Performance of Mobile, Single-Object, Replication Protocols

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 implementation and performance of bounded voting: a new object replication protocol designed for use in mobile and weakly-connected environments. We show that the protocol eliminates several restrictions of previous work, such as the need for (1) strong or complete connectivity, (2) complete knowledge of system membership, and (3) low update rates. The protocol implements an asynchronous, weighted-voting scheme via epidemic information flow, and commits updates in an entirely decentralized fashion. A proxy mechanism is used to enable transparent handling of planned disconnections.

We use a detailed simulation study to characterize the performance of bounded voting under a variety of loads and environment, and to compare it to another decentralized epidemic protocol. We further investigate the performance impact of the proxy mechanism.

1. Introduction

Weighted-voting schemes [10, 13, 18] have long been used in solving distributed consensus problems. Epidemic algorithms [2, 8, 20, 21] have long been used in environments with weak connectivity or uncertain topology. This paper investigates the use of a combination of these two techniques in supporting object replication in mobile and weakly-connected environments.

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 caching and replication mechanisms in order to deliver good performance.

Replication is widely used to enhance both reliability and performance in distributed systems. Traditional replication mechanisms, however, are ill-suited for mobile environments [11]. Mobility and weak connectivity require a critical reassessment of the assumptions underlying traditional replication mechanisms [3]. For instance, one assumption made by master-copy replication schemes [22] is that a single server is always available and accessible by the rest of the system. Clearly, such an assumption may become invalid in mobile environments. Server machines may be disconnected, and therefore inaccessible, at any given time. Furthermore, master-copy replication often assumes that the master server has complete and up-to-date knowledge of system membership, which is difficult to obtain in a mobile environment. As another case in point, consider replication in traditional *voting schemes*. Such schemes typically work by requiring a quorum of simultaneously connected servers to agree on an operation prior to performing it. If such a quorum cannot be established, the operation is aborted. However, mobile replication protocols should ideally allow progress to be made, and updates to be committed, even if a quorum of servers is not simultaneously available. Mobile replication protocols should therefore be *decentralized* and *asynchronous* wherever possible.

Mobility not only requires fundamental changes in the protocols designed for traditional desktop environments, but it also introduces its own variants to existing issues. One such issue involves planned disconnections. A server may declare its intention to disconnect, enabling the protocol to handle the disconnection more effectively than in traditional schemes in which the disconnections are detected and handled only after they occur [3]. The rest of the paper discusses these and other relevant issues, together with our approaches to handling them, in detail.

Proceedings of the 19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)
1060-9857/00 $10.00 © 2000 IEEE

1.1. System model and features

This paper describes the implementation and evaluation of *bounded voting* in Deno, a decentralized object-replication system. Bounded voting can be used to provide replicated-object support for applications in weakly-connected and mobile environments. Bounded voting is designed to support a wide variety of applications ranging from simple shared-calendars to domain-specific databases. More specifically, the 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.

Bounded voting allows the *update anytime-anywhere-anyhow* replication model [11] to be used in order to address requirements of disconnected operation. All servers are treated as peers in their ability to generate updates; no server owns any object. Consequently, bounded voting fundamentally differs from master-copy schemes like Bayou [24], which can be ill-suited for mobile and weakly-connected environments [11].

Bounded voting allows servers to execute updates locally and commit them globally using a decentralized weighted-voting scheme. Updates and voting information are propagated through the system *asynchronously* using an epidemic style of communication (e.g., [2, 8, 20, 21]) that requires only pair-wise communication. Updates gather votes as they pass through servers. An update is committed only when it corners the *plurality* of votes. As a result, no other conflicting update can commit. Update commitment is *decentralized* in that each server *independently* and *locally* commits or aborts updates. However, the same updates eventually commit at all servers and in the same order. Decentralized commitment eliminates the need for synchronous multi-site commits (e.g., two-phase or three-phase commits), which, again, are not well-suited for mobile and weakly-connected environments [11].

