

Decentralized Replicated-Object Protocols

Peter J. Keleher
keleher@cs.umd.edu
Department of Computer Science
University of Maryland
College Park, Maryland 20742

We describe a new replicated-object protocol designed for use in mobile and weakly-connected environments. The protocols use epidemic information propagation. The advantage of epidemic protocols is that data movement only requires pairwise communication. Hence, there is no need for a majority quorum to be available and simultaneously connected at any single time. The protocols increase availability by using voting, rather than primary copy or primary server schemes. Finally, the use of per-object voting currencies allows votes to take place in an entirely decentralized fashion, without any server having complete knowledge of group membership.

We show that currency allocation can be used to implement diverse policies. For example, uniform currency distributions emulate traditional dynamic voting schemes, while allocating all currency to a single server emulates a primary-copy scheme. We present simulation results showing both schemes, as well as the performance advantages of using currency proxies to temporarily reallocate currency during planned disconnections.

1. Introduction

We describe the use of currency-based epidemic algorithms in improving the performance of replication protocols in weakly-connected and mobile environments. Our algorithm description will be presented in the context of Deno, a replicated-object system intended for use with mobile and or weakly-connected hosts. We assume a system that consists of a series of peer shared-object servers, each capable of caching replicas of any object in the system. The protocols discussed in this work assume peer servers with no designated primary copy [6] for any object. By default, all replicas of a given object are equally able to create new updates for the object, and to have them committed.

Replicas are useful for many reasons, including efficiency, availability, and fault tolerance. Replicas increase efficiency by allowing a local rather than a remote copy to be accessed, much in the same way that accessing a processor's memory cache is much faster than accessing memory over the computer's I/O bus. Replicas improve availability by making it possible for applications to make progress even when one or more replicas become temporarily unavailable. Fault tolerance is achieved by ensuring that object data is kept consistent. Loss of any one replica does not result in committed updates being lost if other replicas have copies of the same updates.

The problem with replicas is that they must be kept consistent. Consistency is problematic in distributed systems because updates of multiple sites are generally non-atomic operations. Different sites usually take differing amounts of time to access, meaning that competing *tentative* updates may be seen in different orders at different updates sites. However, consistency requires that any competing updates to the same shared object be *committed* in the same serial order at every replica.

A canonical primary-copy scheme orders updates by when they arrive at the primary copy's server. This is designated as the only correct order, and updates are required to be applied in this order at every replica. This approach has two drawbacks. First, the primary copy can become a performance bottleneck for updates to the object. More importantly in the context of a distributed environment, no updates can be

committed, and no application progress made, without contacting the primary copy. Unavailability of the primary copy brings the entire system to a halt. Administrators often try to minimize the possibility of this occurrence by ensuring that the primary copy resides on a trusted server, protected by a firewall and safeguarded by elaborate battery-backup systems. Any other copy connected to the corporate intranet can communicate with the primary copy.

Unfortunately, progress often needs to be made outside of the corporate boundaries. For example, IBM sales staff have traditionally been expected to be on the road so much that they did not even have offices. If salespeople Frank, Joe, and Nancy collectively cover the state of Texas, they might expect to be able to consolidate their sales data when they meet in Austin. Off-the-shelf hardware like WaveLAN would allow them to open their laptops in a conference room and instantly establish a local network between their machines. Unfortunately, even though all interested parties are present, no updates to shared data can be committed if the primary copy resides in a mainframe in New York. Consider the other alternative: locating the primary copy on one of their machines, such as Nancy's. Problems arise if Nancy then heads to California for a regional sales meeting. Even if Frank and Joe immediately proceed back to New York to update the corporate database, they can not commit any new data until Nancy returns from California.

This area has been the subject of a great deal of recent interest [1-5]. Protocols with widely varying properties have been proposed and implemented in a variety of systems. Many of these systems use a primary copy or commit scheme, also called a *monarchy* [7]. This approach relies on a single distinguished replica to serialize all commits of object updates, effectively holding forward progress in the system hostage to the availability of a single server. One can make the claim that progress is still possible while the primary copy is disconnected because new updates can be generated, just not committed. Various session control guarantees [8] allow such *tentative* updates to be seen by the application or user even before commitment. However, no "progress" can be made in such cases for applications that wish only to see committed data, which is probably the common case.

