Distributed and Parallel Databases, 11, 53–71, 2002


# Light-Weight Currency Management Mechanisms in Mobile and Weakly-Connected Environments

UĞUR ÇETINTEMEL ugur@cs.umd.edu
PETER J. KELEHER keleher@cs.umd.edu
*Department of Computer Science, University of Maryland, USA*

**Recommended by:** Abdelsalam (Sumi) Helal

**Abstract.** This paper discusses the currency management mechanisms used in Deno, an object replication system designed for use in mobile and weakly-connected environments. Deno primarily differs from previous work in implementing an asynchronous weighted-voting scheme via epidemic information flow, and in committing updates in an entirely decentralized fashion, without requiring any server to have complete knowledge of system membership.

We first give an overview of Deno, discussing its voting scheme, proxy mechanism, basic API, and commit performance. We then focus on the issue of currency management. Although there has been much work on currency management in synchronous, strongly-connected environments, this issue has not been explored in asynchronous, weakly-connected environments. We present currency management mechanisms, based on peer-to-peer currency exchanges, that enable light-weight replica creation, retirement, and currency redistribution while maintaining the correctness of the underlying consistency protocol. We also demonstrate that peer-to-peer currency exchanges can be used to exponentially converge to arbitrary target currency distributions, without the need for any server to have global system information.

**Keywords:** epidemic algorithms, replicated data consistency, mobile and weakly-connected systems, weight management

## 1. Introduction

Recent advances in hardware technologies have made mobile computing feasible and practical. Mobile device usage is increasing, as the devices become smaller, cheaper, and more powerful. Mobile users often carry their laptops, PDAs, and other portable devices wherever they go.

Mobile environments differ from typical desktop environments in many ways, including power availability, resources such as CPU, memory, secondary storage, and, above all, in their communication behavior. Mobile systems usually lack continuous connectivity, and typically possess limited communication bandwidth even when they are connected. As a result, mobile and weakly-connected operations rely heavily on replication mechanisms to deliver good performance.

Asynchronous solutions (e.g., [12, 27, 31, 41]) for managing replicated data have a number of advantages over traditional synchronous replication protocols in large-scale, mobile, and weakly-connected environments. They can operate with less than full connectivity, easily adapt to frequent changes in group membership, and make few demands on the

underlying network topology. However, this functionality comes at a price. Asynchronous solutions are generally either slow and require reconciliations [29], or have lower availability because they rely on primary-copy schemes [39].

Deno, a highly-available replicated object storage system intended for use in mobile and weakly-connected environments [28], addresses these concerns through its fully-decentralized, asynchronous replication protocol. Deno differs from previous approaches in that it completely decentralizes all control and information flow. Innovations of Deno include the extension of voting schemes [20, 42] through *fixed* per-object currencies (i.e., weights), and the use of pair-wise epidemic protocols [16] with voting schemes. Deno's replication protocol is highly available, and is able to make progress and eventually commit updates even if there is never a majority of replicas connected to each other simultaneously. No server ever needs to have complete knowledge of group membership, and a given server only needs to be in intermittent contact with at least one other server to take full part in the voting and commitment process. As such, the protocol is highly suited for environments with weak connectivity. Deno's system model is illustrated in figure 1. One or more client processes link to Deno servers, which communicate through pair-wise information propagation. The servers are not necessarily *ever* fully connected.

Deno uses bounded voting [28], in which the total currency in the system is fixed at a system-wide value. Voting schemes [2, 4, 20, 25, 32, 34, 42] allow 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 commit 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 servers have the same independent failure probability of less than $1/2$ [35].

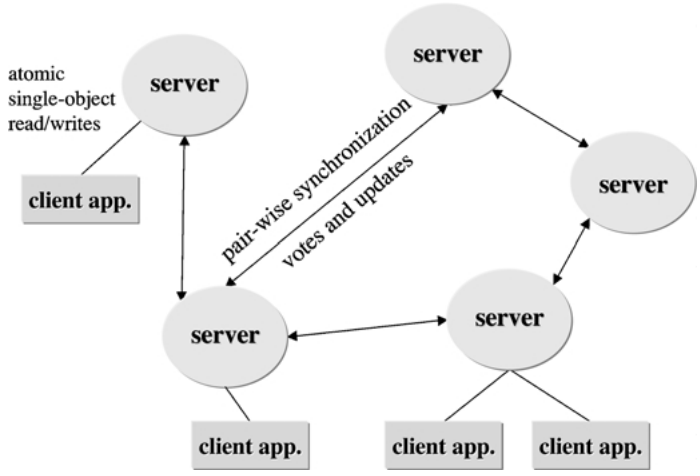


*Figure 1.* Deno system model.

An important issue in any voting scheme is flexible, efficient management of currencies. Light-weight replica management and currency redistribution become especially desirable in highly-dynamic environments due to the need to quickly adapt to changing environmental and application-specific factors and efficiently modify system configuration. Although there has been significant work on currency management for traditional replicated databases and systems, existing proposals are heavy weight in that they typically require the participation of a majority of servers to create/retire replicas and install new currency values [9, 20, 25, 30, 42]. These traditional mechanisms perform inherently poorly under weak connectivity and continuously-changing system configurations due to their synchronous nature and the need to have complete system membership information.

In this paper, we present the light-weight currency management mechanisms used in Deno [13]. The mechanisms presented facilitate efficient replica creation/retirement and currency redistribution operations, while maintaining the correctness of the underlying replica consistency protocol. Unlike previous proposals, our mechanisms require the participation of only two servers to perform these operations, making them especially well-suited for currency management in asynchronous, large-scale, and dynamic environments. In addition, we demonstrate that our currency exchange mechanisms can be used to exponentially converge to arbitrary target currency distributions without the need for any server to have global system information (e.g., current currency distribution, number of servers, etc.).

