A Self-Service Approach to Scalable Service Deployment

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

January 2006
CMU-CS-06-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 that can be decomposed into service objects that are typically accessed individually. Scalability is achieved by migrating 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 wide-area 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 concurrency semantics Operations, both on single objects and multiple objects, are implemented with strong concurrency semantics. Specifically, our approach ensures serializability [44] for operations. 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, object migration is required only for certain types of operations; most importantly a read operation on a single object does not require object migration.

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 have recently been applied to the object may be lost. However, the connected component of the tree containing the server[1] can efficiently regenerate the last version of the object seen in that component when such a disconnection is detected. In this way the server never loses control of objects comprising the service, 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 asking organizations to turn over this information is a significant social barrier. Organizations may be more willing to integrate their data into the classifier locally, since the resulting classifier typically reveals less about its input than the raw data does.

Our approach is not intended for services with highly transient client populations, e.g., as many web services might be, since this client churn might make tree formation and management a significant cost. Moreover, our approach is vulnerable to malicious clients, since if an object is migrated to a malicious client, the client could corrupt it. Consequently, our system is more suited to serving trusted clients or, with

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

the emergence of *trusted platforms* that permit the remote validation of a software platform [32, 41], only clients that are running valid client software.

We report microbenchmarks for our system recorded on PlanetLab [7], though the microbenchmark application involving only trivial operations is not well-suited to self-service. We therefore also report an empirical evaluation of our approach using the aforementioned application involving the distributed construction of network traffic classifiers. Our experiments demonstrate that self-service is compelling for this type of application, e.g., yielding a full order of magnitude improvement in both operation latency and throughput over a centralized implementation in an experiment involving 75 nodes. In addition, we reiterate that our approach better protects client privacy for this application. We are also currently investigating applications of self-service in other domains, including massively multi-player online games (e.g., [21]), large scale scientific analysis (e.g., [29]) and distributed model construction for weather monitoring systems (e.g., [43]).

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 the most relevant prior work of which we are aware that focuses on that pair. We show that 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 [35, 10]. These protocols allow nodes arranged in a tree to locate and retrieve shared objects and perform operations atomically. Another example in this space [24] presents 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 an approach that extends 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 [35, 10].

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 [39, 14] and dynamic web content distribution networks [31, 30]—yield simple consistency protocols. But their "incremental scalability" [14] 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., [36, 33, 45])—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, 34, 38, 47, 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., [9, 12]) 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., [37, 40]) or guarantee consistency for write operations (that 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., [18]) have sacrificed atomicity. Self-service is an alternative that better supports strong consistency and compute-intensive updates.

We additionally comment that client operation privacy is a feature of self-service that, to our knowledge, is not addressed in the classes of systems discussed above.

3 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. In order to interact with the service, a client must *join* the service; in doing so, it is positioned within a tree rooted at the server. A client can also voluntarily *leave* the tree.

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 reinitialized state. In an execution, a client that joins but does not disconnect (though it might leave voluntarily) is called a *connected* client.

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. We assume that any operation that produces an object instance o as output takes as input the previous version of the object, i.e., an object instance o' such that $o'.\mathsf{id} = o.\mathsf{id}$ and $o'.\mathsf{version} + 1 = o.\mathsf{version}$.

It is convenient for discussion that each object instance and each operation be unique—e.g., the implementation could insert a "nonce" in each object instance and operation description. However, we emphasize that this is only to simplify discussion and is not required in the system.

Our system ensures that operations are applied consistently. In particular, for any system execution, there is a set of operations $\mathsf{Durable}$ that includes all operations performed by connected processes (and possibly some operations by clients that disconnect), such that the connected processes perceive the operations in $\mathsf{Durable}$ (and no others) to be executed sequentially. More precisely, our system enforces *serializability* [44]: All connected processes perceive the operations in $\mathsf{Durable}$ to be executed in the same sequential order.

In addition to serializability, our algorithms ensure another important property (though we do not prove it here), termed *coherence* [16]. Informally, coherence guarantees that all connected processes perceive updates to each object individually, in the same sequential order. An important property of coherence is that two distinct update operations involving the same object performed by the same process are perceived by all processes to be in the order in which they were performed, i.e., coherence preserves local process order.

4 Self-Service operations

In this section we describe how update operations (those that produce a new object instance), multi-object operations (those that take multiple objects as input) and single-object read operations are performed in our system. 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 4.3 describes how this implementation enables update and multi-object operations, and the optimizations for single-object read operations.

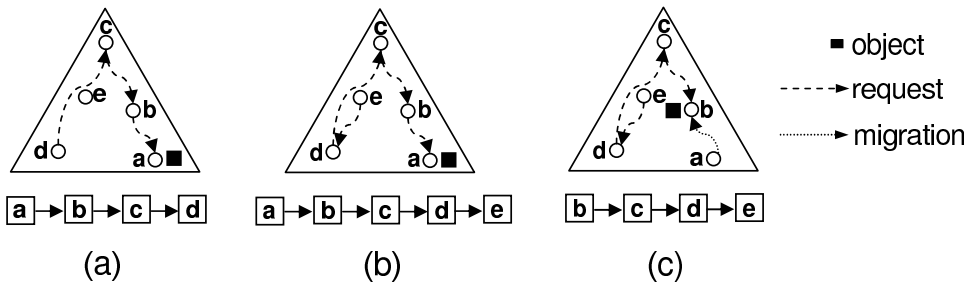

Figure 1: (a) distQ consists of processes a, b, c and d. (b) Process e adds itself to the end of distQ by sending a retrieve request to d. (c) When a completes its operation on the object, it migrates the object to b and drops off distQ.