Epidemic protocols have been adopted by a number of vendors due to increasing replication factors and the need for asynchronous management of replicated data in their products. For instance, Lotus Notes [14] is a widely-used commercial system that uses epidemic propagation. In these protocols, updates are executed at any single server. Asynchronously, servers communicate at a convenient time to exchange information regarding the updates, detecting and bringing the obsolete copies up to date. Epidemic communication uses pair-wise *anti-entropy sessions* to inform servers of the state of other servers. Anti-entropy sessions ensure that all replicas of the same object *eventually* converge to the same final state [8]. Most epidemic protocols take an *optimistic* approach for maintaining data consistency. These protocols allow an update to be locally committed immediately after it is executed. If and when a server detects conflicting updates, it typically resolves the conflict in one of two ways. One approach involves prioritizing the updates based on timestamp, the server that initiated the update, etc. For instance, Lotus Notes employs a timestamp-based mechanism that favors the update having the higher timestamp value. The other update is simply discarded as a stale value. This approach suffers from the infamous *lost update* problem [4] where the effects of a committed update are not reflected in the database. The other approach for conflict resolution relies upon a process called *reconciliation* that attempts to merge the effects of the conflicting updates. Reconciliation of committed updates is feasible only in restricted domains, as, for example, in file systems [16, 17]. However, reconciliations cannot be easily handled in the general application domain that we address. In cases where a situation cannot be reconciled automatically, the protocols resort to manual reconciliation, requiring human intervention. Such a manual approach, clearly, is not scalable. Furthermore, Gray *et al.* argue that reconciliation-based systems suffer from *system delusion* as those systems scale up [11]. As a result, these optimistic approaches lead only to a restricted notion of correctness, which may be sufficient only for some application domains. Specifically, epidemic protocol are adequate for those applications where it can be assumed that most updates are commutative or where conflicts are infrequent and can be reconciled manually or automatically.

Bayou [20, 23, 24] takes a more *pessimistic* (i.e., conflict avoidance-based) approach, ensuring that all committed updates are eventually serialized in the same order at all servers using a primary-copy scheme. More recently, Agrawal *et al.* [2] proposed another pessimistic approach where an update is committed only after the update is certified by all servers that participate in the protocol. When a server detects a conflict among updates, it aborts all the involved updates to ensure correctness. This protocol provides serializability in a transactional framework.

A number of voting protocols have been proposed to improve availability in distributed applications [10, 13, 18, 25]. The fundamental idea behind voting is to synchronize a quorum of servers to agree on an operation prior to performing it. In voting schemes, conflicting operations imply overlapping server quorums so that conflicts are detected before they are performed. Uniform-voting schemes assign a single vote to each replica [25]. Weighted-voting schemes generalize uniform-voting by assigning a non-negative weight to each replica [10]. In [15], we extended previous weighted-voting schemes to allow voting to take place asynchronously, and without complete system membership information. This *bounded* weighted-voting scheme propagates in-

formation asynchronously through epidemic information flow. The use of voting allows higher availability relative to master-copy or master-commit schemes. The use of epidemic information flow allows the system to make progress in weakly-connected environments. This scheme can also emulate diverse configurations through proper currency distribution. For example, traditional dynamic voting schemes are emulated by uniform currency distributions, while a master-copy scheme is emulated by allocating all currency to a single server.

In summary, bounded voting provides the following features and functionalities that we deem highly desirable for our target environment and application domain:

1. *Tolerance for weak connectivity and incomplete information*: Our intended environment includes areas of weak and non-existent connectivity. Additionally, we do not assume that servers are fully connected. Even in the best of cases, individual nodes and devices might have direct contact with only a limited set of other devices. We provide support for arbitrary communication topologies by using a *peer-to-peer* synchronization model, that is, *any* two replicas can synchronize directly. Consequently, co-located or *nearby* machines can synchronize with each other quickly and inexpensively, which is a crucial feature in mobile environments. Furthermore, no server needs to have complete knowledge of the system, or even of the set of servers participating in the protocol, which allows graceful scalability. Updates are committed independently at each server through a decentralized voting protocol.

2. *Reconciliation- and compensation-free replica control*: Epidemic algorithms commonly require all application updates to be commutative. Bounded voting can be extended to take advantage of commuting updates, but the base protocol makes no commutativity assumptions. All updates to the same object are guaranteed to be applied in the same order on all replicas, thereby eliminating the lost update problem. Replicas eventually converge to the same final state. Once a server commits an update, that update will never be rolled back, which avoids system delusion [11] and is a base guarantee needed by many applications. Secondarily, the protocol never aborts all competing updates. As we will demonstrate, this allows progress to be made and updates to be committed regardless of the update rate.

3. *Transparent handling of planned disconnections*: Foreseeable disconnections are handled transparently via *proxies*. Before disconnecting, a server transfers its voting rights to another server. Votes for the disconnected server are cast by its proxy, making the disconnection transparent to other servers. This facility is aided by the protocol's light-weight, dynamic replica management. Any replica can be created or retired dynamically by communicating with any other server already holding a replica.

1.2. Contributions

The primary contributions of this paper are threefold. First, we describe an asynchronous, decentralized protocol specifically designed for mobile and weakly-connected environments, and show that it eliminates several restrictions of previous related work. Second, we use a detailed simulator to characterize the performance of the protocol under a variety of scenarios and environments, and to compare its performance to that of another decentralized epidemic approach. Third, we investigate the performance impact of the extension of the base protocol with a proxy mechanism that facilitates transparent handling of planned disconnections.