The rest of the paper is organized as follows. Section 2 gives an overview of Deno by briefly discussing its voting scheme, proxy mechanism, basic API, and commit performance. Section 3 addresses currency management issues and describes how Deno performs light-weight replica creation, retirement, and currency redistribution. Section 4 discusses related work and Section 5 concludes the paper.

## 2. Deno overview

In this section, we present a brief overview of the Deno object replication system. Deno employs the *update anytime* replication model [21] in order to address requirements of disconnected operation. In this model, all servers are treated as peers in their ability to generate updates. Deno's servers execute updates locally and commit them globally using a decentralized weighted-voting scheme [28]. Updates and voting information are propagated through the system *asynchronously* using an epidemic style of communication (e.g., [1, 16, 38, 41]). Epidemic communication exploits pair-wise *anti-entropy* sessions [16], whereby a server is informed of the state of the other server. Anti-entropy sessions propagate updates through the system and ensure that all replicas of the same object *eventually* converge to the same final state.

Updates gather votes as they pass through servers. An update is committed only if and when it corners a *plurality* (rather than a majority) of votes. Gathering a plurality of votes is sufficient to guarantee that no other update on the same data item can gather more votes. In Deno, update commitment is *decentralized* in that each server independently commits or aborts updates on the basis of local information, eliminating the need for *synchronous* multi-site commit protocols (e.g., two-phase commit [10]). However, the same updates eventually commit at all servers in the same order. Note that, in this paper, we only consider

single-item updates. The extension of our protocol to transactional environments and multi-item updates is orthogonal to the main thrust of this paper, and is discussed elsewhere (see [14]).

### *2.1. Decentralized weighted-voting*

We first briefly describe Deno's asynchronous weighted-voting protocol. A more detailed description of the base protocol, along with the correctness proof, appears in [28].

We assume a model in which the shared state consists of a set of objects replicated across multiple servers. Objects do not need to be replicated at all servers and multiple objects can be replicated at the same server. For simplicity of exposition, however, we limit our discussion to single objects that are replicated at all servers.

Objects are modified by *updates*, which are issued by servers. Updates do not commit globally in one atomic phase. Instead, each server *independently* commits updates on the basis of local information. However, we show below that if an update commits at one server, it eventually commits at all servers, and in the same order with respect to other committed updates.

**Elections.** A clean way of thinking about update commitment is as a series of elections. In the election framework, 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. A candidate wins an election if and when it corners a plurality of the votes. Votes are weighted and the sum of all votes in the system is *bounded* to 1.0. Any election may have multiple candidates, which represent logically concurrent tentative updates. Candidates from different elections might be alive in the system at the same time.

We now present our distributed election algorithm. Initially, we assume that the voter does not propose updates and participates in the protocol only to keep abreast with election winners and system state. Later on, we will explain how a voter can propose an update.

Voting information flows from voter to voter through anti-entropy sessions. For our purposes, an anti-entropy session from server $v'$ to server $v$ is a uni-directional flow of information that specifies the elections that have been won and the votes in the current election. More specifically, an anti-entropy from $v'$ to $v$ causes the following sequence of events to occur as a single (atomic) unit:

(1) If $v'$ knows about more committed elections than $v$ does, $v$ copies all those results as a given, without waiting to find the specific votes that caused those outcomes to occur; and
(2) If $v'$and $v$ both know about the same committed elections,then

- $v$ copies all votes known to $v'$ that it does not know itself, and
- if $v$ has not yet voted and $v'$ has voted, then $v$ votes for the same candidate as $v'$ (or for any other candidate).

A voter $v$ keeps track of the votes of all individual voters and summarizes this election information in two main statistics:

- *votes*($k$), which is the sum of votes that have been cast in favor of candidate $k$ in $v$'s current election, and
- *unknown*, which is the sum of currencies of voters whose vote for $v$'s current election is currently unknown to $v$.

Voter $v$ gathers election information until either it can award its current election to a candidate $k$, or learns from another server that the election has already been committed. Voter $v$ awards the election to $k$ when $v$ finds out that $k$ has won a plurality of votes; that is, if and only if, for all candidates $k \neq j$, either

(1) $votes(k) > votes(j) + unknown$, or
(2) $votes(k) = votes(j) + unknown$ and $k < j$.

Commit rule (1) implies that candidate update $k$ can commit only when it is guaranteed that no other candidate update $j$ can gather more votes than $k$. The protocol, therefore, ensures mutual exclusion by guaranteeing that no two conflicting updates can both commit during the same election. Rule (2) breaks ties with a simple comparison between the indexes of the voters that created the competing updates.

Each individual voter counts votes locally and deduces election outcomes independently. As a result, voter $v$ can commit an update without knowing all the votes, without complete knowledge of which voters have seen the update, and even without knowing which voters replicate the object. After voter $v$ has awarded election $i$ to $k$, it will move on to election $i + 1$.

**Becoming a candidate.** A voter, $v$, may propose an update and become a candidate at any time in the $i$th election as long as:

(1) $v$ has not awarded election $i$ to any candidate, and
(2) $v$ has not yet voted in the $i$th election (a candidate $v$ always votes for itself).

Although the protocol is completely asynchronous and decentralized, it satisfies the global update consistency property as stated by Theorem 1 (see [28] for a proof outline):

**Theorem 1.** *If a voter, $v_1$, awards the ith election to candidate $k$, then when any other voter, $v_2$, completes election $i$, $v_2$will award the election to candidate $k$.*

### *2.2. Currency proxies and fault-tolerance*

Deno achieves fault-tolerance through a proxy mechanism. Proxies represent unavailable servers in the system and are assigned either through (1) *proxy assignments*, or (2) *proxy elections*. In the former case, the unavailable server assigns the proxy itself (in case of planned-for disconnections), whereas in the latter case the servers collectively assign the proxy through a distributed proxy election protocol.

**Proxy assignments.** Deno transparently handles voluntary, planned-for disconnections by having a *primary* server engage a proxy server to vote in its place while the primary is

disconnected. A vote cast by a proxy server is then indistinguishable to other servers from the situation where a server votes and disconnects. 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.

