Light-Weight Currency Management Mechanisms in Deno

Uğur Çetintemel
Department of Computer Science
University of Maryland
ugur@cs.umd.edu

Peter Keleher
Department of Computer Science
University of Maryland
keleher@cs.umd.edu

Abstract

This paper discusses the currency management mechanisms used in Deno, a replicated-object storage 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, briefly discussing its voting scheme, proxy mechanism, basic API, and commit performance. We then present currency management mechanisms, based on peer-to-peer currency exchanges, that enable light-weight replica creation, retirement, and currency redistribution while maintaining protocol correctness. We also demonstrate that peer-to-peer currency exchanges can be used to exponentially converge to arbitrary target currency distributions.

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.

Deno is a highly-available replicated-object server intended for use in mobile and weakly-connected environments [15]. Deno differs from previous approaches in that it completely decentralizes all control and information flow. Innovations of Deno include the extension of voting schemes [10, 24] through *fixed* per-object currencies (i.e., weights), and the use of pair-wise epidemic protocols [8] 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.

In order to address requirements of disconnected operation, Deno employs the *update anytime-anywhere-anyhow* replication model [11]. Our system treats all servers as peers in their ability to generate updates. Deno's servers execute updates locally and commit them globally using a decentralized weighted-voting scheme [15]. Updates and voting information are propagated through the system *asynchronously* using an epidemic style of communication (e.g., [1, 8, 21, 23]). Epidemic communication exploits pair-wise *anti-entropy* sessions [8], 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 when the update corners the *plurality* of votes (i.e., when it is guaranteed 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 [7]). However, the same updates eventually commit at all servers in the same order. Note that, in this paper, we consider non-transactional environments and single-item updates.

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. Existing currency management mechanisms are heavy weight in that they typically require the participation of a majority of servers to create/retire replicas and install new currency values [6, 10, 13, 17, 24]. Deno uses light-weight, peer-to-peer mechanisms that

facilitate these operations, requiring the participation of only two servers.

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 basic 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. Overview

In this section, we present a brief overview of Deno replicated-object storage system.

2.1. Decentralized weighted-voting

We first briefly describe Deno's weighted-voting scheme. The details, with a sketch of the correctness proof, appear in [15].

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 $k < j$.

The voter breaks ties in rule (2) 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 ith election as long as:

(1) *v* has not awarded election *i* to any candidate, and
(2) *v* has not yet voted in the ith 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 [15] 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.*

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 tentative.
int `wait_update`(<update id>)	Waits for an update to be terminated (i.e., either committed or aborted).

Table 1: Basic Deno API

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 by the unavailable server itself (in case of planned disconnections) or through *proxy elections*.

Deno transparently handles voluntary 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.

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 [7], using the decentralized voting scheme described earlier.

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 [14]), 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 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.

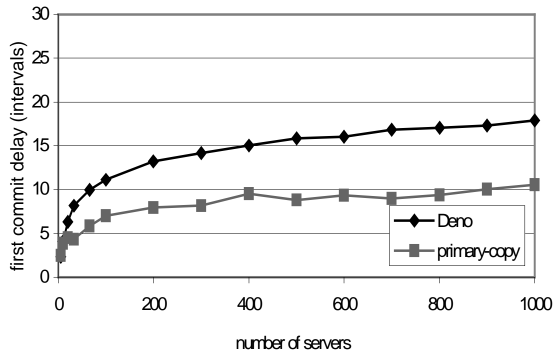

Figure 1: First commit delay (voting vs. primary-copy)

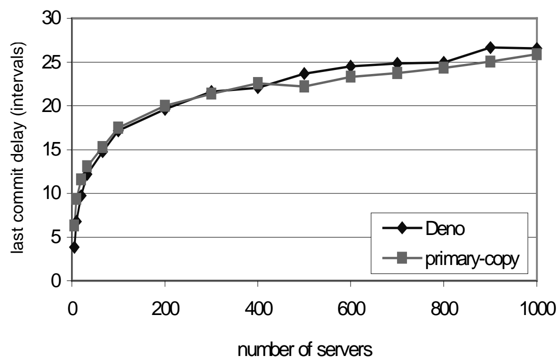

Figure 2: Last commit delay (voting vs. primary-copy)

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 [22].