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.

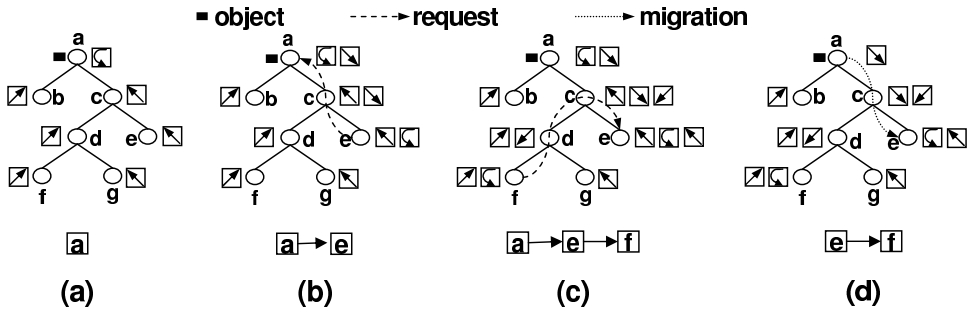

Figure 2: Squares at each process represent its localQ; left-most square is the head and right-most is the tail. The arrow in each square denotes the neighbor to which it points. Initially a has the object. e requests from a and f requests from e. Finally a migrates the object to e.

4.2 distQ implementation

The core of our algorithm deals with the implementation of distQ per object. 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 $p.\mathsf{localQ}[id]$ and adds q to the end of $p.\mathsf{localQ}[id]$ as the new tail. This ensures that the tail of $p.\mathsf{localQ}[id]$ now points in the direction of the new tail of $\mathsf{distQ}[id]$, 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 $p.\mathsf{localQ}[id]$ and forwards the object to the new head of $p.\mathsf{localQ}[id]$. This ensures that the head of $p.\mathsf{localQ}[id]$ points in the direction of the new head of $\mathsf{distQ}[id]$, 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., $p.\mathsf{parent}$, $p.\mathsf{children}$, $p.\mathsf{localQ}[\,]$ and $p.\mathsf{objs}[\,]$. 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[2] $p.\mathsf{sem}[id]$ 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.

4.2.1 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 discussion of localQ so far; it will be discussed in Section 4.3.) 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 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 doRetrieveRequest$(p, id, \perp)$ 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$(from, id, prog)$ not only when it receives a retrieve request from another process (line 20), but also to retrieve the object with identifier id for itself.

Migrating an object with identifier id is handled by the doMigrate function shown in Figure 3. Since the head of $p.\mathsf{localQ}[id]$ is supposed to point toward the current location of the object with identifier id, 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 doRetrieveRequest$(p, id, prog)$; again,

