Consistency Management in Deno

Peter J. Keleher and Ugur Cetintemel
Department of Computer Science
University of Maryland
College Park, Maryland 20742
{keleher,ugur}@cs.umd.edu

We describe a new replicated-object protocol designed for use in mobile and weakly-connected environments. The protocol differs from previous protocols in combining epidemic information propagation with voting, and in using fixed per-object currencies for voting. The advantage of epidemic protocols is that data movement only requires pair-wise 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-commit schemes. Finally, the use of per-object currencies *allows voting 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 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. Furthermore we discuss an initial design of the underlying replicated-object system and present a basic API.*

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 [13], a replicated-object system intended for use with mobile and or weakly-connected hosts. Deno is designed to support a wide variety of applications ranging from simple shared calendars to domain-specific databases. More specifically, Deno's target application domain includes all types of asynchronous collaborative applications, including groupware (e.g., Lotus Notes [12]), mail and bibliographic databases, document editing, CAD, and program development environments for disconnected workgroups.

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 [22] 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 sites. However, consistency requires that any competing updates to the same shared object be *committed* in the same serial order at every replica [4].

A canonical primary-copy scheme [22] 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, 2, 10, 15, 20, 25]. 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 [22], also called a *monarchy* [3]. 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 [24] 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.

Voting schemes [2, 7, 11, 18, 26] 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. Voting has been shown to provide optimal availability when all replicas have the same independent failure probability of less than ½ [19].

This paper has two central contributions. First, we describe how to extend voting schemes through the use of *fixed* per-object currencies [23, 27, 28]. We say that the currency is fixed because there 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 system.

Second, we use these currencies to allow voting to take place asynchronously through a pair-wise *epidemic* protocol. Our currency-based epidemic protocol can make progress and *eventually* commit object updates even if there is never a majority of replicas connected to each other simultaneously. Epidemic protocols [6, 21, 22] are appropriate for situations in which all replicas need to eventually be made consistent, and where disconnections are frequent.

The rest of the paper is organized as follows. Section 2 describes the epidemic weighted-voting scheme used by Deno in detail. Section 3 discusses some important implementation-level issues such as currency allocation, planned disconnections, and anti-entropy mechanisms. Section 4 provides simulation results showing that currency allocation can be used to implement diverse policies. For example, uniform currency distributions emulate traditional voting schemes, while allocating all currency to a single server emulates a primary-copy scheme. Section 5 presents an initial design of Deno along with its basic API and Section 6 briefly discusses related work. Finally, Section 7 concludes the paper with directions for future research.

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 (i.e., *selective* replication is allowed), 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 epidemic information flow 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. A server is analogous to a voter, creating an update is analogous to a voter deciding to run for office, and a committed update is analogous to a candidate winning the election. Voters (and hence candidates) have indexes *0* through *n-1*, where *n* is the total number of voters. We use v_i to refer to the voter with index *i*, and c_i to refer to the candidate with index *i*. 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, uncommitted 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. This method creates situations in which the *current* election of distinct servers is temporarily out of sync. Voter v_i's current election is the election for which v_i is collecting votes. In order to implement this protocol, each voter maintains three pieces of state:

1. v_i*.completed* – the number of elections completed locally.
2. v_i*.[j]* – is either the index of the candidate voted for by v_j in v_i's current election, or $\perp$, which means that v_i has not yet seen a vote from v_j. The size of the array is bounded by the total number of voters.
3. v_i*.curr [j]* – The amount of currency voted by v_j in v_i's current election or $\perp$, which means that v_i has not yet seen a vote from v_j.

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* c_j *wins* v_i*'s current election when:*

1. *votes*(v_i, j) > 0.5, *or* // c_j gathers *majority* of votes
2. $\forall\ k \neq j$, *votes*(v_i, k) + *uncommitted*(v_i) < *votes*(v_i, j) *or* // c_j gathers *plurality* of votes
   ((*votes*(v_i, k) + *uncommitted*(v_i)) = *votes*(v_i, j) *and* ($j < k$)) // tie-break case

Figure 1: Definitions

Note that although total amount of currency in any election is 1.0, the allocation of this currency may change with each election.

