Decentralized Replication Mechanisms in Deno

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

We are currently finalizing the design of Deno, a new shared-object system intended for use with replicated mobile and wide-area data. The broad aim of our research is to develop a framework for highly-available, decentralized shared-object protocols. The key idea is that our protocols will support high availability through a distributed voting scheme. Specifically, we will investigate (a) peer-to-peer updates, which will allow incremental progress to be made in the absence of full connectivity between component servers, (b) voting rather than centralized schemes for committing updates, ensuring that no single point of failure can prevent updates from being committed, and (c) application-specific consistency control, allowing applications to relax coherency constraints in ways that do not break the application's notion of consistency.

Distribution and multiple connectivity modes are becoming the norm rather than the exception in current computing environments. Thus, we expect the impact of our research to be felt in areas as disparate as mobile computing and collaborative data warehousing on the Internet.

1. Introduction

We are currently finalizing the design of Deno, a new shared-object system intended for use with replicated mobile or wide-area data. The broad aim of this project is to develop a framework for decentralized protocols that support replicated shared-object systems. Such systems are used to maintain data coherency for applications used in mobile and other highly dynamic environments. The distinguishing feature of such environments is that timely connectivity to specific hosts is rarely a given. Internet connections can be either slow or non-existent for a variety of reasons, including slow connections (via a modem, for example), unexpected network partitions, and periodic problems such as the data storms that occur daily at noon and 5 p.m. when office workers surf the web. Mobile systems have even more problematic connectivity. Laptops and PDAs are often disconnected the majority of the time, relying on brief periodic connections to synchronize application data on the mobile device with respect to copies of the data residing on corporate intranets.

The problem with this type of connectivity is that most existing consistency protocols use a *primary commit* [1] scheme. In order for any update to be considered permanent and become visible at other sites, it must first be accepted by the server that maintains the primary copy. Regular progress can only be made if the primary copy's server is continuously accessible.

This ubiquitous connectivity is not available in the environments discussed above. The result is that if the primary copy of a collaborative report resides on a worker's laptop, changes to the report can only be committed while the laptop is connected. Contrarily, if the primary copy resides a corporate server, no changes can be committed by disconnected laptops even if the laptops of all active participants in writing the report are connected to each other through an ad hoc wireless network in a conference room.

Our approach to this problem includes the following key ideas:

1) *peer-to-peer synchronization* – Updates to not need to be retrieved from the primary copy's server. Instead, updates propagate through the system through peer-to-per anti-entropy sessions [2].
2) *voting via a currency abstraction rather than primary commit* – The protocols will be highly available because existing replicas vote on whether to commit an update, rather than relying on a possibly unavailable primary copy to order all updates. Votes are conducted through a currency abstraction, which is allocated to replicas at creation time.
3) *fine-grained application control* – The use of the currency abstraction to control voting will enable the use of proxies, decentralized replica creation, and fine-grained control over commit policies. These capabilities will be exported to the applications to allow the use of application-specific notions of coherency.

2. Background

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

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

The most straightforward solution to this problem is to designate one replica as the *primary copy*. The order in which updates arrive at the primary copy is designated as the only correct order, and updates are required to be applied in this order at every replica. This approach has two drawbacks. First, the primary copy can become a performance bottleneck for updates to the object. More importantly in the context of a distributed environment, no updates can be committed, and no application progress made, without contacting the primary copy. Unavailability of the primary copy brings the entire system to a halt.

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

3. Deno design

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

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

The basic Deno API consists of the calls listed in Table 1. These calls allow new servers, objects, and replicas to be created, and replicas to be updated and destroyed. Proxy calls can be used by servers to delegate voting rights for planned disconnections. The sparse interface avoids burdening applications with unwanted or unneeded abstractions and functionality. For example, we provide no means of backing up objects to stable storage. Some applications will have no need for stable storage, while others can provide their own solutions by accessing the objects directly through the object pointers. Deno does provide support for transparent fault tolerance via the replication mechanism.

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

Our initial system will support two types of objects: binary objects and Tcl [3] strings. Binary objects are arbitrary byte-streams. The `Obj` structure used by several of the API calls is a union that contains a pointer and length for binary objects. Calls to `deno_replica_update()` are made on either side of the actual updates in order to delimit the update

Interface Call	Semantics
Deno_server_create([server name])	Creates server with optional name.
deno_object_create(<name> <initial Obj> [exp. #])	Creates new object. Optional third argument gives the expected number of eventual replicas.
Obj deno_replica_create(<name> [<server hint>])	Creates local replica of named object. The optional server hint tells Deno where to look for an existing replica.
deno_object_resize(Obj, int sz)	New size for binary Deno object.
deno_replica _update(<name> <update> [<merge proc>])	Update an object replica. Update and optional merge procedures are specified as Tcl scripts.
deno_replica_proxy(<object name> <server name>)	Delegate authority while disconnected.
deno_replica_unproxy(<object name>)	Retrieve delegated authority.
deno_replica_delete(<name>)	Delete local replica.