Dynamic voting schemes [4, 9, 10] eliminate the single point of failure by allowing a *quorum* of all replicas to commit an update. Quorums are distinct sets that can each commit an update, provided that all replicas of the quorum agree. Serialization of updates is accomplished by requiring that any two potential quorums must share at least one replica. Hence, competing updates can not both be committed without first being serialized by the replicas in the intersection of the quorums that commit them. Dynamic linear voting [11] extends the canonical majority voting schemes with an a priori linear ordering on quorums that can be used to break ties between equal-sized groups of servers. Voting has been shown to provide optimal availability when all processors have the same independent failure probability of less than $\frac{1}{2}$ [12].

This paper has two central contributions. First, we describe how to extend voting schemes through the use of per-object *currencies* [13-15]. An object currency is a fixed amount of currency that is divided among all replicas of a single object. The amount of currency held by a given replica is used as that replica's weight during voting rounds. Replicas do not necessarily have complete information on the amount of currency allocated to other replicas, and currency allocation is not static. Nonetheless, updates can be committed without complete knowledge of the votes of all replicas because the amount of currency remains fixed during failure-free operations. Currencies therefore allow votes to take place in a decentralized fashion, without any server having complete knowledge of group membership. Furthermore, currencies allow the behavior of the protocol to be fine-tuned to match expected system and application behavior. For example, appropriate currency allocation can cause the protocol's behavior to approximate that of a primary-copy or monarchy system.

Second, we use these currencies to allow voting to take place through a pairwise *epidemic* protocol. Currency-based epidemic protocols can make progress and *eventually* commit object updates even if there is never a majority of replicas connected to each other simultaneously. Epidemic protocols [16, 17] are appropriate for situations in which all replicas need to eventually be made consistent, and where disconnections are frequent.

In addition to the description of the new protocol, we provide simulation results showing that currency allocation can be used to implement diverse policies. For example, uniform currency distributions emulate traditional dynamic voting schemes, while allocating all currency to a single server emulates a

primary-copy scheme. We present results showing the rate at which both schemes commit updates, as well as the performance advantages of using *currency proxies* to temporarily reallocate currency during planned disconnections.

2. Theory

We assume a model in which the shared state consists of a set of objects that are replicated across multiple servers. Objects do not need to be replicated at all servers, and servers may replicate multiple objects. For simplicity of presentation, however, we limit our discussion to single objects that are cached at all servers. Our discussion is easily extended to include the more general case.

Objects are modified by *updates*, which are issued by servers. An update consists of either a code fragment or a run-length encoding of binary changes. Updates can be transmitted to other servers and are assumed to execute atomically at remote sites. Given a consistent initial state, application of the same updates in the same order on multiple replicas of the same object result in the same final object state.

Updates do not commit globally in one atomic phase because we assume an epidemic style of updates and poor connectivity. Instead, each server commits updates based on local information. However, we show below that any update that commits at any server eventually commits everywhere, and in the same order with respect to other committed updates.

2.1 Elections

A clean way of thinking about update commitment is as a series of elections. Each new update is a candidate, a server is a voter, and a committed update is said to win the election. Candidates win elections by cornering a plurality of the votes. Each election begins with an underlying agreement of the winners of all previous elections. Once an election is over, a new election commences. Any given election may have multiple candidates (logically concurrent tentative updates), and candidates from different elections might be alive in the system at the same time. In the latter case, however, candidates for any but the most recent election have already lost, but this information has not yet made it to all voters.

Because of the style of information flow, there is no centralized vote-counting. Instead, each voter independently collects votes from other voters and deduces outcomes. In order to implement this protocol, each voter v_i maintains three pieces of state:

1. *completed* – the number of elections completed locally, and
2. *votes[n]* – the set of votes for the current election that are known to v_i , where n is the total number of voters. A vote consists of the index of the server that created the update (candidate). We represent v_i 's knowledge of v_j 's vote in the current election as $v_i[j]$.
3. $v_i.curr_x[n]$ – The amount of currency held by each server in election x . The specific election defaults to the current election if left unspecified. Currency allocation may change with each election. Therefore, $v_i.curr[j]$ might not be known until $v_i[j]$ is known.

