A Self-Service Approach to Scalable Service Deployment

Michael K. Reiter[1] **Asad Samar**[2]

July 2006
CMU-CS-06-101R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report supercedes Technical Report CMU-CS-06-101, January 2006.

Abstract

We describe a system that enables services to scale to large numbers of clients, without the addition of new server resources and without sacrificing service consistency. Our system best supports services whose state can be decomposed into service objects that are typically accessed individually. Scalability is achieved by migrating these objects and outsourcing operation processing to the clients themselves. We present novel algorithms for ensuring consistency of the service and for recovering objects if a client disconnects and leaves the latest versions of objects unreachable. Our system outperforms a centralized service implementation when object state (and thus object migration cost) is small, and when operations are compute-intensive (thus taking advantage of client processing power). In addition, a client executing its own operations enables applications in which the client is unwilling to send its operations elsewhere for processing, due to privacy concerns. We demonstrate these advantages through the evaluation of a prototype network traffic classification service built using our system.

[1] Electrical & Computer Engineering Department and Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; reiter@cmu.edu

[2] Electrical & Computer Engineering Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; asamar@ece.cmu.edu

Keywords: scalable service deployment, consistency protocols, fault tolerance, distributed algorithms

1 Introduction

A typical centralized implementation of a service processes all client operations at a *server*. The resources at the server thus become a significant factor in the service's ability to scale to a large number of clients, particularly when client operations are compute-intensive. In this paper we explore an alternative implementation strategy for services in which the service state can be decomposed into a collection of smaller *objects* such that client operations are typically (though not necessarily always) executed on one object. The approach we consider is to harness *client* resources into the implementation of the service, potentially permitting the service to scale gracefully as the client population grows, without adding resources to the server. While bearing conceptual similarities to *peer-to-peer computing* [1] (to which we compare in detail in Section 2), the approach we consider to harness client resources is, to our knowledge, novel.

In a nutshell, our approach outsources operation processing to the clients themselves; we call this *self-service*. For some types of operations, each involved object is migrated to the client and the client executes its operation locally. This migration occurs within a tree of clients rooted at the server, to which the server adds new clients as they connect. This tree need not be structured in any particular way, but rather can be built as new clients connect so as to accommodate client attributes, e.g., so that only well-connected clients have children, and geographically close clients reside in the same subtree. In addition to harnessing client resources, our system offers the following features:

Strong consistency semantics Operations, both on single objects and on multiple objects, are implemented with strong consistency semantics. Specifically, we present protocols guaranteeing serializability [32, 3] and strict serializability [32], respectively, for durable operations.[1] An ingredient in achieving these semantics is to serialize object migrations, i.e., so that an object is migrated to a client only after the preceding client releases it. This incurs the additional latency of migrations between operations—and so this approach is viable primarily when objects are not too large—but this migration involves moving the object only at most the diameter of the tree. Moreover, single-object reads do not require object migration, and for serializable semantics, the client can perform these reads locally.

Fault recovery If a client *disconnects* while holding an object, either because the client fails, because it can no longer communicate with its parent, or because its parent disconnects, then operations that it recently applied to the object may be lost. However, the connected component of the tree containing the server[2] can efficiently regenerate the last version of the object seen in that component when such a disconnection is detected. Thus, the server never loses control of the service objects, and once an object reaches a portion of the tree that stays connected (except for voluntary departures), all operations it reflects become durable.

Client privacy Because each client applies its own operations locally, it need not reveal these operations to the rest of the system, except to the extent that they are disclosed by the resulting object. This feature is significant for services that build upon contributions of sensitive client data. In particular, a motivating application for this work is the distributed construction of network traffic classifiers. These classifiers are built from network packets and/or flow records, but organizations are often not willing to divulge this data. Organizations may be more willing to integrate their data into the classifier locally, since the resulting classifier typically reveals far less about its input than the raw data does.

Our approach targets collaborative applications, and in doing so it places trust in clients that contribute to the service. (Other, untrusted clients can interact with the service by having a trusted client, or the server itself, perform operations on the untrusted client's behalf.) If one of these trusted clients turns out to be

[1] As such, our operations have consistency semantics similar to database transactions, and our operations can act on multiple objects, just like transactions can. We nevertheless avoid the term "transactions" due to other properties that it connotes in some settings.

[2] We do not address the failure of the server; we presume it is rendered fault-tolerant using standard techniques (e.g., [5]).

malicious, then when an object is migrated to that client, the client could corrupt it. As such, the server should authenticate each client prior to admitting it; with the emergence of *trusted platforms* [33], this authentication might verify that the client is running valid client software. Relaxing trust in clients is an area of ongoing research.

We evaluate our approach using both microbenchmarks and the aforementioned application involving the distributed construction of network traffic classifiers for diagnosis and intrusion detection. Our experiments show that self-service is compelling for this application, e.g., yielding up to an order of magnitude improvement in update latency and throughput over a centralized implementation in an experiment involving 70 nodes. In addition, we reiterate that our approach better protects client privacy for this application. This application is representative of a broader class that is well-suited to self-service, including numerous applications in distributed data mining [23] and collaborative filtering for web content or email, such as spam filtering (e.g., [47, 14]). Here we use one such application to analyze the performance benefits that self-service offers, though the application itself will be detailed elsewhere.

2 Related work

Scalability, fault recovery and consistency have been studied for decades. Many systems have excelled at two of these, but it appears to be harder to achieve them all. Below we discuss pairs of these properties and prior work that focuses on that pair. To our knowledge, self-service is a different point in the design space of tradeoffs among these goals than has been explored previously.

Consistency and scalability Our design of self-service was influenced most directly by work in this space, notably token-based distributed mutual exclusion protocols [36, 29, 9]. These protocols allow nodes arranged in a tree to locate and retrieve shared objects and perform operations atomically. Another example in this space [24] is a distributed hash table design that supports atomic operations. These approaches achieve scalability and consistency, but do not address failures. Self-service can be viewed as extending this category of research to recover from failures (though operations by clients that disconnect may not be durable, see Section 3). Our work also enables consistent multi-object operations and optimizations for single-object reads that are not possible in the works from which we most closely build [36, 29, 9].

Consistency and fault recovery Prior systems that have focused on consistency and fault recovery typically take the form of cluster-based solutions in which object updates are processed at a cluster of machines (e.g., [4]). Objects are either hosted on a single cluster or different clusters. Approaches that employ a single cluster for updates—e.g., cluster-based internet services [40, 12] and dynamic web content distribution networks [31, 30]—yield simple consistency protocols. But their "incremental scalability" [12] remains a barrier, i.e., to support more clients, resources must be added to the cluster. Systems that host objects on different clusters—such as peer-to-peer systems with explicit support for consistent updates (e.g., [37, 34, 44])—scale better, but typically do not support multi-object updates. Self-service is a different choice in this design space: it overcomes "incremental scalability" by utilizing clients' resources for update processing while implementing consistent single- and multi-object update operations. However, it offers weaker fault recovery than a well-designed cluster solution, which can fully mask failures of cluster nodes.

Fault recovery and scalability Mechanisms that achieve better than incremental scalability typically fall in the category of peer-to-peer systems, e.g., [42, 35, 39, 46, 25]. Moreover, these systems necessarily provide recovery from faults of unreliable peers. However, most applications of these peer-to-peer substrates either support only read operations (e.g., [8, 10]) or support updates but with weak forms of consistency: Examples of the latter class include peer-to-peer file systems that achieve only *eventual consistency* (e.g., [38, 41]) or guarantee consistency for write operations (that blindly overwrite the previous state) but not for more general update operations (e.g., [28]). In [20], replica versioning provides probabilistic consistency guarantees. Even the database systems built over peer-to-peer technology of which we are aware (e.g., [17]) have sacrificed atomicity. Self-service is an alternative that better supports strong consistency and compute-intensive updates.

3 Terminology and goals

Our system implements a service with a designated *server* and an unbounded number of *clients*. The *processes* in the system include the clients and the server. To interact with the service, a client *joins* the service; in doing so, it is positioned within a tree rooted at the server. A client can also voluntarily *leave* the service.

If a process loses contact with one of its children, e.g., due to the failure of the child or of the communication link to the child, then the child and all other clients in the subtree rooted at the child are said to *disconnect*. To simplify discussion, we treat the disconnection of a client as permanent, or more specifically, a disconnected client may re-join the service but with a re-initialized state. In an execution, a client that joins but does not disconnect (though it might leave voluntarily) is called *connected*.

The service enables clients to invoke *operations* on *objects*. These operations may be *reads* or *updates*. Updates compute *object instances* from other object instances. An object instance o is an immutable structure with several fields, including an *identifier* field $o.\mathsf{id}$ and a *version* field $o.\mathsf{version}$. We refer to object instances with the same identifier as versions of the same object. Any operation that produces an object instance o as output takes as input the previous version, i.e., an instance $\acute{o}$ such that $o'.\mathsf{id} = o.\mathsf{id}$ and $o'.\mathsf{version} + 1 = o.\mathsf{version}$.

Our system applies operations consistently: for any system execution, there is a set of operations Durable that includes all operations performed by connected processes (and possibly some by clients that disconnect), such that the connected processes perceive the operations in Durable (and no others) to be executed sequentially. More precisely, we present two variations of our algorithm. One enforces *serializability* [32, 3]: all connected processes perceive the operations in Durable to be executed in the same sequential order. The other enforces an even stronger property, *strict serializability* [32]: the same sequential order perceived by processes preserves the real-time order between operations. Recall that strict serializability implies linearizability [16] for applications that employ only single-object operations.

4 Object management

We begin by describing a high-level abstraction in Section 4.1 that enables our solution, and then discuss the implementation of that abstraction in Section 4.2. Section 5 describes how this implementation enables update and multi-object operations, and the optimizations for single-object read operations.

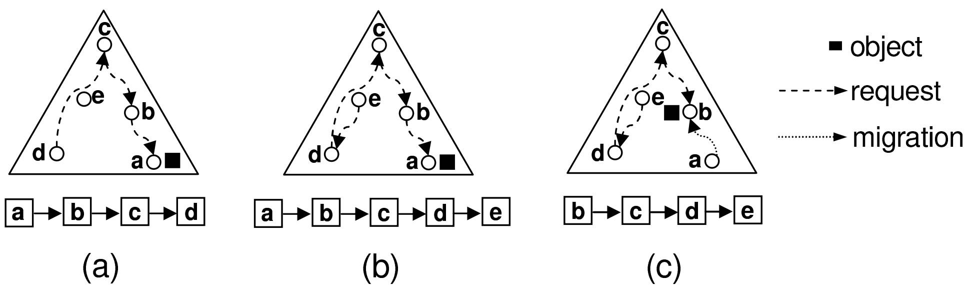


Figure 1: (a) distQ consists of processes a, b, c and d. (b) e adds itself to the end of distQ by sending a retrieve request to d. (c) When a completes its operation, it migrates the object to b and drops off distQ.

4.1 distQ abstraction

For each object, processes who wish to perform operations on that object arrange themselves in a logical distributed FIFO queue denoted distQ, and take turns according to their positions in distQ to perform those operations. The process at the front of distQ is denoted as the *head* and the one at the end of distQ is denoted as the *tail*. Initially, distQ consists of only one process—the server. When an operation is invoked at a process p, p sends a *retrieve request* to the current tail of distQ. This request results in adding p to the end of distQ, making it the new tail; see Figure 1-(b). When the head of distQ completes its operation, it drops off the queue and *migrates* the object to the next process in distQ, which becomes the new head; see Figure 1-(c). This distributed queue ensures that the object is accessed sequentially.

A process performs an operation involving multiple objects by retrieving each involved object via its distQ. Once the process holds these objects, it performs its operation and then releases each such object to be migrated to the process next in that object's distQ.

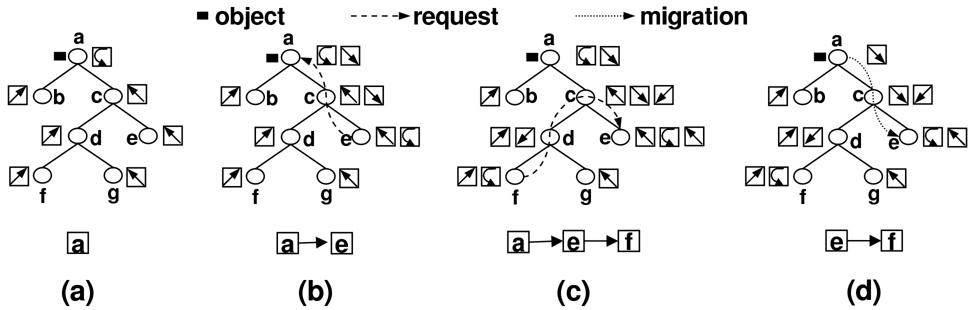


Figure 2: Squares at each process represent its localQ; left-most square is the head and right-most is the tail. The arrow in each square denotes the neighbor to which it points. Initially a has the object. e requests from a and f requests from e. Finally a migrates the object to e.

4.2 distQ implementation

distQ for the object with identifier id (henceforth, distQ$[id]$) is implemented using a local FIFO queue p.localQ$[id]$ at every process p. Elements of p.localQ$[id]$ are neighbors of p. Intuitively, p.localQ$[id]$ is maintained so that the head and tail of p.localQ$[id]$ point to p's neighbors that are in the direction of the head and tail of distQ$[id]$, respectively. Initially, the server has the object and it is the only element in distQ$[id]$. Thus, p.localQ$[id]$ at each client p is initialized with a single entry, p's parent, the parent being in the direction of the server (Figure 2-(a)).

