failure that characterizes primary commit approaches [2]. Instead, the unavailability of any individual replica is not prohibitive of the progress of the update commitment process. Moreover, commitment agreement is accomplished without the need for a plurality quorum of replica servers to be simultaneously accessible, as happens with dynamic voting schemes [3, 4]. Instead, voting information flows epidemically between replicas and update commitment is based solely on local information.

The solution we propose has two main contributions. Firstly, our voting algorithm introduces a significant optimization over basic epidemic weighted voting solutions by allowing multiple update candidates to participate in an election. By using version vectors, candidates consisting of one or more causally related updates may be voted and committed by running a single distributed election round.

As a result, the overall number of anti-entropy sessions required to commit updates is decreased when compared to a basic weighted voting protocol. Hence, update commitment delay is minimized and so eventual strong consistency guarantees are more rapidly delivered to applications. Namely, such reduction is substantial in scenarios where frequent causally related updates are generated tentatively by applications. The examples presented above are representative of such update patterns. In worst case scenarios, our protocol is similar to basic weighted voting protocols.

Secondly, we show how dynamic version vectors can be easily incorporated into the voting protocol so as to keep the version vector size to a minimum and to eliminate the need for a complete knowledge of the set of replicas. In particular, we introduce a novel compression algorithm that only requires consensus from a plurality quorum before removing inactive entries from dynamic version vectors.

Finally, we present and discuss simulated results that compare the behavior of our proposed solution with other reference protocols. Namely, we measure the protocols adequacy to non-uniform update models that are expected to better reflect typical real usage scenarios [5, 6].

The remainder of the paper is organized as follows. Section 2 describes related work, while Section 3 introduces the proposed consistency algorithm using static version vectors and Section 4 explains how dynamic version vectors are incorporated into such algorithm. Section 5 then evaluates the algorithm. Finally, Section 6 draws some conclusions.

2 Related Work

The issue of optimistic data replication for mobile and loosely coupled environments has been addressed by a number of projects [7, 1], with the common intent of offering high data availability. Most of the proposed solutions share the goal of our work by being able to eventually enforce convergence from weakly consistent tentative replica values towards a strongly consistent stable form.

Roam [5] is a scalable optimistic replicated file system intended for mobile environments. Roam ensures an eventual convergence to a total order among causally related replica updates. Nevertheless, Roam's consistency protocol does not regard any notion of update commitment, which means that it cannot assert whether the replica values accessed by applications are strongly consistent or not.

Other solutions overcome such important limitation by complementing the tentative view of a replica with a strongly consistent view obtained from the exclusive application of committed updates. Three main approaches for update commitment can be distinguished in related work.

One approach proposed by Golding [8] relies on the maintenance, at each replica, of a worst case estimate of the updates received at every replica, through the use of *ack vectors*. Based on such estimate, each individual server commits updates when it is certain that an update has been received by every replica. A main limitation of this solution is that the unavailability of any single replica stalls the entire commitment process.

On the other hand, a primary commit strategy centralizes the commitment process in a single distinguished primary replica that establishes a total commit order over the updates it receives. Primary commit is able to rapidly commit updates, since if

suffices for an update to be received by the primary replica to become committed, provided that no conflict is found. However, should the primary replica become unavailable, the commitment progress of updates generated by replicas other than the primary is inevitably halted. The Bayou System [9] is an optimistic database replication system that employs a primary commit approach. Bayou also relies on application-specific conflict detection and resolution procedures to attain adaptable consistency guarantees. Furthermore, Haddock-FS [10], a highly available replicated file system designed for resource constrained mobile environments, is also based on the primary commit approach.

Finally, a third approach uses voting so as to allow a plurality quorum to commit an update. In particular, Deno [11] relies on an epidemic voting protocol to support object replication in a transactional framework for loosely-connected environments. Deno requires one entire election round to be completed in order to commit each single update, if only non-commutable updates are considered. This is acceptable when applications are interested in knowing the commitment outcome of each tentative issued update before issuing the next one. However, in the usage scenarios addressed by this paper, users and applications will often be interested in tentatively issuing multiple, causally related tentative updates before acknowledging their commitment. In such situations, the commitment delay imposed by Deno's voting algorithm becomes unacceptably higher than that of a primary commit approach.

3 Consistency Protocol

The following discussion considers a model where a set of logical objects is replicated at multiple server hosts. An object replica at a given server provides local applications with access to a version of the object contents, as stored by the replica. Such accesses may read or modify the object contents. In the case of the latter, an update is issued by the server and applied to the replica.

Updates issued at a given replica are propagated to other servers in an epidemic fashion in order to eventually achieve object consistency. The local execution of an update is assumed to be recoverable, atomic and deterministic. The former means that a replica will not reach an inconsistent value if it fails before the update execution completes. It follows from the other two properties that the execution of the same ordered sequence of updates at two distinct replicas in the same initial consistent state will yield an identical final state.

Hereafter, we assume an asynchronous system in which servers can only fail silently. Network partitions may also occur, thus restricting connectivity between servers that happen to be located in distinct partitions.

For simplicity, we consider that each logical object is replicated at every server in the system. Nevertheless, the consistency protocol is trivially extensible to support selective replication [12], i.e. where each object may be replicated at a sub-set of servers. For the sake of generality, the set of replicas may be dynamic, and thus change with the creation or removal of new servers.

3.1 Overview

Due to the optimistic nature of the consistency protocol, an update issued at a local replica is not immediately committed at every remaining replica. Instead, such update is considered to be in a tentative form since conflicting updates may still be issued at other replicas. The consistency protocol is responsible for committing such tentative updates into a total order that will be eventually reflected at every replica. Our protocol achieves this goal through a weighted voting approach [13].

In a weighted voting consistency protocol, concurrent tentative updates are regarded as rival candidates in an election. The servers replicating a given logical object act as voters whose votes determine the outcome of each election between candidate updates to the object. A candidate update wins an election by collecting a plurality of votes, in which case it is committed and its rival candidates are discarded.