[2]To remind the reader, a semaphore s is a concurrency control primitive that represents a non-negative integer counter 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 [11].

```
doRetrieveRequest(from, id, prog)                 /* Invoked locally on request by from (from could be p). Run prog on completion */
1.  ⟨q, prog'⟩ ← p.localQ[id].tail                /* q made the last request for this object */
2.  p.localQ[id].addToTail(⟨from, prog⟩)          /* 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.  ⟨q, prog⟩ ← 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 ← {id' : id ⟹^{p.Deps} id'}          /* ... then find which other objects this object depends on */
14.     Objs ← {p.objs[id'] : id' ∈ IDs}          /* ... collect all these objects */
15.     DepSet ← p.Deps ∩ (IDs × 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 ← p.Deps \ 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], ∅, ∅) 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, ⊥)                /* Invoke doRetrieveRequest on from's behalf with an empty program */

Upon receiving (migrate : o, Objs, DepSet)        /* o being migrated that depends on copied objects Objs with relation DepSet */
21. p.objs[o.id] ← o                              /* Save the migrated object o */
22. foreach o' ∈ Objs                             /* For each one of the copied objects ... */
23.     p.objs[o'.id] ← o'                        /* ... save the copied object */
24. p.Deps ← p.Deps ∪ 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

we defer discussion of $prog$ to Section 4.3. 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 4.2.2.

4.2.2 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.parent only if all object instances on which o depends are already recorded at p.parent. Otherwise, p.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.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 $id \in IDs$ and $p.\mathsf{objs}[id] \Rightarrow o'$, then add $o'.\mathsf{id}$ to IDs. p then copies $\{p.\mathsf{objs}[id]\}_{id \in IDs}$ 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 $p.\mathsf{objs}[id] \stackrel{op}{\Rightarrow} p.\mathsf{objs}[id']$, then p adds (id, id') to $p.\mathsf{Deps}$. (This will be further detailed in Section 4.3.) 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 $\stackrel{p.\mathsf{Deps}}{\Longrightarrow}$ in line 13 of Figure 3—and copies both $Objs = \{p.\mathsf{objs}[id']\}_{id' \in IDs}$ (line 14) and $DepSet = p.\mathsf{Deps} \cap (IDs \times IDs)$ (line 15) along with the migrating object (line 16). Finally, p updates $p.\mathsf{Deps} \leftarrow p.\mathsf{Deps} \setminus DepSet$ (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 $p.\mathsf{Deps} \leftarrow p.\mathsf{Deps} \cup DepSet$ (line 24).

4.3 Operation implementation

In order to achieve our desired concurrency semantics, for each object we enforce sequential execution of all update and multi-object operations (Section 4.3.1) 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 4.3.2) that dominate many workloads.

4.3.1 Update and multi-object operations

We first discuss how we use the mechanisms from Section 4.2 to invoke update and multi-object operations, and then discuss operation durability.

Invoking operations Consider an update or multi-object operation op, and let $id_1, \ldots, id_k$ denote distinct identifiers of the objects involved (read or updated) in the operation. In order 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:

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

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 4.2.2) with the new dependencies introduced by op. Finally, $prog_0$ executes V operations on the semaphores for each of the objects, permitting them to migrate to other processes. Viewing the semaphores $p.\mathsf{sem}[id_1]$, …, $p.\mathsf{sem}[id_k]$ as locks and the V operations as releasing the locks, $prog_k$ can be viewed as implementing strict two-phase locking [2]. 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 4.2.2) 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 as stated require only that a process forces its updates to be durable when it leaves the tree (Section 5.3), so that operations by a process that remains connected until it leaves are durable.

4.3.2 Single-object read operations

Due to the serial execution of update and multi-object operations, as described in Section 4.3.1, single-object reads so as to achieve serializability [44] for all operations can be implemented with local reads. That is, to perform an operation that involves reading a single object with identifier id, process p simply returns $p.\mathsf{objs}[id]$ immediately.

5 Tree management

To this point we have deferred discussion of joins, leaves and disconnections. Here we briefly describe the join protocol we have implemented, and then detail how to adapt our algorithm to address disconnections (Section 5.2) and processes leaving voluntarily (Section 5.3).

5.1 Joins

In order to join the service, a client contacts the root of the tree—the server. 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.

This simple mechanism enables the server to perform any access control mechanisms desired by the service. In addition, different application-specific tree construction policies can be implemented. In our implementation, processes construct a k-ary balanced tree. Examples of other policies include clustering clients in subtrees based on their geographical location, network proximity or shared interest in certain service objects.

5.2 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 4.

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 updates by now-disconnected clients that are lost. p is the best candidate since among the still-connected processes, p is the last to have saved the object before it was migrated to the disconnected subtree (line 21, Figure 3). Note that in case of multiple

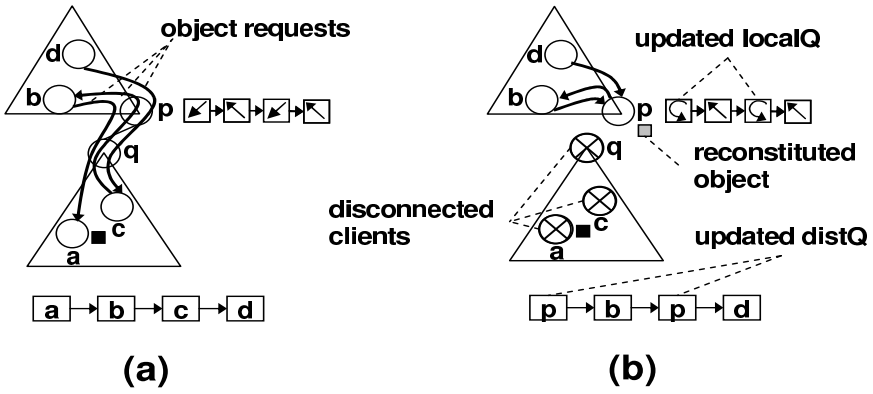

Figure 4: q loses contact with parent p and its subtree disconnects. p replaces disconnected clients in distQ and reconstitutes the object so b and d can make progress.

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 distQ$[id]$.

The pseudocode that p executes when its child q disconnects is the childDisconnected(q) routine in Figure 5. Specifically, p replaces all instances of q in p.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 4-(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 5) to initiate the migration of p.objs$[id]$ to the neighbor following (this instance of) p in p.localQ$[id]$, once p has the object. If the disconnected child was at the head of p.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).

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

Qreplace(id, q)                                                    /* Invoked locally by p */
7.  foreach i = 1, ..., p.localQ[id].size − 1                      /* For each element of localQ, except the last */
8.      if p.localQ[id].elmt[i] = ⟨q, ∗⟩                           /* If the element points to q ("∗" is wild-card)... */
9.          p.localQ[id].elmt[i] ← ⟨p, ||V(p.sem[id])||⟩           /* ...change it to point to myself */
10.         t ← new thread(||P(p.sem[id]); doMigrate(id)||)        /* ...create a thread that waits for object and then migrates it */
11.         t.enable()                                             /* ...run-enable the thread */
12. if p.localQ[id].tail = ⟨q, ∗⟩                                  /* If the last element is the disconnected child... */
13.     p.localQ[id].tail ← ⟨p, ||V(p.sem[id])||⟩                  /* ...then replace it by myself; no need to start thread here */
```

Figure 5: Disconnection-handling at process p

5.3 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 read and retrieve requests forwarded through p must not block as a result of p leaving the tree.

If p is a leaf node, then it simply 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 4.3.1), and departs. If p is an internal node then it forces its updates to be durable, and then arbitrarily chooses one of its children q and *promotes* q. The promotion includes updating q's state according to the state at p, and updating 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[ ] and p.Deps. When q receives this message it executes the pseudocode shown in Figure 6. The steps performed by q to assume p's role are mostly straightforward and include updating its objects (lines 2–3), parent (line 11), children (line 12), and object dependencies (line 13).