When a process p receives a retrieve request for the object with identifier id from its neighbor q, it forwards the request to the tail of p.localQ$[id]$ and adds q to the end of p.localQ$[id]$ as the new tail. Thus, the tail of p.localQ$[id]$ now points in the direction of the new tail of distQ$[id]$, which must be in the direction of q since the latest retrieve request came from q; see Figures 2-(b) and 2-(c). When a process p receives a migrate message containing the object, it removes the current head of p.localQ$[id]$ and forwards the object to the new head of p.localQ$[id]$. This ensures that the head of p.localQ$[id]$ points in the direction of the new head of distQ$[id]$, see Figure 2-(d).

Pseudocode for this algorithm is shown in Figure 3. We use the following notation throughout for accessing localQ: localQ.head and localQ.tail are the head and the tail. localQ.elmt$[i]$ is the i^{th} element (localQ.elmt$[1]$ = localQ.head). localQ.size is the current number of elements. localQ.removeFromHead() removes the current head. localQ.addToTail(e) adds the element e to the tail. localQ.hasElements() returns true if localQ is not empty. Initialization of a process upon joining the tree is not shown in the pseudocode of Figure 3; we describe initialization here. When a process p joins the tree, it is initialized with a parent $p.\mathsf{parent}$ ($\perp$ if p is the server). Each process also maintains a set $p.\mathsf{children}$ that is initially empty but that grows as other clients are added to the tree. For each object identifier id, p initializes a local queue $p.\mathsf{localQ}[id]$ by enqueuing p if p is the server and $p.\mathsf{parent}$ otherwise. In addition, for each object identifier id, the server p initializes its copy of the object, $p.\mathsf{objs}[id]$, to a default initial state.

Each process consists of several threads running concurrently. The global state at a process p that is visible to all threads is denoted using the "$p.$" prefix, e.g., $p.\mathsf{parent}$. Variable names without the "$p.$" prefix represent state local to its thread. In order to synchronize these threads, the pseudocode of process p employs a semaphore[3] $p.\mathsf{sem}[id]$ per object identifier id, used to prevent the migration of object $p.\mathsf{objs}[id]$ to another process before p is done using it. $p.\mathsf{sem}[id]$ is initialized to one at the server and zero elsewhere. In our pseudocode, we assume that any thread executes in isolation until it completes or blocks on a semaphore.

```
doRetrieveRequest($from, id, prog$)                      /* Invoked locally on request by $from$ ($from$ could be $p$) */
1.  $\langle q, prog' \rangle \leftarrow p$.localQ[$id$].tail          /* $q$ made the last request for this object */
2.  $p$.localQ[$id$].addToTail($\langle from, prog \rangle$)         /* The next request will be forwarded to $from$ */
3.  if $q = p$                                          /* If I last requested this object ... */
4.      $P(p$.sem[$id$]$)$                                  /* ... then wait till I am done using it */
5.      doMigrate($id$)                                 /* ... and then initiate migration to the requesting process */
6.  else                                                /* If I am not the last process to request this object ... */
7.      send (retrieveRequest : $p, id$) to $q$              /* ... then forward the request to who last requested it (to reach tail of distQ[$id$]) */

doMigrate($id$)                                         /* Invoked locally when migration is initiated by $p$, destined for $p$ or goes through $p$ */
8.  $p$.localQ[$id$].removeFromHead()                    /* This object's owner does not lie toward the current head any more */
9.  $\langle q, prog \rangle \leftarrow p$.localQ[$id$].head           /* $q$ requested this object, object will be migrated to $q$ and so $q$ is the new head */
10. if $q = p$                                          /* If I requested this object ... */
11.     $prog$                                          /* ... then execute the program that was registered for my request */
12. else if $q = p$.parent                              /* If this object is for someone else who is toward my parent ... */
13.     $IDs \leftarrow \{id' : id \overset{p.\mathsf{Deps}}{\Longrightarrow} id'\}$              /* ... then find which other objects this object depends on */
14.     $Objs \leftarrow \{p$.objs[$id'$] $: id' \in IDs\}$             /* ... collect all these objects */
15.     $DepSet \leftarrow p$.Deps $\cap (IDs \times IDs)$           /* ... and the dependency relations between them */
16.     send (migrate : $p$.objs[$id$], $Objs, DepSet$) to $q$   /* ... migrate the object, copy objects it depends on and the dependencies to parent */
17.     $p$.Deps $\leftarrow p$.Deps $\setminus DepSet$              /* ... finally remove the dependencies so same values not copied to the parent again */
18. else                                                /* If this object is for someone else who is toward a child ... */
19.     send (migrate : $p$.objs[$id$]$, \emptyset, \emptyset$) to $q$         /* ... then no need to copy any other objects, just migrate this object */

Upon receiving (retrieveRequest : $from, id$)           /* Request received for object with identifier $id$ from another process $from$ */
20. doRetrieveRequest($from, id, \perp$)                 /* Invoke doRetrieveRequest on $from$'s behalf with an empty program */

Upon receiving (migrate : $o, Objs, DepSet$)            /* Object $o$ being migrated that depends on copied objects $Objs$ with relation $DepSet$ */
21. $p$.objs[$o$.id] $\leftarrow o$                             /* Save the migrated object $o$ */
22. foreach $o' \in Objs$                               /* For each one of the copied objects ... */
23.     $p$.objs[$o'$.id] $\leftarrow o'$                        /* ... save the copied object */
24. $p$.Deps $\leftarrow p$.Deps $\cup DepSet$                      /* Update the local dependency relation with the one in the message */
25. doMigrate($o$.id)                                   /* Invoke doMigrate for the object with identifier $id$ */
```

Figure 3: Object management pseudocode for process p

4.3 Retrieving one object

The routing of retrieve requests for objects is handled by the doRetrieveRequest function shown in Figure 3. When p executes doRetrieveRequest($from, id, prog$), it adds $\langle from, prog \rangle$ to the tail of $p.\mathsf{localQ}[id]$ (line 2), since $from$ denotes the process from which p received the request for id. ($prog$ has been elided from

[3] To remind the reader, a semaphore s represents a non-negative integer with two atomic operations: $V(s)$ increments s by one; $P(s)$ blocks the calling thread while $s = 0$ and then decrements s by one.

discussion of localQ so far; it will be discussed in Section 5.) p then checks if the previous tail (lines 1, 3) was itself. If so, it awaits the completion of its previous operation (line 4) before it migrates the object to from by invoking $\mathsf{doMigrate}(id)$ (line 5, discussed below). If the previous tail was another process q, then p sends $(\mathsf{retrieveRequest} : p, id)$ to q (line 7); when received at q, q will perform $\mathsf{doRetrieveRequest}(p, id, \bot)$ similarly (line 20). In this way, a retrieve request is routed to the tail of $\mathsf{distQ}[id]$, where it is blocked until the object is migrated to the requesting process. Note that p invokes doRetrieveRequest not only when it receives a retrieve request from another process (line 20), but also to retrieve the object for itself.

Migrating an object with identifier id is handled by the doMigrate function. Since the head of $p.\mathsf{localQ}[id]$ should point toward the current location of the object, p must remove its now-stale head (line 8), and identify the new head q to which it should migrate the object to reach its future destination (line 9). If that future destination is p itself, then p runs the program prog (line 11) that was stored when p requested the object by invoking $\mathsf{doRetrieveRequest}(p, id, \mathit{prog})$; again, we defer discussion of prog to Section 5. Otherwise, p migrates the object toward that destination (line 16 or 19). If p is migrating the object to a child (line 19), then it need not send any further information. If p is migrating the object to its parent, however, then it must send additional information (lines 13–16) that is detailed in Section 4.4.

4.4 Object dependencies

There is a natural dependency relation $\Rightarrow$ (pronounced "depends on") between object instances. First, define $o \overset{op}{\Rightarrow} o'$ if in an operation op, either op produced o and took o' as input, or o and o' were both produced by op. Then, let $\Rightarrow \; = \; \bigcup_{op} \overset{op}{\Rightarrow}$. Intuitively, a process p should pass an object instance o to $p.\mathsf{parent}$ only if all object instances on which o depends are already recorded at $p.\mathsf{parent}$. Otherwise, $p.\mathsf{parent}$ might receive only o before p disconnects, in which case atomicity of the operation that produced o cannot be guaranteed. Thus, to pass o to $p.\mathsf{parent}$, p must *copy* along all object instances on which o depends. Note that copying has different semantics than migrating, and in particular copying an object instance does not transfer "ownership" of that object.

Because each process holds only the latest version it has received for each object identifier, however, it may not be possible for p to copy an object instance o' upward when migrating o even if $o \Rightarrow o'$: o' may have been "overwritten" at p, i.e., $p.\mathsf{objs}[o'.\mathsf{id}].\mathsf{version} > o'.\mathsf{version}$. In this case, it would suffice to copy $p.\mathsf{objs}[o'.\mathsf{id}]$ in lieu of o', provided that each o'' such that $p.\mathsf{objs}[o'.\mathsf{id}] \Rightarrow o''$ were also copied—but of course, o'' might have been "overwritten" at p, as well. As such, in a refinement of the initial algorithm above, when p migrates o to its parent, it computes an identifier set IDs recursively according to the following rules until no more indices can be added to IDs: (i) initialize IDs to $\{o.\mathsf{id}\}$; (ii) if $id \in \mathit{IDs}$ and $p.\mathsf{objs}[id] \Rightarrow o'$, then add $o'.\mathsf{id}$ to IDs. p then copies $\{p.\mathsf{objs}[id]\}_{id \in \mathit{IDs}}$ to its parent.

It is not necessary for each process p to track $\Rightarrow$ between all object instances in order to compute the appropriate identifier set IDs. Rather, each process maintains a binary relation $p.\mathsf{Deps}$ between object identifiers, initialized to $\emptyset$. If p performs an update operation op such that an output $p.\mathsf{objs}[id] \overset{op}{\Rightarrow} p.\mathsf{objs}[id']$, then p adds (id, id') to $p.\mathsf{Deps}$. In order to perform $\mathsf{doMigrate}(id)$ to $p.\mathsf{parent}$, p determines the identifier set IDs as those indices reachable from id by following edges (relations) in $p.\mathsf{Deps}$—reachability is denoted $\overset{p.\mathsf{Deps}}{\Longrightarrow}$ in line 13 of Figure 3—and copies both $\mathit{Objs} = \{p.\mathsf{objs}[id']\}_{id' \in \mathit{IDs}}$ (line 14) and $\mathit{DepSet} = p.\mathsf{Deps} \cap (\mathit{IDs} \times \mathit{IDs})$ (line 15) along with the migrating object (line 16). Finally, p updates $p.\mathsf{Deps} \leftarrow p.\mathsf{Deps} \setminus \mathit{DepSet}$ (line 17), i.e., to remove these dependencies for future migrations upward.

If p receives a migration from a child with copied objects Objs and copied dependencies DepSet, then p saves Objs in $p.\mathsf{objs}$ (lines 22–23) and sets $p.\mathsf{Deps} \leftarrow p.\mathsf{Deps} \cup \mathit{DepSet}$ (line 24).

5 Operation implementation

In order to achieve our desired consistency semantics, for each object we enforce sequential execution of all update and multi-object operations (Section 5.1) that involve that object. Fortunately, for many realistic workloads, these types of operations are also the least frequent, and so the cost of executing them sequentially need not be prohibitive. In addition, this sequential execution of update and multi-object operations enables significant optimizations for single-object reads (Section 5.2) that dominate many workloads.

5.1 Update and multi-object operations

Invoking operations Let $id_1, \ldots, id_k$ denote distinct identifiers of the objects involved (read or updated) in an update or multi-object operation op. To perform op, process p recursively constructs—but does not yet execute—a sequence $prog_0, prog_1, \ldots, prog_k$ of programs as follows, where "$\|$" delimits a program:

```
$prog_0 \leftarrow \| \; op;$
        $NewDeps \leftarrow \{ \; (id, id') \;\; :$
                    $p.\mathsf{objs}[id] \overset{op}{\Rightarrow} p.\mathsf{objs}[id'] \};$
        $p.\mathsf{Deps} \leftarrow p.\mathsf{Deps} \cup NewDeps;$
        $V(p.\mathsf{sem}[id_1]); \ldots; V(p.\mathsf{sem}[id_k]) \; \|$
$prog_i \leftarrow \| \; \mathsf{doRetrieveRequest}(p, id_i, prog_{i-1}) \; \|$
```

Process p then executes $prog_k$. Note that $prog_k$ requests id_k and, once that is retrieved, $prog_{k-1}$ is executed (at line 11 of Figure 3). This, in turn, requests id_{k-1}, and so forth. Once id_1 has been retrieved, $prog_0$ is executed. This performs op and then updates the dependency relation $p.\mathsf{Deps}$ (see Section 4.4) with the new dependencies introduced by op. Finally, $prog_0$ executes a V operation on the semaphore for each object, permitting it to migrate. Viewing the semaphores $p.\mathsf{sem}[id_1]$, . . ., $p.\mathsf{sem}[id_k]$ as locks, $prog_k$ can be viewed as implementing strict two-phase locking [3]. So, to prevent deadlock, $id_1, \ldots, id_k$ must be arranged (i.e., the "locks" must be obtained) in a canonical order.