We now illustrate the proxy assignment process by a simple example. Consider a server, $s_i$, planning to disconnect from the network. Server $s_i$ contacts another server, $s_j$, assigns $s_j$ as its proxy, and disconnects from the network. Now suppose $s_j$ sees a new candidate update. When processing the candidate, $s_j$ not only casts a vote for itself, but also casts a vote in place of $s_i$ using $s_i$'s currency. This vote is then no different from the vote that would be cast and disseminated by $s_i$ if $s_i$ were connected. Other servers have no way of knowing about the proxy assignment, and therefore the whole proxy mechanism is transparent to the rest of the system.

**Proxy elections.** In case of unexpected disconnections, failures, or network partitions, Deno servers collectively elect a server to act as a proxy to the unavailable, failed server(s). Proxy elections are performed similarly to coordinator elections protocols widely used by many distributed protocols [10], using the decentralized voting scheme described earlier. 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 the elected proxy server or obtain currency from other servers through peer-to-peer currency exchanges.

### *2.3. Deno design*

Deno is a runtime library that can be linked directly with application instances. Any process that is linked to a copy of the Deno library is considered to be a Deno server. Deno's target application domain includes all types of asynchronous collaborative applications, including collaborative groupware (e.g., Lotus Notes [27]), mail and bibliographic databases, document editing, CAD, and program development environments for disconnected workgroups.

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.

Table 1 lists the basic Deno API calls. These calls allow new servers, objects, and object replicas to be created, and object replicas to be updated and destroyed. Servers use proxy calls to delegate voting rights before planned disconnections. Notification calls are used

*Table 1.* Basic Deno API.

| Interface call | Semantics |
|---|---|
| `server_create`([server name]) | Creates server with optional name. |
| `object_create`(<name> <initial Obj> [exp. #]) | Creates new object. Optional third argument gives the expected number of eventual replicas. |
| Obj `replica_create`(<name> [<server hint>]) | Creates local replica of named object. The optional server hint tells Deno where to look for an existing replica. |
| `object_resize`(Obj, int sz) | New size for binary Deno object. |
| int `replica_update`(<name> <update>) | Updates an object replica. |
| `replica_proxy`(<object name> [<server hint>]) | Delegates authority while disconnected. |
| `replica_unproxy`(<object name>) | Retrieves delegated authority. |
| `replica_delete`(<name> [<proxy hint>]) | Deletes local replica and transfer currency. |
| int `update_status`(<update id>) | Identifies current status of an update. An update can be committed, aborted, or still be tentative. |
| int `wait_update`(<update id>) | Waits for an update to be terminated (i.e., either committed or aborted). |

to learn about the termination status of the updates. The sparse interface avoids burdening applications with unwanted or unneeded abstractions and functionality.

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 `replica_create()`. There are no distinguished servers; any server is capable of creating new objects and providing object replicas to other servers. Servers are all peers; they differ only in the amount of per-object currency that they hold.

Calls to `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 to objects and are accomplished by calling `object_resize()`.

A server that plans to disconnect can use the call `replica_proxy()` to transfer its voting rights to a proxy server. When the server reconnects, it calls `replica_unproxy()` in order to regain its voting rights from its proxy.

Two calls are used by applications to gain information regarding the termination status of updates: `update_status()` and `wait_update()`. 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 until a given update is either committed or aborted. Using these calls and maintaining enough information to back out of tentative updates, Deno can provide any type of session guarantees [40], which define the set of updates visible to client applications.

### *2.4. Basic performance*

The primary goal of our protocol is 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.

This section investigates the basic update performance characteristics of our voting protocols using a simulation study. We simulate a system where time is broken into fixed-length intervals. Each server initiates a synchronization session and synchronizes with another (randomly-selected) server using a uniform distribution with a mean of one interval length (i.e., each server pulls information once every interval on the average). A *pull-pull* type of synchronization model is employed; i.e., both synchronization partners pull information from each other in a single synchronization session. For purposes of the evaluation, we assume that the system is fully connected and there are no failures. We also assume, unless stated otherwise, that currency is distributed uniformly across all servers. All the numbers presented are the averaged results of at least ten independent runs, where each run involves executing 1000 updates.

In addition to our voting protocol, we also investigate the performance of two other epidemic protocols that appeared in the literature: epidemic primary-copy and write-all protocols. In all the schemes examined, updates are executed locally, are disseminated across the system using pair-wise epidemic synchronization sessions, and are committed or aborted globally. In the primary-copy scheme an update is committed when it is serialized at the primary-copy server. This is essentially the approach used by the Bayou weakly-connected storage system [41]. It is well known that such a primary-copy approach suffers from single-point failures [10]: if the primary-copy server is unavailable, then no updates can be committed. The other protocol, which we refer to as write-all, is a Read-One-Write-All (ROWA) type epidemic protocol. In write-all, an update is committed after it is certified at all servers. This scheme aborts all conflicting updates in order to provide consistency. Agrawal et al. [1] proposed a similar ROWA type epidemic protocol that supports multi-item operations in a transactional framework.

Figure 2 shows a plot of the average number of intervals (i.e., synchronization periods) needed to commit an update versus the number of servers for Deno's default (uniform)

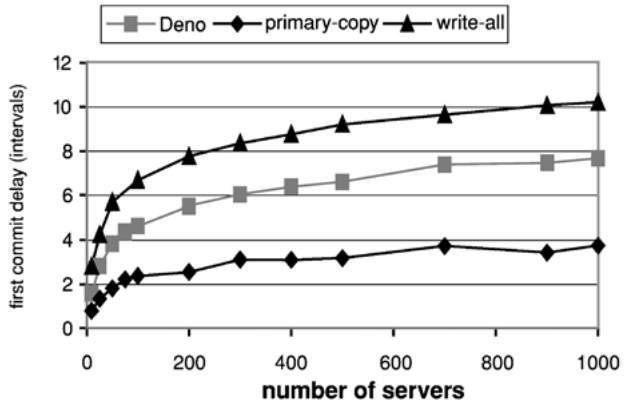


*Figure 2*. First commit delay.