Elections consider a fixed per-object currency scheme, in which each voter is associated with a given amount of currency that determines its weight during voting rounds. The global currency of a logical object, distributed among its replica servers, equals a fixed amount of 1. Currencies can be exchanged between servers and the currency held by failed servers can be recovered by running a *currency reevaluation* election, as discussed in [14].

3.1.1 Version Vector Candidates

In some cases, applications will be interested in generating more than one tentative update prior to its commitment decision. These may include disconnected mobile applications and ad-hoc groups of mobile applications working cooperatively in the absence of a plurality quorum. Since the commitment decision may not be taken in the short-term, these applications may wish to issue a sequence of multiple, causally ordered tentative updates.

In order to efficiently accommodate for such update scenarios, the novel solution proposed in this paper employs version vectors [15, 16] to identify candidate updates in a weighted voting algorithm. A version vector, v, is a logical time stamp comprised by an array of N integers, one for each replica of the logical object. Given a replica version stamped with a version vector v_k, each entry $v_k[i]$, for $i = 0, 1, .., N$, represents the number of updates issued at replica i that affect the version in consideration. Version vectors are defined as follows:

1. The initial version of an object is denoted by v_0, where $v_0[i] = 0$ for $i = 0, 1, .., N$;
2. An update issued at object replica r with version v_k generates a new version $v_j = advance_r(v_k)$, defined by:
   (a) $\forall i \neq r, advance_r(v_k)[i] = v_k[i]$;
   (b) $advance_r(v_k)[r] = v_k[r] + 1$.

For simplicity of presentation, the notation $v(u_1, .., u_n)$ is used in the remainder of the paper to represent the version vector that is obtained if a sequence of updates, $u_1, .., u_n$, respectively issued at replicas $r_1, .., r_n$, are applied to a null version vector, v_0, as given by $advance_{r_n}(...(advance_{r_1}(v_0)))$.

Once identified by version vectors, v_1 and v_2, two object versions can be compared as follows:

1. $v_1 = v_2$ iff $v_1[i] = v_2[i]$ for $i = 0, 1, .., N$;
2. $v_1 \leq v_2$ iff $v_1[i] \leq v_2[i]$ for $i = 0, 1, .., N$;

Important statements can be made about the causality between two distinct replica versions identified by version vectors v_1 and v_2. Firstly, it can be proven that if and only if $v_1 \leq v_2$ and $v_1 \neq v_2$, or simply $v_1 < v_2$, then version v_2 causally succeeds v_1 according to the *happened before* relation defined by Lamport [17]. Otherwise, if neither $v_1 \leq v_2$ nor $v_2 \leq v_1$, then both versions are causally concurrent, or $v_1 \parallel v_2$.

The flexibility brought by identifying candidates using version vectors allows a sequence of one or more updates to run for the current election as a whole. In this case, the candidate is represented by the version vector corresponding to the tentative version obtained if the entire update sequence was applied to the replica. As the next sections explain, the voting algorithm is responsible for deciding if the update sequence or a prefix of it will become committed.

Moreover, the inherent expressiveness of version vector candidates allows for candidates consisting of one or more causally related updates to be committed on a single distributed election round. In disconnected or loosely-coupled usage scenarios such as those enabled by mobile environments, where such update patterns are expectably dominant, a substantial optimization in update commitment delays is therefore achievable.

The consistency protocol requires each replica r to maintain the following state:

- $stableTS_r$, which consists of a version vector that identifies the most recent stable version that is currently known by replica r, obtained after the ordered application of all committed updates;
- $votes_r[1..N]$, which stores, for each server $k = 1, 2, .., N$, the version vector corresponding to the candidate voted for by k, as known by r; or $\perp$, if the vote of such server has not yet been known to r;
- $cur_r[1..N]$, which stores, for each server $k = 1, 2, .., N$ whose vote replica r has knowledge of, the currency associated with such vote;

As the next sections describe in greater detail, voting information flows in an epidemic fashion among servers and the decision to commit an update is based only on local replica information. These are important properties for operation under mobile and loosely-coupled environments. In particular, Section 3.2 addresses the issues of tentative update storage and their corresponding commitment upon a replica

value. Furthermore, Section 3.3 then addresses the access to the replica by applications and users. Section 3.4 then describes the epidemic flow of consistency information and Section 3.5 finally defines how candidate are elected.

3.2 Update Commitment

The protocol proposed hereafter is orthogonal to the issues of actual transference and storage of tentative updates. In particular, the protocol does not impose the decision of whether to transfer and store, at each individual replica, the tentative updates belonging to every candidate in the current election or, alternatively, only those concerning the replica's own candidate.

This means that, at the time a server determines that a given candidate has won the election and, thus, its updates should be committed, such updates may not be immediately available. Instead, they will be eventually collected through anti-entropy sessions with other servers. Consequently, there may occur a discrepancy between the most recent stable version identified by the consistency protocol and the actual stable value that is locally accessible.

In order to allow for such discrepancy, an additional element is maintained at the state of each replica r:

- c_r, which consists of an integer value representing the number of updates in the stable path that have already been committed by replica r;

The value of c_r may be lower than the number of updates that have actually been determined by the consistency algorithm as belonging to the stable path. In such case, the replica's stable value does not yet reflect the most recent stable version r is aware of. As a consequence, the protocol is flexible enough to support servers with differing memory limitations.

On one hand, servers with rich memory resources may store every update associated with each candidate, hence being able to immediately gain access to the most recent known stable value as each new stable version is determined by the protocol. On the other hand, memory-constrained devices may opt to restrict themselves to storing only the updates of their own candidate and, thus, allow for occasional delays in the availability of the most recent stable value when rival candidates win an election. In either case, however, the efficiency of the protocol in determining commitment decisions is not affected. Furthermore, both strategies may transparently co-exist in a system of replicas of the same logical object.