Table 1: Basic Deno API

interval to the underlying system. The actual updates consist of simple writes and/or calls to `deno_object_resize()`. Modifications to the object are detected through simple byte comparisons between before and after versions of the object.

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

Note that the system can easily maintain enough information to back out of either type of update. Deno could therefore provide any type of session guarantees[4]. By default, however, only committed values are visible.

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

3.1 Pairwise Information Dissemination

Deno will use anti-entropy sessions to propagate updates among replicas. This approach has been investigated in the context of a *primary commit* scheme in the Bayou [5] system. Our protocols will differ significantly in that they will rely on fewer assumptions on the completeness of available replica information. For example, our protocols will need to propagate updates to shared objects in the absence of knowledge of the complete set of replicas, or even of a primary copy that has pointers to all extent replicas.

The above problem, and many others, is greatly complicated by the fact that we assume peer-to-peer data exchanges, rather than more centralized update schemes. Both tentative and committed updates propagate between servers indirectly through periodic *anti-entropy* sessions. An anti-entropy session consists of two peers updating each other on all object updates not seen by the other. Hence, data moves through the system slowly, one step at a time.

The advantage of such a scheme is that all copies will eventually be updated, without any requirement that there ever be a direct connection between every pair of replicas. The disadvantage is that data moves slowly through the system, making global consensus more difficult to achieve.

3.2 Merge procedures

Deno supports the use of *merge procedures*, which are used to determine whether a given update can be applied without breaking application constraints. The restriction that no two entities can reserve the same meeting room at the same time in a meeting room scheduler is an example of one such object constraint. Merge procedures are specified in Tcl, and accompany tentative updates as they are propagated from one replica to another.

4. Committing Updates

Intuitively, progress should be possible in both of the situations enumerated in Section 2. Assume that there are four copies of the shared data: one copy each on the laptops of Frank, Joe, and Nancy, and one copy on the corporate mainframe. Three copies are in contact in both of the above scenarios. Recall that consistency is maintained if updates are eventually applied in the same order on every machine. One way to achieve this in the above scenarios is to say that any three copies can agree

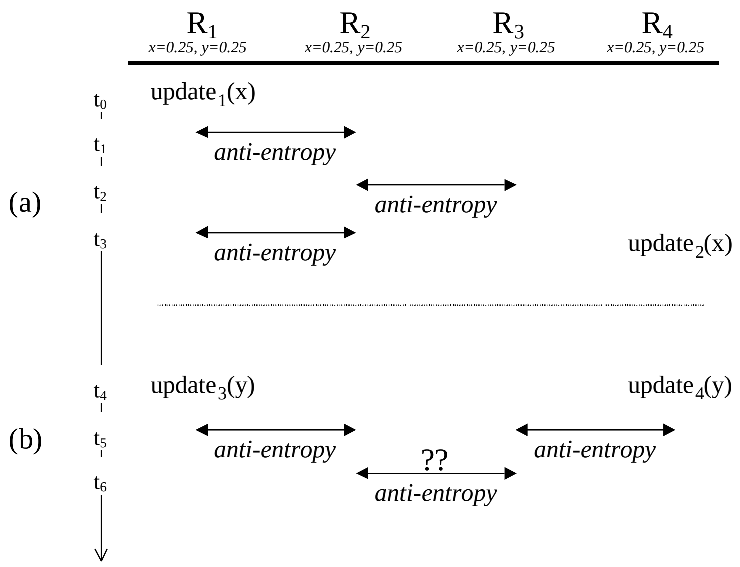

Figure 1: Four replicas each of objects *x* and *y*. Currency is divided evenly for both replicas. **(a)** shows the progress of an update from R_1. The update is committed because a majority of the object's currency "sees" it before any competing update. **(b)** shows two competing to *y*. At time t_6,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

to commit an update. This rule allows progress to be made in both situations, and prevents any conflicting updates from being generated on the remaining disconnected machine. This rule is an example of a voting or consensus approach to distributed agreement. Voting policies can be used in many situations to ensure coherence.