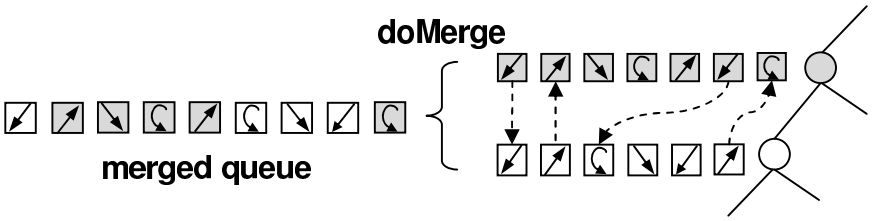

Figure 6: Queue merge. Shaded and unshaded elements are parent's and child's neighbors (and self pointers), respectively. Dashed arrows are from a skipped element to the element in the other queue added next. Elements between curved arrows are added to $mergedQ$ in order.

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 6. To merge p.localQ$[id]$ and q.localQ$[id]$, q begins with q.localQ$[id]$ if its head points to p and p.localQ$[id]$ otherwise. q adds elements from the chosen queue, say p.localQ$[id]$, to a newly created $mergedQ$ until an instance of q is reached (line 19 of Figure 7), say at the i^{th} index, i.e., p.localQ$[id]$.elmt$[i] = q$. The merge algorithm then skips this i^{th} element and begins to add elements from q.localQ$[id]$ until an instance of p is found. This element is skipped and the algorithm switches back to p.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 Qreplace(id, p) defined in Figure 5.

This merging and subsequent Qreplace has the following effects: Any outstanding retrieve requests that were initiated by p (represented by instances of p in p.localQ$[id]$) now appear as initiated by q since all instances of p from p.localQ$[id]$ are copied to $mergedQ$ and then replaced by q. Retrieve requests that were forwarded through p but not q now appear as forwarded through q as all elements in p.localQ$[id]$ are added to $mergedQ$, except instances of q. Retrieve requests that were forwarded through q and not p appear as before since q.localQ$[id]$ elements are all added to $mergedQ$, except instances of p. Finally, retrieve requests that were forwarded through both p and q now appear as forwarded through only q, due to skipping elements in p.localQ$[id]$ that point to q and vice-versa.