Moreover, it is assumed that a log of committed updates is maintained, including the following information:

- $issuer_r[1..c_r]$, which stores, for each update committed so far in replica r, the server that generated it.

From the consistency protocol's viewpoint, the procedure for committing a sequence of updates $u_1, .., u_n$, generated by servers $i_1, ..i_n$, respectively, is thus comprised by the following steps:

1. For each update, u_k, $issuer_r[c_r + k] \leftarrow i_k$;
2. $c_r \leftarrow c_r + n$;

3.3 Access to Stable and Tentative Views

Each server is able to offer two possibly distinct views over the value of a replica to its applications and users: the stable and tentative views. The first view reflects the value of the replicated object that is identified by $stableTS$. This value is obtained by the ordered application of each committed update that has resulted from the elections that have already been completed at the local server.

Issuing an update from this view causes further local accesses to the replica to be blocked until the respective election is locally completed and the update is committed or discarded. It follows from the latter that the stable view offers traditional sequential consistency guarantees [18], acceptable for applications with strong consistency demands.

On the other hand, the tentative view exposes the value that corresponds to the candidate version that is currently voted by the local server. If no vote has yet been cast by the local server, both the stable and tentative views yield the same replica value.

If waiting for a commitment agreement for each update is not acceptable, applications may opt to access the weakly consistent value provided by the tentative view. In this case, generating an update is a non-blocking operation that simply suffixes the new tentative update to the candidate currently voted by

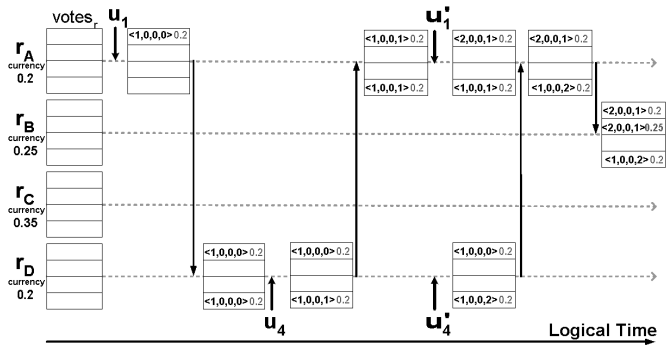

Figure 1: Example of update generation and propagation through anti-entropy sessions, using tentative view. Four replicas of the same logical object, r_1, r_2, r_3 and r_4, start from a common initial stable version, $\langle 0,0,0,0\rangle$ and currency is unevenly divided among the replicas.

the server in consideration. Since the tentative view reflects the version determined by such candidate, the newly issued update is immediately visible in further accesses to the tentative view.

Summing up, issuing an update on a replica r causes a new candidate to run for the current election according to the following rules:

1. If $votes_r[r] = \perp$,
   then $votes_r[r] \leftarrow advance_r(stableTS_r)$ and $cur_r[r] = currency_r$;
2. Otherwise (and only if *tentative view* is selected),
   $votes_r[r] \leftarrow advance_r(votes_r[r])$;

3.4 Anti-entropy

Voting information is propagated through the system by anti-entropy sessions established between pairs of accessible replicas. An anti-entropy session is an unidirectional pull-based interaction in which a requesting replica, A, updates its local election knowledge with information obtained from another replica, B. In case B has more up-to-date election information, it transfers such information to A. Furthermore, if A has not yet voted for a candidate that is concurrent to the one voted for by B, A accepts the latter, thus contributing to its election.

Each anti-entropy session is carried out according to the following procedure, which should be executed atomically:

1. If $stableTS_A < stableTS_B$ then
   (a) $stableTS_A \leftarrow stableTS_B$;
   (b) $\forall k$ s.t. $votes_A[k] \| stableTS_A$ or $votes_A[k] \leq stableTS_A$,
   $votes_A[k] \leftarrow \perp$;
2. If ($votes_A[A] = \perp$ and $stableTS_A < votes_B[B]$)
   or $votes_A[A] < votes_B[B]$ then
   $votes_A[A] \leftarrow votes_B[B]$ and $cur_A[A] \leftarrow currency_A$;
3. $\forall k$ s.t. ($votes_A[k] = \perp$ and $stableTS_A < votes_B[k]$)
   or $votes_A[k] < votes_B[k]$,
   $votes_A[k] \leftarrow votes_B[k]$ and $cur_A[k] \leftarrow cur_B[k]$.
4. If $c_A < c_B$ then commit update sequence issued by $issuer_B[c_A + 1], .., issuer_B[c_B]$

The first step ensures that, in case r_B knows about a more recent stable version, r_A will adopt it. This means that r_A will regard the elections that originated such new stable version as completed and so begin a new election from that point. Such new election is prepared by keeping only the voting information that will still be meaningful for the outcome of the election. Namely, these are the votes on candidates that causally succeed the stable version.

As a second step, r_A is persuaded to vote for the same candidate as the one voted by r_B, provided that r_A has not yet voted for a concurrent candidate. Subsequently, r_A updates its current knowledge of the current election with relevant voting information that may be held by r_B. Namely, r_A stores each vote that it is not yet aware of or whose candidate is more complete than the one it currently has knowledge of.

Finally, the set of committed updates held by B that are not yet locally available at replica A are collected and committed by the latter. An example of update generation and propagation through anti-entropy is illustrated in Figure 1.

3.5 Election decision

The candidates being voted in an election represent update paths that traverse through one of more versions beyond the initial point defined by the stable

version, $stableTS$. These possibly divergent candidate update paths may share common prefix sub-paths. The definition of the maximum common version expresses such notion.