The value of any $v_i[j]$ is either the index of the candidate voted for by v_j , or \perp , which means that v_i has not yet seen a vote from v_j . The total amount of currency in any election is 1.0.

Definition 1: Define $uncommitted(v_i)$ as: $\sum_{j=1}^n v_i.curr[j]$, s.t. $v_i[j]$ is equal to \perp .

Definition 2: Define $votes(v_i, k)$ as $\sum_{j=1}^n v_i.curr[j]$, s.t. $v_i[j]$ is equal to k .

Definition 3: A candidate j wins (update created by server j commits) with voter (at server) v_i when:

1. $votes(v_i, j) > 0.5$, or
2. $\forall k, votes(v_i, k) + uncommitted(v_i) < votes(v_i, j)$, or
3. $\forall k, votes(v_i, k) + uncommitted(v_i) < votes(v_i, j)$ or
 $((votes(v_i, k) + uncommitted(v_i)) = votes(v_i, j) \text{ and } (j < k))$

Definition 3 essentially says that a candidate wins with a voter if it has a majority or plurality of the vote. Ties are broken with a simple deterministic comparison of server ids. The winner of the j^{th} vote at v_i is denoted $v_i.\text{commit}(j)$. When an election is won at v_i , all votes $v_i[j]$ are reset to \perp .

It follows naturally from the above definitions that candidates can win without all the votes being known. Similarly, updates can be committed by a server without complete knowledge of which servers have seen the update, or even complete knowledge of which servers cache the object.

2.2 Anti-entropy

Election information flows from voter to voter through anti-entropy sessions. In terms of elections, an anti-entropy session is a uni-directional flow of information specifying elections that have been won, and votes in the current election. More specifically, an anti-entropy session from v_i to v_j causes the following events to occur as a single atomic unit:

1. If $v_i.\text{completed} > v_j.\text{completed}$, then $v_j.\text{completed} = v_i.\text{completed}$ and $\forall k, v_j[k] \leftarrow \perp$, and:

$$\begin{array}{l} v_i.\text{commit} \\ \forall \\ k = v_j.\text{commit} + 1 \end{array} v_j.\text{commit}(k) \leftarrow v_i.\text{commit}(k) .$$
2. If $v_j.\text{completed} = v_i.\text{completed}$, then $\forall k$ s.t. $v_j[k] = \perp, v_j[k] \leftarrow v_i[k]$.
3. If $v_j[j] = \perp$, then $v_j[j] \leftarrow v_i[j]$.

The first rule states that if v_i is aware of the outcome of more elections than v_j , v_j accepts these results as a given, without waiting to find out the specific votes that caused these outcomes to occur. The second rule says that if both voters are holding the same election, then v_j will copy all of the votes known to v_i that it does not yet know itself. The final rule says that if v_j has not yet voted, it will vote the same as v_i . In both of these last two rules, the ‘‘vote’’ being copied may be \perp . However, as this value is only overwriting \perp in the first place, no consistency problems occur.

2.3 Becoming a candidate

Voters may become candidates (new updates may be created) in any election at any time, provided that:

- a) the election has not been decided for that voter yet, and
- b) the voter has not yet voted in the election.

Becoming a candidate merely consists of setting $v_i[i]$ to i .

2.4 Correctness

Given the above definitions, we can finally show that distinct voters arrive at the same election results.

Theorem 1: After all elections have been completed by all voters:

$$\forall i, j, k: v_i.\text{commit}(k) = v_j.\text{commit}(k).$$

Sketch of Proof: For reasons of brevity, we present only a proof outline. The proof proceeds along the following lines. Given the probabilistic way in which anti-entropy targets are chosen, all votes will eventually propagate to all voters. Second, if $v_i[j] = k$, for any i, j , and k , then $v_l[j]$ will be either k or \perp for all other voters l . Assume v_i commits update k . Let S be the set of servers that v_i records voting for k . For all servers l , $v_l[j]$ must be either k or \perp , for all j in S . Therefore, the currency represented by these servers either has to be recorded as voting for x or as uncommitted. In either case, this amount of currency prevents Definition 3 from allowing any other update to be committed. Therefore, all servers must eventually deduce the same outcomes, or be told of the common outcome by other voters, and will come to the same conclusions.