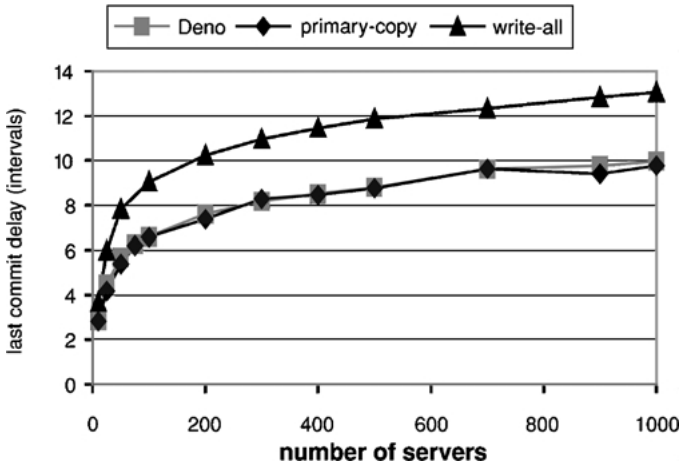


*Figure 3*. Last commit delay.

voting scheme, the primary-copy scheme (modeled by allocating all currency at a randomly-selected site), and the write-all scheme. The metric labeled with "first commit" is the traditional *commit delay*, which denotes time between the initiation of an update and the time it is *first* committed at a server. The figure reveals that the primary-copy scheme commits updates significantly faster than the voting scheme. In the primary-copy scheme, an update commits when the primary copy receives the update—unless the update has already become obsolete by the commitment of a previously committed conflicting update. Therefore such a scheme clearly commits an update much faster than a uniform voting approach, which requires an update to visit at least a majority of servers (assuming a uniform currency distribution). Both the primary-copy and voting approaches are significantly better than the write-all approach, because the write-all approach requires *all* servers to receive the update before the update can be committed.

Figure 3 concentrates on the *last commit delay*, which is the commit

delay for the server that is the last to commit the update. The figure reveals that the performance difference between the primary-copy and the voting approach is largely lost, and the two configurations perform quite similarly. The way the commit information is propagated in the system is the reason for this interesting result. Although primary-copy commits an update pretty fast, this information propagates to other servers relatively slower. This is because all servers, except the one that owns the primary-copy, learn the commitment from other servers that have already learned the commitment. In the voting protocol, on the other hand, distinct servers may either learn the commitment from other servers (as in the case of primary-copy), or commit the update themselves *independently*. In the presented example, for instance, about 45% of the servers committed the update independently. The delay between the first and subsequent commits is thus quite small.

In an environment where updates are propagated asynchronously, average commit delay, and probably last commit delay, are also as important as the first commit delay, because

a committed update is not useful until its effects are reflected to a replica. One important implication is that the performance penalty of using voting rather than a primary-copy approach is not as large as commonly assumed in the kinds of environments we address in this paper.

## 3. Light-weight currency management

Timely update commitment depends on being able to assemble a majority to vote on updates. The cost of assembling a majority is highly dependent on the currency distribution of the object replicas. The best currency distribution depends on the non-trivial interplay among several factors such as expected availability of individual servers, interconnectivity, and application characteristics. In general, replicas that are more reliable or better interconnected should receive more currency [7].

In this section, we discuss mechanisms that enable the implementation of arbitrary currency distribution policies while still maintaining the correctness of the voting protocol. Note that the issue of finding *optimal* currency distributions has been addressed by previous work (e.g., [5, 7, 9, 25, 30]), and is outside the scope of this paper.

We first describe how replicas are created and currency is initially allocated. We then discuss protocols for dynamically reallocating currency while maintaining the mutual exclusion properties of our voting protocol. We also investigate the cost of migrating currency distributions towards target distributions when initial allocations are not ideal.

### *3.1. Replica creation and retirement*

Objects are initially created with a total currency of 1.0 (or any other system-wide fixed amount), held by the creating server. A new replica is created through a request to a server that already has a replica. The response to such a request contains both an object replica and some amount of currency that is subtracted from the currency held by the responding server. A replica can be retired using a similar pair-wise mechanism in which the currency held by the retired replica is transferred to another replica.

Initial currency allocation is non-trivial because not only servers do not have complete knowledge of the size of the anticipated set of servers, but also there is generally not even a central location that can be expected to receive all currency requests. Instead, each server receives an initial block of currency from the server who responds to its initial request to create a replica. This respondent can be any server, so we clearly cannot guarantee to achieve a given distribution merely by allocation.

However, Deno applications can direct initial currency allocation by providing a hint at object creation as to how many replicas are expected to be created (see Section 2.3). 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. This choice can also be controlled through runtime hints.

### *3.2. Currency redistribution mechanisms*

Without any restricting assumptions, it is not likely that initial currency allocations will approach the target distributions. Furthermore, the *optimal* distribution in dynamic environments and systems may change continuously. It is crucial, therefore, to provide mechanisms to redistribute currency dynamically throughout the lifetime of the object.

Deno uses peer-to-peer *currency exchanges* to incrementally change existing currency distributions into arbitrary target distributions. A peer-to-peer currency exchange involves a pair of servers communicating and redistributing their total currency according to some redistribution policy.

**Requirements.** We now describe in detail how to implement peer-to-peer currency exchanges while maintaining the correctness of the voting protocol. Let $s_i$ and $s_j$ be two servers that exchange currency, and, without loss of generality, let $x$ be the currency to be transferred from $s_i$ to $s_j$. Further, let $e_i$ denote the most recent election in which $s_i$ voted, and $e_j$ denote the current election of $s_j$. For correctness, the protocol has to guarantee that:

(1) $x$ is not used more than once in any election, and
(2) $x$ is available to every election.