Definition 1: *Maximum common version.*

Given two version vectors, $v_1 = \langle e_{1,1}, e_{1,2}, .., e_{1,N} \rangle$ and $v_2 = \langle e_{2,1}, e_{2,2}, .., e_{2,N} \rangle$, their maximum common version, $mcv(v_1, v_2)$, is given by a version vector $\langle m_1, m_2, .., m_N \rangle$ where $\forall k, m_k = min(e_{1,k}, e_{2,k})$. For simplicity, we assume $mcv(v_1, v_2, .., v_m)$ to be obtained by $mcv(mcv(mcv(v_1, v_2)), ...), v_m)$.

Theorem 1: Let $v_1, .., v_m \in votes_r$, be one or more candidate versions known by replica r, each connoting a tentative update path starting from the stable version, $stableTS_r$. Their maximum common version, $mcv(v_1, .., v_m)$, constitutes the farthest version of an update sub-path that is mutually traversed by the update paths of $v_1, .., v_m$. Complementarily, the total currency voted for such common sub-path is obtained by $voted_r(mcv(v_1, .., v_m)) = cur_r[1] + ... + cur_r[N]$.

Sketch of Proof: Assume, by absurd, that $m = mcv(v_1, .., v_m)$, is not the farthest version of an update sub-path that is shared by the update paths of $v_1, .., v_m$; instead, such version is given by $m' \neq m$. By definition of mcv, $m' \not> m$; otherwise, $m' \leq v_i, \forall i$, wouldn't be verified. So, $m' < m$, which implies that $\exists k$ s.t. $m'[k] < m[k] \leq v_1[k]$ and $m'[k] < m[k] \leq v_2[k]$. Since m' identifies the farthest common sub-path, the differences between $m'[k]$ and $v_1[k]$, as well as $m'[k]$ and $v_1[k]$, must have respectively resulted from concurrent tentative updates generated by replica k. However, replica k is not allowed to issue concurrent updates when holding the same $stableTS_k$: if $votes_k[k] = \perp$ (Section 3.3), it means that $nexistsi \neq k$ s.t. $votes_k[i]$ represents any tentative update from k. By anti-entropy, this is also verified at any other replica.

It follows from the definition of a fixed currency scheme, as mentioned in Section 3.1, that the total currency amount stored in cur_r at each replica r is lower than or equal to 1. We define the $uncommitted_r$ value at each replica r at a given moment to be:

Definition 2: $uncommitted_r = \sum cur[k] : votes_r[k] \neq \perp$

The voting algorithm is responsible for progressively determining common sub-paths of candidate versions that manage to obtain a plurality of votes. This decision is based on the definition of maximum common version among the set of candidate versions voted at a given replica and on the value of $uncommitted_r$, according to the following definition:

Definition 3: Let w be a version vector s.t. $w = mcv(v_1, .., v_m)$ where $v_1, .., v_m \in votes_r$. w wins an election when:

1. $voted_r(w) > 0.5$, or
2. $\forall k$ s.t. $k = mvc(v_{k_1}, .., v_{k_n}), v_{k_1}, .., v_{k_n} \in votes_r$ and $k \parallel w$,
   (a) $voted_r(w) > voted_r(k) + uncommited_r$, or
   (b) $voted_r(w) = voted_r(k) + uncommited_r$ and $w <_{lex} k$.

The above rules state the conditions that guarantee that a candidate has collected sufficient votes to win an election. The votes may constitute a majority, when the amount of currency voted on the winning candidate surpasses 0.5; or a simple plurality, when the voted currency is greater than the maximum potentially obtainable currency of any other rival candidate. Ties are decided by choosing the candidate whose version vector is lexically lower. If one represents each version vector as a number whose digits are the elements of the vector, such representation can be numerically compared, thus inducing a lexical order, $<_{lex}$, in the version vector space.

Determining if a candidate has won an election depends exclusively on information that is locally available at each replica. This means that, once having collected enough voting information, a given replica is able to decide, by its own, to commit a candidate version that locally fulfills the election winning conditions. Hence, update commitment is accomplished in a purely decentralized manner. An example is depicted in Figure 2.

After finding a new winner version vector, w, a replica r takes the following steps atomically in order

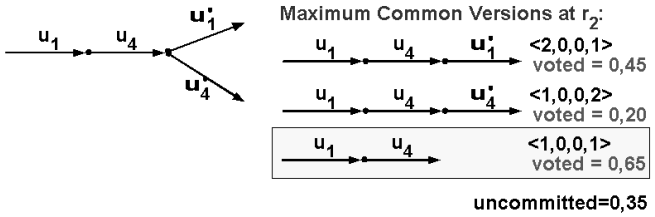

Figure 2: Election decision for replica r_2 at the final state in Figure 1. Among the potential maximum common versions, obtained from the candidates known by r_2, candidate $\langle 1,0,0,1\rangle$ is found to have collected a plurality of votes and, thus, updates u_1 and u_4 will be committed in that order.

to accept the election decision and prepare for the next election:

1. $stableTS_r \leftarrow w$;
2. $\forall v_k \in votes_r$ s.t. $v_k \| w$ or $v_k \leq w$, $votes_r[k] \leftarrow \perp$;
3. If the sequence of updates that comprise the update path defined between versions $stableTS_r$ and w is locally available, then commit it;

After accepting the election result by setting the winning version as the new stable version, the second step resets all the defeated candidates to $\perp$. Depending on the local availability of the updates that belong to the winning candidate, they may be committed into the replica's stable value; otherwise, further anti-entropy sessions will ensure that such updates are eventually collected and committed. A new election can then take place.

Theorem 2: After all elections have been completed at every replica and all updates belonging the resulting stable path have been committed at every replica in the system:

$\forall r, t$, replica r has committed the same ordered sequence of updates as t.