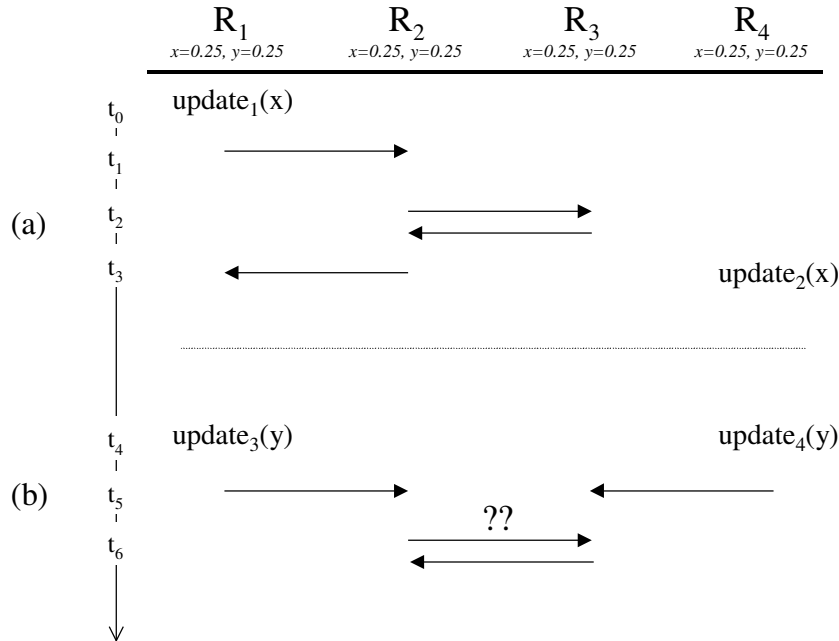


Figure 1: Four replicas each of objects x and y . Currency is divided evenly for both replicas. (a) shows the progress of an update from R_1 . The update is committed because a majority of the object’s currency “sees” it before any competing update. (b) shows two competing updates to y . At time t_6 , each update has been seen by replicas with a combined currency of 0.50.

3. Practice

Updates are either *tentative* or *committed*. Newly created updates are tentative, and may be rolled back without ever being committed. Tentative updates may or may not be visible to the application, depending on the type of session guarantees needed by the application. Updates are *committed* when servers holding a plurality of the object’s currency agree that they are acceptable.

Consider Figure 1 (a). Objects x and y are replicated at sites R_1 through R_4 . Each site has currency of 0.25 for both objects. R_1 creates an update to x at time t_0 . At this point the update is tentative. At time t_1 , R_1 sends information to R_2 , and at time t_2 , R_2 sends to R_3 . At this point, three of the four replicas know of the tentative update and have ordered it before any other tentative updates to x . These replicas can be said to have voted to commit $update_1$ because they control 75% of the object y ’s currency. However, R_3 knows this. In fact, R_4 does not even yet know of the update’s existence. This makes it possible for R_4 to naively create a new update, $update_2$ at time t_3 . This update will be aborted at t_5 when R_4 learns that a quorum has already determined that the next committed update to x will be $update_1$.

Figure 1 (b) shows an example of two competing updates being started at time t_4 . Each synchronizes with one other replica at t_5 , leading to a stalemate in which each competing update has 50% of the currency. While currency allocation schemes could be rigged to prevent this from occurring in the case of two competing updates, three or more competing updates could still lead to the same problem. The lexicographic tie-breaker will favor $update_3$ over $update_4$.

3.1 Voting

Deno’s replication protocol makes few assumptions on the completeness of available replica information. For example, Deno propagates updates to shared objects in the absence of knowledge of the complete set of replicas, or even of a primary copy that has pointers to all extent replicas. This problem, and many others, is greatly complicated by the peer-to-peer communication. This communication pattern results in data moving slowly through the system, one step at a time.

Objects are initially created with a total currency of 1.0, which is held by the creating server. New replicas are created by sending requests to servers that have existing replicas. The response to such requests contains both the object's data and some amount of currency. This amount is subtracted from the currency held by the existing replica. The total amount of currency in the system remains constant during failure-free operation.