The rest of the paper is organized as follows. Section 2 describes the base bounded-voting scheme. Section 3 discusses implementation issues. In particular, we discuss how to create and retire replicas, how to allocate and redistribute the currency, how to transparently handle planned disconnections using proxies. We describe our experimental environment in Section 4, and characterize Deno's protocol performance in Section 5. We discuss related work in Section 6, and conclude in Section 7.

2. Decentralized weighted voting

We now briefly describe the bounded voting scheme. The details of the base protocol, along 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 (i.e., selective replication) and multiple objects can be replicated at the same server. For simplicity of exposition, however, we limit our discussion to single objects that are cached at all servers. Our discussion is easily extended to include the more general case.

Objects are modified by *updates*, which are issued by servers. Updates can be transmitted to other servers and are assumed to execute atomically at remote servers. Updates do not commit globally in one atomic phase, as we use pair-wise synchronization and assume poor connectivity. 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

Proceedings of the 19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)
1060-9857/00 $10.00 © 2000 IEEE

Definition 1: *Define* uncommitted(v_i) *as*:

$$\sum_{j=1}^{n} v_i.curr[j] \text{, s.t. } v_i[j] \text{ is equal to } \perp.$$

Definition 2: *Define* votes(v_i, k) *as*:

$$\sum_{j=1}^{n} v_i.curr[j] \text{ s.t. } v_i[j] \text{ is equal to } k.$$

Definition 3: *A candidate c_j wins v_i's current election when:*

1. *votes(v_i, j) > 0.5, or*
2. $\forall\, k \neq j$,
   (a) *votes(v_i, k) + uncommitted(v_i) < votes(v_i, j) or*
   (b) *(votes(v_i, k) + uncommitted(v_i)) = votes(v_i, j) and ($j < k$)*

Figure 1: Definitions

eventually commits everywhere, and in the same order with respect to other committed updates.

2.1. Elections

A clean way of thinking about update commitment is as a series of elections. A server is analogous to a voter, creating an update is analogous to a voter deciding to run for office, and a committed update is analogous to a candidate winning the election. Voters (and hence candidates) have indexes 0 through n-1, where n is the total number of voters. We use v_i to refer to the voter with index i, and c_i to refer to the candidate with index i. Candidates win elections by cornering a plurality of the votes. Each election begins with an underlying agreement of the winners of all previous elections. Once an election is over, a new election commences. Any given election may have multiple candidates (logically concurrent tentative updates), and candidates from different elections might be alive in the system at the same time. In the latter case, however, uncommitted candidates for any but the most recent election have already lost, but this information has not yet made it to all voters.

Because of the style of information flow, there is no centralized vote counting. Instead, each voter independently collects votes from other voters and deduces outcomes. This method creates situations in which the *current* election of distinct servers is temporarily out of sync. Voter v_i's current election is the election for which v_i is collecting votes. In order to implement this protocol, each voter maintains three pieces of state:

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

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

Figure 1 presents some important definitions used in this section. Definition 3 essentially says that a candidate wins with a voter if it has a majority or plurality of the vote. Ties are broken with a simple deterministic comparison between the indexes of the servers that created the competing updates. The winner of the jth vote at v_i is denoted v_i*.commit*(j). When an election is won at v_i, all votes $v_i[j]$ are reset to $\perp$.

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

2.2. Illustration

Newly created updates are *tentative*, and may be discarded without ever being committed. Tentative updates may or may not be visible to the application, depending on the type of session guarantees needed by the application [23]. Updates are *committed* when servers holding a plurality of the object's currency agree that they are acceptable. We now illustrate how the protocol works by two examples:

Example 1: Figure 2(a). Objects x and y are replicated at servers v_1 through v_4. Each server has currency of 0.25 for both objects. Server v_1 creates a tentative update to x at time t_0. At time t_1, v_1 sends information to v_2, and at time t_2, v_2 sends to v_3. At this point, three of the four replicas know of the tentative update and have ordered it before any other tentative updates to x. These replicas can commit $u_{1,1}$ because they control 75% of the object x's currency. However, only v_3 knows this. Not knowing of the first election's outcome, v_4 naively creates a new update, $u_{4,1}$ at time t_3. This update will be aborted at t_4 when v_4 learns that a majority has already determined that $u_{1,1}$ should be committed.

Example 2: Figure 2(b) shows an example of two competing updates being started at time t_5. Each synchronizes with one other replica at t_6, leading to a potential stalemate in which each competing update has 50% of the currency. While currency allocation schemes could be rigged to prevent this from occurring in the case of two competing updates, three or more competing updates could still lead to the same problem. The lexicographic tie-breaker will favor $u_{1,2}$ over $u_{4,2}$.