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. We built a simple simulator to investigate Deno's basic protocol performance. We simulate a system in which time is broken into uniform intervals. Each server initiates a randomly-directed anti-entropy session during each interval. For purposes of the experiments, we assume uniform distribution of currency and a completely available, fully-connected system.

Figure 1 shows a plot of the average number of intervals needed to commit an update versus the number of servers for Deno's default (uniform) voting scheme and a Bayou-like [23] epidemic primary-copy scheme. In the latter scheme, updates are disseminated using epidemic flow and an update is committed when it is serialized at the primary-copy server. Such a primary-copy approach suffers from single-point failures; if the primary-copy server is unavailable, then no updates can be committed. The figure reveals that the primary-copy scheme commits updates significantly faster than the voting scheme. However, the time at which the *first* server commits an update is not necessarily the quantity that best predicts application performance. Since all servers have an equal chance of being read, a second interesting metric would be the time at which the *last* server commits an update. Figure 2 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.

In Figure 3, we compare the commit performance of Deno to a "write-all" type of epidemic protocol in which an update is committed only after *all* servers validate that the update can commit. If and when a server detects a conflict, the server aborts all the updates involved in the conflict to ensure correctness. Agrawal *et al.* [1] recently described a similar approach that, unlike the voting protocol presented in this paper, supports transactional multi-item updates and ensures serializability. Figure 4 suggests that the voting mechanism used by Deno consistently commits updates about 30-40% faster than write-all. This improvement is basically due to the fact that while write-all requires an update to be validated by all servers before committing the update, it is sufficient for an update to be validated by a majority of servers in Deno (assuming a uniform currency distribution). This feature not only yields performance improvements over the write-all scheme, but it also turns out to be crucial for making progress during times of low availability, accessibility and network partitions.

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 [4]. In this section, we investigate mechanisms that enable the implementation of arbitrary currency distribution policies while

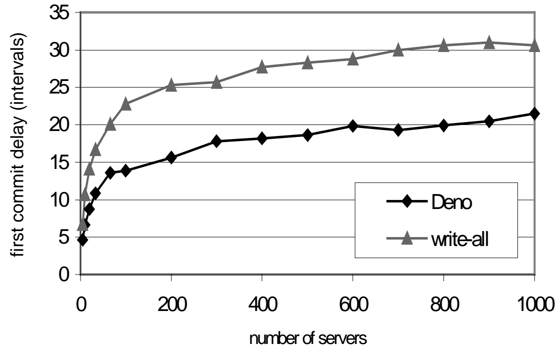

Figure 3: First commit delay (voting vs. write-all)

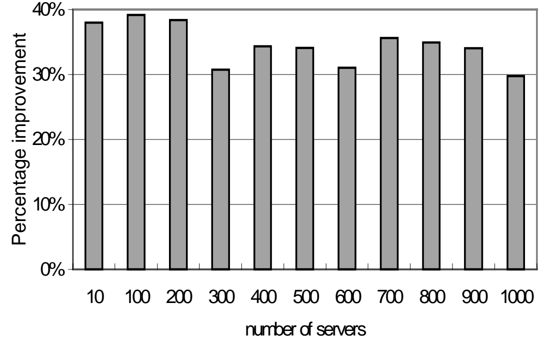

Figure 4: Percentage improvement in commit delays (voting over write-all)

still maintaining the correctness of the voting protocol. Note that the issue of finding *optimal* currency distributions is outside the scope of this paper and have been addressed by several previous work (e.g., [3, 4, 6, 13, 17]).

We first describe how replicas are created and currency is initially allocated. We then discuss protocols for dynamically re-allocating 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, which is held by the creating server. A new replica is created through a request to a server that already has a replica (Section 2.3). 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 pairwise 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 can clearly not guarantee to achieve a given distribution merely by allocation.

However, Deno applications can direct 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 mechanism

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 throughout the lifetime of the object.

Deno uses peer-to-peer *currency exchanges* to incrementally change existing distributions into arbitrary target distributions. A pair of servers communicates and redistributes their total currency according to some redistribution policy.

We now describe how to implement peer-to-peer currency exchanges while maintaining the correctness of our 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.