Update durability A process that performs an update operation can force the operation to be durable, by copying each resulting object instance o (and those on which it depends, see Section 4.4) to the server, allowing each process p on the path to save o if $p.\mathsf{objs}[o.\mathsf{id}].\mathsf{version} < o.\mathsf{version}$. That said, doing so per update would impose a significant load on the system, and so our goals (Section 3) do not require this. Rather, our goals require only that a process forces its updates to be durable when it leaves the tree (Section 6.2), so that operations by a process that remains connected until it leaves are durable.

5.2 Single-object read operations

We present two different protocols implementing a read operation that involves a single object. Depending on which of these two protocols is employed, our system guarantees either serializability or strict serializability when combined with the implementation of update and multi-object operations from Section 5.1.

5.2.1 Serializability

Due to the serial execution of update and multi-object operations (Section 5.1), single-object reads so as to achieve serializability [3] can be implemented with local reads. That is, to perform a read operation involving a single object with identifier id, process p simply returns $p.\mathsf{objs}[id]$. See Section 7 for the correctness proof.

5.2.2 Strict Serializability

Recall that all update and multi-object read operations involving the same object are performed serially (Section 5.1). Therefore, in order to guarantee strict serializability, we only need to ensure that a single-object read operation op on an object with identifier id invoked by a process p, reads the latest version of this object produced before op is invoked. One way to achieve this would be to serialize op along with the update and multi-object operations in $\mathsf{distQ}[id]$. However, this would require op to wait for the completion of the concurrent update and multi-object operations (those performed by processes preceding p in $\mathsf{distQ}[id]$).

A more efficient solution is to request the latest version from the process at the head of $\mathsf{distQ}[id]$—the process that is the current "owner" of the object with identifier id. Our algorithms already provide a way to route to the head of $\mathsf{distQ}[id]$, using $\mathsf{localQ}[id]\mathsf{.head}$ at each process. Thus a read request for id follows $p.\mathsf{localQ}[id]\mathsf{.head}$ at each process p until it reaches a process p' such that either $p'.\mathsf{localQ}[id]\mathsf{.head} = p'$ (i.e., p' holds the latest object version), or $p'.\mathsf{localQ}[id]\mathsf{.head} = p''$ is the process that forwarded this read request to p'. In the latter case, p' forwarded $p'.\mathsf{objs}[id]$ to p'' in a migration concurrently with p'' forwarding this read request to p' (since $p''.\mathsf{localQ}[id]\mathsf{.head} = p'$ when p'' did so), and so it is safe for p' to serve the read request with $p'.\mathsf{objs}[id]$.

The initiator p of the read request could encode its identity within the request, allowing the responder p' to directly send a copy of the object to p outside the tree. However, to facilitate reconstituting the object in case it is lost due to a disconnection (see Section 6.1 for details of this mechanism), we require that the object be passed through the tree to the highest process in the path from p' to p, i.e., the lowest common ancestor p'' of the initiator and responder of the read request. After receiving the object in response to the read request, p'' directly sends the object to p (the initiator) outside the tree, see Figure 4. Note that since the requested object is copied upwards in the tree from p' to p'' (unless $p' = p''$), any objects that the requested object depends upon, must also be copied along using the techniques described in Section 4.4. A proof that this protocol yields strict serializability is presented in Section 7.

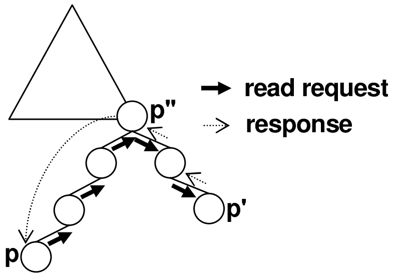


Figure 4: p initiates a single object read request that reaches p'. p' sends the response through the tree to the highest process p'' in the path. p'' then copies the requested object directly to p outside the tree.

6 Tree management

To this point we have deferred discussion of joins, leaves and disconnections. Joins are straightforward: a client contacts the root of the tree—the server. Based on some tree construction policy, the server either adds the client as its child and notifies it, or forwards the join request to an existing child. Each client that receives a join request from its parent follows the same procedure. We invest the balance of this section to detail how to adapt our algorithm to address disconnections (Section 6.1) and processes leaving voluntarily (Section 6.2).

6.1 Disconnections

Recall that when a process loses contact with a child, all clients in the subtree rooted at that child are said to *disconnect*. The child (or, if the child failed, each uppermost surviving client in the subtree), informs its subtree of the disconnection, to enable clients to reconnect (after reinitializing) if desired. Of concern here, however, is that some of these disconnected clients may have earlier issued retrieve requests for objects, and for each such object with identifier id, the disconnected client may appear in $\mathsf{distQ}[id]$. In this case, steps must be taken to ensure that the connected processes preceded by a disconnected process in $\mathsf{distQ}[id]$ continue to make progress. To this end, all occurrences of the disconnected clients in $\mathsf{distQ}[id]$ are replaced with the parent p of the uppermost disconnected client q, see Figure 5.

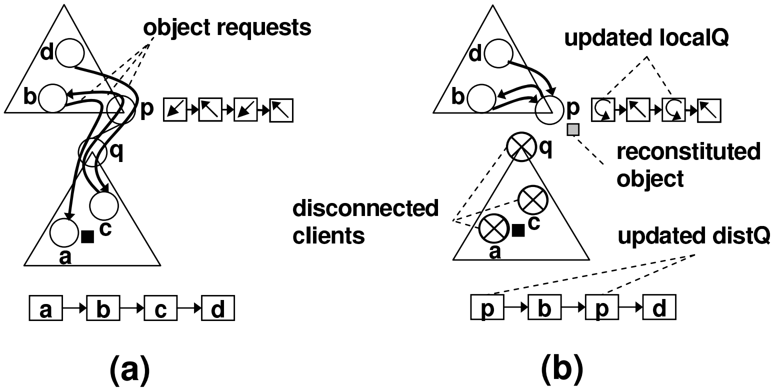


Figure 5: q loses contact with parent p and its subtree disconnects. p replaces disconnected clients in distQ and reconstitutes the object so b and d can make progress.

Choosing p to replace the disconnected clients is motivated by several factors: First, p is in the best position to detect the disconnection of the subtree rooted at its child q. Second, as we will see below, in our algorithm p need only take local actions to replace the disconnected clients; as such, this is a very efficient solution. Third, in case the head of $\mathsf{distQ}[id]$ is one of the disconnected clients, the object with identifier id must be in the disconnected component. This object needs to be reconstituted using the local copy at one of the processes still connected, while minimizing the number of updates by now-disconnected clients that are lost. p is the best candidate among the still-connected processes: p is the last to have saved the object as it was either migrated toward q (migrations are performed through the tree), or copied upward from q in response to a strictly-serializable single-object read request (the response travels upward along the tree, see Section 5.2). Note that in case of multiple simultaneous disconnections, only one connected process—that which has the object in its disconnected child's subtree—will reconstitute the object from its local copy, becoming the new head of $\mathsf{distQ}[id]$.

In order to replace all occurrences of the disconnected clients from $\mathsf{distQ}[id]$ by itself, p replaces all instances of q in $p.\mathsf{localQ}[id]$ with itself. As such, any retrieve request that was initiated at a connected process and blocked at a disconnected client is now blocked at p, see Figure 5-(b). The pseudocode that p executes when its child q disconnects is the $\mathsf{childDisconnected}(q)$ routine in Figure 6. Specifically, p replaces all instances of q in $p.\mathsf{localQ}[id]$ with itself and a "no-op" operation to execute once p obtains the object (line 8–9 and 12–13). As such, any retrieve request that was initiated at a connected process and blocked at a disconnected client is now blocked at p, see Figure 5-(b). For each of these requests that are now blocked at p, p creates and run-enables a new thread (lines 10–11 of Figure 6) to initiate the migration of $p.\mathsf{objs}[id]$ to the neighbor following (this instance of) p in $p.\mathsf{localQ}[id]$, once p has the object. If the disconnected child was at the head of $p.\mathsf{localQ}[id]$, then p reconstitutes the object simply by making its local copy (which is the latest at any connected process) available (lines 5–6). p also responds to any strictly-serializable single-object read requests initiated by a still-connected process and forwarded by p to q, and for which p has not observed a response (not shown in Figure 6).

```
childDisconnected($q$)                                          /* Invoked at $p$ when $p$'s child $q$ disconnects */
$p$.children $\leftarrow$ $p$.children $\setminus \{q\}$        /* Remove $q$ as a child */
foreach $id$                                                    /* For each object...*/
    $q' \leftarrow p$.localQ$[id]$.head                         /* ...save the current head of localQ */
    Qreplace$(id, q)$                                           /* ...run Qreplace for this object */
    if $q' = q$                                                 /* If the disconnected child was the head before Qreplace... */
        $V(p.\mathsf{sem}[id])$                                 /* ...then make the object available, as I am the new head */

Qreplace$(id, q)$                                               /* Invoked locally by $p$ */
foreach $i = 1, \ldots, p$.localQ$[id]$.size $- 1$              /* For each element of localQ, except the last */
    if $p$.localQ$[id]$.elmt$[i] = \langle q, * \rangle$        /* If the element points to $q$ ("$*$" is wild-card)... */
        $p$.localQ$[id]$.elmt$[i] \leftarrow \langle p, \|V(p.\mathsf{sem}[id])\| \rangle$   /* ...change it to point to myself */
        $t \leftarrow$ new thread$(\|P(p.\mathsf{sem}[id])$; doMigrate$(id)\|)$   /* ...create a thread that waits for object and then migrates it */
        $t$.enable()                                            /* ...run-enable the thread */
if $p$.localQ$[id]$.tail $= \langle q, * \rangle$               /* If the last element is the disconnected child... */
    $p$.localQ$[id]$.tail $\leftarrow \langle p, \|V(p.\mathsf{sem}[id])\| \rangle$   /* ...then replace it by myself; no need to start thread here */
```

Figure 6: Disconnection-handling at process p

6.2 Leaves

In order to voluntarily leave the tree, a client p must ensure that any objects in the subtree rooted at p are still accessible to connected nodes, once p leaves. Furthermore, outstanding retrieve requests forwarded through p must not block as a result of p leaving the tree.

If p is a leaf node, then it serves any retrieve requests blocked on it, migrates any objects held at p to its parent (Section 4.2), forces its updates to be durable (Section 5.1), and departs. If p is an internal node then it forces its updates to be durable, and then chooses one of its children q to *promote*. The promotion updates q's state according to the state at p, and updates all other neighbors of p to recognize q in p's place.

Before promoting q, p notifies its neighbors (including q) to temporarily hold future messages destined for p, until they are notified by q that q's promotion is complete (at which point they can forward those messages to q and replace all instances of p in their data structures with q). p then sends to q a promote message containing p.parent, p.children, p.localQ[ ], p.objs[ ] (or, rather, only those object versions that q does not yet have) and p.Deps. When q receives these, it updates its parent, children, objects and object dependencies according to p's state.

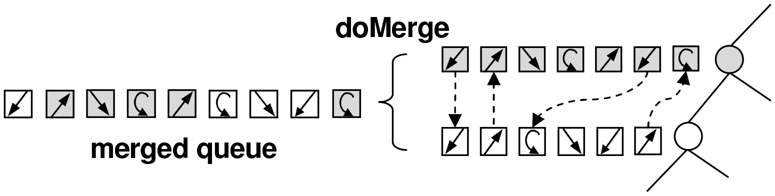


Figure 7: Queue merge. Shaded and unshaded elements are parent's and child's neighbors (and self pointers), respectively. Dashed arrows are from a skipped element to the element in the other queue added next. Elements between curved arrows are added to $mergedQ$ in order.