Going back to the example discussed in Section 1, assume that each replica has an equal amount of currency. Any three replicas control 75% of the currency, and can conclude that no other set of replicas is concurrently committing updates to the same object. Hence, they can commit updates and application progress can be achieved.

Progress is achieved in the above examples because one set of replicas had more than half of the currency. What happens if two disjoint sets of replicas each have exactly half of the currency? More generally, consider the case where multiple tentative updates each gain currency support of less than 50%, but all currency is consumed.

We handle conflicts by generalizing the quorum-voting scheme to commit updates that fail to achieve a majority. An update can be committed if no other update can garner more currency, *and* the update is chosen by the tie-breaking procedure. Deno breaks ties through a lexicographic comparison between the server ID's of the servers that created the updates. This procedure does not require the participation of all replicas, but it does require that the amount of unaccounted-for currency not be enough to change the update chosen to be committed. Conflicting updates can therefore slow the process of committing updates because more complete information is needed.

It is also worth noting that the primary copy and voting approaches to update commitment are not necessarily mutually exclusive. Currencies can be allocated in ways that prefer quorums containing specific replicas, or more than half of the currency can be retained by a given replica. The latter situation reduces to a primary copy scheme.

3.2 Currency allocation

Timely update commitment depends on being able to assemble a quorum to vote on updates. The cost of assembling a quorum is highly dependent on the availability and currency distribution of the object replicas. There are a number of different strategies that could be pursued in currency allocation. The best choice can depend on application semantics, expected availability of individual servers, and network topology. A peer-to-peer application might work best with currency evenly distributed among the replicas, while a client-server application might work better if any one client and the server together constitute a quorum. Note that a uniform distribution of currency is not necessarily easy to achieve unless the number of replicas is known. Even if the number of replicas is known a priori, poor distributions can result when replicas are created by other than the first replica. The problem is that currency is split between any new replica and the replica that created it. Unless the existing replica has twice the eventually desired average currency, both will have only half the desired values.

Deno applications can direct currency allocation by providing a hint at object creation as to how many replicas are expected to be created. This hint allows Deno to allocate currency to replica requests in a way that provides a uniform level of currency for the expected number of replicas. For this to work, new replicas must be created from the original replica.

3.3 Proxies

Proxies are often used to represent unavailable devices in distributed systems. A *primary* can engage a proxy to vote in its place in commit quorums. The use of proxies can prevent degradation in the overall commit rate when devices have expected, planned-for disconnections. In fact, proxies can even improve commit latency because currency is concentrated in fewer servers, and fewer rounds of communication are required to establish a quorum. An example where proxies would be useful is when a laptop is taken on a trip where no other servers will be available. The laptop's currency can be transferred to a desktop machine for the trip's duration.

There are two obvious approaches to including proxies in currency-based replication protocols. The first is to explicitly transfer currency to the proxy. The proxy's weight in subsequent votes temporarily increases to encompass both its own currency and that of the proxy's primary. One drawback is that proxies become visible to all servers. Problems can arise from race conditions between the information about a proxy being engaged or disengaged, and tentative updates.

A less intrusive approach is to have the proxy tell other servers that the primary's vote is the same as its own while the proxy is engaged. A proxy vote is then indistinguishable to other servers from the situation where a server votes and then disconnects. When a primary reconnects, it updates its own information to match that of the proxy, including votes on prior and current tentative updates. The primary treats any votes cast in its behalf as if they had been cast directly. In particular, any votes cast for tentative updates remain cast. The result is that there are no race conditions, and the entire proxy engagement is transparent to the rest of the system.

Proxies whose primaries fail can permanently vote the primary's currency. The advantage of this approach is that even the failure is transparent to the other servers. The orphaned data structures will continue to collect in long-running computations as more servers fail. A garbage-collection mechanism could periodically reclaim data structures pertaining to failed servers.

The default behavior can be used to deal with proxies that fail. Consider a primary that reconnects, only to find that its proxy has failed. If a failure update for the proxy has been committed, but no such update has been committed for primary, the primary can immediately resume voting without further mechanism. If failure updates have been committed for both, the normal mechanism for reconstituting failed servers is used.