6 Correctness

Definition 1 (reads-from, $\rightarrow_{\mathrm{rf}}$, $\rightarrow^*_{\mathrm{rf}}$)**.** *An operation op* reads from *op′, denoted op′* $\rightarrow_{\mathrm{rf}}$ *op, if op inputs an object instance produced by op′.* $\rightarrow^*_{\mathrm{rf}}$ *denotes the transitive closure of* $\rightarrow_{\mathrm{rf}}$.

```
Upon receiving (promote :gParent, siblings, parentQ[ ],
                        parentObjs[ ], parentDeps)   /* Message received by q from q.parent that is voluntarily leaving */
1.  foreach id                                       /* For each object...*/
2.      if parentObjs[id].version > q.objs[id].version   /* If the parent's version is newer than my version... */
3.          q.objs[id] ← parentObjs[id]              /* ...then replace my instance with parent's instance */
4.      mergedQ[id] ← ∅                              /* Start with a fresh mergedQ */
5.      if q.localQ[id].head = ⟨q.parent, ∗⟩         /* If the head of my localQ points to my parent... */
6.          doMerge(q.localQ[id], parentQ[id],       /* ...then start the merge operation with my localQ */
                    q, q.parent, mergedQ[id])
7.      else                                         /* If the head of my localQ does not point to my parent... */
8.          doMerge(parentQ[id], q.localQ[id],       /* ...then start the merge operation with parent's localQ */
                    q.parent, q, mergedQ[id])
9.      q.localQ[id] ← mergedQ[id]                   /* Set localQ to the newly created mergedQ */
10.     Qreplace(id, q.parent)                       /* Run Qreplace on the new localQ to replace parent with myself */
11. q.parent ← gParent                               /* The old grand-parent is now my parent */
12. q.children ← (q.children ∪ siblings) \ {q}       /* Old siblings are now my children */
13. q.Deps ← q.Deps ∪ 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.     ⟨r, prog⟩ ← localQ.removeFromHead()          /* ...then remove its head */
16.     if r ≠ p′                                    /* If the head does not point to the other process... */
17.         mergedQ.addToTail(⟨r, prog⟩)             /* ...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 7: Pseudocode run at q for its promotion

Lemma 1. *Let op_1 and op_2 denote distinct operations that output object instances o_1 and o_2, respectively, where o_1.id $= o_2$.id and o_1.version $= o_2$.version. Then there are no operations op_3 and op_4 (distinct or not) performed by connected processes such that $op_1 \rightarrow^*_{\text{rf}} op_3$ and $op_2 \rightarrow^*_{\text{rf}} op_4$.*

Proof sketch: Among the connected processes, the localQ.tail pointers implement the Arrow protocol by Demmer and Herlihy [10] (augmented to account for disconnections as described in Section 5.2). 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., [10, 17, 22]. 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_1 and o_2 with the same object identifier and version number implies that at least one of op_1 and op_2, say op_1, was performed by a client that disconnects. Moreover, the process that performs op_1 must disconnect prior to migrating o_1 (or having it copied due to the migration of an object instance that depends on o_1) out of the subtree that disconnects. Otherwise, the lowest connected ancestor in the tree, who reconstitutes the object following the disconnection, would reconstitute o_1 or a later version (see Section 5.2). So, o_1 is never visible in the connected component containing the server. This also implies that for each object instance o such that $o \Rightarrow o_1$, o is not visible in the connected component: if o is migrated (or copied) up to the connected component, then o_1 (or a later version) must be copied along with it (see Section 4.2.2). Therefore, none of the other object instances produced by op_1 are visible in the connected component, as each of these instances depends on o_1. As a consequence, none of the instances produced by op_1 is ever read by a connected process and so $op_1 \not\rightarrow^*_{\text{rf}} op_3$. □

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 (Durable)**.** *The set* Durable *is defined inductively to include operations according to the following two rules (and no other operations):*

1. *If op was executed at a connected process, then $op \in$* Durable*.*
2. *If $op \in$* Durable *and $op' \rightarrow^*_{\text{rf}} op$, then $op' \in$* Durable*.*

Below we prove that the operations in Durable are serializable, and as such they are durable (since "losing" an update could violate 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 ($\rightarrow_{\text{ver}}$)**.** *The version precedence relation, denoted $\rightarrow_{\text{ver}}$, is defined for operations as follows: For distinct operations op_i, op_j and op_k, let op_k read an object instance o produced by op_j and op_i produce an object instance o' such that $o.\mathsf{id} = o'.\mathsf{id}$. If $o'.\mathsf{version} < o.\mathsf{version}$ then $op_i \rightarrow_{\text{ver}} op_j$, otherwise $op_k \rightarrow_{\text{ver}} op_i$.*

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 $op_i \rightarrow_{\text{rf}} op_j$ or $op_i \rightarrow_{\text{ver}} op_j$ 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 Durable$'$ denote the subset of 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 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 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_1 and op_2, if $op_1 \rightarrow_{\text{rf}} op_2$ then $ts(op_1) < ts(op_2)$. 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 $ID_{in}(op)$ denote the set of identifiers of object instances input to an operation op, i.e., $ID_{in}(op) = \{o.\mathsf{id} : o \in input(op)\}$.