The interesting part of q's promotion is how it *merges* q.localQ$[id]$ with p.localQ$[id]$ for each id, so that any outstanding retrieve requests for id that were blocked at p or q, or simply forwarded to other processes by

p or q or both, will make progress as usual when q's promotion is complete, see Figure 7. Figure 8 presents the pseudocode used by a promoted child q to merge $q.\mathsf{localQ}[id]$ with its parent p's $p.\mathsf{localQ}[id]$ for each identifier id, as the parent voluntarily leaves the service. In order to merge $p.\mathsf{localQ}[id]$ and $q.\mathsf{localQ}[id]$, q begins with $q.\mathsf{localQ}[id]$ if its head points to p and $p.\mathsf{localQ}[id]$ otherwise. q adds elements from the chosen queue, say $p.\mathsf{localQ}[id]$, to a newly created $mergedQ$ until an instance of q is reached (line 19 of Figure 8), say at the i^{th} index, i.e., $p.\mathsf{localQ}[id].\mathsf{elmt}[i] = q$. The merge algorithm then skips this i^{th} element and begins to add elements from $q.\mathsf{localQ}[id]$ until an instance of p is found. This element is skipped and the algorithm switches back to $p.\mathsf{localQ}[id]$ adding elements starting from the $(i+1)^{st}$ index. This algorithm continues until both queues have been completely (except for the skipped elements) added to $mergedQ$. After merging the two queues, q replaces all occurrences of p in $mergedQ$ by itself, using $\mathsf{Qreplace}(id, p)$ defined in Figure 6.

```
Upon receiving (promote :$gParent, siblings, parentQ[\,],$
                   $parentObjs[\,], parentDeps$)     /* Message received by $q$ from $q.\mathsf{parent}$ that is voluntarily leaving */
1.  foreach $id$                                     /* For each object...*/
2.      if $parentObjs[id].\mathsf{version} > q.\mathsf{objs}[id].\mathsf{version}$   /* If the parent's version is newer than my version... */
3.          $q.\mathsf{objs}[id] \leftarrow parentObjs[id]$    /* ...then replace my instance with parent's instance */
4.      $mergedQ[id] \leftarrow \emptyset$               /* Start with a fresh $mergedQ$ */
5.      if $q.\mathsf{localQ}[id].\mathsf{head} = \langle q.\mathsf{parent}, * \rangle$   /* If the head of my localQ points to my parent... */
6.          doMerge($q.\mathsf{localQ}[id], parentQ[id],$   /* ...then start the merge operation with my localQ */
                  $q, q.\mathsf{parent}, mergedQ[id]$)
7.      else                                        /* If the head of my localQ does not point to my parent... */
8.          doMerge($parentQ[id], q.\mathsf{localQ}[id],$   /* ...then start the merge operation with parent's localQ */
                  $q.\mathsf{parent}, q, mergedQ[id]$)
9.      $q.\mathsf{localQ}[id] \leftarrow mergedQ[id]$     /* Set localQ to the newly created $mergedQ$ */
10.     Qreplace($id, q.\mathsf{parent}$)              /* Run Qreplace on the new localQ to replace parent with myself */
11. $q.\mathsf{parent} \leftarrow gParent$               /* The old grand-parent is now my parent */
12. $q.\mathsf{children} \leftarrow (q.\mathsf{children} \cup siblings) \setminus \{q\}$   /* Old siblings are now my children */
13. $q.\mathsf{Deps} \leftarrow q.\mathsf{Deps} \cup parentDeps$   /* Add parent's object dependencies */

doMerge(localQ, localQ$'$, $p, p', mergedQ$)          /* Invoked locally to merge my localQ with parent's localQ */
14. while localQ.hasElements()                      /* If there are more elements in the first queue... */
15.     $\langle r, prog \rangle \leftarrow$ localQ.removeFromHead()   /* ...then remove its head */
16.     if $r \neq p'$                                 /* If the head does not point to the other process... */
17.         $mergedQ$.addToTail($\langle r, prog \rangle$)   /* ...then add this element to the tail of $mergedQ$ */
18.     else                                        /* If the head points to the other process... */
19.         doMerge(localQ$'$, localQ, $p', p, mergedQ$)   /* ...then skip this element and recurse with the other queue */
```

Figure 8: Pseudocode run at q for its promotion

At this point, any outstanding retrieve requests that were initiated by p (represented by instances of p in $p.\mathsf{localQ}[id]$) now appear as initiated by q since all instances of p from $p.\mathsf{localQ}[id]$ are copied to $mergedQ$ and then replaced by q. Retrieve requests forwarded through p but not q now appear as forwarded through q, as all elements in $p.\mathsf{localQ}[id]$ are added to $mergedQ$, except instances of q. Retrieve requests forwarded through q and not p appear as before since $q.\mathsf{localQ}[id]$ elements are all added to $mergedQ$, except instances of p. Finally, requests forwarded through both p and q now appear as forwarded through only q, due to skipping elements in $p.\mathsf{localQ}[id]$ that point to q and vice-versa.

7 Correctness

Definition 1 (reads-from, $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{rf}}_*$)**.** *An operation* op_i reads from op_j, *denoted* $op_j \xrightarrow{\text{rf}} op_i$, *if* op_i *inputs an object instance produced by* op_j*.* $\xrightarrow{\text{rf}}_*$ *denotes the transitive closure of* $\xrightarrow{\text{rf}}$*.*

Lemma 1. *Let op_i and op_j denote distinct operations that output object instances o_i and o_j, respectively, where $o_i.\mathsf{id} = o_j.\mathsf{id}$ and $o_i.\mathsf{version} = o_j.\mathsf{version}$. Then there are no operations op_k and op_l (distinct or not) performed by connected processes such that $op_i \xrightarrow{\text{rf}}_* op_k$ and $op_j \xrightarrow{\text{rf}}_* op_l$.*

Proof. Among the connected processes, the $\mathsf{localQ.tail}$ pointers implement the Arrow protocol by Demmer and Herlihy [9] (augmented to account for disconnections as described in Section 6.1). This protocol ensures that per object identifier, migrations among connected processes occur serially. We do not recount the proof of this fact here; interested readers are referred to, e.g., [9, 15, 21]. This fact implies that there is a unique object instance bearing a particular identifier and version number that is retrieved by connected processes.

As a result, the existence of two object instances o_i and o_j with the same object identifier and version number implies that at least one of op_i and op_j, say op_i, was performed by a client that disconnects. Moreover, the process that performs op_i must disconnect prior to migrating o_i (or having it copied due to the migration, Section 5.1, or the strictly serializable read, Section 5.2.2, of an object instance that depends on o_i) out of the subtree that disconnects. Otherwise, the lowest connected ancestor in the tree, who reconstitutes the object following the disconnection, would reconstitute o_i or a later version (see Section 6.1). So, o_i is never visible in the connected component containing the server. This also implies that for each object instance o such that $o \Rightarrow o_i$ (o depends on o_i), o is not visible in the connected component: if o is migrated (or copied) up to the connected component, then o_i (or a later version) must be copied along with it (see Section 4.4). Therefore, none of the other object instances produced by op_i are visible in the connected component, as each of these instances depends on o_i. As a consequence, none of the instances produced by op_i is ever read by a connected process and so $op_i \overset{\text{rf}}{\not\rightarrow}_* op_k$. □

Lemma 1 ensures that per object identifier, there is a unique sequence of object instances (ordered by version number) that are visible to connected processes. In addition, Lemma 1 also provides an avenue by which we can define the Durable set for our protocol, i.e., to consist of those update operations that produce object instances visible to the connected processes and those read operations that observe those object instances.

Definition 2 ($\mathsf{Durable}$)**.** *The set* $\mathsf{Durable}$ *is defined inductively to include operations according to the following two rules (and no other operations):*

1. *If op_i was executed at a connected process, then $op_i \in \mathsf{Durable}$.*

2. *If $op_i \in \mathsf{Durable}$ and $op_j \xrightarrow{\text{rf}}_* op_i$, then $op_j \in \mathsf{Durable}$.*

Below we prove that the operations in $\mathsf{Durable}$ are serializable when the updates and multi-object reads are implemented as in Section 5.1 and single object reads are implemented as in Section 5.2.1. Furthermore operations in $\mathsf{Durable}$ are strictly serializable for the other incarnation of our system, i.e., when the updates and multi-object reads are implemented as in Section 5.1 and single object reads are implemented as in Section 5.2.2. Note that in either case, operations in $\mathsf{Durable}$ are in fact durable, since “losing” an update could violate serializability or strict serializability. Finally, note that Lemma 1 holds for either incarnation of our system.

7.1 Proof of serializability

Multi-version Serializability theory Our system maintains multiple versions of the same object at the same time (although not at the same process), therefore we argue the serializability of our algorithms using multi-version serializability theory [3]. Multi-version serializability theory allows us to argue the serializability of a set of operations through the acyclicity of a particular graph, called the *multi-version serialization graph*.

Definition 3 ($\xrightarrow{\text{ver}}$)**.** *The version precedence relation, denoted* $\xrightarrow{\text{ver}}$*, is defined for operations as follows: For distinct operations* op_i*,* op_j *and* op_k*, let* op_k *read an object instance* o_j *produced by* op_j *and* op_i *produce an object instance* o_i *such that* $o_i.\mathsf{id} = o_j.\mathsf{id}$*. If* $o_i.\mathsf{version} < o_j.\mathsf{version}$ *then* $op_i \xrightarrow{\text{ver}} op_j$*, otherwise* $op_k \xrightarrow{\text{ver}} op_i$*.*

Definition 4 (Multi-version serialization graph)**.** *A multi-version serialization graph of a set* S *of operations, denoted* $MVSG(S)$*, is a directed graph whose nodes are operations in* S *and there is an edge from operation* op_i *to operation* op_j *if* $op_i \xrightarrow{\text{rf}} op_j$ *or* $op_i \xrightarrow{\text{ver}} op_j$ *or both.*

In order to prove that the set S of operations is serializable, it is both necessary and sufficient to prove that $MVSG(S)$ is acyclic [3, Theorem 5.4].

We prove the acyclicity of $MVSG(\mathsf{Durable})$ in two steps: First we prove that its subgraph consisting only of update and multi-object read operations (and the corresponding edges) is acyclic. We then prove that adding single-object read operations and the corresponding edges to this acyclic subgraph does not introduce any cycles.

Let $\mathsf{Durable}'$ denote the subset of $\mathsf{Durable}$ consisting only of update and multi-object operations. In order to prove the acyclicity of $MVSG(\mathsf{Durable}')$, we describe a technique to assign timestamps to operations in $\mathsf{Durable}'$, and then prove that all edges in $MVSG(\mathsf{Durable}')$ are in timestamp order. Since timestamp order is acyclic, this proves the acyclicity of $MVSG(\mathsf{Durable}')$. Note that these timestamps serve only to argue about the order of operations and do not add functionality to our algorithms.

Assigning timestamps Let $ts(op)$ denote the timestamp assigned to an operation op. Let $input(op)$ and $output(op)$ denote the set of object instances input to and produced by operation op, respectively. We assign timestamps to update and multi-object operations such that for each pair of operations op_i and op_j, if $op_i \xrightarrow{\text{rf}} op_j$ then $ts(op_i) < ts(op_j)$. Timestamps with these properties can be assigned as follows: Store a timestamp $ts\_recent(o)$ for each object instance o. For each update or multi-object read operation op, define $maxTs(op)$ as:

$$maxTs(op) = \max_{o \in input(op)} ts\_recent(o)$$

Then assign the timestamp to op as follows:

$$ts(op) \leftarrow maxTs(op) + 1$$

The timestamp for each object instance involved in the operation op is updated as follows:

$$\forall o \in input(op) \bigcup output(op) : \; ts\_recent(o) \leftarrow maxTs(op) + 1$$

Let $ID_{in}(op)$ and $ID_{out}(op)$ denote the set of identifiers of the object instances input to and output by an operation op, respectively, i.e., $ID_{in}(op) = \{o.\mathsf{id} : o \in input(op)\}$ and $ID_{out}(op) = \{o.\mathsf{id} : o \in output(op)\}$.

Lemma 2. *Let* op_i *and* op_j *be distinct update or multi-object read operations in* $\mathsf{Durable}'$ *performed by processes* p_i *and* p_j*, respectively, such that* $ID_{in}(op_i) \cap ID_{in}(op_j) \neq \emptyset$*. If for some* $id \in ID_{in}(op_i) \cap ID_{in}(op_j)$*,* p_i *retrieves an instance* o_i *with* $o_i.\mathsf{id} = id$ *before* p_j *retrieves an instance* o_j *with* $o_j.\mathsf{id} = id$*, then* $ts(op_i) < ts(op_j)$*.*

Proof. Since updates and multi-object operations retrieve instances with the same identifier serially and there is a unique sequence of instances with the same identifier (Lemma 1), p_j cannot retrieve o_j before p_i invokes a $V(p_i.\mathsf{sem}[id])$. This $V(p_i.\mathsf{sem}[id])$ is performed only after p_i completes op_i (last statement of

$prog_0$, see Section 5.1) and therefore, only after assigning $ts\_recent(o_i') \leftarrow ts(op_i)$, where o_i' is either o_i or its newer version in case op_i updates o_i. Since ts_recent can only grow and op_j is assigned a timestamp greater than ts_recent of all instances in $input(op_j)$, $ts(op_j) \geq ts(op_i) + 1$. □

Lemma 3. *Let op_i and op_j be distinct operations in* $\mathsf{Durable}'$*, such that* $op_i \xrightarrow{\text{rf}} op_j$*. Then* $ts(op_i) < ts(op_j)$*.*

Proof. Let o_i be an object instance produced by op_i at process p_i and input by op_j at process p_j. p_j can retrieve o_i only after p_i performs a $V(p_i.\mathsf{sem}[o_i.\mathsf{id}])$, which is done after op_i completes. Therefore, p_i must complete the retrieval of an instance with identifier $o_i.\mathsf{id}$ (the instance input to op_i) before p_j retrieves o_i as input to op_j, and so by Lemma 2, $ts(op_i) < ts(op_j)$. □

Lemma 4. *Let op_i and op_j be distinct operations in* $\mathsf{Durable}'$*, such that* $op_i \xrightarrow{\text{ver}} op_j$*. Then* $ts(op_i) < ts(op_j)$*.*

Proof. $op_i \xrightarrow{\text{ver}} op_j$ can be a result of one of the following two cases.

Case 1: op_i, op_j and op_k are distinct operations performed by processes p_i, p_j and p_k, respectively, such that op_k inputs instance o_j produced by op_j, op_i produces instance $o_i : o_i.\mathsf{id} = o_j.\mathsf{id}$ and $o_i.\mathsf{version} < o_j.\mathsf{version}$. Since instances with the same identifier are retrieved serially and form a unique sequence ordered by version numbers (Lemma 1), p_i must retrieve instance with identifier $o_i.\mathsf{id}$ (for input to op_i) before p_j does (for input to op_j) and therefore by Lemma 2, $ts(op_i) < ts(op_j)$.