Restriction (1) is needed in order to prevent servers from reaching different conclusions on the outcome of a single election. The need for restriction (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:

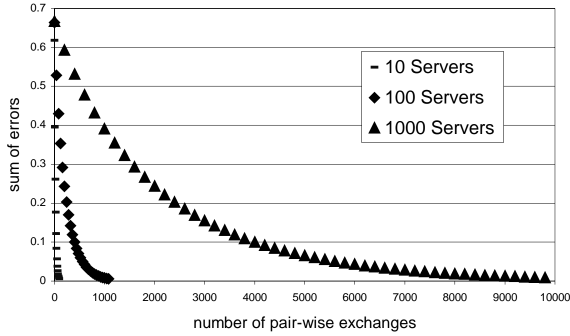

Figure 5: Converging to a target distribution with randomly selected peer-to-peer currency exchanges.

(i) if $e_i < e_j$, then $e = e_j$

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

Case (i) implies that it is possible for s_j to increase its vote during the same election. 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. Notice 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 currency distribution given by Amir and Wool [3], 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). Currency redistribution can also be used to improve commit performance, for instance, by redistributing currencies according to update creation frequencies at the servers.

Note that an existing currency distribution can be migrated to a target distribution without the need for any server to have global information (e.g., number of servers, current 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 also investigated the convergence speed of the pair-wise currency redistribution mechanism. We observed that randomly-selected, pair-wise currency exchanges allow an existing distribution to converge *exponentially* fast to any target distribution. We proved this result analytically for three servers, and the experimental results in Figure 5 suggest that the proposition generalize when there are more than three servers.

Figure 5 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. At each currency exchange, two servers redistribute their currencies proportional to their weights in the target distribution as described previously. The shapes of the plots in the figure demonstrate that the difference between the target and the existing distributions diminishes exponentially fast. As expected, the larger the number of replicas, the more the number of currency exchanges required to converge to the target distribution.

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 [6].

4. Related work

There has been significant work in the area of data and consistency management in mobile and weakly-connected environments [2, 5, 9, 12, 16, 18-20, 23]. Of particular relevance to our work are those proposals that exploit epidemic algorithms to propagate updates [1, 8, 14, 18, 21, 23]. 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., [14, 18]) 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 [23] 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. More recently, Agrawal *et al*. proposed a "write-all" 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. Deno differs from these pessimistic approaches in its use of a novel epidemic voting scheme primarily to achieve higher availability.

Voting schemes [10, 24] improve availability by allowing a *quorum* of all replicas to commit an update. Work on currency (e.g., weight) management mainly focused on *policies* that are used to *reassign* votes after site or link failures to improve availability [3, 6, 13, 17]. 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.

We note that recent work [25] investigated why voting systems (i.e., quorums) 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., 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 replicated-object storage system, and described how Deno implements a novel, decentralized weighted-voting scheme via epidemic information flow. We focused on the important issue of currency management, and described mechanisms that facilitate light-weight replica creation, retirement, and dynamic currency redistribution. Unlike previous protocols that typically require 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, without any server having complete knowledge of state of the system. Using simulation, we demonstrated that this convergence happens exponentially fast.

In terms of future work, we plan to extend Deno to perform transactional multi-item updates. We also plan to investigate dynamic currency redistribution and anti-entropy policies. We are currently implementing the Deno prototype on top of Windows32/WinCE environments to investigate these issues.

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] 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.

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

[4] 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.

[5] 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.

[6] 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, pp. 394-426, 1989.

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

[8] 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.

[9] M. Dunham and A. Helal, "Mobile Computing and Databases: Anything New?," *SIGMOD Record*, vol. 24, pp. 5-9, 1995.

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

[11] 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.

[12] T. Imielinski and B. R. Badrinath, "Wireless Mobile Computing: Challenges in Data Management," *Communications of the ACM*, vol. 37, pp. 19-28, October 1994.

[13] 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.

[14] 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.

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

[16] 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.

[17] A. Kumar and A. Segev, "Cost and Availability Tradeoffs in Replicated data concurrency control," *ACM Transactions on Database Systems*, vol. 18, pp. 102-131, March 1993.

[18] 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.

[19] 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.

[20] 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.

[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] 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.

[23] 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.

[24] 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.

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