Requirement (1) is needed in order to prevent servers from reaching different conclusions on the outcome of a single election. The need for requirement (2) is less obvious. Any amount of currency that effectively *disappears* from an election can prevent an election from closing. In the case of server failures, the rest of the system cooperates to reallocate the lost server's currency. However, in this case no server has failed, and without restriction (2), a loss of currency could halt the entire system.

In order to satisfy the two correctness requirements presented above, we define $e$, the election in which $s_i$ decreases the amount of currency it uses by $x$, and $s_j$ increases the amount of currency it uses by $x$, as:

(i) if $e_i < e_j$, then $e = e_j$
(ii) if $e_i \geq e_j$, then $e = e_i + 1$

Case (i) indicates that $s_i$ last used $x$ in an election which has already been completed by $s_j$. Server $s_j$ can immediately use $x$ in its current election since the requirement (1) presented earlier is guaranteed. In fact, $s_j$ *must* use $x$ in its current election in order to satisfy requirement (2). Case (ii) complements the former case by handling the situation in which $s_i$ last used $x$ in an election that has *not* yet been completed by $s_j$. In this case, $s_j$ cannot use $x$ before it completes all those elections in which $s_i$ has used $x$ in order not to invalidate requirement (1).

Case (i) also implies that it is possible that $s_j$ increase its vote during an election for which $s_j$ has already cast a vote. Therefore, a server that observes two different votes from the same server for the same election uses the vote with more currency, since cases (i) and (ii) together guarantee that it is not possible for a server to decrease its currency in an election it has already voted. On the other hand, $s_j$ *cannot* change the candidate update for which it has already voted in a given election.

**Currency representation.** The above currency exchange protocol requires that servers maintain extra information regarding the amount of their currency they can use in a given election. Specifically, each server $s_i$ maintains a set of ⟨ *currency, election number* ⟩ pairs as follows:

$$\{\langle c_{i1}, e_{i1}\rangle, \langle c_{i2}, e_{i2}\rangle, \ldots, \langle c_{im}, e_{im}\rangle\},$$

where $e_{ij}$ represents the election number in which $s_i$ starts voting $c_{ij}$ of its currency, $\forall j = 1 \ldots m$. The currency voted by $s_i$ at election $e$ is then:

$$\sum c_{ij}, \quad \text{s.t.}, e_{ij} \leq e$$

Currencies transferred during exchanges are also represented using the same set representation. It is, therefore, possible for a server to exchange the currency that the server has not even started using yet without sacrificing correctness. The protocol guarantees that for every election $j$:

$$\sum_{i=1}^{n} c_{ij} = 1.0,$$

where $n$ is the number of servers in the system. Note that the protocol presented above also applies to the currency transfers performed during replica creation and retirement.

An important feature of peer-to-peer exchanges is that the final currency distribution does not have to be known by any participating server. Rather, each server indicates a target weight and receives currency proportional to this weight. More formally, let $c_i$ and $c_i'$ denote the currencies that $s_i$ holds before and after a currency exchange, respectively. Assume that two servers, $s_i$ and $s_j$, that desire to eventually hold target currency levels of $t_i$ and $t_j$ respectively, perform a currency exchange. In this case, the new currency values after the currency exchange will be:

$$c_i' = (t_i/(t_i + t_j))(c_i + c_j)$$
$$c_j' = (c_i + c_j) - c_i'$$

### *3.3. Currency redistribution policies*

Given any initial distribution, randomized peer-to-peer currency exchanges can be used to converge to *any* target distribution, even without complete knowledge of the servers in the system. For example, consider the optimal availability currency distribution given by Amir and Wool [5], where currency is distributed proportionally to the individual availability of servers. Without complete knowledge of all availabilities in the system, it is not possible for any individual server to determine its own target currency. However, two servers participating in a peer-to-peer currency exchange can converge to these unknown targets by redistributing their own currencies proportionally to their own availabilities (i.e. $t_i$ is set equal to the availability of $s_i$). Therefore, it is sufficient for each server to have knowledge

of only its own availability. For instance, servers can converge to a uniform distribution without knowing the total number of servers; during a pair-wise currency exchange, two servers can simply share their total currency equally (i.e., $t_i$ and $t_j$ are set equal).

In synchronous quorum systems, it is known that when all servers have the same independent failure probability $f$, then availability is maximized using a primary-copy system if $f > 1/2$, or using a majority quorum system if $f < 1/2$ [5]. If the failure probabilities are different and all of them are smaller than $1/2$, then optimal currencies are defined as:

$$c_i = \log_2\left(\frac{1 - f_i}{f_i}\right),$$

where $c_i$ is the currency at $s_i$, and $f_i$ is the failure probability of $s_i$.

On the other hand, 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 impact of disconnections by looking at their effects on commit performance.

We conducted a serious of experiments to quantify the potential performance improvements attainable through currency redistribution based on individual server availabilities. For purposes of these experiments, servers disconnect at any synchronization period with some probability. The disconnection probabilities are assigned to servers using a zipf distribution. During each synchronization session, two servers redistribute their total currency proportional to their availabilities, where the availability of a server is defined to be 1.0 less the disconnection probability of the server. As in the earlier experiments, currency is initially uniformly distributed in the system, and servers perform synchronization once every synchronization period on average.

Figure 4 shows the change in the commit performance of the system as the system performs currency exchanges. Commit delays decrease as the number of currency exchanges performed in the system increases. Initially, i.e., when the currency is distributed evenly across servers, the commit delay is about 27 intervals. The commit delay decreases down

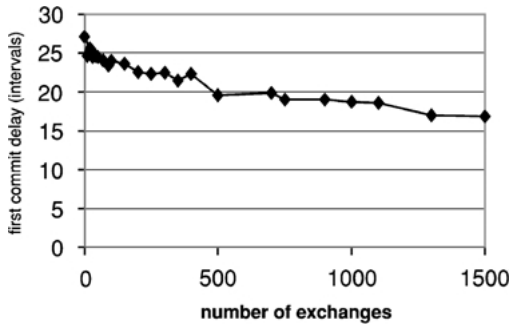


*Figure 4*. Effects of availability-based currency redistribution (100 servers).