The main drawback of traditional voting policies in a distributed, possibly disconnected environment is that it requires consensus on group membership. A positive "vote" requires that more than half of the replicas vote in favor of an update before it can be committed. However, the system can not establish this fact without having some sort of enumeration of group membership. Understating the number of replicas might allow multiple disconnected groups of replicas to commit conflicting updates. Overstating the number of replicas might prevent any update from ever occurring.

4.1 Voting via currency

We eliminate the need for knowledge of group membership by basing our votes on a logical currency. Briefly, any server that creates an object also creates a currency specific to that object. We say that the currency is *held* by the replica, even though it is actually the replica's server that holds the currency. This currency is used by replicas to participate in voting. The creator initially has a total currency of 1.0. New replicas are created by sending requests to servers that have existing replicas. The response to such requests contains both the object's data and some amount of currency. This amount is subtracted from the currency held by the existing replica. Hence, the total amount of currency in the system remains constant at all times during failure-free operation.

Going back to the example discussed in Section 2, assume that each replica has an equal amount of currency. Any three replicas control 75% of the currency, and can conclude that no other set of replicas is concurrently committing updates to the same object. Hence, they can commit updates and application progress can be achieved.

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

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

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

4.2 Committing updates

Updates are either *tentative* or *committed.* Newly created updates are tentative and may be rolled back without ever being committed. Tentative updates may or may not be visible to the application, depending on the type of session guarantees needed by the application. Updates are *committed* when a quorum agrees that they are acceptable. However, updates are not committed through two-phase or similar protocols, because all information propagates slowly through the system via pair-wise synchronization. Instead, replicas become aware of new updates at distinct times, and they become aware that others are aware in similar fashion.

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

Figure 1 (b) shows an example of two competing updates being started at time t_4. Each synchronizes with one other replica at t_5, leading to a stalemate in which each competing update has 50% of the currency. While currency allocation schemes could be rigged to prevent this from occurring in the case of two competing updates, three or more competing updates could still lead to the same problem. The lexicographic tie-breaker will favor $update_3$ over $udpate_4$.

4.3 Currency allocation

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

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

Note also that servers can transparently gift other servers with currency, allowing the system to stabilize in a state with uniform currency distribution regardless of the initial configuration.

4.4 Naming

Object names have two parts, a name that is unique among all objects created by the same server, and the server's name. This scheme is sufficient to guarantee unique system-wide object names, provided that servers also have unique names.

Server names are constructed from the IP address of the machine on which the server process is running and the current time. Where this is ambiguous (multi-processor machines) or not desirable (the user wants friendlier names, such as "Pete's chat" and "Jeff's chat"), applications can provide their own names.

4.5 Proxies

A server that expects to disconnect soon can designate another replica as its *proxy*. A proxy has voting rights on its *primary's* currency until the primary explicitly requests the currency back. This mechanism makes the voting mechanism less unwieldy and less dependent on fortuitous currency allocations. Proxies permanently inherent currency from failed primaries.

4.6 Failure detection

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

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

5. Related Work

The Deno project uses ideas from many areas. The general software architecture is modeled after the Bayou system [5]. Our ideas on lottery scheduling have been influenced by the entire field of quorum consensus, but were most influenced by the use of currency abstractions in proportional-share schedulers [6] and the market-oriented approaches of wide-area databases like Mariposa [7].

6. Conclusions and current status

In summary, there is an ever-increasing range of applications that require coherent access to replicated data in environments with variable connectivity, including special-purpose distributed databases and Internet information sources. Technological advances in network hardware and software, together with the sociological trends that have made the Internet a household presence, have made such applications possible. Through properly structured consistency protocols, we hope to make them efficient.

The designs of Deno's interface and underlying protocols are essentially complete. Work is currently going forward in two directions. We are building a simulator to evaluate information propagation and commit rates in our expected environments. We are also starting to code the initial Deno prototype in a C core. Deno will include a Tcl interpreter to aid in evaluating merge procedures and updates to Tcl objects.

7. References

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

[2] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the 6th Symposium on Principles of Distributed Computing*, August 1987.

[3] J. K. Osterhout, "Tcl: An Embeddable Command Language," in *USENIX Winter Conf.*, 1990.

[4] D. B. Terry, A. J. Derners, K. petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session Guarantees for Weakly Consistent Replicated Data," in *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, 1994.

[5] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing Update Conflicts in a Weakly Connected Replicated Storage System," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[6] C. A. Waldspurger and W. E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management," in *Proc. of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[7] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A Wide-Area Distributed Database System," *VLDB Journal*, vol. 5, pp. 48-63, 1996.