Proceedings of the 19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)
1060-9857/00 $10.00 © 2000 IEEE

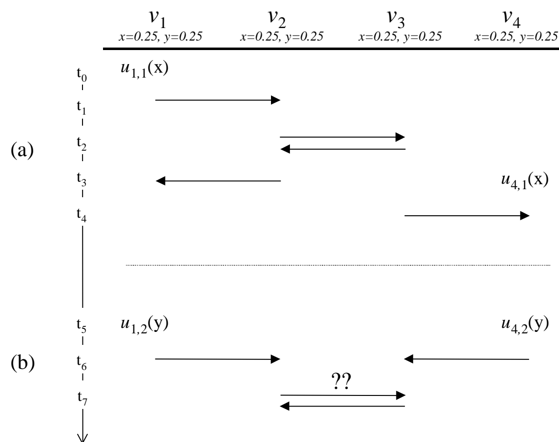

Figure 2: Protocol illustration - Four replicas each of objects x and y. $u_{i,j}$ is the update created by v_i in election j. Currency is divided evenly for both replicas. **(a)** shows the progress of update $u_{1,1}$ from v_1. The update is committed because a majority of the object's currency *observes*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

3. Protocol implementation

This section describes implementation issues and an extension to the basic protocol. Objects are initially created with a total currency of 1.0, which is held by the creating server. A new replica can simply be created by sending a request to a server that already has a replica. The response to such requests contains both the object's data and some amount of currency. This amount is subtracted from the currency held by the existing replica. The total amount of currency in the system remains constant during failure-free operation. A replica can be retired using a similar pair-wise communication in which the currency held by the retired replica is transferred to another replica. These mechanisms enable light-weight replica creation and retirement as currency transfers need to involve no more than *two* servers.

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

3.1. 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 availability and currency distribution of the object replicas. We divide currency management into three parts. First, a target currency distribution has to be identified. Second, an allocation strategy, which specifies how currency is handed out when replicas are first created, must be defined. Finally, there must be a policy specifying what currency exchanges are allowed at runtime, if any.

In general, the *best* currency distribution depends on application semantics, expected availability of individual servers, and network topology. Initial allocation is non-trivial not only because no server can have accurate knowledge about the size of the anticipated set of servers, but also there is generally not a specific server that receives all the allocation requests. The respondent can be any server, therefore we cannot guarantee to reach a target currency distribution merely by allocation. Deno uses light-weight *peer-to-peer currency exchanges* [6] to incrementally change existing currency distributions into arbitrary target distributions. An important feature of peer-to-peer exchanges is that servers can reach arbitrary global currency distributions *exponentially* fast and using only local information, without the need for global synchronization. A detailed discussion of currency management in Deno appears in [6].

3.2. Fault tolerance

This section presents an overview of failure detection and handling. Deno achieves fault-tolerance through a proxy mechanism. Proxies represent *failed* servers in the system and are selected either by the failed server itself (in case of expected disconnections) or through proxy elections. We first introduce the notion of currency proxies and how they enable transparent handling of planned disconnections. We then discuss how the same mechanism can be used to tolerate real failures.

Planned disconnections and currency proxies: Predictable, planned disconnections constitute a benign failure mode unique to mobile environments. Unlike *real* failures that are detected only after they occur, planned disconnections enable special actions to be taken before the failure (i.e., disconnection) occurs. Deno uses a proxy mechanism to *transparently* handle planned disconnections. The basic idea is to have a *primary* engage a proxy to vote in its place while the primary is disconnected. The use of proxies in this fashion can prevent degradation in the overall commit rate when devices have expected, planned-for disconnections. An example where proxies would be useful is when a laptop is taken on a trip where no other servers will be available. The laptop's currency can be transferred to a desktop machine for the trip's duration.

Deno's approach is to have the proxy server vote the primary's currency as its own while the proxy server is engaged. A proxy vote is then indistinguishable to other servers from the situation where a server votes and then disconnects. When a primary reconnects, it updates its

Proceedings of the 19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)
1060-9857/00 $10.00 © 2000 IEEE

own information to match that of the proxy, including votes on prior and current tentative updates. The primary treats any votes cast in its behalf as if they had been cast directly by the primary. In particular, any votes cast for tentative updates remain cast. The result is that there are no race conditions, and the entire proxy engagement is *transparent* to the rest of the system.

Proxies whose primaries fail can permanently vote the primary's currency. The advantage of this approach is that even the failure is transparent to the other servers, although the failure eventually has to be made explicit and addressed. Proxies can be transferred when proxies plan a disconnection. Reconnecting primaries can locate their proxy by checking auxiliary data appended to any proxy vote. This data specifies which server voted for the primary. The returning primary can retrieve its currency directly from this server. A proxy that fails unexpectedly prevents the primary from participating in elections until either the proxy re-connects, or is judged failed.

Failures and proxy elections: Failure detection in the domain of mobile applications is difficult because servers may be out of contact either temporarily or permanently. No action should be taken in the former case, but action must be taken in the latter because the currency held by the server can prevent updates from committing.