Lemma 2. *Let op_i and op_j be distinct update or multi-object read operations in* Durable′ *performed by processes p_i and p_j, respectively, such that $ID_{in}(op_i) \bigcap ID_{in}(op_j) \neq \emptyset$. If for some $id \in ID_{in}(op_i) \bigcap ID_{in}(op_j)$, p_i retrieves an instance o with $o.\mathsf{id} = id$ before p_j retrieves an instance o' with $o'.\mathsf{id} = id$, 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' 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 4.3.1) and therefore, only after assigning $ts\_recent(o'') \leftarrow ts(op_i)$, where o'' is either o or its newer version in case op_i updates o. 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* Durable′*, such that $op_i \rightarrow_{\mathrm{rf}} op_j$. Then $ts(op_i) < ts(op_j)$.*

Proof. Let o 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 only after p_i performs a $V(p_i.\mathsf{sem}[o.\mathsf{id}])$, which is done after op_i completes. Therefore, p_i must complete the retrieval of an instance with identifier $o.\mathsf{id}$ (the instance input to op_i) before p_j retrieves o 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* Durable′*, such that $op_i \rightarrow_{\mathrm{ver}} op_j$. Then $ts(op_i) < ts(op_j)$.*

Proof. $op_i \rightarrow_{\mathrm{ver}} op_j$ 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 produced by op_j, op_i produces instance o' and $o'.\mathsf{id} = o.\mathsf{id}$, $o'.\mathsf{version} < o.\mathsf{version}$. Since instances with the same identifier are retrieved serially and form a unique sequence ordered by version number (Lemma 1), p_i must retrieve instance with identifier $o'.\mathsf{id}$ (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 produced by op_k, op_j produces instance o' and $o'.\mathsf{id} = o.\mathsf{id}$, $o'.\mathsf{version} > o.\mathsf{version}$. Since op_i inputs a version earlier than the one produced by op_j and object instances are retrieved serially and have a unique sequence ordered by version number (Lemma 1), p_i must have retrieved o before p_j retrieved an instance with identifier $o'.\mathsf{id}$ 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 (and the corresponding edges) from* 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 in order, into $\mathsf{Durable}'$. 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 read by op_i, and a version precedence edge from op_i to each update operation op_j that produces an instance o' with $o'.\mathsf{id} = o.\mathsf{id}$ and $o'.\mathsf{version} > o.\mathsf{version}$.

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 $op_k \rightarrow_{\mathrm{rf}}^{*} op_j$: 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) in $MVSG(\mathsf{Durable}')$ even before adding op_i, a contradiction due to Theorem 1. □

Theorem 2. 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 Evaluation

We evaluated the performance of our self-service system in two settings. First, we conducted experiments on PlanetLab [7] (Section 7.2) to measure the throughput and latency 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 (Section 7.3) 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, and has characteristics that are better suited to self-service. In this evaluation the self-service approach dramatically outperformed a centralized implementation. This service involves computationally intensive operations, and as such it was not feasible to run these experiments on PlanetLab due to the small CPU resources we were allocated: during the time we used PlanetLab, we saw an average of 7-10% CPU available for our slice on most PlanetLab nodes. Therefore, we performed these experiments on an isolated 75-node cluster.

7.1 Experimental system

Our self-service system is implemented in Java JDK 5.0 and consists of less than 1000 lines of code. The following choices influenced system performance:

Object compression Large objects (objects in the application discussed in Section 7.3 can be a few megabytes) are stored and transmitted in compressed form. The memory and bandwidth savings due to compression far outweigh the small computation costs. We use the LZO compression library[3], which is invoked from Java through the Java Native Interface.

Deep copy reads A node makes a local, deep 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

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

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 overall 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 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 ten 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.

Our experiments were characterized by certain parameters. Each experiment was conducted on n_{nodes} nodes, including all the clients and the server, arranged in a k-ary tree. Operations were performed on a subset of the total n_{objects} shared objects. Each node performed n_{updates} 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. Nodes kept n_{plReads} read and $n_{\mathsf{plUpdates}}$ update operations outstanding at a time, i.e., after joining the tree each node initiated n_{plReads} read and $n_{\mathsf{plUpdates}}$ update operations each, in parallel. Subsequent completion of an operation resulted in a new operation of the same type (read/update), until n_{updates} update operations had been completed.

To evaluate multi-object updates we sampled the number of objects from a Gaussian distribution with mean $\mu_{\mathsf{objsPerUp}}$ before each operation. Samples less than one were rounded to one and real numbers greater than one were rounded to the closest integers. Reads in each experiment operated on single objects, since multi-object reads are performed using the same protocol as updates. For each update and read, a node sampled the needed number of objects uniformly at random, without replacement.

7.2 Microbenchmarks on PlanetLab

Experiments conducted on PlanetLab used Internet2, CAnet and university machines in the US and Canada.