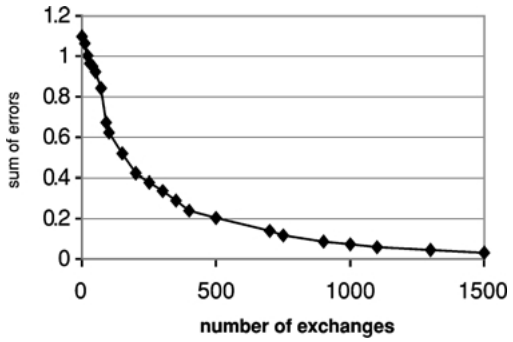


*Figure 5*. Difference between existing and desired currency distribution (100 servers).

to 24 intervals after 50 exchanges, 20 intervals after 500 exchanges, and 17 intervals after 1500 exchanges. These figures correspond to, respectively, 9%, 28%, and 38% improvements with respect to the initial commit speed of the system. After 1500 currency exchanges (not shown), we do not observe further improvement in commit performance, and the performance essentially remains the same. Figure 5 investigates the reason for this behavior by plotting the sum of differences between the existing currency distribution and the distribution where currency is distributed proportional to relative availabilities at servers as the system performs currency exchanges. The figure reveals that this difference diminishes very fast. In fact, the difference becomes after zero after approximately 1500 exchanges, implying that the system converged to its target distribution.

Note that an existing currency distribution can be migrated to a target distribution without the need for any server to have global system information such as the number of servers, existing currency distribution, etc. The ability to achieve global goals with only local information is one of the reasons that this mechanism is especially suited for highly-dynamic environments and systems.

### *3.4. Convergence rates*

We further investigated the convergence speed of our pair-wise currency redistribution mechanism. We observed that randomly-selected, pair-wise currency exchanges allow an existing currency distribution to converge *exponentially* fast to any target distribution. We proved this result analytically for three servers, and the experimental results in figures 6 and 7 suggest that the proposition generalize when there are more than three servers.

Figure 6 shows the mean difference between thousand pairs of randomly chosen initial and target currency distributions versus the number of (randomly-selected) pair-wise currency exchanges performed in the system. Each currency exchange involves two servers redistributing their currencies proportional to their weights in the target distribution as described previously. As expected, the larger the number of replicas, the larger the number of currency exchanges required to converge to the target distribution. However, the shapes of the plots in the figure demonstrate that the difference between the target and the existing

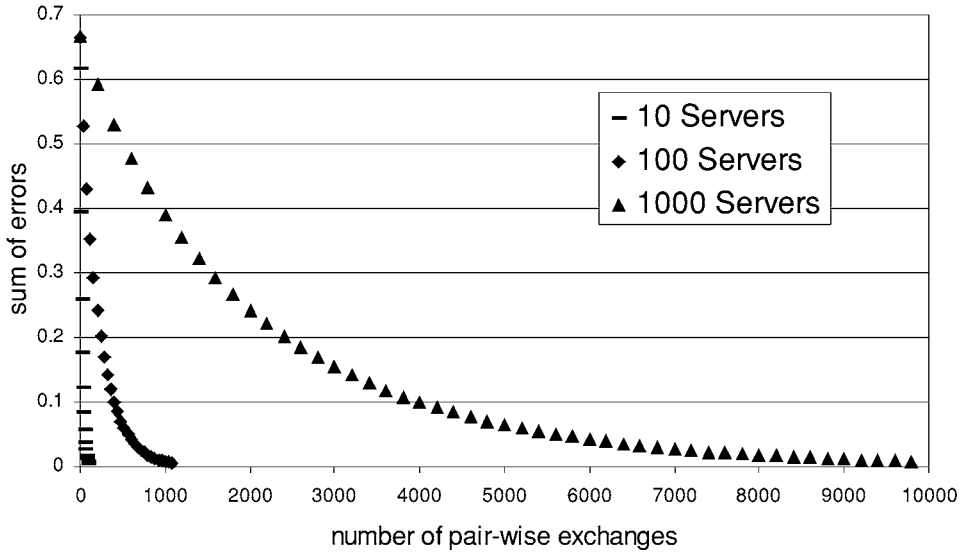


*Figure 6*. Converging to a target distribution with randomly selected peer-to-peer currency exchanges.

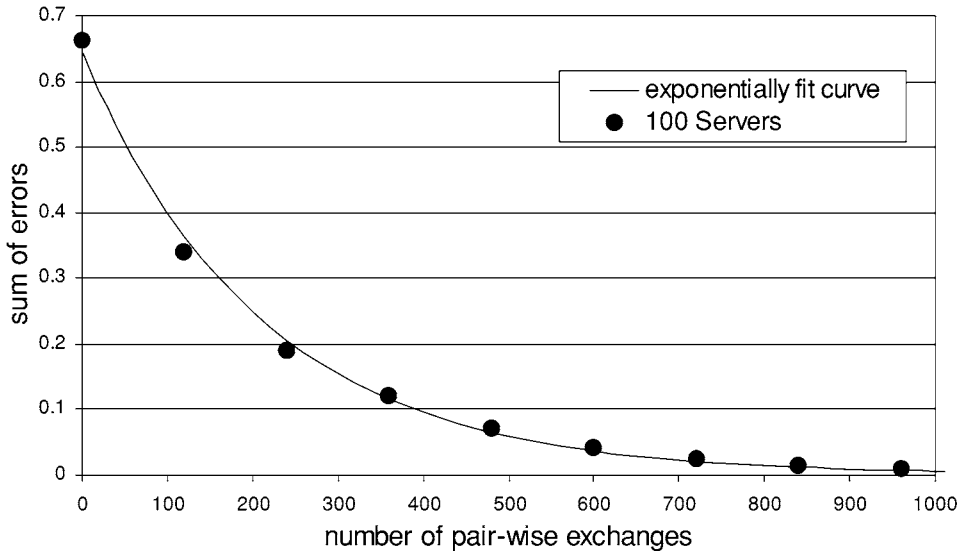


*Figure 7*. Exponentially diminishing error (100 servers).