Sketch of Proof: The proof is outlined as follows. Assume that all replicas share a common initial stable version, $stable_0$. Let $w = mcv(v_1, .., v_m)$ where $v_1, .., v_m \in votes_r$ for a given replica r. Consider also that $v_1, .., v_m$ are the votes corresponding to the set of replicas S. Then, according to the anti-entropy procedure, for all replicas k, $\forall s \in S, votes_k[s]$ will either be $\perp$ or v, where $v \leq w$ or $v \geq w$. Now assume that r decides that w wins the election, thus setting $stableTS_r = w$. Hence, an amount of $cur_r(w)$ is either set to $\perp$ or v, where $v \leq w$ or $v \geq w$, at all the remaining replicas k, which means that no concurrent candidate, $c \| w$, will ever be declared winner, according to Definition 3. Therefore, all replicas must either (a) eventually collect the necessary votes from replicas S and accept w as winner or (b) receive information of the elections outcome from replicas whose $stableTS$ is already set to w. Since any candidate is enforced to be formed by a totally ordered sequence of updates and, by hypothesis, every update belonging to the stable path has already been committed, the set of updates belonging to winner candidate w is guaranteed to be committed in a total order at every replica.

4 Dynamic Version Vector Candidates

The replication protocol, as defined in Section 3, would suffer from two fundamental drawbacks when compared to a basic fixed currency weighted voting protocol [13]. Such drawbacks arise from the usage of static version vectors which, by definition, comprise one integer entry for each existing replica of the logical object in consideration.

Firstly, identifying replica versions would require maintaining a complete knowledge of group membership in what concerns the set of existing replicas in the system. In contrast, no such knowledge is imposed by a basic fixed currency weighted voting protocol, which facilitates the dynamic arrival and removal of new replicas. Secondly, while the latter uses simple integer values to represent candidates, static version vectors require an *N-entry* array of integers for the same function, where N is the number of replicas. Consequently, the size of voting information stored at servers and propagated among them at each election round is multiplied by N.

In order to solve the first problem and minimize the impact of the second, this section describes how a dynamic version vector (DVV) approach can be integrated into the consistency protocol. The basic idea behind DVVs is to represent only the entries corresponding to those replicas that are actively generating updates, i.e. whose update counter in a version vector is not zero-valued. Additionally, a DVV main-

tenance algorithm ensures that such collection of entries is confined to the minimal set of active replicas whose representation in a version vector is relevant to the underlying consistency protocol. As will be discussed, such algorithm is integrated into the weighted voting protocol defined earlier and neither affects its behavior and high availability nor imposes additional network messages to be exchanged.

As a result, the need for a complete knowledge of group membership over the set of replicas is eliminated, since replica creation and removal are implicitly handled by the dynamic nature of DVVs. Furthermore, the size of version time stamps is substantially reduced if the minimal set of consistency-relevant replica entries is kept small. As Section 5 describes, experimental evidence suggests that such assumption is correct in most real systems.

4.1 Dynamic Version Vectors

In contrast to static version vectors, a DVV is an associative array that contains a variable number of entries. Each entry is a pair $\langle replica : counter \rangle$, where *replica* is a replica identifier and *counter* is an integer that is equivalent to the entry for the given replica in a version vector.

Hereafter, we will use the notation $v[r]$ to represent the *counter* field of the entry in v that corresponds to replica r. Moreover, $v[r] = \perp$ means that v does not hold any entry for replica r, in which case replica r should be regarded as having a zero-valued counter.

DVVs are defined in an equivalent way to version vectors, as presented in Section 3.1.1. The only difference comes from the fact that entries with null counter values are absent in the representation of a DVV. Therefore, the initial version of an object is an empty DVV; also, $advance_r(dvv_k)$ causes the entry associated with replica k to be incremented if it is present in dvv_k or, otherwise, inserted with a counter value of 1.

Additionally, dynamic version vectors introduce the concept of vector compression, defined as follows.

Definition 4: *Vector compression.*

Let dvv_k and dvv_c be two DVVs so that $dvv_k \geq dvv_c$. The vector compression of dvv_k by dvv_c, represented by $compress_{dvv_c}(dvv_k)$, is a dynamic version vector defined by:

1. $\forall r$ s.t. $v_c[r] = i$ $(i \in N)$,
   (a) If $dvv_k[r] > i$, $compress_{dvv_c}(dvv_k)[r] = dvv_k[r] - i$;
   (b) Otherwise, $compress_{dvv_c}(dvv_k)[r] = \perp$;
2. $\forall r$ s.t. $dvv_c[r] = \perp$, $compress_{dvv_c}(dvv_k)[r] = dvv_k[r]$.

By treating absent entries as zero-counter entries, two DVVs can still be causally ordered by the happened-before relationship presented in Section 3.1.1, provided that both vectors have seen an identical sets of vector compressions [19]. The latter is a fundamental condition for the correctness of the causality statements that may be extracted from the comparison of DVVs.

4.2 Dynamic Vector Maintenance

As mentioned earlier, a DVV ought to comprise a variable number of entries that correspond to the minimal set of active replicas that are relevant to the underlying consistency protocol at each moment. In the case of a weighted voting protocol, such set is composed by the replicas that have issued the tentative updates that are still being voted at a given moment.

In order to accomplish such minimal representation, DVVs must be dynamically adjusted when: (a) a previously absent replica issues a candidate update or, inversely, (b) all the updates issued by a given replica have become committed and so its corresponding entry in the DVV becomes redundant. These events are respectively handled by vector expansion and compression.

4.2.1 Vector Expansion

Vector expansion is trivially achieved. When a given replica wishes to generate an update upon a version identified by DVV dvv_k, the new version's DVV is simply obtained by $advance_r(dvv_k)$. If that replica was not represented in the original DVV, dvv_k, this means that a new entry for the new writer replica is inserted in the new DVV. The propagation of voting information, performed by the underlying replication protocol, will then automatically propagate the expanded entry to the remaining replicas.