Our first experiment used single-object operations with $n_{\mathsf{updates}} = 150$ and $n_{\mathsf{objects}} = 100$, while varying $n_{\mathsf{nodes}} \in \{15, 25, 35, 45, 55\}$. We also used $k = 5$, which is a smaller degree than we would suggest in practice, but we chose this so that the tree would gain depth as n_{nodes} was increased. Figure 8 shows the results for updates with $n_{\mathsf{plUpdates}} \in \{5, 10\}$ and for reads with $n_{\mathsf{plReads}} = 30$, to mimic a read-intensive workload. The self-service throughput increased with an increase in the number of nodes, e.g., the update throughput for $n_{\mathsf{plUpdates}} = 10$ increased from 99.25 updates/second with 15 nodes to 242.38 updates/second with 55 nodes. Latency also increased as expected. The sharper latency increase between $n_{\mathsf{nodes}} = 25$ and $n_{\mathsf{nodes}} = 35$ results from the growth in the maximum tree depth in a tree of degree $k = 5$. Reads are local and performed very well.

Figure 9 shows microbenchmarks for multi-object operations. We used $n_{\mathsf{nodes}} = 45$, $k = 5$, $n_{\mathsf{updates}} = 150$, $n_{\mathsf{objects}} = 100$, $n_{\mathsf{plReads}} = n_{\mathsf{plUpdates}} = 5$ and varied $\mu_{\mathsf{objsPerUp}} \in \{1, 2, 3, 4, 5\}$. Note that when $\mu_{\mathsf{objsPerUp}} = 1$, the number of objects per update could be one or more than one (as sampled) and so this case is not the same as the single-object case. The 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 4.3.1); 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.

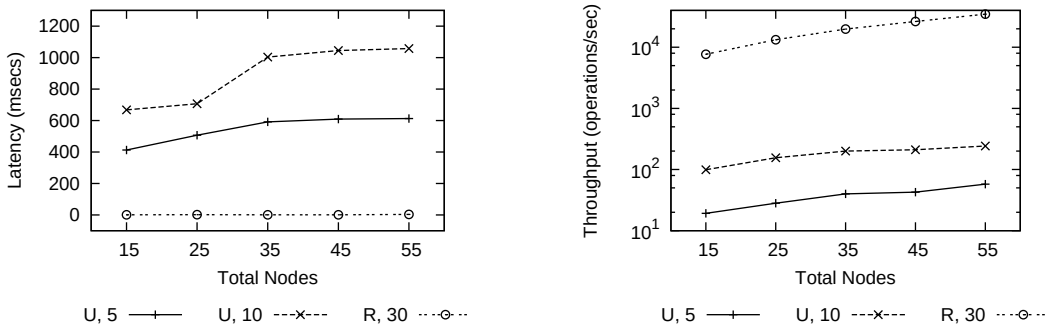

Figure 8: Microbenchmarks on PlanetLab. Mean latency/operation and throughput for updates (U) with $n_{\mathsf{plUpdates}} \in \{5, 10\}$ and for reads (R) with $n_{\mathsf{plReads}} = 30$.

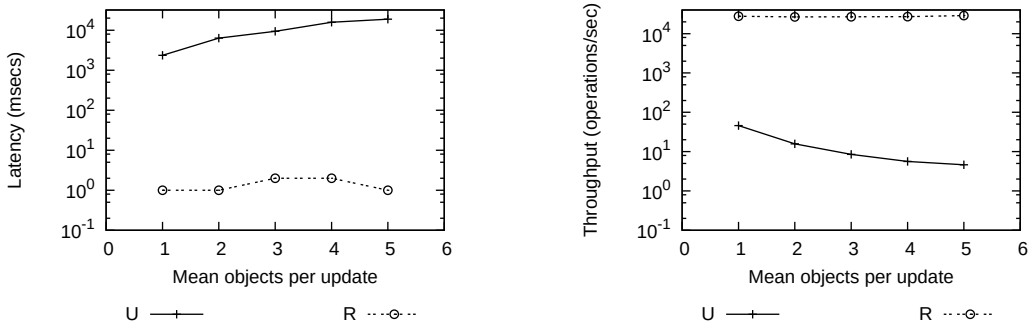

Figure 9: Microbenchmarks on PlanetLab with 45 nodes. Latencies and throughputs for updates (U) and reads (R) against mean objects per update.

7.3 Network traffic classification service

As discussed previously, the microbenchmarks of Section 7.2 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, and that may be of independent interest. We briefly motivate and describe this service 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., see [23, 46]). Our thesis is that models for performing this classification can be built more effectively by aggregating contributions from many networks, rather than each administrator constructing classifiers based on his own network's traffic.

We are thus building a service through which networks can contribute labeled traffic records toward the construction of classifiers for network traffic. In this application the server is run by some coordination center (e.g., CERT/CC), the clients are various networks that contribute records and the shared objects are the classifiers. We envision a number of such classifiers, parameterized by application (HTTP, BitTorrent, etc.), transport (TCP, UDP, etc.), and/or attack attributes ("attack" vs. "normal"). The classifiers that our service supports are Support Vector Machines (SVMs) [8], a popular learning mechanism used for classification and regression and that are particularly well-suited to data with many features. More specifically, we use a variant of traditional SVMs called incremental SVMs [15, 6] that allow the models to be constructed incrementally as new contributions are received. SVMs have previously been used to characterize

network traffic [13, 27], though not in a distributed setting. Our implementation uses the LIBSVM library[4] to construct SVM models from raw data.

7.3.1 Experiment setup

Since we used our service for performance evaluation, we tried to construct SVMs as realistically as possible, and for this purpose we needed network traffic records from which to build these SVMs. We used the KDD Cup 1999 intrusion detection dataset[5] as raw data. This data consists of labeled connection records, each consisting of 41 features related to the connection including the application protocol, the transport protocol, protocol flags, connection length, "attack" vs. "normal", etc. We formatted and scaled this raw data according to LIBSVM's requirements and then divided it among the clients.

We found SVM training on this data to be sufficiently computationally intensive that it was not feasible to perform these experiments on PlanetLab. Instead we used a 75-node rack consisting of Intel Pentium 4 2.8GHz machines, each with 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. In this configuration, one of the 75 machines was used as the server; other nodes acted as clients. Each client performed n_{updates} update operations, each operation updating a classifier with b new records at that client.

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 b 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 deep copies and keeping objects in memory—were preserved in the centralized implementation.

7.3.2 Results

Our first experiment evaluated single-object operations using $n_{\mathsf{nodes}} = 75$, $n_{\mathsf{updates}} = 150$, $n_{\mathsf{objects}} = 50$, and $b = 500$. For the self-service implementation we chose $k = 5$. We varied $n_{\mathsf{plReads}} = n_{\mathsf{plUpdates}} \in \{1, 5, 10, 15, 20\}$. Figure 10 plots the results for all cases. Our experiments showed that self-service throughput was much higher and latency much lower than the centralized case for both updates and reads—e.g., for $n_{\mathsf{plReads}} = n_{\mathsf{plUpdates}} = 20$, the self-service implementation performed 127.48 updates/second at 3.7 secs/update, whereas the centralized server could only manage 5.37 updates/second at 153.37 seconds/update. Update throughput (in either implementation) did not increase due to the compute intensive nature of updates (the CPUs could manage only so much computation regardless of the number of parallel operations being requested by each client). Self-service update latencies increased with more parallel operations since this increased per-object collisions. In the centralized server case, however, even updates to different objects competed with each other for server CPU, yielding far worse performance. The contention at the centralized server also impacted read processing. In contrast, read throughput in the self-service implementation increased with an increase in parallel operations, since (single-object) reads are served locally.