Figure 1 presents some important definitions used in this section. 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 between the indexes of the servers that created the competing updates. The winner of the j^{th} vote at v_i is denoted $v_i.commit(j)$. 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. Figure 2 describes the steps to be executed (as a single atomic unit) during an anti-entropy session from v_i to v_j.

Rule 1 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. Rule 2 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. Rule 3 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 only overwrites $\perp$, no consistency problems occur.

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

$$\forall_{k = v_j.commit + 1}^{v_i.commit} \; v_j.commit(k) \leftarrow v_i.commit(k) .$$

2. If $v_j.completed = v_i.completed$, then $\forall\, k$ s.t. $v_j[k] = \perp$, $v_j[k] \leftarrow v_i[k]$.

3. If $v_j[j] = \perp$, then $v_j[j] \leftarrow v_i[i]$.

Figure 2: Steps to be executed during an anti-entropy session

2.3 Becoming a Candidate

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

1. the election has not been decided for that voter yet, and
2. the voter has not yet voted in the election (i.e., $v_i[i]=\perp$).

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

2.4 Correctness

Given the above definitions, we can 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.commit(k) = v_j.commit(k).

Sketch of Proof: For reasons of brevity, we present only a proof outline. The proof proceeds along the following lines. Restrict the discussion to a single election. 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 in Figure 1 from allowing any other update to be committed. Therefore, all servers must eventually deduce the same outcome, or be told of the common outcome by other voters (Step 1 in Figure 2), and will come to the same conclusions.

2.5 Commutativity Tables

Most databases, Deno included, expect a single ordering of all updates to a single object. However, Deno will also allow application-specific functions to modify the system's consistency requirements. The first way in which Deno will allow consistency to be relaxed is through *commutativity tables*. Operations in typical database systems are not commutable, but many operations in collaborative and groupware applications are. We can take advantage of this by allowing applications to define operation *templates*, lacking only the instantiation of the

	Credits	Debits	Interest
Credits	x	x	
Debits	x	x	
Interest			x

Table 1: Commutativity Table

template parameters. Applications can then record information on which operations are commutative through two-dimensional grids called commutativity tables, which indicate commutability for each possible pair of operation types.

As an example, consider a scalar object representing the balance of a checking account, shown at the right in Table 1 in which the marked entries identify the operation pairs that commute. Simple credits and debits can be executed in any order without changing the final balance. However, calculating and crediting the account for earned interest based on the current balance does not commute with respect to credits and debits.

Operation templates must be defined in advance in order to be included in the tables. However, the data used by these operations need not be static. For example, the specific amounts credited or debited to an account in Table 1 are irrelevant[1]. Moreover, all operations do not need to be defined in advance. By default, Deno assumes that operations not defined in advance are not commutative with respect to any other operation.

More specifically, we can think of each update being generated in a given *context*, where a context is the current election number of a given object. Without commutativity tables, all except the winning update created during in a given context are aborted. With commutativity tables, all losing updates are compared against the winning update to check for commutativity, and the commutative updates are reborn in the next election. As commutativity tables are created at object creation time, this process can be repeated deterministically at each server.

We can generalize the commutativity table into general-purpose *commutativity procedures* in order to exploit more sophisticated inter-relationships. An update-specific commutativity procedure can be supplied with each update. Analogously to the above, each losing update with a commutativity procedure has the procedure run against the contents of all local data objects *after* the winning update has been applied. Allowing all objects to be inspected opens the possibility of the procedures returning different results at different sites. This does not affect correctness, but can be difficult to reason with. As an optimization, procedures can be limited to inspecting only the current object.

3. Practice

This section discusses our approach to issues that will arise when implementing this protocol in a real system. The first issue is whether or not to let applications see uncommitted updates. Newly created updates are *tenta-*

[1] Ignoring error conditions for the moment. It is certainly possible that processing all debits before credits might result in a bank shutting down an account unnecessarily.