Case 2: op_i, op_j and op_k are distinct operations performed by connected processes, such that op_i inputs instance o_k produced by op_k, op_j produces instance $o_j : o_j.\mathsf{id} = o_k.\mathsf{id}$ and $o_j.\mathsf{version} > o_k.\mathsf{version}$. Since op_i inputs a version older than the one produced by op_j and object instances are retrieved serially and have a unique sequence ordered by version numbers (Lemma 1), p_i must have retrieved o_k before p_j retrieved an instance with identifier $o_k.\mathsf{id}$ and performed op_j. Hence, by Lemma 2, $ts(op_i) < ts(op_j)$. □

Theorem 1. $MVSG(\mathsf{Durable}')$ *is acyclic.*

Proof. All edges in $MVSG(\mathsf{Durable}')$ are in timestamp order (Lemmas 3 and 4) and timestamp order is acyclic. □

Lemma 5. *Adding single-object read operations as described in Section 5.2.1 (and the corresponding* $\xrightarrow{\text{rf}}$ *and* $\xrightarrow{\text{ver}}$ *edges) from* $\mathsf{Durable}$ *to the acyclic* $MVSG(\mathsf{Durable}')$ *does not introduce any cycles.*

Proof. Arbitrarily order the single-object read operations in $\mathsf{Durable} \setminus \mathsf{Durable}'$, and consider inserting them one-by-one into the acyclic (Theorem 1) $\mathsf{Durable}'$. Update the corresponding $MVSG$ by adding a node for each new operation and any new $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ver}}$ edges induced by this new node. For a contradiction, let $op_i \in \mathsf{Durable} \setminus \mathsf{Durable}'$ be the first single-object read operation whose insertion results in a cycle. The insertion of op_i adds the following edges: A single reads-from edge from the update operation op_k that produced the object instance o_k read by op_i, and a version precedence edge from op_i to each update operation op_j that produces an instance o_j with $o_j.\mathsf{id} = o_k.\mathsf{id}$ and $o_j.\mathsf{version} > o_k.\mathsf{version}$.

Assume for contradiction that these new edges and the node op_i introduce a cycle in the multiversion seralizability graph. This is possible only if there already exists a path from some such op_j to op_k. But there also already exists a path from op_k to op_j in $MVSG(\mathsf{Durable}')$ as $op_k \xrightarrow{\text{rf}}_* op_j$: op_j produces a newer version of the instance output by op_k and the retrievals are serialized for instances with the same identifier (Lemma 1). Thus, there must already be a cycle (from op_k to op_j and back to op_k) even before adding op_i, a contradiction. □

Theorem 2. $\mathsf{Durable}$ *is serializable.*

Proof. $MVSG(\mathsf{Durable}')$ is acyclic (Theorem 1) and adding single-object read operations and the corresponding edges to this subgraph does not introduce any cycles (Lemma 5). Therefore, $MVSG(\mathsf{Durable})$ is acyclic and thus Durable is serializable [3]. □

7.2 Proof of strict serializability

In order to achieve strict serializability, the updates and multi-object reads are performed in the same way as for the serializable version of our protocols, i.e., updates and multi-object reads involving the same objects are serialized, see Section 5.1. However, single object reads are performed as in Section 5.2.2, instead of reading the local copy of the object as in the serializable algorithm. Strict serializability requires that all (connected) processes perceive the operations to be in the same sequential order (serializability) and furthermore, this sequential order must preserve the real-time order between operations, i.e., if op_i completes before op_j is invoked, then op_i must precede op_j in the sequential order perceived by the processes. We first prove that the subset Durable′ of Durable containing all the updates and multi-object reads in Durable, and no other operations, is strictly serializable. We then prove that the single object reads implemented as described in Section 5.2.2 do not violate strict serializability.

Definition 5 (real-time order, $\xrightarrow{\text{rt}}$)**.** *We say $op_i \xrightarrow{\text{rt}} op_j$, if op_i completes before op_j is invoked.*

Definition 6 (Multiversion strict serialization graph)**.** *A multi-version strict serialization graph of a set S of operations, denoted $MVSSG(S)$, is the graph $MVSG(S)$, with an additional edge between each $op_i, op_j \in S$, if $op_i \xrightarrow{\text{rt}} op_j$.*

We prove the strict serializability of operations in Durable by showing that if $MVSG(\mathsf{Durable})$ is acyclic, then $MVSSG(\mathsf{Durable})$ is also acyclic. Note that if $MVSSG(\mathsf{Durable})$ is acyclic, then a topological sort of $MVSSG(\mathsf{Durable})$ yields a strict serialization of the operations in the set Durable.

We define $\xrightarrow{\text{rf,ver}}$ as $\xrightarrow{\text{rf}} \cup \xrightarrow{\text{ver}}$ and $\xrightarrow{\text{rf,ver,rt}}$ as $\xrightarrow{\text{rf,ver}} \cup \xrightarrow{\text{rt}}$. So if $op_i \xrightarrow{\text{rf,ver,rt}} op_j$, then at least one of the three relations, $op_i \xrightarrow{\text{rf}} op_j$, $op_i \xrightarrow{\text{ver}} op_j$, $op_i \xrightarrow{\text{rt}} op_j$, holds. Finally we define, $\xrightarrow{\text{rf,ver}}_*$ and $\xrightarrow{\text{rf,ver,rt}}_*$ as the transitive closure of $\xrightarrow{\text{rf,ver}}$ and $\xrightarrow{\text{rf,ver,rt}}$ respectively.

Lemma 6. *Let op_i and op_j be distinct operations in* Durable′ *such that $ID_{in}(op_i) \cap ID_{in}(op_j) \neq \emptyset$. If $ts(op_i) < ts(op_j)$, then op_i completes before op_j completes.*

Proof. Assume for contradiction that op_j performed by process p_j completes before op_i performed by process p_i completes. Then there must exist an identifier $id \in ID_{in}(op_i) \cap ID_{in}(op_j)$ such that p_j retrieves an instance $o_j : o_j.\mathsf{id} = id$ for input to op_j before p_i retrieves an instance $o_i : o_i.\mathsf{id} = id$ for input to op_i: otherwise, if p_i retrieves o_i before p_j retrieves o_j, then p_j must wait for p_i to release the object with identifier id, which is done via a $V(p_i.\mathsf{sem}[id])$ only after p_i completes op_i (last statement of $prog_0$, see Section 5.1). Since p_j retrieves instance $o_j : o_j.\mathsf{id} = id$ for input to op_j before p_i retrieves instance $o_i : o_i.\mathsf{id} = id$ for input to op_i, it must be the case that $ts(op_j) < ts(op_i)$ (Lemma 2), a contradiction. □

Lemma 7. *Let op_i and op_j be distinct operations in* Durable′*. If $op_i \xrightarrow{\text{rf,ver}}_* op_j$ and all the operations that make up the sequence in this transitive relation are in* Durable′*, then op_i completes before op_j completes.*

Proof. We first note that $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ver}}$ preserve the timestamp order, i.e., if $op_i \xrightarrow{\text{rf}} op_j$, then $ts(op_i) < ts(op_j)$ (Lemma 3) and if $op_i \xrightarrow{\text{ver}} op_j$, then $ts(op_i) < ts(op_j)$ (Lemma 4). Therefore, $op_i \xrightarrow{\text{rf}} op_j$ and $op_i \xrightarrow{\text{ver}} op_j$ each implies that op_i completes before op_j completes (Lemma 6). Finally, note that the “completes before” relation is transitive, i.e., if op_i completes before op_k completes and op_k completes before op_j completes, then op_i completes before op_j completes. □

Corollary 1. *Let op_i and op_j be distinct operations in* $\mathsf{Durable}'$. *If* $op_i \xrightarrow{\text{rf,ver,rt}}_* op_j$ *and all the operations that make up the sequence in this transitive relation are in* $\mathsf{Durable}'$*, then op_i completes before op_j completes.*

Proof. This is a direct consequence of (i) Lemma 7, (ii) the fact that if $op_i \xrightarrow{\text{rt}} op_j$, then op_i completes before op_j is invoked, and therefore before op_j completes, and (iii) that $\xrightarrow{\text{rt}}$ is transitive. □

Theorem 3. *$MVSSG$($\mathsf{Durable}'$) is acyclic.*

Proof. We know that $MVSG(\mathsf{Durable}')$ is acyclic (Theorem 1). Assume for contradiction that $MVSSG(\mathsf{Durable}')$ has a cycle. Now construct $MVSSG(\mathsf{Durable}')$ by adding each real-time order edge to $MVSG(\mathsf{Durable}')$ one by one. Let $op_i \xrightarrow{\text{rt}} op_j : op_i, op_j \in \mathsf{Durable}'$ be the first edge that creates a cycle during the construction of $MVSSG(\mathsf{Durable}')$. This cycle is possible only if there already existed a path from op_j to op_i before adding $op_i \xrightarrow{\text{rt}} op_j$ to the graph being constructed. This path consists of $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ver}}$ edges from $MVSG(\mathsf{Durable}')$, and possibly some $\xrightarrow{\text{rt}}$ edges added to the graph before adding $op_i \xrightarrow{\text{rt}} op_j$. We can therefore state the relation between op_j and op_i as $op_j \xrightarrow{\text{rf,ver,rt}}_* op_i$. This implies that op_j completes before op_i completes (Corollary 1), and therefore, $op_i \not\xrightarrow{\text{rt}} op_j$, a contradiction. □

The remaining part of the proof deals with the single object read operations as implemented in Section 5.2.2, and proves that these operations do not introduce any cycles when added to the acyclic $MVSSG(\mathsf{Durable}')$.

Lemma 8. *Let $op_i \in \mathsf{Durable} \setminus \mathsf{Durable}'$ be a single object read of an object with identifier id implemented as described in Section 5.2.2. Let $op_j \in \mathsf{Durable} : id \in ID_{out}(op_j)$ be the most recent such update operation to complete before op_i is invoked, and let $o_j : o_j.\mathsf{id} = id$ be an instance produced by op_j. Then op_i either reads o_j or an instance $o_k : o_k.\mathsf{id} = id, o_k.\mathsf{version} > o_j.\mathsf{version}$.*

Proof. Let p_i be the process that performs the single object read op_i and p_j be the process that completes op_j. op_j completes before op_i is invoked, i.e., before p_i initiates the read request for op_i. Once initiated the request follows localQ.head pointers towards the current owner of the object with identifier id. This current owner is either (i) p_j itself, or (ii) a process p_k that either performs an operation $op_k : id \in ID_{in}(op_k)$ after p_j completes op_j, or p_k is in the migration path of this object as it is being migrated to some third process. In case (i) (p_j is the current owner), p_j responds to the read request with $p_j.\mathsf{objs}[id] = o_j$, and the lemma holds. In case (ii) (p_k is the current owner, or is in the migration path), p_k responds with $o_k = p_k.\mathsf{objs}[id]$. Since, there is a unique sequence of object instances with the same identifier (Lemma 1), and objects are migrated serially (Lemma 2), it must be the case that $o_k.\mathsf{version} \geq o_j.\mathsf{version}$ and so the lemma holds. □

Corollary 2. *Let $op_i \in \mathsf{Durable} \setminus \mathsf{Durable}'$ be a single object read of an object with identifier id implemented as described in Section 5.2.2, and let $op_j \in \mathsf{Durable}$. If $op_i \xrightarrow{\text{ver}} op_j$, then op_j completes after op_i is invoked, i.e., $op_j \not\xrightarrow{\text{rt}} op_i$.*

Proof. Let $o_i : o_i.\mathsf{id} = id$ be the object instance read by the operation op_i. Then $op_i \xrightarrow{\text{ver}} op_j$ implies that op_j produces an instance $o_j : o_j.\mathsf{version} > o_i.\mathsf{version}$. (This is the only possible reason for the edge $op_i \xrightarrow{\text{ver}} op_j$ when op_i is a single object read operation.) Assume for contradiction, that op_j completes before op_i is invoked. Then, the object instance read by op_i must have $o_i.\mathsf{version} \geq o_j.\mathsf{version}$ (Lemma 8), and as a result $op_i \not\xrightarrow{\text{ver}} op_j$, a contradiction. □

Lemma 9. *Let op and op' be distinct operations in* $\mathsf{Durable}$. *If $op \xrightarrow{\text{rf,ver,rt}}_* op'$ and $op \in \mathsf{Durable}'$, then op completes before op' completes. (Note that we only restrict the first operation op to be in* $\mathsf{Durable}'$. *All other operations involved are in* $\mathsf{Durable}$.*)*

Proof. Since $\xrightarrow{\text{rf,ver,rt}}_*$ is a transitive closure of $\xrightarrow{\text{rf,ver,rt}}$, there exist a finite sequence of operations $op_i, 1 \leq i \leq n$, such that $op \xrightarrow{\text{rf,ver,rt}} op_1 \xrightarrow{\text{rf,ver,rt}} \ldots \xrightarrow{\text{rf,ver,rt}} op_n \xrightarrow{\text{rf,ver,rt}} op'$. The case where all operations in the sequence are in $\mathsf{Durable}'$ is handled by Corollary 1. Here we focus on the cases where single object read operations may be part of the sequence. We first prove that for each op_i ($1 \leq i \leq n$), such that $op_i \in \mathsf{Durable} \setminus \mathsf{Durable}'$, op_{i-1} (the operation immediately preceding op_i in the sequence) completes before op_{i+1} (the operation immediately succeeding op_i in the sequence) completes. We then handle the case when the last operation $op' \in \mathsf{Durable} \setminus \mathsf{Durable}'$. Finally, we handle the case when $n = 0$, i.e., $op \xrightarrow{\text{rf,ver,rt}} op'$.