Detecting permanent disconnections is the first problem. Simple timeouts are not workable because disconnection is the rule rather than the exception. Disconnections are not only potentially frequent, but might be quite lengthy. A second approach is to count the updates that commit without a vote from the server in question. The advantage of this approach is that servers planning disconnections will designate proxies to vote their currency. Hence, votes are only not cast by servers that are unexpectedly out of touch with the rest of the system.

Once a permanent disconnection is detected, action must be taken to recoup the currency held by the disconnected server. Loss of this currency can either slow or completely prevent updates from being committed. The protocol can compensate for failed replicas via *proxy elections*.

The main idea is to collectively elect a server to act as a proxy to the failed server. Proxy elections are performed similarly to coordinator election protocols widely used by many distributed protocols [4]. After detecting a failure, a server initiates a proxy election update. As with other changes to objects, a proxy election update is a special type of update 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 weighted-voting scheme to commit them. One implication is that a proxy election can only occur if a majority of the current currency is available. This 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, it resets its current currency to zero. The server may then request its currency back from its proxy or obtain currency from other servers through peer-to-peer exchanges (Section 3.1).

Parameter	Description	Setting
UPS	Mean global update rate (uniform)	0.0-10.0 (updates/s)
EPS	Mean anti-entropy rate (uniform)	0.1, 1.0, 10.0 (entropies/s)
MsgLatency	Mean message latency (exp.)	75 (msec)
NumSites	Number of object replicas	10-1000

Table 1: Primary simulation parameters

4. Experimental environment

Deno's bounded voting protocol removes reliance on any single master server, and allows progress to be made without synchronous global consensus. Clearly, however, these advantages are not without cost. Before quantifying the performance of bounded voting, we first describe our simulation environment.

4.1. Simulation model and assumptions

We implemented a detailed simulator using the CSIM simulation package [1]. Although the simulator is quite detailed and provides high accuracy, we give only a high-level description of it due to space limitations.

Table 1 summarizes the main simulation parameters and settings used in the experiments. The simulator models a number of distributed servers that replicate objects and perform updates on their copies using bounded voting. Servers periodically initiate anti-entropy sessions, with a rate denoted by *EPS,* and update their states. Updates are generated according to a global update rate given by *UPS*. After an update is generated, it is injected to a server selected at random. Servers have two modes of operation: (1) connected, and (2) disconnected. Server connection and disconnection periods are determined independently at distinct servers, and are derived using an exponential function. When a server is disconnected, it cannot send or receive messages, but it can still create tentative updates. The default model employs a fully-connected communication topology with uniform communication delays given by *MsgLatency* (set to model a wide-area network). Unless otherwise stated, we assume uniformly distributed currencies, and no disconnections.

4.2. Performance metrics

One metric we focus on is the *commit rate* that denotes the number of committed updates per second.

Proceedings of the 19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)
1060-9857/00 $10.00 © 2000 IEEE

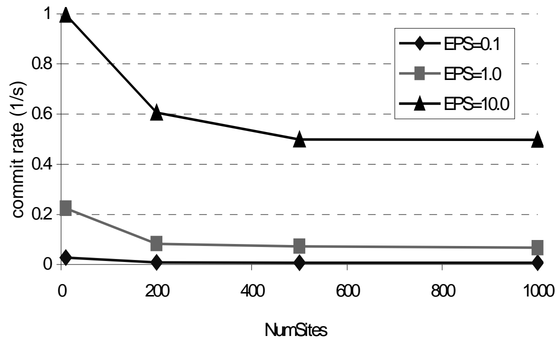

Figure 3: Base commit rates with fixed update rate (UPS=1.0)

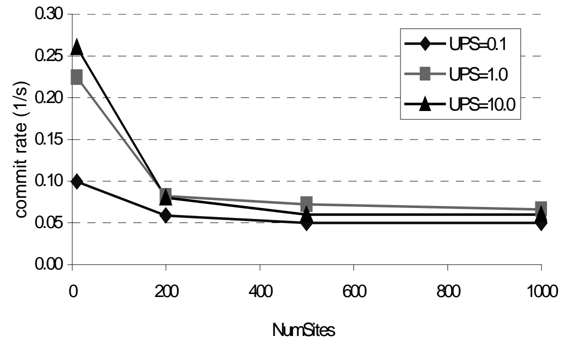

Figure 4: Base commit rates with fixed anti-entropy rate (EPS=1.0)

Whenever appropriate, we also present *commit percentage* results that denote the percentage of updates that are committed. As the protocols we investigate are asynchronous and servers typically learn of committed updates at different times, we report two different commit delays. *First commit delay* (*FCD*) represents the difference between the time an update is initiated and the time it is *first* committed at some server. *Last commit delay* (*LCD*) is the time until all servers commit the update. In an environment where updates are propagated asynchronously, LCD is also a significant metric because all replicas have the same probability of being read and a committed update is not useful until it is available to a server.