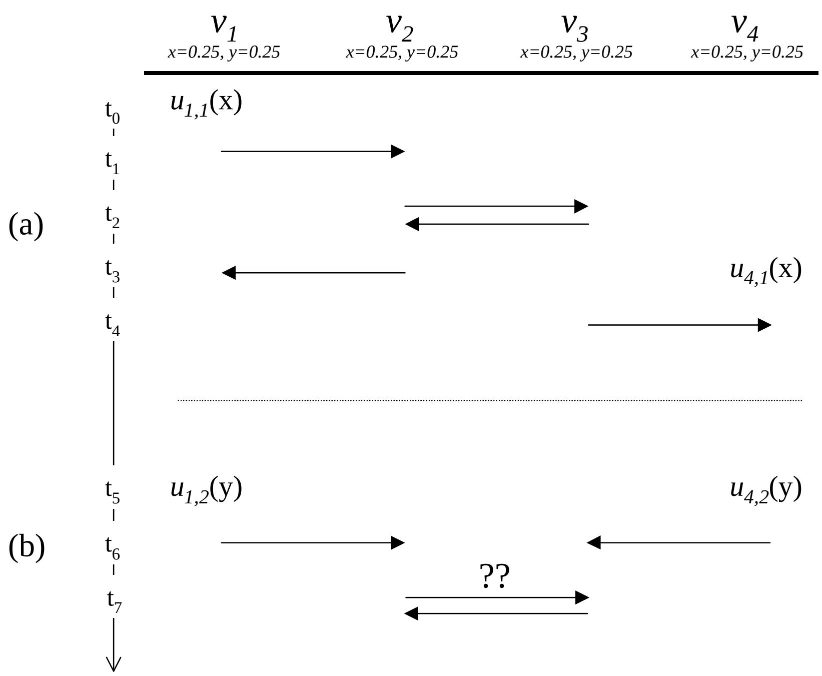


Figure 3: Four replicas each of objects *x* and *y*. $u_{i,j}$ is the update created by v_i in election *j*. Currency is divided evenly for both replicas. (a) shows the progress of update $u_{1,1}$ from v_1. The update is committed because a majority of the object's currency "sees" it before any competing update. (b) shows two competing updates to *y*. At time t_6, both $u_{1,2}$ and $u_{4,2}$ have been seen by replicas with a combined currency of 0.50.

tive, 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 3(a). Objects *x* and *y* are replicated at sites v_1 through v_4. Each site has currency of 0.25 for both objects. Server v_1 creates a tentative update to *x* at time t_0. At time t_1, v_1 sends information to v_2, and at time t_2, v_2 sends to v_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 commit $u_{1,1}$ because they control 75% of the object *x*'s currency. However, only v_3 knows this. Not knowing of the first election's outcome, v_4 naively creates a new update, $u_{4,1}$ at time t_3. This update will be aborted at t_4 when v_4 learns that a quorum has already determined that $u_{1,1}$ should be committed.

Figure 3(b) shows an example of two competing updates being started at time t_5. Each synchronizes with one other replica at t_6, leading to a potential 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 $u_{1,2}$ over $u_{4,2}$.

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. If initial allocations are not ideal, Deno servers can perform peer-to-peer currency exchanges to incrementally improve existing currency distributions.

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 it's 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

In this section, we present an overview of the failure detection and handling protocol used by Deno. The details of the protocol are beyond the scope of this paper and will appear elsewhere.

Failure detection in the domain of mobile applications is difficult because servers may be out of contact either temporarily or permanently. 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 frequent, but might also 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 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 via *proxy elections*.

The main idea is to collectively elect a server to act as a proxy for the failed server(s). Proxy elections are performed similarly to coordinator election protocols widely used by many distributed protocols [4]. After detecting a failure, a server initiates a *proxy election update* that indicates the server's intention to become the proxy for the failed server. As with other changes to objects, a proxy election update is a special type of operation on an object. The election update, therefore, must be committed before it can take effect. Deno treats all updates, including proxy election updates, uniformly and uses its voting scheme to commit them. One implication is that a proxy election can occur if a majority of the current currency is available. Such a restriction is necessary to prevent parallel proxy elections in multiple partitions after a network failure. When a failed server rejoins the computation and learns about the proxy election, the server resets its currency to zero. The server may then request its currency back from its currency or obtain currency from other servers through peer-to-peer currency exchanges.