Let op_i ($1 \leq i \leq n$) be any single object read operation in the sequence. In order to prove that op_{i-1} completes before op_{i+1} completes, there are only four cases to consider (this is because if op_i is a single object read, then $op_j \xrightarrow{\text{ver}} op_i$ and $op_i \xrightarrow{\text{rf}} op_j$ are not possible, for any operation $op_j \in \mathsf{Durable}$):

Case 1 ($op_{i-1} \xrightarrow{\text{rf}} op_i \xrightarrow{\text{rt}} op_{i+1}$): Since op_i reads an instance produced by op_{i-1}, op_{i-1} must complete before op_i completes (processes do not make a new version available until the operation producing this version completes). Also since $op_i \xrightarrow{\text{rt}} op_{i+1}$, op_i completes before op_{i+1} is invoked. Thus op_{i-1} completes before op_{i+1} is invoked, and therefore, before op_{i+1} completes.

Case 2 ($op_{i-1} \xrightarrow{\text{rf}} op_i \xrightarrow{\text{ver}} op_{i+1}$): In this case, both op_{i-1} and op_{i+1} produce a new instance of the object read by op_i, therefore $op_{i-1}, op_{i+1} \in \mathsf{Durable}'$. Furthermore, $ID_{in}(op_i) \in ID_{in}(op_{i-1}) \cap ID_{in}(op_{i+1})$. Let id be the identifier of the object read by op_i. Then op_{i-1} produces an instance $o_{i-1} : o_{i-1}.\mathsf{id} = id$ that is read by op_i, and op_{i+1} produces an instance $o_{i+1} : o_{i+1}.\mathsf{id} = id$ and $o_{i+1}.\mathsf{version} > o_{i-1}.\mathsf{version}$ (hence the relation $op_i \xrightarrow{\text{ver}} op_{i+1}$). Therefore, the process performing op_{i-1} must have retrieved an instance with identifier id for op_{i-1} before an instance with identifier id was retrieved by the process performing op_{i+1}, and so $ts(op_{i-1}) < ts(op_{i+1})$ (Lemma 2). So op_{i-1} completes before op_{i+1} completes (Lemma 6).

Case 3 ($op_{i-1} \xrightarrow{\text{rt}} op_i \xrightarrow{\text{ver}} op_{i+1}$): op_{i-1} completes before op_i is invoked and op_{i+1} completes after op_i is invoked (Corollary 2). Therefore, op_{i-1} completes before op_{i+1} completes.

Case 4 ($op_{i-1} \xrightarrow{\text{rt}} op_i \xrightarrow{\text{rt}} op_{i+1}$): op_{i-1} completes before op_i is invoked and op_i completes before op_{i+1} is invoked. Therefore, op_{i-1} completes before op_{i+1} is invoked, and so before op_{i+1} completes.

Note that these cases handling the intermediate single object operations, together with Corollary 1, prove that op completes before op_n completes, and extend to op' if $op' \in \mathsf{Durable}'$. Now in case op' (the last operation in the sequence) is a single object operation, we can either have $op_n \xrightarrow{\text{rf}} op'$ or $op_n \xrightarrow{\text{rt}} op'$. Note that in either case op_n completes before op' completes, and so the lemma statement holds.

Finally, if $n = 0$, i.e., $op \xrightarrow{\text{rf,ver,rt}} op'$, and $op' \in \mathsf{Durable}'$, then the lemma holds due to Corollary 1. If $op \xrightarrow{\text{rf,ver,rt}} op'$ and op' is a single-object read operation, then we can either have $op \xrightarrow{\text{rf}} op'$ or $op \xrightarrow{\text{rt}} op'$, and in either case op completes before op' completes. □

Lemma 10. *Let op and op' be distinct operations in* $\mathsf{Durable}$. *If $op \xrightarrow{\text{rf,ver,rt}}_* op'$, then op is invoked before op' completes. (Note that this statement does not restrict op to be in* $\mathsf{Durable}'$ *as in Lemma 9, and is therefore, stronger than Lemma 9.)*

Proof. In case $op \in \mathsf{Durable}'$, the result holds directly due to Lemma 9. We now consider the case when $op \in \mathsf{Durable} \setminus \mathsf{Durable}'$. $op \xrightarrow{\text{rf,ver,rt}}_* op'$ is represented by the finite sequence $op \xrightarrow{\text{rf,ver,rt}} op_1 \xrightarrow{\text{rf,ver,rt}} \ldots \xrightarrow{\text{rf,ver,rt}} op_n \xrightarrow{\text{rf,ver,rt}} op'$ (as in Lemma 9). Let op_j represent the operation that immediately succeeds op in this sequence, i.e., $op_j = op_1$ if $n \neq 0$, and $op_j = op'$ if $n = 0$. If op is a single object read operation then there are only two possibilities:

Case 1 ($op \xrightarrow{\text{ver}} op_j$): In this case, op is invoked before op_j completes (Corollary 2). Therefore, if $op_j = op'$ ($n = 0$), then the statement holds. In case $op_j = op_1$ ($n \neq 0$), we note that $op_j \in \mathsf{Durable}'$ (since it *produces* a version later than the one read by op) and so we can apply Lemma 9, i.e., op_j completes before op' completes. This implies that op is invoked before op' completes, and the statement holds.

Case 2 ($op \xrightarrow{\text{rt}} op_j$): In this case op completes before op_j is invoked. Therefore, the statement holds if $op_j = op'$ ($n = 0$). If $op_j = op_1$ ($n \neq 0$) and all operations in the sequence $\{op_1, \ldots, op_n, op'\}$ are single-object read operations, then it must be the case that $op_1 \xrightarrow{\text{rt}} op_2 \xrightarrow{\text{rt}} \ldots \xrightarrow{\text{rt}} op_n \xrightarrow{\text{rt}} op'$, as these are the only possible edges between successive single-object read operations, and the lemma statement is obviously true. If all operations in this sequence are not single-object read operations, then let op_k be the first operation in the sequence that is in $\mathsf{Durable}'$. In this case, we make three observations: (a) Applying Lemma 9 to the sequence $\{op_k, op_{k+1}, \ldots, op_n\}$, we note that op_k completes before op_n completes. (b) Since all operations preceding op_k in the sequence are single-object read operations, therefore, it must be the case that $op \xrightarrow{\text{rt}} op_1 \xrightarrow{\text{rt}} \ldots \xrightarrow{\text{rt}} op_{k-1}$. Therefore, op completes before op_{k-1} is invoked. (c) Finally, note that the only possible edges from $op_{k-1} \in \mathsf{Durable} \setminus \mathsf{Durable}'$ to $op_k \in \mathsf{Durable}'$ are $op_{k-1} \xrightarrow{\text{rt}} op_k$ (in which case op_{k-1} completes before op_k is invoked), and $op_{k-1} \xrightarrow{\text{ver}} op_k$ (in which case op_{k-1} is invoked before op_k completes, due to Corollary 2). Observations (a), (b) and (c) together prove the lemma. □

Theorem 4. *$MVSSG$($\mathsf{Durable}$) is acyclic.*

Proof. Assume, for contradiction, that $MVSSG$($\mathsf{Durable}$) has a cycle. Construct $MVSSG$($\mathsf{Durable}$) by starting with the acyclic $MVSSG$($\mathsf{Durable}'$) (Theorem 3), and adding the node and corresponding edges for each (single object read) operation in $\mathsf{Durable} \setminus \mathsf{Durable}'$, one after the other. Consider the first $op \in \mathsf{Durable} \setminus \mathsf{Durable}'$ which when added along with the corresponding edges, results in a cycle. The insertion of $op \in \mathsf{Durable} \setminus \mathsf{Durable}'$ that reads a single object instance $o : o.\mathsf{id} = id$, results in the addition of the following edges: (i) A single $op_i \xrightarrow{\text{rf}} op$ edge from $op_i \in \mathsf{Durable}'$ that produces the instance o read by op, (ii) a number of $op_j \xrightarrow{\text{rt}} op$ edges from each operation op_j that completes before op is invoked, (iii) a number of $op \xrightarrow{\text{ver}} op_k$ edges for each op_k that produces an instance $o' : o'.\mathsf{id} = id, o'.\mathsf{version} > o.\mathsf{version}$, and (iv) a number of $op \xrightarrow{\text{rt}} op_l$ edges for each op_l that is invoked after op completes. Note that (i) and (ii) are incoming edges, i.e., those directed towards op, while (iii) and (iv) are outgoing edges. We consider each possible combination of these edges, and prove that the combination could not result in a cycle.

Case 1 ($op_i \xrightarrow{\text{rf}} op \xrightarrow{\text{rt}} op_l$): If these two edges result in a cycle, then there must already exist a path from op_l to op_i, i.e., $op_l \xrightarrow{\text{rf,ver,rt}}_* op_i$. This implies that op_l is invoked before op_i completes (Corollary 10), and therefore before op completes (since op_i completes before op completes, $op_i \xrightarrow{\text{rf}} op$). However, this is a contradiction since $op \xrightarrow{\text{rt}} op_l$. Therefore, these two edges cannot create a cycle in the graph.

Case 2 ($op_i \xrightarrow{\text{rf}} op \xrightarrow{\text{ver}} op_k$): If these two edges result in a cycle, then there must already exist a path from op_k to op_i. However, since $op_k, op_i \in \mathsf{Durable}', ID_{in}(op) \in ID_{in}(op_k) \cap ID_{in}(op_i)$ and op_k produces a later version of an object instance produced by op_i, it must be the case that $op_i \xrightarrow{\text{rf}}_* op_k$ and so there must already be a path from op_i to op_k. This implies that there must already exist a cycle in the graph (op_i to op_k to op_i), a contradiction. Therefore, these two edges cannot create a cycle in the graph.

Case 3 ($op_j \xrightarrow{\text{rt}} op \xrightarrow{\text{ver}} op_k$): If these two edges result in a cycle, then there must already exist a path from op_k to op_j, i.e., $op_k \xrightarrow{\text{rf,ver,rt}}_* op_j$. Since $op_k \in \mathsf{Durable}'$ (it produces a new version of the object read by op), we can apply Lemma 9, and state that op_k completes before op_j completes, and therefore before op is invoked. However, this is a contradiction since $op \xrightarrow{\text{ver}} op_k$ (due to Corollary 2). Therefore, these two edges cannot create a cycle in the graph.

Case 4 ($op_j \xrightarrow{\text{rt}} op \xrightarrow{\text{rt}} op_l$): If these two edges result in a cycle, then there must already exist a path from op_l to op_j, i.e., $op_l \xrightarrow{\text{rf,ver,rt}}_* op_j$. This implies that op_l is invoked before op_j completes (Corollary 10), and therefore before op is invoked. But this is a contradiction since $op \xrightarrow{\text{rt}} op_l$. Therefore, these two edges cannot create a cycle in the graph.

Therefore $MVSSG$(Durable) is acyclic, and hence Durable is strictly serializable. □

8 Evaluation

We evaluated the performance of our self-service system in two types of experiments. First, we measured the performance of a trivial service in which operations require no processing. These microbenchmarks illustrate the inherent costs of our implementation. However, the self-service approach is poorly suited to a service with these characteristics. After all, harnessing client resources to perform only trivial operations is of little use, and incurs the unnecessary overhead of object migrations.

We thus performed a second evaluation of an application better suited to self-service, and indeed that partially motivated it. This service enables the construction of network traffic models from distributed data sources. This service involves computationally intensive operations, making it better suited to our approach, but also more difficult to run on, e.g., PlanetLab.[4] So, we performed these experiments, as well as our microbenchmarks for comparison purposes, on a 70-node cluster.

8.1 Experimental system

Our system is implemented in Java 5.0 and, at the time of this writing, is relatively unoptimized. It does, however, employ the following optimizations:

Object compression Nontrivial objects (objects in the application discussed in Section 8.3 approach a few hundred kilobytes) are stored and transmitted in compressed form. The memory and bandwidth savings due to compression far outweigh the computation costs. We use the LZO compression library[5], invoked from Java via the Java Native Interface.

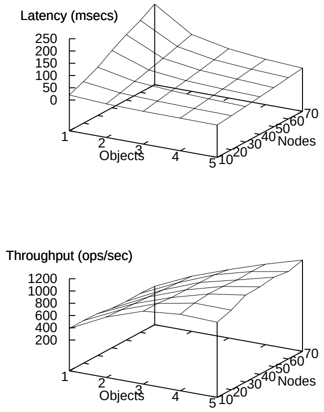
Serving reads from copies A node makes a local copy of an object before updating it, so it can serve reads while the object is being modified. This improves the performance of reads when updates are computationally expensive—the setting targeted by self-service. If an object is not being modified, reads are served directly from the object.

In-memory objects Objects are kept in memory in their compressed forms and are decompressed when needed to perform operations.

To control the experiments and measure the system performance, we used a *monitor* that ran on a dedicated machine and communicated with all nodes in an experiment. In each experiment, each client joined the tree, notified the monitor when its join procedure was complete, performed read and update operations (and possibly left and rejoined the tree, depending on the experiment) and finally reported per-operation latency and the total number of operations performed to the monitor. The monitor computed the average latency across all nodes and the overall system throughput—operations per second performed by the system as a whole. Each experiment was repeated five times with a random node chosen as the server each time. Each client waited a random amount of time before sending its join request to the server, resulting in a different tree configuration for each run.