distributions diminishes quickly. In fact, the convergence to target distribution happens *exponentially* fast. We illustrate this result in figure 7, which plots the corresponding data points for 100 servers along with an exponential curve fitted to our data. The exponential curve shown in the figure has the form $y = ae^{bx}$ where $(a, b) = (0.6453, -0.0047)$. The corresponding exponentially fit curves for 10 and 1000 servers (not shown) can be obtained, respectively, using the settings of $(0.6107, -0.0514)$ and $(0.6431, -0.0004)$.

It is also worth noting that Barbara and Garcia-Molina demonstrated that autonomous, incremental methods for determining new currency distributions, while being more flexible, can yield as much availability as those methods that require having complete knowledge of the state of the system [9].

## 4. Related work

In this section, we review related research efforts by first discussing previous research relevant to Deno in general, and then focusing on currency management in voting systems.

There has been significant work in the area of data and consistency management in mobile and weakly-connected environments [3, 8, 15, 17, 18, 24, 29, 33, 36, 37, 41]. Of particular relevance to Deno's replication protocol are those proposals that exploit epidemic algorithms to propagate updates [1, 16, 27, 33, 38, 41]. In epidemic protocols, updates are executed at any single server. Asynchronously, servers communicate in a pair-wise fashion to exchange information regarding the updates, detecting and bringing the obsolete copies up to date. Epidemic algorithms are a natural fit for mobile and weakly-connected environments: they do not rely on static communication topologies or fail during temporary disconnections.

Many epidemic systems take an optimistic approach and use reconciliation-based protocols (e.g., [27, 33]) that are only viable in certain domains such as file systems. This choice in domain allows the use of strong assumptions on the relative scarcity of update contention. Additionally, reconciliation can be automated for many types of files.

Bayou [41] takes a more pessimistic (i.e., conflict avoidance-based) approach, ensuring that all committed updates are serialized in the same order at all servers using a primary-copy scheme; i.e., updates can only be committed at the primary-copy server. Agrawal et al. proposed a ROWA type pessimistic approach that ensures serializability in a transactional framework [1]. In their approach, an update transaction is committed only after it is validated at all servers. Both the primary-copy approach and the ROWA approach typically have low reliability as they suffer from the infamous single-point-of-failure problem.

Recently, we implemented a Deno prototype on top of Win32 and Linux platforms, and extended the basic single-item Deno protocol to handle multi-item transactional updates [14]. The extended transactional protocols we proposed provide two levels of consistency: strict serializability [10] and update serializability [11, 19]. Independent from our research on transactional voting protocols, Holliday et al. [22] also proposed a transactional quorum based approach that provides strict serializability. This work assumes static, globally-known currencies and does not address currency management issues.

The impact of mobility on data management and replication has been discussed in [3, 6, 8, 17, 24]. Pitoura and Bhargava proposed a two-level, cluster-based consistency model for intermittently-connected environments [36]. Huang et al. analyzed different data replication

methods with the objective of minimizing the communication costs between the mobile and stationary hosts in a mobile environment [23]. Jing et al. presented a framework for information access under wireless client-server computing systems [26].

Voting schemes [20, 42] improve availability by allowing a quorum of all replicas to commit an update. Work on currency management mainly focused on *policies* that are used to reassign votes after server or link failures to improve availability [5, 9, 25, 30]. The weight reassignments, as well as replica creation and retirement operations, are typically installed at servers using heavy-weight mechanisms that require the participation of (at least) a majority of servers to maintain mutual exclusion properties. To the best of our knowledge, Deno is the only voting scheme that allows for light-weight replica creation and retirement, requiring the participation of only two servers.

Finally, we note that recent work [43] investigated why quorum systems have yet to become widespread in real-world applications. One of the conclusions is that voting does 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 addressed in this work fit neither category. Most failures (e.g., voluntary disconnections) are likely to be independent, and network partitions, while possible, are not the dominant cause of unavailability.

## 5. Conclusions

In this paper, we presented an overview of the Deno highly-available object replication system, and described how it implements a novel, decentralized voting scheme via epidemic information flow. We also briefly discussed the proxy mechanism used to transparently tolerate disconnections. Using a simulation study, we characterized the commit performance of Deno, along with other comparable epidemic consistency protocols that appeared in the literature. One of our interesting findings is that the presumed performance advantage of the primary-copy approach over voting approaches is not as significant with asynchronous epidemic protocols as it is with synchronous protocols.

We then focused on the important issue of currency management, and described mechanisms that facilitate light-weight replica creation, retirement, and dynamic currency redistribution in mobile and weakly-connected environments. Unlike previous heavy-weight protocols that typically require complete system state and at least a majority of servers to create new replicas or install new currency values, the mechanisms we proposed are based on peer-to-peer currency exchanges, thereby requiring the participation of only two servers. Furthermore, these mechanisms can be used to converge to arbitrary target currency distributions exponentially fast, without any server requiring complete knowledge of system state. These unique features make our mechanisms especially well-suited for use in weakly-connected and highly-dynamic environments.

In terms of future work, we plan to investigate dynamic synchronization policies (i.e., when, what, and with whom to synchronize?). A synchronization policy needs to consider a variety of environmental factors such as the available bandwidth, cost of communication,

server availability, and currency information as well as application-dependent factors such as update generation rate. Furthermore, since many of the mentioned factors typically demonstrate dynamic behavior, adaptive policies are required. We have so far assumed a flat organization of Deno servers. As with any distributed system, flat organizations, especially peer-to-peer models, suffer from low scalability. Significant performance improvements can be attained if more structured synchronization topologies are used. Adaptive synchronization policies and support for hierarchical organizations will form the basis of our future work.