3.5 Anti-entropy Mechanisms

The pair-wise 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.

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 [6]. 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 $i+1^{th}$ iteration is just:

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

which converges rapidly. The corresponding recurrence for pushes is:

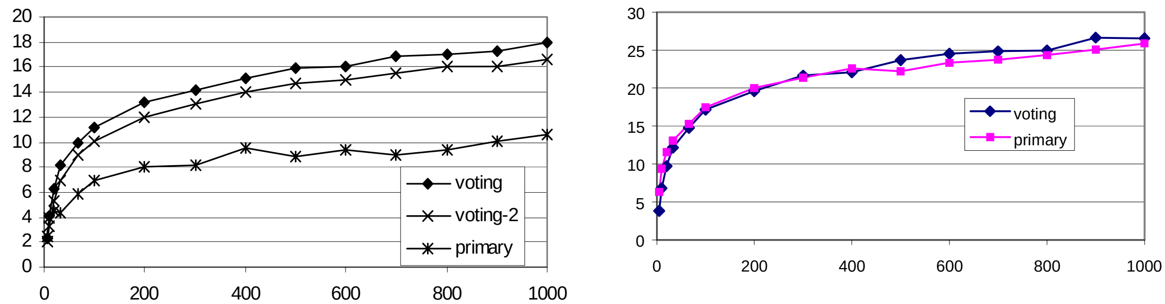
$$p_{i+1} = p_i (1 - \frac{1}{n})^{n(1-p_i)}$$

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. Experimental Study

The primary goal of our protocols will be to improve the ability of the system to make progress during times of low connectivity. This includes improving read availability, and the ability to commit updates. However, poor performance and speed at committing could make a system unusable during periods of good connectivity. We



(a) **(b)**

Figure 4: 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.

built a simple simulator in order to gain an intuitive into the protocol's behavior in our expected environments. We simulate a 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 4(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 accurately 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 4(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 5 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

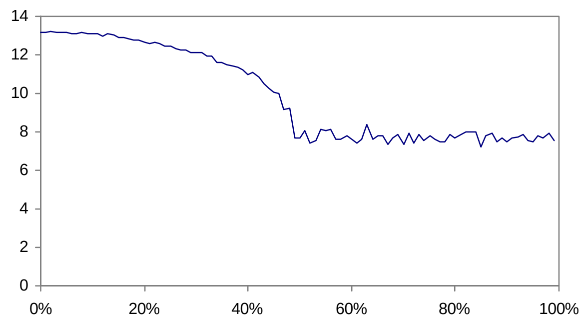


Figure 5: Commit cost (in intervals) versus percentage of currency given to single replica (200 replicas).

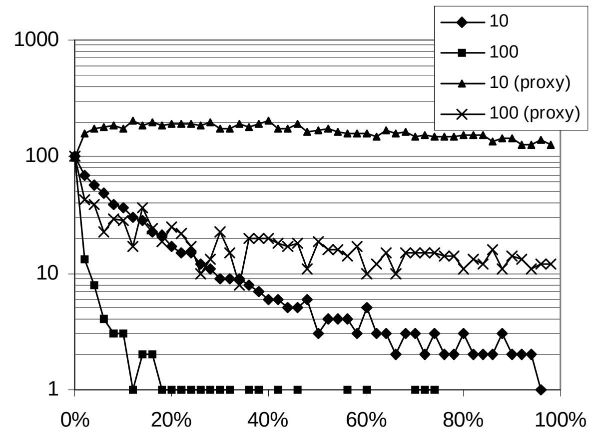


Figure 6: Committed updates in 2000 intervals versus unavailability: The *x* axis is the probability that a server will disconnect in any given interval.

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 6 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.