3.4 Failure detection and handling

Simple timeouts are not workable in the domain of mobile computing because disconnection is the rule rather than the exception. Disconnections are not only potentially frequently, but might be quite lengthy. A second approach is to count the updates that commit without a vote from the server in question. Note that votes will be seen from disconnected servers with proxies, so this method will only identify servers that disconnected or failed unexpectedly.

Servers can individually detect (via timeouts) failures and remove failed servers from their tables. However, currency is lost when servers fail without designating proxies. Loss of this currency can either slow or completely prevent updates from being committed. The protocol can compensate for lost replicas by *revaluating* the currency. Revaluation does not change the absolute amount of currency held by any server. Instead, it changes the thresholds at which an update can be committed. If currency in the amount of 0.25 is lost, a revaluation would allow updates to commit at a value of .375 in the absence of contention.

As with other changes to objects, a currency revaluation is a special type of update operation on an object. Revaluations must be committed before they can take effect. One implication is that revaluation can only occur if at least 50% of the current currency is available to vote. This is necessary to prevent parallel currency revaluations after network partitions.

3.5 Anti-entropy

The pairwise communication between servers in epidemic protocols is called anti-entropy because each such session reduces differences between servers, thereby decreasing total entropy. A Deno anti-entropy session consists of one server, s_1 , picking a second server, s_2 to pull information from. The selections of s_2 will initially be made at random among other servers known to s_1 . However, this choice could be skewed according to some scheme that eliminates redundancy in highly-available environments. A server votes for the first update for an object that it "sees" after the last was committed.

The initiating server *pulls* information from the responding server. This is in contrast to a server *pushing* information to another server. Pushing information is inexpensive, and allows a number of non-traditional transmission mediums, such as floppies, email, or satellite transmission.

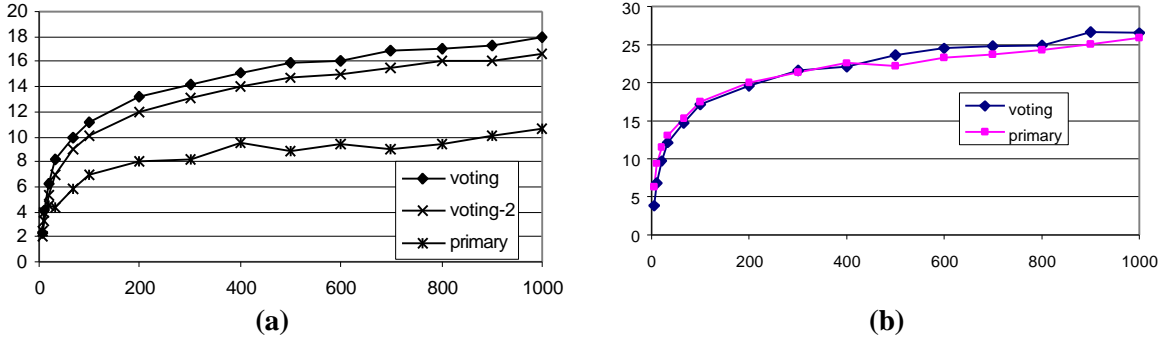


Figure 2: Commit rates: (a) shows the average number of intervals needed for the first replica to commit an update versus the number of replicas for the default voting scheme, voting assuming reliable communication, and a primary-copy scheme. (b) shows the number of intervals for *last* replica to commit updates.

However, pulling information allows the initiating server to summarize its state to the responding server. This summary allows the responding server to respond with only new information. By contrast, push transmissions have to be conservative about underlying assumptions of the data that has been seen by the destination. Without any knowledge of the destination, the initiator of a push would have to send all updates in order to ensure that any of the information can be used. Consider the alternative. If the initiator of a push transfer knows of 20 updates to object x and assumes that the destination must know of at least 10 of the updates, it will only transmit updates 11 through 20. However, if the destination had only seen the first 9 updates, it can not use any of the later updates because updates must be applied in order. The result is that push transfers tend to be conservative, and result in wasted resources.