## 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. M. Ahamad and M.H. Ammar, "Multidimensional voting," ACM Transactions on Computing Systems, vol. 9, no. 4, pp. 399–431, 1991.
3. R. Alonso and H.F. Korth, "Database system issues in nomadic computing," in Proc. of the ACM SIGMOD Int. Conf. on Management of Data, Washington, DC, May 1993.
4. Y. Amir and A. Wool, "Evaluating quorum systems over the internet," in Fault-Tolerant Computing Symposium (FTCS), June 1996.
5. Y. Amir and A. Wool, "Optimal availability quorum systems: Theory and practice," Information Processing Letters, vol. 65, pp. 223–228, 1998.
6. D. Barbara, "Mobile computing and databases," IEEE Transactions on Knowledge and Data Engineering, vol. 1, pp. 108–117, 1999.
7. D. Barbara and H. Garcia-Molina, "Optimizing the reliability provided by voting mechanisms," in Proc. of the International Conf. on Distributed Computing Systems, San Francisco, October 1984.
8. D. Barbara and H. Garcia-Molina, "Replicated data management in mobile environments: Anything new under the Sun?" in IFIP Working Conference on Applications in Parallel and Distributed Computing, April 1994.
9. D. Barbara, H. Garcia-Molina, and A. Spauster, "Increasing availability under mutual exclusion constraints with dynamic voting assignment," ACM Transactions on Computing Systems, vol. 7, no. 4, pp. 394–426, 1989.
10. P.A. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley: Reading, MA, 1987.
11. P. Bober and M. Carey, "Multiversion query locking," in Proc. of the VLDB Conference, British Colombia, Canada, 1992.
12. Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silbershatz, "Update propagation protocols for replicated databases," in Proc. of the ACM SIGMOD Int. Conf. on Management of Data, Philadelphia, PA, 1999.
13. U. Cetintemel and P.J. Keleher, "Light-weight currency management mechanisms in Deno," in Proc. 10th IEEE Workshop on Research Issues in Data Engineering (RIDE), San Diego, February 2000.
14. U. Cetintemel, P.J. Keleher, and M.J. Franklin, "Support for speculative update propagation and mobility in Deno," In IEEE Intl. Conf. on Distributed Computing Systems (ICDCS), Phoenix, 2001.
15. S. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a partitioned network: A survey," ACM Computing Surveys, vol. 17, no. 3, pp. 341–370, 1985.
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 Proc. of the Symposium on Principles of Distributed Computing, August 1987.
17. M. Dunham and A. Helal, "Mobile computing and databases: Anything new?" SIGMOD Record, vol. 24, no. 4, pp. 5–9, 1995.
18. L. Frank, "Evaluation overview of the replication methods for high availability databases," in ECOOP, 1998.
19. H. Garcia-Molina and G. Wiederhold, "Read-only transactions in a distributed database system," ACM Transactions on Database Systems, vol. 7, no. 2, pp. 209–234, 1982.

20. D.K. Gifford, "Weighted voting for replicated data," in Proceedings of the ACM Symposium on Operating Systems Principles, 1979.
21. 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.
22. J. Holliday, R. Steinke, D. Agrawal, and A.E. Abbadi, "Epidemic quorums for managing replicated data," in IPCCC'2000, Phoenix, Arizona, 2000.
23. Y. Huang, A.P. Sistla, and O. Wolfson," Data replication for mobile computers," in Proc. of the ACM SIGMOD Int. Conf. on Management of Data, 1994.
24. T. Imielinski and B.R. Badrinath, "Wireless mobile computing: Challenges in data management," Communications of the ACM, vol. 37, no. 10, pp. 19–28, 1994.
25. S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database," ACM Transactions on Database Systems, vol. 15, no. 2, pp. 230–280, 1990.
26. J. Jing, A. Helal, and A. Elmagarmid, "Client-server computing in mobile environments," Computing Surveys, vol. 31, no. 2, pp. 117–157, 1999.
27. 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.
28. P.J. Keleher, "Decentralized replicated-object protocols," in Proc. of the Symposium on Principles of Distributed Computing, May 1999.
29. 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.
30. A. Kumar and A. Segev, "Cost and availability tradeoffs in replicated data concurrency control," ACM Transactions on Database Systems, vol. 18, no. 1, pp. 102–131, 1993.
31. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," ACM Transactions on Computing Systems, vol. 10, no. 4, pp. 360–391, 1992.
32. 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.
33. 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, no. 2, pp. 155–180, 1998.
34. J.-F. Pâris and D. Long, "Efficient dynamic voting algorithms," in Proc. of the Int. Conf. on Data Engineering, Los Angeles, California, February 1988.
35. D. Peleg and A. Wool, "The availability of quorum systems," Information and Computation, vol. 123, no. 2, pp. 210–223, 1995.
36. E. Pitoura and B. Bhargava, "Maintaining consistency of data in mobile distributed environments," in Proc. of the International Conference on Distributed Computing Systems, May 1995.
37. R. Prakash and M. Singhal, "Dynamic hashing + quorum = Efficient location management for mobile computing systems," in Proc. of the Principles of Distributed Computing, Santa Barbara, CA, August 1997.
38. 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.
39. M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES," IEEE Transactions on Software Engineering, vol. 3, no. 3, pp. 188–194, 1979.
40. 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.
41. 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.
42. R.H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," ACM Transactions on Database Systems, vol. 4, no. 2, pp. 180–209, 1979.
43. A. Wool, "Quorum systems in replicated databases: Science or fiction?" Bulletin of the Technical Committee on Data Engineering, vol. 21, no. 4, pp. 3–11, 1998.