5. Deno Design

Deno is a library that can be linked directly with application instances, such as bibliographic databases, chat servers, or collaborative groupware applications. Objects can be of any size, although our current mechanisms will work best with relatively small numbers of objects. Any process that is linked to a copy of the Deno library is considered to be a Deno server. However, servers do not replicate all objects. Object replication is only on demand, and entire databases do not need to be replicated as a unit.

Interface Call	Semantics
`deno_server_create`([server name])	Creates server with optional name.
`deno_object_create`(<name> <initial Obj> [exp. #])	Creates new object. Optional third argument gives the expected number of eventual replicas.
Obj `deno_replica_create`(<name> [<server hint>])	Creates local replica of named object. The optional server hint tells Deno where to look for an existing replica.
`deno_object_resize`(Obj, int sz)	New size for binary Deno object.
int `deno_replica_update`(<name> <update>)	Updates an object replica. Updates are specified as Tcl scripts.
`deno_replica_proxy`(<object name> [<server hint>])	Delegates authority while disconnected.
`deno_replica_unproxy`(<object name>)	Retrieves delegated authority.
`deno_replica_delete`(<name> [<proxy hint>])	Deletes local replica and transfer currency.
int `deno_update_status`(<update id>)	Identifies current status of an update. An update can be committed, aborted, or tentative.
int `deno_wait_update`(<update id>)	Waits for an update to be terminated (i..e., either committed or aborted).

Table 2: Basic Deno API

The overriding goal of the Deno project is to investigate replica consistency protocols. We are therefore not motivated to build large and complicated interfaces to the object system. By the same token, we feel that lightweight interfaces are the appropriate choice for many applications, and that more complex services can be efficiently built on top of Deno services if needed.

The basic Deno API consists of the calls listed in Table 2. These calls allow new servers, objects, and replicas to be created, and replicas to be updated and destroyed. Servers use proxy calls to delegate voting rights before planned disconnections. Notification calls are used to learn about the termination status of the updates. The sparse interface avoids burdening applications with unwanted or unneeded abstractions and functionality. For example, we provide no means of backing up objects to stable storage. Some applications will have no need for stable storage, while others can provide their own solutions by accessing the objects directly through the object pointers. Deno does provide support for transparent fault tolerance via the replication mechanism.

Likewise, our interface does not include any sort of query interface, even over the namespace of local objects. In other words, there is no way for an application to query a server to list local replicas that are replicated locally. Such interfaces are not needed for applications that have only a few, statically-defined objects. More dynamic or complex applications could build directory services on top of Deno’s mechanisms through a well-known directory object.

We currently expect applications to provide the name of a machine that is running a Deno server with an existing replica. With name in hand, the new server can talk to a well-known port and obtain object replicas by calling `deno_replica_create()`. As an example, consider a chat application based on Deno mechanisms. The database will consist of a single object, the chat log. The first chat process that starts will create a new log object. Subsequent chat processes can start up and obtain replicas of the log object by connecting with any existing server. There are no distinguished servers, any server is capable of creating new objects and providing object replicas to other servers. As discussed before, there is also no notion of a primary server for any object. Servers are all peers, differing only in the amount of per-object currency that they hold.

Our initial system will support two types of objects: binary objects and Tcl [16] strings. Binary objects are arbitrary byte-streams. The `Obj` structure used by several of the API calls is a union that contains a pointer and length for binary objects. Calls to `deno_replica_update()` are made on either side of the actual updates in order to delimit the update interval to the underlying system. The actual updates consist of simple writes and/or calls to `deno_object_resize()`. Modifications to the object are detected through simple byte comparisons between before and after versions of the object.

Tcl objects are simple strings, and are not modified directly by the application. Instead, a Tcl code fragment is passed to the `deno_replica_update()`call. This fragment is atomically applied to the object by Deno. The `deno_object_resize()`call is not used for Tcl objects.

A server that plans to disconnect can use the call `deno_replica_proxy()`to transfer its currency and voting rights to a proxy server. The optional argument specifies where to look for a proxy. When the server reconnects, it calls `deno_replica_unproxy()`in order to regain its currency and voting rights from its proxy.