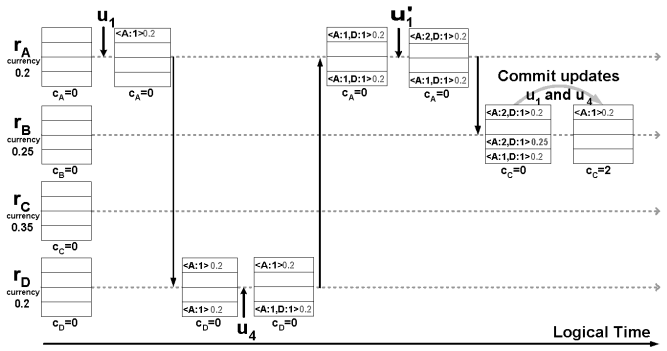

Figure 3: Example of update generation and propagation using tentative view, under the consistency algorithm with dynamic version vectors. Four replicas of the same logical object, r_A, r_B, r_C and r_D evolve towards the commitment of updates u_1 and u_4, accordingly restricting their DVVs to a minimal set of entries.

4.2.2 Vector Compression

On the other hand, vector compression is more challenging. In contrast to vector expansion, an entry from a DVV can only be removed if an agreement to do so is reached with every other replica. Otherwise, the causality relations between DVVs would not always be valid and so the replication protocol would not be correct.

Our solution accomplishes such requirement by relying on the update commitment agreement that is already provided by the underlying weighted voting consistency protocol. By taking into account the behavior of the weighted voting consistency protocol, the vector compression algorithm is able to overcome the minimality result by Charron-Bost [20], which assumes the case where no information is available about the data causality relations between replicas. Most importantly, our contribution neither (a) requires every replica to be contacted before purging a vector entry, nor (b) assumes a group management service that correctly detects failed replicas nor (c) introduces additional network traffic, as happens with alternative vector compression approaches [19, 21].

The basic idea consists of extending the update commitment phase with an implicit vector compression phase of all the DVVs stored by a given replica, concerning the entries that correspond to the replicas that have issued the committed updates. Accordingly, the steps taken by a replica r for committing a sequence of updates $u_1, .., u_n$, generated by servers $i_1, ..i_n$, as described in Section 3.2, are complemented as follows:

3. $stableTS_r \leftarrow compress_{v(u_1,..,u_n)}(stableTS_r)$;
4. $\forall k$ s.t. $votes_r[k] \neq \perp$, $votes_r[k] \leftarrow compress_{v(u_1,..,u_n)}(votes_r[k])$;

Consequently, as updates become committed, the representation of the DVVs stored at each replica is successively restricted to the entries that result from the updates that still remain uncommitted. Not only does this include the tentative updates represented by each election candidate, but also the updates that already comprise the stable version but have not yet been locally committed. Eventually, as the latter become collected and committed by anti-entropy, the local representation of DVVs is then reduced to the entries corresponding to the pending tentative updates.

In order to accommodate for vector compression, any comparison between version identifiers held by distinct replicas must take into account their possibly differing vector compression background. For this purpose, the definition of *DVV compression adjustment* is necessary.

Definition 5: *DVV compression adjustment.*

Given two replicas, r_i and r_j, with possibly differing commitment counters, respectively c_i and c_j, the compression adjustment of a DVV, dvv_j, held by r_j, against replica r_i, is a DVV defined as:

1. If $c_i < c_j$ then $adjust_{i,j}(dvv_j) = advance_{issuer_j[c_j]}(...(advance_{issuer_j[c_i+1]}(dvv_j))$;
2. If $c_i > c_j$ then $adjust_{i,j}(dvv_j) = compress_{v_\Delta}(dvv_j)$, where $v_\Delta = v(issuer_i[c_j + 1], .., issuer_i[c_i])$;
3. Otherwise, $adjust_{i,j}(dvv_j) = dvv_j$.

By recalling the information of past vector compressions that is available in the *issuer* structures held by each replica, the compression backgrounds of DVVs from distinct replicas are thus synchronized. In particular, if r_i has seen less vector compressions than r_j (1), their DVVs may be safely compared by

rolling back the additional compressed entries according to the information in $issuer_j$. Inversely, in case the DVVs held by r_i have seen a supplemental sequence of vector compressions (2), applying such sequence to the DVVs held by r_j will equally ensure a common vector compression background.

Accordingly, the anti-entropy procedure (described in Section 3.4), between a requesting replica r_A and another replica r_B, is simply redefined as follows in order to ensure the correctness of DVV comparisons:

1. If $stableTS_A < adjust_{A,B}(stableTS_B)$ then
   (a) $stableTS_A \leftarrow adjust_{A,B}(stableTS_B)$;
   (b) $\forall k$ s.t. $votes_A[k] \| stableTS_A$ or $votes_A[k] \leq stableTS_A$, $votes_A[k] \leftarrow \perp$;
2. If $(votes_A[A] = \perp$ and $stableTS_A < adjust_{A,B}(votes_B[B]))$ or $votes_A[A] < adjust_{A,B}(votes_B[B])$ then
   $votes_A[A] \leftarrow adjust_{A,B}(votes_B[B])$ and $cur_A[A] \leftarrow currency_A$;
3. $\forall k$ s.t. $(votes_A[k] = \perp$ and $stableTS_A < adjust_{A,B}(votes_B[k]))$ or $votes_A[k] < adjust_{A,B}(votes_B[k])$,
   $votes_A[k] \leftarrow adjust_{A,B}(votes_B[k])$ and $cur_A[k] \leftarrow cur_B[k]$.
4. If $c_A < c_B$ then commit update sequence issued by $issuer_B[c_A + 1], .., issuer_B[c_B]$

Figure 3 shows an example of the modified consistency protocol.

Theorem 3: After all elections have been completed at every replica and all updates belonging the resulting stable path have been committed at every replica in the system:

$\forall r, t$, replica r has committed the same ordered sequence of updates as t.