Another advantage of pull transfers is that tend to commit updates more quickly than push transfers [16]. Let p_i be the probability that a server has not seen a new update after the i^{th} interval after the creation. Then the probability that the server has not seen the update after the $i+1^{\text{th}}$ iteration is just:

$$p_{i+1} = p_i^2$$

which converges rapidly. The corresponding recurrence for pushes is:

$$p_{i+1} = p_i \left(1 - \frac{1}{n}\right)^{n(1-p_i)}$$

This second recurrence converges (commits updates) more slowly than the first. Deno supports push transfers as well as pull, but uses pulls by default.

Note also that servers can transparently gift other servers with currency, allowing the system to stabilize in a state with uniform currency distribution regardless of the initial configuration. However, care must be taken to ensure that knowledge about the currency transaction moves with at least as fast as knowledge of any vote. In other words, changing currency requires each “vote” to be accompanied by the amount of currency held by the server when the vote was made. Additionally, care needs to be taken to avoid transferring currency from a server that has voted on a given update to one that has not.

4. Simulation

This section provides some simple verification of our intuitions of Deno’s protocol performance. We simulate a simple system in which time is broken into uniform intervals. Each server initiates a randomly-directed anti-entropy session during each interval. The initial metric of interest is commit speed versus the number of servers.

Figure 2 (a) shows a plot of the average number of intervals needed to commit an update versus the number of servers. We assume uniform distribution of currency and a completely available, fully-connected system. We show three protocols: “primary” is a simple primary copy scheme with a randomly chosen primary copy, “voting” is Deno’s default voting scheme, and “voting-2” is this same scheme assuming a reliable underlying communication protocol. Reliable communication allows the responding server to accu-

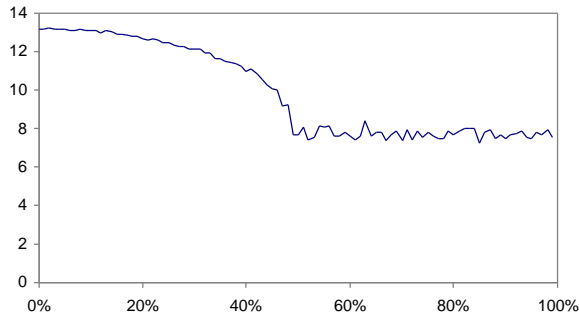


Figure 3: Commit cost (in intervals) versus percentage of currency given to single replica (200 replicas).

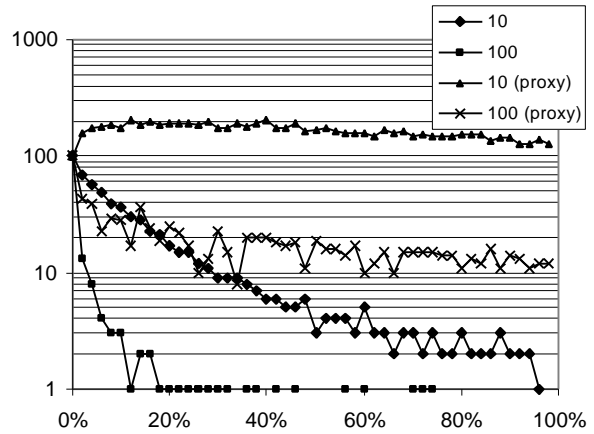


Figure 4: Committed updates in 2000 intervals versus unavailability: The x axis is the probability that a server will disconnect in any given interval.

rately predict when the initiating server will vote for an update based on the responding server’s information. This results in slightly faster information propagation, but the resulting performance is still short of the primary copy scheme. This is to be expected, as a primary-copy scheme can potentially commit updates with much less communication.

However, the time at which the *first* server commits an update is not necessarily the quantity that best predicts application performance. Since all servers have an equal chance of being read, a second interesting metric would be the time at which the *last* server commits an update. Figure 2 (b) shows that the rate at which the Deno’s protocol commits updates everywhere in the system is virtually identical to that of the primary copy. The metric of most use to applications probably lies somewhere between the two.

Deno’s currency mechanism allows currency allocation to be used in tuning protocol performance. Figure 3 shows commit costs (first commit) versus the degree to which object currency is skewed towards a single replica. A skew of 0% results in the default uniform distribution of currency. A skew of 100% emulates a primary-copy scheme. The plot suggests that a sophisticated replication protocol might benefit from skewing currency towards a single copy in times of high connectivity, and from smoothing out the distribution during times of low connectivity.