The calls `deno_update_status()`and `deno_wait_update()`are used by applications to gain information regarding the termination status of updates. The former call returns the current status of a given update, indicating whether the update is committed, aborted, or still tentative. The latter call blocks the application till a given update is either committed or aborted. Using these calls and maintaining enough information to back out of either type of update, Deno can provide any type of session guarantees [24]. By default, however, only committed values are visible.

6. Related Work

We discuss related systems below. Related work on voting and transaction semantics is referenced in the text where appropriate.

Coda [14] and Ficus [17] share many of the goals of our work in the more limited domain of distributed file systems. This choice in domain allows the use of strong assumptions on the relative scarcity of contention. Additionally, reconciliation can be automated for many types of files. Hence, these systems both use replication that is optimistic in the sense of allowing conflicting transactions to commit. Our work makes stronger consistency guarantees at the expense of committing fewer updates.

Bayou [25] also uses epidemic information flow via anti-entropy sessions. However, Bayou objects are committed through a primary-copy rather than a voting scheme. Rather than making guarantees that an update commits only in the context in which it was created, Bayou allows all updates to compete and be committed. Conflicts are detected through dependency-check procedures (similar to our commutativity procedures) that are supplied with each update. These procedures are run at each server in order to decide whether an update can be committed there. Note that these procedures need to be deterministic with respect to the sites that they execute on, while non-determinism of commutativity procedures only affects the rate at which updates commit, not correctness.

Dan [5] pointed out several shortcomings of the traditional ACID transactional model [9] when applied to Internet environments. Primarily, entities are less concerned with the consistency of local databases with respect to partner databases than they are about ensuring that transactions, including legal obligations, are durably recorded. Coyote applications can describe *compensating transactions* that can be used to recover from transactions that need to be retracted. This approach assumes more optimism than ours. However, a similar approach could be used to extend Deno's mechanisms in order to allow more updates to commit, at the cost of the corresponding compensating transactions.

Gray *et al.* [8] categorized replication protocols along two dimensions: 'master/group' object ownership and 'lazy/eager' update propagation. They argued that eager schemes are not suitable for mobile environments due to frequent disconnections and proposed a two-tier lazy protocol for scalable data replication. Under their categorization, Deno can be classified as a lazy group protocol since all replicas are peers and updates are propagated asynchronously. Our pessimistic voting scheme, however, guarantees that conflicting updates are not committed. Deno, therefore, does not require reconciliation, or suffer from *system delusion* [8].

We note that recent work [29] has investigated why quorum systems have yet to become widespread in real-world applications. One of the conclusions is that quorums do not enhance availability because either failures are positively correlated (when servers are on a single LAN) or network partitions occur (when servers are distributed across multiple LANs). In the latter case, a quorum constructed on a single LAN has higher availability than quorums constructed across multiple LANs. However, the weakly-connected environments discussed in this work fit neither category. Disconnections are likely to be independent, and network partitions, while possible, are not the dominant cause of unavailability.

7. Conclusions and Future Work

Mobility and weakly-connectivity pose special problems to object replication systems. We have described a new protocol that uses a combination of voting with fixed currencies and epidemic information flow to allow updates to commit in such environments. This approach is well-suited to weakly-connected environments specifically because it is highly decentralized. However, this decentralization could make the protocol unwieldy in times of high connectivity. For example, users of interactive groupware applications are likely to tolerate slow response times when intermittently connected, but will expect low response times when connected to the corporate backbone. This type of behavior can be built on top of the protocol described above by increasing the frequency of and directing the destinations of the anti-entropy sessions, somewhat similarly to rumor-mongering [6]. We are currently building the Deno prototype to investigate these and other issues.

8. References

[1] D. Agrawal, A. E. Abbadi, and R. Steinke. Epidemic Algorithms in Replicated Databases. In *Proc. of the Symposium on Principles of Database Systems*, Tucson, Arizona, May 1997.