Sketch of Proof: A proof may be outlined as follows. Assume the lemma that the causality statements obtained by comparison of any version identifier (namely DVVs or the s_r measure), are correct in the consistency algorithm with vector compression. Therefore, the algorithm is equivalent to the one with static version vectors which, in turn, is correct. Finally, the lemma is proven in result of the following: (a) vector compressions within the DVVs of the same replica are performed atomically, so such DVVs have always a common vector compression history; and (b) any comparison between DVVs from distinct replicas during the anti-entropy procedure is guaranteed, by the application of the DVV compression adjustment, to be performed only after both DVVs have seen a common vector compression sequence.

5 Evaluation

This section presents and discusses simulated results that compare the behavior of our proposed solution to reference protocols mentioned in Section 2. *C#* implementations of the primary commit (*Primary*), basic weighted voting (*Basic WV*) and version vector weighted voting (*VVWV*) protocols were measured under a simulated environment.

The simulator includes a collection of replicas of a common logical object that are able to issue updates and mutually propagate such update information. Time is divided into logical time slices where each replica (1) pulls anti-entropy information from a partner randomly selected from the set of available replicas and, according to a certain probabilistic update model, (2) generates a maximum of one tentative update.

Each experiment was performed by running the three protocols side-by-side under the same exact conditions, until a total number of 20 updates had been committed or discarded by every replica. In order to obtain accurate estimates, each experimental setting was tested 10 times and the average measurements were considered. All measurements were obtained with the following basic parameters: 10 running replicas and a global probability of 70% that one among all replicas would issue one update at each time slice.

Three update models are analyzed. Firstly, a uniform update model where updates may be issued at any replica with the same probability. In this case, the update probability at every replica was of 7%. Furthermore, we consider two additional models that assume non-uniform update behavior, suggested from empirical evidence described in related work: a *hot-spot* update model, which assumes that updates typ-

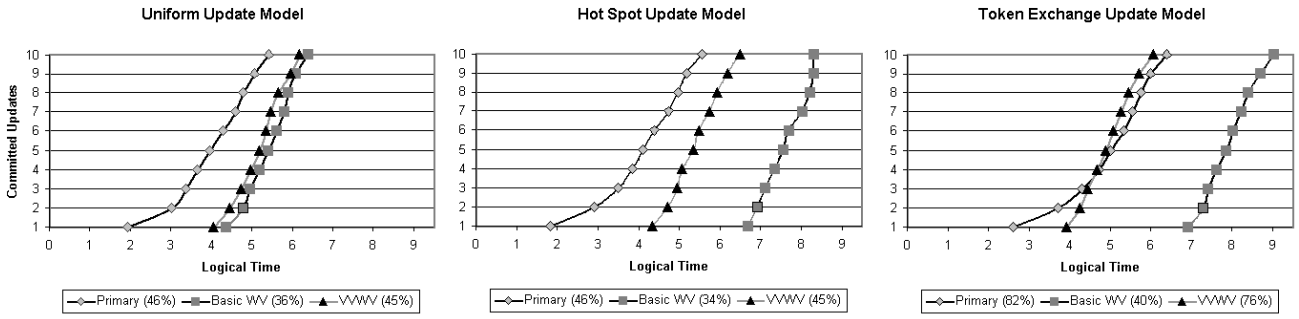

Figure 4: Update commitment times for the simulated update models with complete server availability. The measured commitment rates of each protocol are shown in parenthesis.

ically occur in a small set of replicas [5]; and a *token exchange* model, where users, through their social interaction and semantic knowledge, ensure that only a single up-to-date replica is, with greater probability, updated at each moment [6].

A first experiment measured commitment delay, i.e. the time taken between the creation of a tentative update and its commitment, for each protocol on each update model. Since, at every protocol, update commitment is a local decision of each replica, different commitment delays are observed by each replica, as shown in Figure 4. As expected, primary commit accomplishes the lowest average delay times, since it requires fewer messages to be propagated in order to commit an update, provided that the optimal anti-entropy partners are selected.

Regarding both weighted voting alternatives in a uniform update model, no significant difference is observed, which is entailed by the relatively low frequency of commitment of multiple update candidates. However, as one evolves to the non-uniform update models, where multiple update candidates are expectedly more frequent, our protocol achieves a substantial optimization in commitment delay (28.7% and 37.4% average reduction against basic weighted voting for hot-spot and token exchange models, respectively). The change induced by the non-uniform models is also verified with respect to the primary commit protocol, where the average commitment delay difference is strongly reduced: to 32.3% and 0.7% overheads, respectively. More precisely, in the case of the latter, an average of 7 replicas is able to commit an update more rapidly using our protocol than using a primary commit protocol.

Using the same experimental settings, the vector compression algorithm proves to be highly efficient in reducing the storage overhead of DVVs. At each time slice, the number of entries of the DVVs stored at each replica were measured. The average number of DVV entries was very approximate to 1, while the maximum number of entries ever held at a DVV was of 3 entries. Namely: in the uniform update model, an average of 1.1 and an average maximum of 2.8 entries; in the hot-spot update model, an average of 1.0 and an average maximum of 2.0 entries; and, in the token-exchange update model, an average of 1.2 and an average maximum of 3.0 entries.

Nevertheless, comparing update commitment delays is not sufficient to provide a complete view over the behavior of protocols since the discarded updates are not accounted. Furthermore, the issue of server availability needs to be considered as a central variable for the efficiency of each protocol.

With this intent, a second experiment was performed in which each server had a parameterizable probability of becoming disconnected from the remaining group of servers at the beginning of each time slice. Each such disconnected server was then unable to communicate with other servers but was still allowed to issue updates. With a 10% probability, a disconnected server would again become available at the end of each time slice. Such probability yields an expected disconnection time of 10 time slices, which was considered to be acceptable for the usage scenarios that are mainly addressed by this paper.