4.3. Protocols studied

In addition to the Deno protocol, we also investigate the performance of a "Read-One, Write-All" (ROWA) [4] type pessimistic epidemic protocol. For the sake of brevity, we only give a high-level description of the protocol. This asynchronous protocol, which we refer to as *Write-All* (*WA*), works by disseminating log records that corresponds to updates using an epidemic model. Similar to Deno, updates are executed locally and then committed globally. When a server learns about an update, it checks whether there exist any conflicting updates. If a server detects a conflict, it aborts the conflicting updates. The abort records are also propagated to other servers to ensure that an update, if aborted, is aborted at all servers. An update is committed if it is certified at all servers. Agrawal *et al*. proposed a ROWA-type epidemic protocol similar to what we have described above, that, however, also supports transactional semantics and serializability [2].

5. Experimental results

We are now in a position to present the results of performance experiments that illustrate the performance of Deno. We first present results that demonstrate the performance of the base bounded-voting protocol, comparing it to the WA protocol, and exploring its performance under a variety of connectivities. We then characterize the performance impact of extending the base protocol via a proxy mechanism.

5.1. Basic performance

Figure 3 and Figure 4 show the rate at which Deno commits updates versus the number of servers for several different anti-entropy and update rates, where the currency is uniformly distributed and servers perform randomly-directed anti-entropy sessions. The leveling-off of commit rates results from conflicting updates. Recall that updates compete to win elections, with losing updates being aborted by default. As the average election has increasing numbers of participants, a lower percentage is committed, and those that do commit are committed more slowly because winning an election in the face of competition requires a higher percentage of servers to participate before the winning update is determined.

Figure 5 shows the commit percentage results for Deno and the WA approach for different levels of update contention (notice the exponential scale on y-axis). As expected both approaches suffer from increased update contention. However, we observe that Deno significantly outperforms WA over the entire range of update rates, committing orders of magnitude more updates than WA under high contention rates with relatively large number of servers. Even with 10 servers, Deno commits about twice as many updates as WA under high update rates. Another important point to observe is that although WA cannot commit any updates beyond some update rate (recall from Section 4.3 that WA aborts all conflicting updates to ensure consistency), Deno continues to make progress and commit updates *regardless of the update rate*.

Proceedings of the 19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)
1060-9857/00 $10.00 © 2000 IEEE

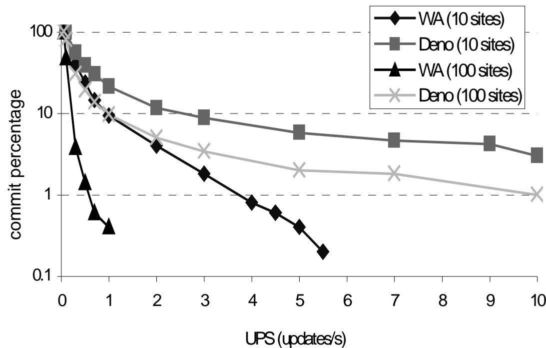

Figure 5: Percentage of committed updates for Deno and WA (EPS=1.0)

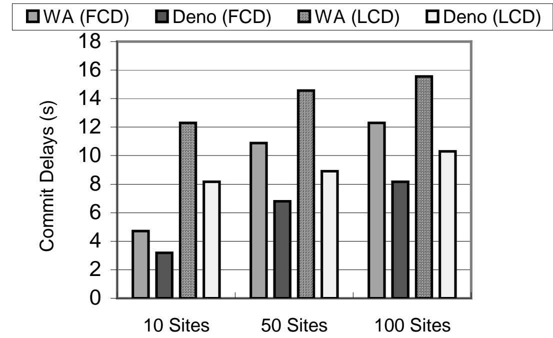

Figure 6: Commit delays for Deno and WA w/o update contention

Figure 6 compares Deno and the WA approach in terms of how fast they commit updates *without* update contention; i.e., commit percentage is 100%. This is achieved by initiating and committing individual updates in isolation. We observe that Deno consistently commits an update about 30-40% faster than WA. This is basically due to the fact that while WA requires an update to be certified by all servers before committing it, it is sufficient for an update to be certified by a plurality of servers in Deno (in case of no contention, plurality is practically majority). We observe that the propagation delay of a committed update is also significantly smaller for Deno. This is due to the decentralized commitment enabled by Deno: In WA, an update is committed only at a single server and the commit decision is then propagated to other servers. In Deno, however, the same update can be committed independently at different servers (typically by using different quorums), significantly reducing the time for all the servers to learn about the commitment of the update.

Our approach differs from typical primary-server architectures both in that we use a voting scheme, and in that we use background anti-entropy messages to decide elections. The use of anti-entropy adds a clear performance penalty versus systems where a primary copy can be contacted directly. Set against this are the advantages of being able to make progress in cases of low connectivity and less than fully-connected topologies.