[4] We tried, and during the time we used PlanetLab, we saw an average of only 7-10% CPU available for our slice on most PlanetLab nodes. This was insufficient for our experiments.

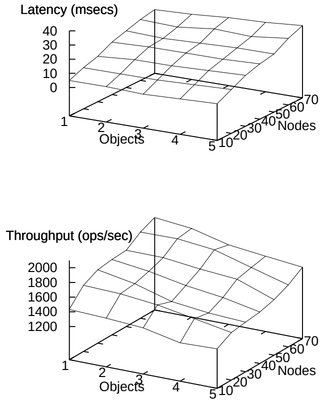
[5] `http://www.oberhumer.com/opensource/lzo/`



		nodes						
objects		10	20	30	40	50	60	70
1	ms	21 [14,38]	44 [32,77]	71 [55,110]	108 [86,174]	141 [117,219]	168 [141,286]	203 [167,312]
	ops/s	389	392	383	343	330	336	325
2	ms	12 [1,28]	25 [2,50]	41 [3,85]	56 [4,109]	72 [10,161]	89 [4,167]	107 [5,203]
	ops/s	683	674	653	638	627	614	599
3	ms	9 [1,24]	19 [3,44]	30 [3,72]	40 [8,81]	51 [4,118]	62 [6,150]	76 [12,141]
	ops/s	888	889	861	860	856	854	813
4	ms	8 [1,21]	16 [2,43]	24 [3,58]	33 [5,74]	42 [4,116]	51 [5,120]	60 [5,158]
	ops/s	945	1003	1027	1024	1004	1016	1011
5	ms	7 [1,22]	14 [2,35]	20 [3,46]	28 [3,79]	34 [4,101]	44 [5,108]	51 [6,107]
	ops/s	925	996	1116	1125	1157	1123	1187
30	ms	5 [0,16]	13 [1,41]	17 [1,41]	23 [1,65]	26 [2,62]	32 [2,74]	39 [2,92]
	ops/s	976	1026	1363	1457	1444	1357	1478
40	ms	5 [0,16]	14 [0,44]	19 [1,55]	23 [1,46]	29 [1,60]	32 [2,65]	40 [2,96]
	ops/s	1102	1268	1320	1323	1346	1458	1459
50	ms	6 [0,17]	13 [0,37]	15 [1,31]	23 [1,64]	27 [1,56]	34 [2,81]	37 [2,93]
	ops/s	1171	1213	1202	1314	1363	1371	1436

Single-object updates

Single-object reads (strict serializability)



		nodes						
objects		10	20	30	40	50	60	70
1	ms	5 [1,14]	9 [1,28]	12 [1,49]	17 [1,81]	20 [1,87]	23 [1,109]	25 [1,116]
	ops/s	1434	1649	1747	1774	1787	1921	2007
2	ms	5 [1,15]	9 [1,26]	14 [1,47]	18 [1,62]	19 [1,80]	22 [2,98]	26 [2,120]
	ops/s	1412	1631	1655	1711	1784	1929	1956
3	ms	4 [1,12]	9 [1,27]	14 [1,50]	18 [1,72]	20 [2,79]	25 [2,108]	28 [2,130]
	ops/s	1378	1577	1512	1616	1727	1875	1831
4	ms	5 [1,17]	10 [1,28]	15 [1,49]	18 [1,73]	21 [2,88]	24 [2,105]	29 [2,152]
	ops/s	1272	1488	1453	1550	1686	1733	1774
5	ms	6 [1,15]	11 [1,36]	16 [1,63]	19 [1,77]	21 [2,83]	27 [2,139]	31 [2,161]
	ops/s	1288	1396	1425	1451	1506	1593	1722
30	ms	6 [1,17]	10 [1,24]	12 [1,30]	17 [2,48]	20 [3,44]	25 [3,65)	29 [3,79]
	ops/s	1156	1061	1288	1309	1402	1588	1602
40	ms	6 [1,17]	9 [1,24]	13 [1,45]	14 [2,40]	17 [2,44]	20 [3,63]	28 [3,91]
	ops/s	959	1089	1218	1379	1453	1592	1656
50	ms	6 [1,17]	8 [1,20]	12 [2,22)	17 [2,41]	24 [2,51]	30 [2,71]	31 [2,101]
	ops/s	932	1152	1397	1341	1481	1563	1633

Figure 9: Mean latencies (ms) and throughputs (ops/s) for single-object updates (top) and reads (bottom) of trivial objects in a static tree. Latencies are written in the form “mean [5th percentile, 95th percentile]”.

Our experiments were characterized by certain parameters. Each experiment was conducted on a fixed number nodes of nodes, including all the clients and the server, arranged in a 5-ary tree. Five is a smaller degree than we would suggest in practice, but we chose this so that the tree would gain depth as nodes was increased. Each experiment had a fixed number objects of objects. Each node performed 150 update operations and as many read operations as possible in this time (to keep the system loaded with reads throughout), before sending results to the monitor. Each node kept one read and one update operation outstanding at a time, i.e., after joining the tree each node initiated one read and one update operation, in parallel. Once a node completed an operation, it started a new operation of the same type (read/update) until it completed 150 update operations. Each update and read operation was performed on objsPerUp and objsPerRd objects, respectively, selected uniformly at random (without replacement, per operation) from the objects objects. Since serializable reads are performed locally and thus have negligible cost in comparison to other operations (see Section 5.2), all reads in our experiments were strictly serializable reads.

The experiments reported here were conducted on (up to) 70 nodes, each with an Intel P-IV 2.8GHz processor, 1GB of memory and an Intel PRO/1000 network interface card. The machines were connected with an HP ProCurve Switch 4140gl specified with a maximum throughput of 18.3Gbps.

8.2 Microbenchmarks

Our microbenchmark application is a trivial application in which each object is an integer counter that can be updated (incremented) or read.

Static tree, single-object operations Our first set of experiments performed only single-object operations ($\mathsf{objsPerUp} = \mathsf{objsPerRd} = 1$) in static trees with varying numbers of nodes and objects. The results of these tests are shown in Figure 9.

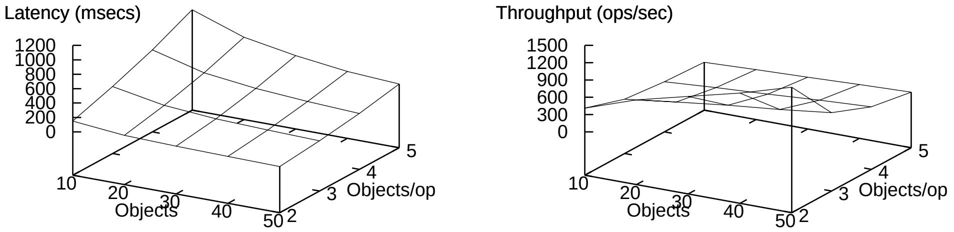
Two features of these results warrant discussion. First, update latency suffers for small numbers of objects and large numbers of nodes; e.g., for a single object, the update latency grew by an order of magnitude as the number of nodes ranged from 10 to 70. This occurs because these increasing numbers of updates often contend for the same object (or always do, in the case of one object). Since contending updates are processed sequentially, greater contention unavoidably increases the average latency. Note that these costs dissipate as the number of objects increases, and so contention decreases.

Second, read and update throughput exhibit very different trends; the former decreases and the latter increases as the number of objects increases. Update throughput benefits with more objects for the same reason that latency decreased, i.e., because there is less contention and hence more simultaneous updates occurring. Read throughput, on the other hand, *benefits* from update contention on an object that is the target of many reads, since the rapid migration of the object can improve servicing of read operations. Recall from Section 5.2 that a node p receiving a read request for object id can service the request if $p.\mathsf{localQ}[id].\mathsf{head}$ is the neighbor from whence the request came. When the object is migrating rapidly, this optimization can be exercised more often, increasing read throughput for an object with high update contention. Indeed, in tests with reads executing in isolation (not shown), i.e., with no concurrent writes, the throughput for small numbers of objects approached those of larger numbers of objects.

Static tree, multi-object operations Figure 10 shows performance for multi-object updates. We used $\mathsf{nodes} = 70$ and varied $\mathsf{objects}$ and $\mathsf{objsPerUp}$. Since for a trivial service, multi-object reads perform identically to multi-object updates, we plot only the update curves. For a fixed number of objects, update latency grows as the number of objects per operation grows. This results from two factors: (i) each node retrieves the objects sequentially (see Section 5.1); and (ii) the frequency of operations involving the same object increases with the number of objects per operation, and so the number of operations that conflict grows. The results also show that these factors are mitigated as the number of objects grows.

Dynamic tree, single-object operations Our third type of test evaluated the performance of our protocols during changes to the tree composition. Because accommodating leaves and joins is more involved than recovering from disconnections (which is a purely local algorithm, see Section 6.1), inducing leaves and joins yields a more conservative evaluation of our protocols when the tree is dynamic. In these tests, each client, after completing an update and before starting the next, chooses instead to leave the tree with probability $\Pr(\text{leave})$. If it chooses to not leave, then it commences its next update operation. Otherwise, it initiates the leave protocol described in Section 6.2. Upon completing the leave protocol, it immediately rejoins the tree using the join protocol, and then commences its next update (after which it will again leave the tree with probability $\Pr(\text{leave})$). As such, $\Pr(\text{leave})$ is roughly the ratio of leaves to update operations in the experiment.

We calculate latency as before, though we modify the way in which we calculate throughput, because the time a client spent leaving and rejoining the tree should not count toward the denominator of its operation throughput calculation. As such, during each run of the experiment, we calculate the time each client spent in the tree (actively performing operations), "pausing" this measurement when the client initiates a leave



		objsPerUp			
objects		2	3	4	5
10	ms	151 [4,822]	329 [5,1246]	538 [6,1525]	794 [7,1868]
	ops/s	412	194	120	82
20	ms	83 [5,697]	200 [7,1317]	349 [8,1783]	544 [10,2347]
	ops/s	719	307	181	117
30	ms	63 [5,508]	142 [7,1209]	264 [10,1865]	418 [11,2695]
	ops/s	940	413	229	148
40	ms	52 [6,327]	118 [8,1055]	213 [9,1731]	329 [12,2459]
	ops/s	1160	502	290	183
50	ms	41 [6,201]	98 [8,623]	177 [11,1707]	284 [13,2463]
	ops/s	1426	609	335	215

Figure 10: Mean latencies (ms) and throughputs (ops/s) for multi-object updates of trivial objects in a static tree (nodes $= 70$). Latencies are written in the form "mean [5th percentile, 95th percentile]".

and "resuming" it when the client completes a join. Then, we calculate throughput as the ratio of the total number of operations completed to the *average* time clients spent in the tree.

Example results of these tests are shown in Figure 11. Comparing these latency and throughput numbers to their corresponding numbers in Figure 9, we see that that leaves and joins impact these numbers modestly. For example, setting $\Pr(\text{leave}) = .005$, which induced between 19 and 23 leaves and rejoins among the nodes $= 40$ nodes in the tests, resulted in latency increases of approximately 6% and 18% for single-object updates and reads, respectively, in a system with one object. This was accompanied by decreases in throughput of 13% and 3% for the corresponding operation types. In a system of 50 objects with $\Pr(\text{leave}) = .005$, latencies for updates and reads grew by 4% and 12%, respectively, while update and read throughput suffered by 33% and 14%.

		Single-object updates		**Single-object reads**	
		$\Pr(\text{leave})$		$\Pr(\text{leave})$	
objects		.001	.005	.001	.005
1	ms	110 [19,134]	115 [8,128]	19 [0,63]	20 [0,79]
	ops/s	324	298	1321	1713
50	ms	20 [2,53]	24 [2,71]	19 [2,51]	19 [1,76]
	ops/s	996	885	1279	1151

Figure 11: Impact of leaves and joins on single-object operations. $\Pr(\text{leave})$ is the approximate ratio of leaves to updates in the experiment. nodes $= 40$.

In examining the sources of these costs, we found that they are not mainly due to computation or communication induced by the leaves or joins, but rather are due to the synchronization delays induced by leaves in our present implementation. Recall from Section 6.2 that when p leaves it informs its neighbors to suspend sending messages to p until its promoted replacement is ready. Our present implementation coarsely suspends the neighbors from sending *all* messages to *all* their neighbors; though only a brief suspension, it impacts throughput more substantially with large numbers of objects. We are presently refining our implementation to suspend sends more judiciously, and will report numbers for this implementation in the final version of this paper.

8.3 Network traffic classification service

As discussed previously, the microbenchmarks of Section 8.2 are pessimistic, in that a service with very low-cost operations is one that is poorly suited to the strengths of self-service. In this section we evaluate the implementation of a service that better represents the types of applications for which self-service was designed. Length restrictions preclude us from detailing this service fully, but we will briefly motivate and describe it here.

Today, network traffic characterization is an area of active research, including techniques to classify traffic as that of a particular application (e.g., see [26, 19] and the references therein) or as anomalous and thus indicative of an attack (e.g., [22, 45]). Much work suggests that models for performing this classification can be built more effectively by aggregating contributions from many networks (e.g., [43, 18, 2]). We are thus building a service through which networks can contribute traffic records toward the construction of classifiers for network traffic. In this application, the server is run by some coordination center, the clients are various networks that contribute records, and the shared objects are the classifiers. Our application supports an arbitrary number of classifiers, e.g., parameterized by application (port), attack attributes (“attack” vs. “normal”) or other characteristics. Strictly serializable semantics ensure reads see the latest models, in addition to offering atomic updates.