[2] Y. Amir and A. Wool. Evaluating Quorum Systems over the Internet. In *Fault-Tolerant Computing Symposium (FTCS)*, June 1996.

[3] Y. Amir and A. Wool. Optimal Availability Quorum Systems: Theory and Practice. *Information Processing Letters*, vol. 65, pp. 223-228, April 1998.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*: Addison-Wesley, 1987.

[5] A. Dan and F. Parr. The Coyote approach for Network Centric Service Applications: Conversational Service Transactions, a Monitor and an Application Style. In *High Performance Transaction Systems Workshop (HPTS)*, 1997.

[6] 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 *Proc. of the Symposium on Principles of Distributed Computing*, August 1987.

[7] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1979.

[8] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replications and a Solution. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, June 1996.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*: Morgan Kaufmann, 1992.

[10] M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computing Systems*, vol. 4, pp. 32-53, February 1986.

[11] 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.

[12] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozie, and L. Greif. Replicated Document Management in a Group Communication System. In *Proceedings of the 2nd Conference on Computer Supported Cooperative Work*, 1988.

[13] P. J. Keleher. Decentralized Replicated-Object Protocols. In *Proc. of the Symposium on Principles of Distributed Computing*, May 1999.

[14] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. of the ACM Symposium on Operating Systems Principles*, October 1991.

[15] 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.

[16] J. K. Osterhout. Tcl: An Embeddable Command Language. In *USENIX Winter Conf.*, 1990.

[17] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on Optimistically Replicated Peer-to-Peer Filing. *Software--Practice and Experience*, vol. 28, pp. 155-180, February 1998.

[18] J.-F. Pâris and D. Long. Efficient Dynamic Voting Algorithms. In *Proc. of the Int. Conf. on Data Engineering*, Los Angeles, California, February 1988.

[19] D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation*, vol. 123, pp. 210-223, 1995.

[20] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. of the ACM Symposium on Operating System Principles*, Saint-Milo France, October 1997.

[21] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable Update Propagation in Epidemic Replicated Databases. In *Proc. of the Int.Conf. on Extending Database Technology*, Avignon, France, March 1996.

[22] M. Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. *IEEE Transactions on Software Engineering*, vol. 3, pp. 188-194, May 1979.

[23] 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.

[24] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Int. Conf. on Parallel and Distributed Information Systems*, September 1994.

[25] 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 *Proc. of the ACM Symposium on Operating Systems Principles*, December 1995.

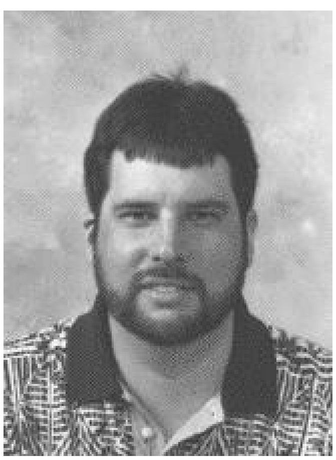
[26] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, vol. 4, pp. 180-209, 1979.

[27] 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.

[28] 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.

[29] A. Wool. Quorum Systems in Replicated Databases: Science or Fiction? *Bulletin of the Technical Committee on Data Engineering*, vol. 21, pp. 3-11, 1998.

Ugur Cetintemel is a Ph.D. candidate at the Computer Science Department, University of Maryland, College Park. His current research interests include replication and data dissemination in distributed systems, and systems performance evaluation. He received B.Sc. and M.S. degrees in Computer Science from Bilkent University, Turkey in 1994 and 1996, respectively.



Dr. Keleher received a PhD in computer science from Rice University in 1995. He is currently an assistant professor in the Computer Science Department at the University of Maryland, College Park. Professor Keleher's primary interests are in the design and analysis of distributed computing infrastructure, including global resource management, distributed shared memory, and communication performance. He is currently leading the Deno project, an infrastructure for replicated storage in mobile environments, and Active Harmony, a new framework for global resource management in dynamic, heterogeneous environments. Professor Keleher received the National Science Foundation CAREER award in 1996. Member, ACM.