The primary reason for using a voting scheme is to increase availability. Assuming independent failure modes, voting is provably optimal when all servers have a failure probability of less than 1/2 [19]. With failure probabilities at least 1/2, fewer than half of the servers will be connected at any one time on average, and a simple weighted-voting scheme will be unable to commit any updates most of the times. A monarchy or master-copy approach would clearly be preferable.

The above argument assumes that a majority of the servers need to be simultaneously connected in order to make progress. However, Deno uses anti-entropy to move information, arriving at decisions to commit updates asynchronously. Hence, Deno's weighted-voting protocol can commit updates even when no more than two servers are ever simultaneously connected. This is one of the clear advantages of using an asynchronous approach.

Figure 7 shows commit rates versus the percentage of servers that are available (i.e., connected) simultaneously. For the purposes of this experiment, the percentage of connected servers is made constant by having one server connecting while another is disconnecting. The figure clearly demonstrates the ability of Deno to make progress and commit updates at all availability levels. Note that while commit rates increase almost linearly with connect probability, there is no knee in the curve at 50%, the point at which a conventional voting protocol would cease to commit any updates. The reason that commitments can continue is that the asynchronous anti-entropy sessions allow servers to communicate indirectly with other servers that are not simultaneously connected.

5.2. Impact of the proxy mechanism

In Section 3.2 we introduced our proxy mechanism and how it can be used for fault-tolerance. Here, we show that proxies are also useful in improving system performance. Figure 8 shows commit rates both with and without proxies, where each server has a probability 0.5 of being up at any given time. Either one or ten percent of the servers (i.e. proxy group ratio 0.01 or 0.1) serve as *proxy servers*. For purposes of this experiment, proxy servers are assumed to be reliable, while other servers connect and disconnect with mean intervals of one second. Not only do proxies reduce performance degradation when availability decreases, but they can even improve commit latency versus the case with completely available servers. This is because proxies cause currency to be concentrated in fewer servers, and fewer rounds of communication are required to establish a majority and commit an update.

Proceedings of the 19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)
1060-9857/00 $10.00 © 2000 IEEE

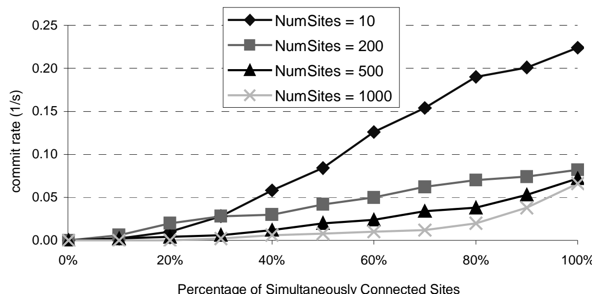

Figure 7: Commit rates as the percentage of simultaneously connected servers are varied, EPS=1.0, UPS=1.0

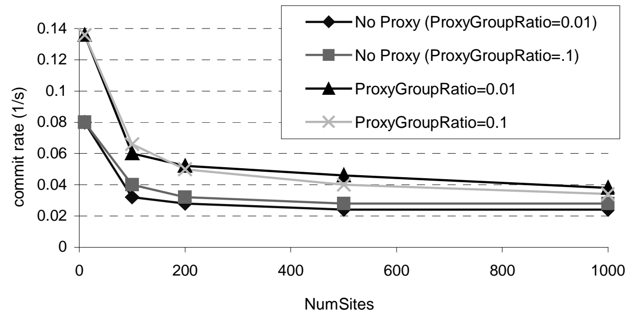

Figure 8: Commit rates with and without proxies, EPS=1.0, UPS=1.0

6. Related work

We discuss related previous work below. Further related work on voting, epidemic protocols, and transaction semantics is referenced in the text where appropriate.

Coda [16] and Ficus [17] share many of the goals of our work in the more limited domain of distributed file systems. This choice in domain allows the use of strong assumptions on the relative scarcity of contention. Additionally, reconciliation can be automated for many types of files. Hence, these systems both use replication that is optimistic in the sense of allowing conflicting transactions to commit. Bayou [24] uses epidemic information flow via anti-entropy sessions. However, Bayou differs from Deno in that objects are committed through a master-copy rather than a voting scheme.

Rabinovich *et al*. [21] addressed the issue of reducing the amount of data transferred during anti-entropy sessions. Agrawal *et al*. proposed a ROWA class of epidemic algorithms for transactional multi-item updates [2]. The protocol described in this paper is intended for atomic single-item updates suitable for non-transactional environments and applications (e.g., file systems), and does not provide transactional semantics. Deno, however, differs from the ROWA-type epidemic approaches in at least two fundamental ways. First, our protocol is designed to make progress and eventually commit updates even if there are conflicting, logically concurrent updates, whereas a ROWA approach has to abort all logically concurrent updates, losing all the progress made that far. Second, our protocol is highly decentralized which, in addition to making it well-suited for mobile-environments, yields major performance benefits. An epidemic ROWA approach requires the participation of all servers before an update can commit, whereas our approach eliminates such a severe restriction.