Figure 5 shows the update commitment rate of each protocol for different server disconnection probabilities in each update model. As expected, the update commitment rate is strongly dependent on the

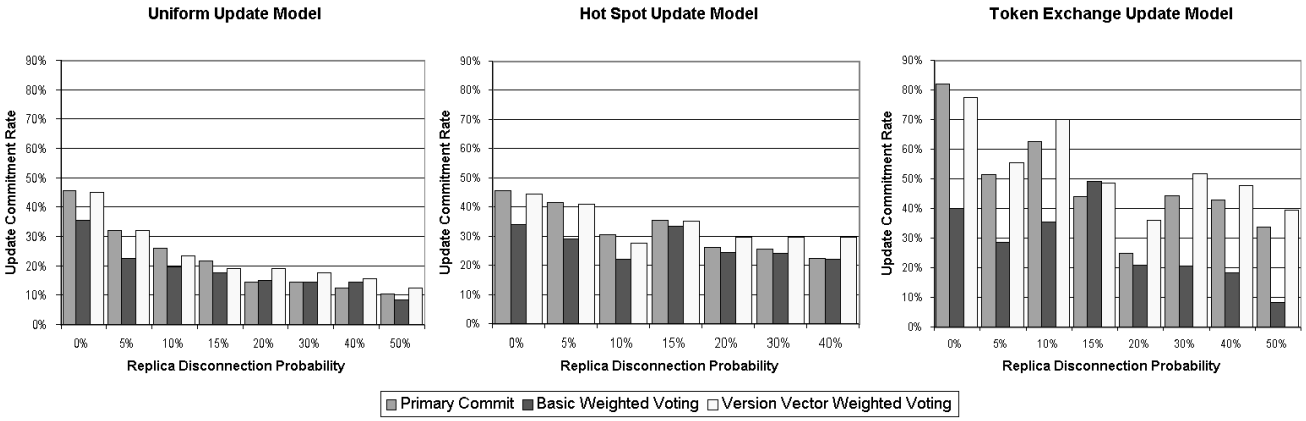

Figure 5: Commitment rates for the simulated update models in the presence of temporarily disconnected servers.

time a protocol takes to commit tentative updates. A lower commitment delay of the latter means that updates will be in a tentative state for a shorter duration, therefore reducing the possibility of tentative update concurrency and, consequently, the rate of discarded updates. With 0% disconnection, update commitment rates at all update models are exclusively determined by the average commitment delays analyzed in the first experiment: primary commit achieves a slight advantage over our protocol (0.1%, 1.0%, 4%), increased with respect to the basic weighted voting protocol (10.0%, 8.0%, 42.0%).

However, with non-null disconnection probabilities, the impact of server unavailability becomes very influential on the efficiency of the protocols. As a result, the observed commitment rate of every protocol decreases as the disconnection probability grows. In the case of the primary commit protocol, this is explained by the consequences of occasional unavailability periods of the primary replica to the commitment progress; on the other hand, weighted voting protocols take longer to reach a plurality quorum when the number of available voters is reduced.

Nevertheless, it is suggested that the impact of temporary unavailability of servers has a greater impact in the case of the primary commit protocol, due to its reliance on a single point of failure. In particular, the observed commitment rate of the primary commit protocol is lower than that of our protocol for most of the disconnection situations shown in Figure 5.

6 Conclusions

Mobile and loosely-coupled environments call for decentralized optimistic replication protocols that provide highly available access to shared objects. A fundamental property of most optimistic protocols is to guarantee an eventual consensus on a commit order among the set of tentatively issued updates so as to deliver eventual strong guarantees to applications.

In this paper we propose a replicated object protocol that employs a novel epidemic weighted voting algorithm based on version vectors for achieving such goal. This algorithm introduces an optimization over the basic weighted voting solution by allowing multiple causally ordered update candidates to be committed at a single election round. Complementarily, the paper discusses how a dynamic version vector maintenance algrithm is incorporated into the protocol so as to effectively eliminate the need for a complete knowledge of group membership and to minimize the storage overhead of version vectors.

From the results obtained from a side-by-side execution of reference protocols in a simulated environment, we demonstrate that our solution is advantageous in realistic non-uniform update models, both with respect to basic weighted voting and primary commit protocols.

References

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

[2] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.

[3] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Fault Tolerant Computing Symposium (FTCS)*, June 1996.

[4] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. In *ACM Transactions on Database Systems*, volume 15, pages 230–280, 1990.

[5] D. Ratner, P. Reiher, and G. Popek. Roam: A scalable replication system for mobile computing. In *Mobility in Databases and Distributed Systems*, 1999.

[6] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Softw. Pract. Exper.*, 28(2):155–180, 1998.

[7] Yasushi Saito and Marc Shapiro. Optimistic replication. Technical Report MSR-TR-2003-60, Microsoft Research, October 2003.

[8] 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, February 1993.

[9] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[10] J. Barreto and P. Ferreira. A replicated file system for resource constrained mobile devices. In *Proceedings of IADIS International Conference on Applied Computing*, 2004.

[11] U. Cetintemel, P. J. Keleher, B. Bhattacharjee, and M. J. Franklin. Deno: A decentralized, peer-to-peer object replication system for mobile and weakly-connected environments. *IEEE Transactions on Computer Systems (TOCS)*, 52, July 2003.

[12] David H. Ratner, Peter L. Reiher, Gerald J. Popek, and Richard G. Guy. Peer replication with selective control. *Lecture Notes in Computer Science*, 1748, 1999.

[13] P. Keleher. Decentralized replicated-object protocols. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)*, 1999.

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

[15] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[16] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

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

[18] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, September 1979.

[19] David Howard Ratner. Roam: A Scalable Replication System for Mobile and Distributed Computing. Technical Report 970044, University of California, 31, 1998.

[20] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991.

[21] Richard A. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, Computer and Information Sciences Board, University of California at Santa Cruz, 1992.