The classifiers that our service presently implements are support vector machines (SVMs) [7], a popular learning mechanism used for classification and regression and that is particularly well-suited to data with many features. More specifically, we use a variant of traditional SVMs called incremental SVMs [13, 6] that allow the models to be constructed incrementally as new contributions are received. SVMs have previously been used to characterize network traffic [11, 27], though not in a distributed setting. Our implementation uses the LIBSVM library[6] to construct SVM models from raw data.

For the purposes of this evaluation, the raw data consisted of pre-recorded connection records, each consisting of 41 features related to the connection including the application protocol, the transport protocol, protocol flags, connection length, etc. Each update operation updated a classifier with 500 new records. A CDF of the time required for updates and the sizes of the resulting models (compressed and uncompressed) are shown in Figure 12; as can be seen, updates are indeed computationally intensive.

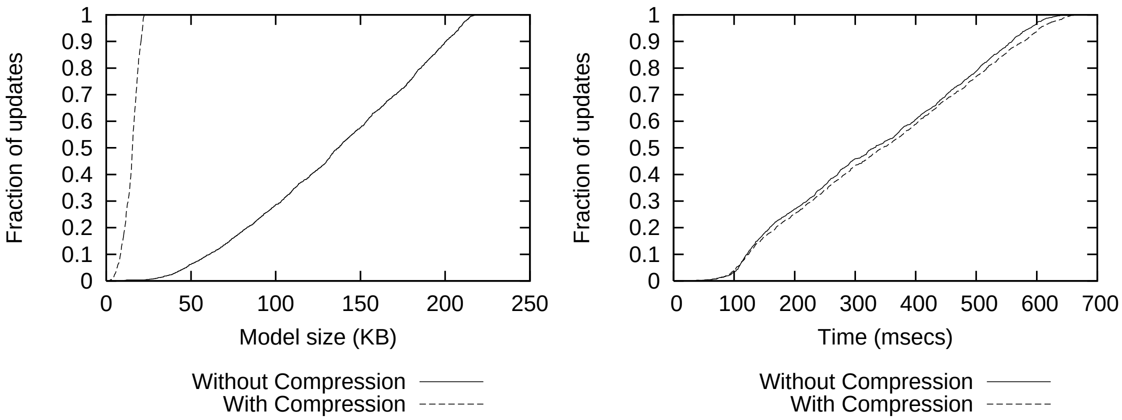


Figure 12: Cumulative distribution functions for update times and resulting model sizes.

[6] http://www.csie.ntu.edu.tw/~cjlin/libsvm

We compared the performance of our self-service implementation of this service against the same service built using a centralized implementation, which we optimized to the best of our ability. This implementation served read and update operations using different threads, so reads were not queued behind computationally expensive updates. To update a classifier, a client sent 500 connection records to the server who updated the corresponding classifier with this data. The server responded to a read operation by sending the requested classifier back to the client. The optimizations discussed for the self-service implementation—compressing objects, serving reads from copies and keeping objects in memory—were preserved in this implementation.

Static tree, single-object operations Our first experiment evaluated single-object operations using $\mathsf{objects} = 50$. Figure 13 plots the results; note that the vertical axes of these graphs are log scale. Our experiments showed that self-service update latency and throughput were dramatically superior to those of the centralized server, by roughly an order of magnitude or more in all cases. Moreover, the trends suggest that as the number of nodes increases past our ability to test, the performance difference for updates might become even more pronounced since, e.g., the update throughput is trending downward for the centralized server but upward for self-service. The performance improvement for updates that self-service yielded was the result of harnessing client cycles to contribute to the service computation. At the same time, the read performance of each implementation was comparable.

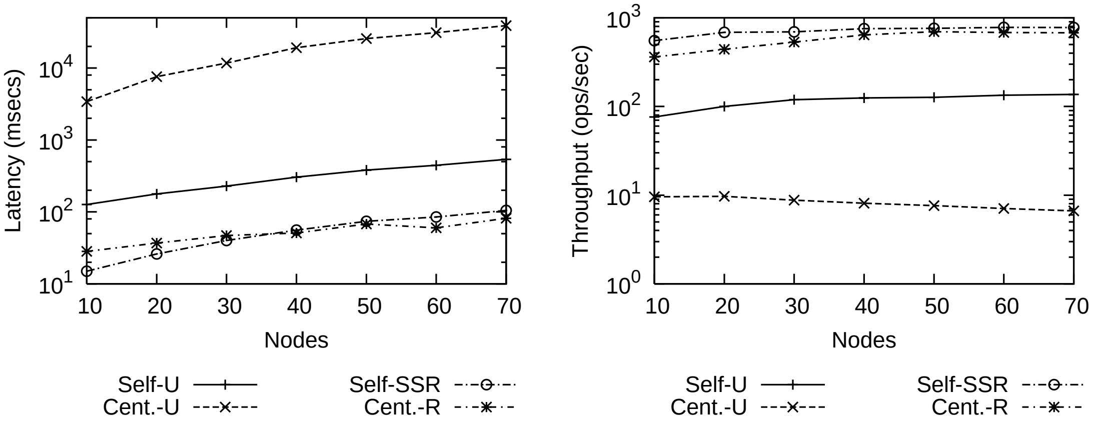


Figure 13: Building traffic models: Performance for self-service updates (Self-U) and strictly serializable reads (Self-SSR), and centralized updates (Cent.-U) and reads (Cent.-R). $\mathsf{objects} = 50$. $\mathsf{objsPerRd} = \mathsf{objsPerUp} = 1$.

Static tree, multi-object operations For the types of traffic models we envision, we expect single-object operations to be the norm in this application. However, multi-object updates could naturally arise, e.g., to incorporate the same traffic records into distinct but related models (e.g., a model for BitTorrent and a model for all file sharing protocols in aggregate). Thus, in our experiments, a multi-object update incorporates the same data into multiple models. Our experiments with multi-object operations are illustrated in Figure 14. As $\mathsf{objsPerUp}$ increased, the greater contention for retrieving objects impacted the self-service update throughput. Nevertheless, self-service still achieved much better update throughput and latency than the centralized server; the centralized server's processor was the bottleneck in this case. On the other hand, the centralized server's multi-object reads outperformed self-service multi-object reads, since in the multi-object case, self-service implements reads using the same protocol as updates. So, these reads contend with the updates for retrieving the relevant objects.

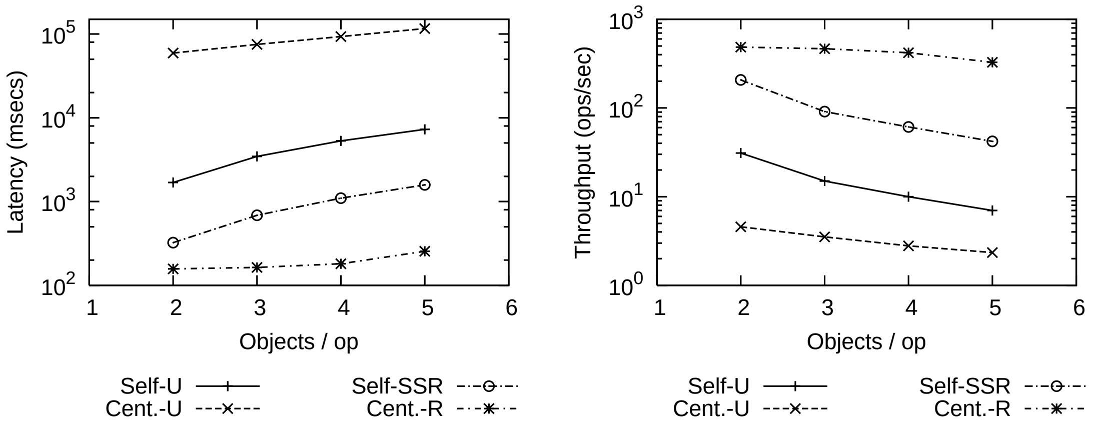


Figure 14: Building traffic models: Performance for self-service updates (Self-U) and strictly serializable reads (Self-SSR), and centralized updates (Cent.-U) and reads (Cent.-R). $\mathsf{nodes} = 70$. $\mathsf{objects} = 50$.

9 Conclusion

We presented a self-service approach to implementing highly scalable services without the need to add to server resources and while providing strong consistency semantics. Our approach is well-suited to services where state can be decomposed into small objects that are typically accessed individually, and where operation processing is compute intensive. Our algorithms allow objects to be migrated to clients so clients can perform their own operations, enabling the service to scale gracefully and in the process, preserving clients' privacy. Update operations are serialized while efficient single-object read operations are supported. Clients may join, leave or disconnect from the service. In case of disconnects, the service recovers objects whose latest versions are left unreachable. Clients performing their own operations helps to preserve clients' privacy, a significant factor for some applications. We evaluated self-service through microbenchmarks and a prototype network traffic classification service built using self-service.

References

[1] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, Dec. 2004.

[2] M. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, and D. Watson. Data reduction for the scalable automated analysis of distributed darknet traffic. In *Proceedings of the Internet Measurement Conference*, Oct. 2005.

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

[4] K. Birman, T. Joseph, T. Raeuchle, and A. Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):502–508, 1985.

[5] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary–backup approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison-Wesley, second edition, 1993.

[6] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. *Advances in Neural Information Processing Systems*, 13, 2001.

[7] C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symp. on Operating Syst. Principles*, 2001.

[9] M. J. Demmer and M. P. Herlihy. The Arrow distributed directory protocol. In *Proc. 12th Intl. Symposium of Distributed Computing*, pages 119–133, 1998.

[10] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symp. on Operating Syst. Principles*, 2001.

[11] E. Eskin, A. Arnold, M. Preraua, L. Portnoy, and S. J. Stolfo. A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. *Applications of Data Mining in Computer Security*, 2002.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. 16th ACM Symp. on Operating Systems Principles*, 1997.

[13] G. Fung and O. L. Mangasarian. Incremental support vector machine classification. In *Proc. 2nd SIAM Intl. Conference on Data Mining*, pages 247–260, 2002.

[14] A. Gray and M. Haahr. Personalised, collaborative spam filtering. Technical Report TCD-CS-2004-36, Computer Science Department, The University of Dublin, Trinity College, Aug. 2004.

[15] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 127–133, Aug. 2001.

[16] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[17] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the 29th Conference on Very Large Databases*, Sept. 2003.

[18] X. Jiang and D. Xu. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

[19] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel traffic classification in the dark. *ACM SIGCOMM Computer Communication Review*, 35(4):229–240, 2005.

[20] P. Knezevic, A. Wombacher, and T. Risse. Highly available DHTs: Keeping data consistency after updates. In *Proc. 4th Intl. Wkshp on Agents and Peer-to-Peer Computing*, 2005.

[21] F. Kuhn and R. Wattenhofer. Dynamic analysis of the arrow distributed protocol. In *Proc. 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 294–301, 2004.

[22] W. Lee, S. J. Stolfo, and K. W. Mok. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, pages 120–132, 1999.

[23] K. Liu, H. Kargupta, K. Bhaduri, and J. Ryan. Distributed data mining bibliography. Available at `http://www.cs.umbc.edu/~hillol/DDMBIB/` as of April 2006.

[24] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in distributed hash tables. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 295–305, 2002.

[25] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of 1st Intl. Wkshp on Peer-to-Peer Systems (IPTPS)*, 2002.

[26] A. Moore and D. Zuev. Internet traffic classification using Bayesian analysis techniques. In *Proceedings of ACM SIGMETRICS '05*, June 2005.

[27] S. Mukkamala and A. H. Sung. Identifying significant features for network forensic analysis using artificial intelligent techniques. *Intl. Journal of Digital Evidence*, 1(4):1–17, 2003.

[28] A. Muthitacharoen, R. Morris, T. Gil, , and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[29] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.

[30] A. G. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Scalable consistency maintenance in content distribution networks using cooperative leases. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):813–828, 2003.

[31] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *Conference on Innovative Data Systems Research*, 2005.

[32] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[33] S. Pearson. *Trusted Computing Platforms: TCPA Technology in Context*. HP Professional Series. Prentice Hall, first edition, 2002.

[34] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. Networked Syst. Design and Implementation*, 2004.

[35] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, 2001.

[36] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, Feb. 1989.

[37] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the Oceanstore prototype. In *Proc. 2nd USENIX Conference on File and Storage Technologies*, 2003.

[38] B. Richard, D. M. Nioclais, and D. Chalon. Clique: A transparent, peer-to-peer collaborative file sharing system. Technical Report HPL-2002-307, HP Labs, 2002.

[39] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Intl. Conference on Distributed Systems Platforms (Middleware)*, 2001.

[40] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. In *Proc. 17th ACM Symp. on Operating Systems Principles*, 1999.

[41] Y. Saito and C. Karamanolis. The Pangaea symbiotic wide-area file system. In *Proc. 10th ACM-SIGOPS European Wkshp*, 2002.

[42] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, 2001.

[43] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the DOMINO overlay system. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2004.

[44] H. Yu and A. Vahdat. Consistent and automatic replica regeneration. In *Proc. Networked Syst. Design and Implementation*, 2004.

[45] S. Zanero and S. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *Proc. ACM Symposium on Applied Computing*, 2004.

[46] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[47] F. Zhou, L. Zhuang, B. Y. Zhao, L. Huang, A. D. Joseph, and J. D. Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, June 2003.