We introduced our bounded weighted-voting scheme in [15], and discussed its theoretical aspects and correctness. We also presented a high-level, preliminary description of the Deno weakly-consistent storage system. We described the light-weight currency management mechanisms used in Deno in [6].

More 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 [7]. The extended transactional protocols we proposed provide two levels of consistency; strong serializability [4] and update serializability [5, 9]. Independent of our research on transactional voting protocols, Holliday *et al.* also proposed a quorum-based epidemic approach that provides strong serializability and transactional semantics [12]. Holliday's work assumes a more traditional replicated database environment, and static, globally-known currencies, whereas our work is geared more towards environments with weak-connectivity and incomplete system information.

7. Conclusions and future work

We described an asynchronous, decentralized replicated-object protocol for mobile and weakly-connected environments, and demonstrated that this protocol eliminates several significant restrictions inherent to previous work. The protocol implements a new decentralized weighted-voting scheme using epidemic information flow. The protocol is pessimistic, and therefore requires neither compensation nor reconciliation for consistency. The voting scheme guarantees to commit an update out of each group of competing updates, and makes progress in a variety of situations in which previous protocols would abort all conflicting updates to ensure correctness. The protocol transparently handles planned disconnections, a frequent activity in mobile environments.

We investigated the performance of our protocol under different workloads and configurations using a detailed simulation model. Comparison with a ROWA-type decentralized epidemic protocol showed that the voting protocol performs better for all update rates. In addition to characterizing the performance of our base protocol, we also investigated the performance impact of the proxy mechanism, and demonstrated that cur-

Proceedings of the 19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)
1060-9857/00 $10.00 © 2000 IEEE

rency proxies cannot only handle planned disconnections transparently, but also increase system performance.

We have recently implemented a Deno prototype that runs on top of Win32 and Linux platforms [7]. We plan to investigate dynamic synchronization policies (i.e., when, what, and with whom to synchronize?) using our prototype. A synchronization policy needs to consider a variety of environmental factors such as the available bandwidth, communication costs, 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. Adaptive synchronization policies will form the basis of our future work.

8. References

[1] "CSIM 18 Simulation Engine Manual (C++ version)," Mesquite Software, Inc.

[2] D. Agrawal, A. E. Abbadi, and R. Steinke, "Epidemic Algorithms in Replicated Databases," in *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 1997.

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

[5] P. Bober and M. Carey, "Multiversion Query Locking," in *Proc. of the VLDB Conference*, British Colombia, Canada, 1992.

[6] U. Cetintemel and P. J. Keleher, "Light-Weight Currency Management Mechanisms in Deno," in *The 10th IEEE Workshop on Research Issues in Data Engineering (RIDE'2000)*, February 2000.

[7] U. Cetintemel, P. J. Keleher, and M. J. Franklin, "Support for Speculative Update Propagation and Mobility in Deno," UMIACS, UMIACS-TR-99-70, Oct. 29, 1999.

[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*, 1987.

[9] H. Garcia-Molina and G. Wiederhold, "Read-Only Transactions in a Distributed Database System," *ACM Transactions on Database Systems*, vol. 7, pp. 209-234, June 1982.

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

[11] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The Dangers of Replication and a Solution," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, June 1996.

[12] J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi, "Epidemic Quorums for Managing Replicated Data," in *IPCCC'2000*, Phoenix, Arizona, 2000.

[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 *Proc. of the Conf. on Computer Supported Cooperative Work*, 1988.

[15] P. J. Keleher, "Decentralized Replicated-Object Protocols," in *The 18th Annual Symposium on Principles of Distributed Computing (PODC '99)*, May 1999.

[16] J. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

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

[18] J.-F. Pâris and D. D. E. Long, "Efficient Dynamic Voting Algorithms," in *Proceedings of the Fourth International Conference on Data Engineering*, February 1988.

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

[20] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible Update Propagation for Weakly Consistent Replication," in *16th ACM Symposium on Operating System Principles*, Saint-Milo France, October 1997.

[21] M. Rabinovich, N. H. Gehani, and A. Kononov, "Scalable Update Propagation in Epidemic Replicated Databases," in *International Conference on Extending Database Technology (EDBT)*, 1996.

[22] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRESS," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 188-194, May 1979.

[23] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session Guarantees for Weakly Consistent Replicated Data," in *3rd International Conference on Parallel and Distributed Information Systems (PDIS 94)*, September 1994.

[24] 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*, 1995.

[25] 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.

Proceedings of the **19th IEEE Symposium on Reliable Distributed Sustems (SRDS'00)**
1060-9857/00 $10.00 © 2000 IEEE