Measuring the availability of an epidemic protocol is not necessarily well defined. The availability of a typical quorum protocol is the percentage of time that a quorum is simultaneously connected and able to communicate. However, epidemic protocols do not require any server to be able to talk to more than one other server in order to make progress. While this implies that availability might be a poor metric, we can capture the affects of disconnections by looking at its effect on commit rates.

Figure 4 shows plots of commit rates versus the probability that a server will disconnect in any given interval. The commit rates are from 2000-interval runs, with new updates created every 20 intervals if previous updates have been committed. Disconnection probabilities are assumed to be uniform. We show curves for two different disconnection durations, and both with and without proxies. Section 3.3 alluded to the fact that the use of proxies can actually improve performance by effectively concentrating the currency in fewer replicas. This can be seen in the line for duration 10 with proxies. Proxies dramatically improve commit rates for both durations.

Note that these availability curves assume independent failures, i.e. no network partitions. One of the main advantages of voting schemes is that a single network partition can not prevent updates from being committed. Multiple partitions can be tolerated if they do not result in a single partition of less than half of the original replicas, or if the revaluation protocol is run between partitions.

5. Conclusions

We have described the replication protocol used in Deno, a replicated-object library intended for use in mobile and weakly-connected environments. Deno’s replication protocol uses voting and per-object cur-

rencies to commit object updates, and anti-entropy sessions to propagate information. Anti-entropy sessions allow progress to be made during times of very low connectivity. The use of per-object currencies allows updates to be committed in the absence of complete information about servers, and provides fine-grained control over the protocol behavior.

6. References

- [1] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing Update Conflicts in a Weakly Connected Replicated Storage System," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
- [2] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible Update Propagation for Weakly Consistent Replication," in *16th ACM Symposium on Operating System Principles*, Saint-Milo France, October 1997.
- [3] E. Y. Lotem, I. Keidar, and D. Dolev, "Dynamic Voting for Consistent Primary Components," in *17th ACM Symposium on Principles of Distributed Computing*, June 1997.
- [4] Y. Amir and A. Wool, "Evaluating Quorum Systems over the Internet," in *Fault-Tolerant Computing Symposium (FTCS)*, June 1996.
- [5] M. Herlihy, "A Quorum-Consensus Replication Method for Abstract Data Types," in *TOCS*, February 1986.
- [6] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRESS," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 188-194, May 1979.
- [7] Y. Amir and A. Wool, "Optimal Availability Quorum Systems: Theory and Practice," *Information Processing Letters*, vol. 65, pp. 223-228, April 1998.
- [8] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session Guarantees for Weakly Consistent Replicated Data," in *3rd International Conference on Parallel and Distributed Information Systems (PDIS 94)*, September 1994.
- [9] J.-F. Pâris and D. D. E. Long, "Efficient Dynamic Voting Algorithms," in *Proceedings of the Fourth International Conference on Data Engineering*, February 1988.
- [10] S. Jajodia and D. Mutchler, "Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database," *ACM Transactions on Database Systems*, vol. 15, pp. 230-280, 1990.
- [11] S. Jajodia, "Managing Replicated Files in Partitioned Distributed Database Systems," in *IEEE International Conference on Data Engineering*, 1987.
- [12] D. Peleg and A. Wool, "The availability of quorum systems," *Information and Computation*, vol. 123, pp. 210-223, 1995.
- [13] C. A. Waldspurger and W. E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management," in *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.
- [14] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A Wide-Area Distributed Database System," *VLDB Journal*, vol. 5, pp. 48-63, 1996.
- [15] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, "Spawn: A distributed Computational Economy," *IEEE Transactions on Software Engineering*, vol. 18, pp. 103-117, February 1992.
- [16] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the 6th Symposium on Principles of Distributed Computing*, August 1987.
- [17] M. Rabinovich, N. H. Gehani, and A. Kononov, "Scalable Update Propagation in Epidemic Replicated Databases," in *International Conference on Extending Database Technology (EDBT)*, 1996.