Next we conducted experiments with multi-object operations. In this application, an example of a multi-object operation would be updating two models with the same traffic records, e.g., models for TCP and HTTP traffic. Thus, a multi-object operation updates all the models involved using the same data. Here we used $n_{\mathsf{nodes}} = 75$, $n_{\mathsf{updates}} = 150$, $n_{\mathsf{objects}} = 50$, $n_{\mathsf{plReads}} = n_{\mathsf{plUpdates}} = 5$ and $b = 500$. We varied $\mu_{\mathsf{objsPerUp}} \in \{1, 2, 3, 4, 5\}$. Figure 11 plots the results. This experiment showed that even for

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

[5] `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`

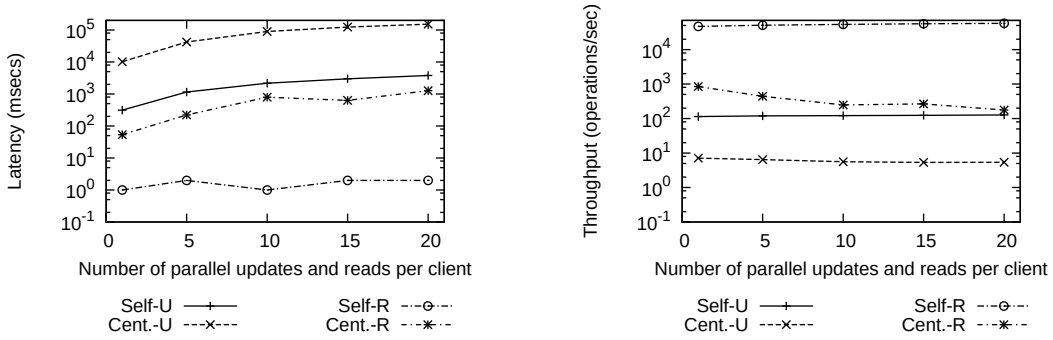

Figure 10: Building traffic models on 75 nodes: Latencies and throughputs for self-service updates (Self-U) and reads (Self-R), and centralized updates (Cent.-U) and reads (Cent.-R) for $n_{\mathsf{plReads}} = n_{\mathsf{plUpdates}} \in \{1, 5, 10, 15, 20\}$.

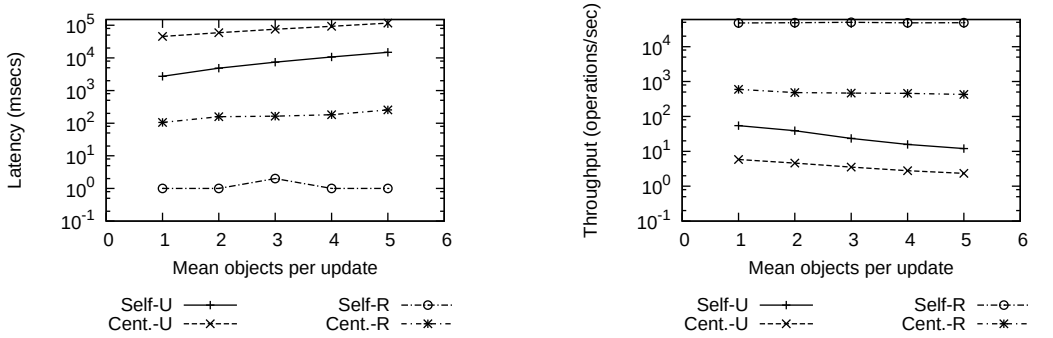

Figure 11: Building traffic models on 75 nodes: Latencies and throughputs for self-service updates (Self-U) and reads (Self-R) and centralized updates (Cent.-U) and reads (Cent.-R) against mean objects per update.

multi-object operations where the self-service implementation needed to sequentially retrieve objects, it still performed much better than the centralized server that had to perform all the processing itself. For example, for $\mu_{\mathsf{objsPerUp}} = 5$ self-service performed 11.99 updates/second at 14.8 seconds/update compared to only 2.34 updates/second at 116.2 seconds/update by the centralized service. Reads in the centralized server case performed much worse than the self-service local reads. However, the centralized server reads did not suffer as much as they did in the previous experiment (Figure 10) since per-operation processing (e.g., connection establishment) at the server was amortized in multi-object operations.

Our final experiment showed the effect of an increasing number of nodes on the latency and throughput of read and update operations; see Figure 12. For this experiment we used single-object operations with $n_{\mathsf{updates}} = 150$, $n_{\mathsf{objects}} = 50$, $n_{\mathsf{plReads}} = n_{\mathsf{plUpdates}} = 5$, and $b = 500$, and varied $n_{\mathsf{nodes}} \in \{10, 20, \ldots, 70\}$. The centralized server's performance (both latency and throughput) suffered drastically as the number of clients increased, whereas the self-service sustained its performance much better. For example as the nodes increased from 10 to 70, the centralized server's throughput decreased from 9.58 to 6.65 updates/second and the latency increased from 3.4 seconds/update to 38.9 seconds/update. In contrast, for the same increase in n_{nodes}, the self-service throughput increased from 82.5 updates/second to 147.67 updates/second without a very large increase in latency (from 0.42 to 1.38 seconds/update). The self-service update throughput did not increase much more since adding nodes also resulted in more per-object collisions.

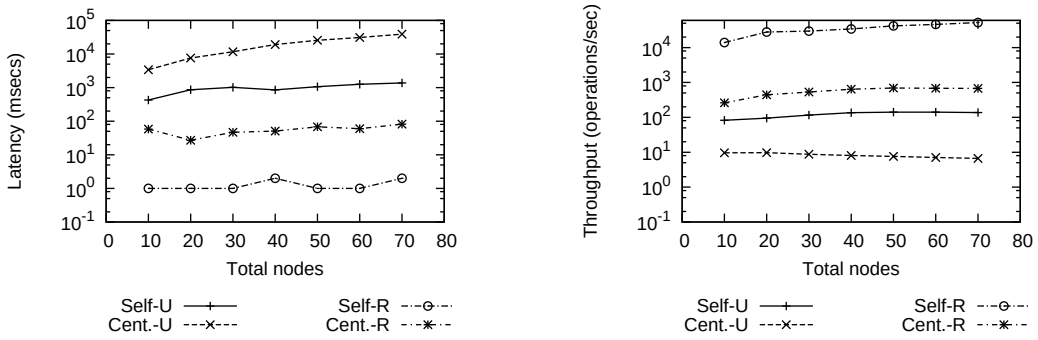

Figure 12: Building traffic models on 75 nodes: Latencies and throughputs for self-service updates (Self-U) and reads (Self-R), and centralized updates (Cent.-U) and reads (Cent.-R) against the number of nodes.

8 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, operation processing is compute intensive and client churn is low. 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. We demonstrated these advantages through the evaluation of a large-scale network traffic classification service built using self-service, which convincingly outperformed a centralized implementation.

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

[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] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

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

[9] 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.

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

[11] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, The Mathematics Department, Technological University, Eindhoven, 1965.

[12] 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.

[13] 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.

[14] 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.

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

[16] J. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989.

[17] 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.

[18] 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.

[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] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *Proceedings of IEEE Infocom*, Mar. 2004.

[22] 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.

[23] 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.

[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] H. Newman. Worldwide distributed analysis and data grids for next-generation physics experiments. In *Proc. 7th Intl. Wkshp on Adv. Comp. and Analysis Techniques in Physics Research*, 2000.

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

[33] 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.

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] S. W. Smith. *Trusted Computing Platforms Design and Applications*. Springer, 2005.

[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] G. Tucker, N. Gasparini, R. Bras, S. Rybarczyk, and S. Lancaster. An object-oriented framework for distributed hydrologic and geomorphic modeling using triangulated irregular networks. In *Proc. 4th Intl. Conference on GeoComputation*, 1999.

[44] M. Yannakakis. Serializability by locking. *Journal of the ACM*, 31(2):227–244, Apr. 1984.

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

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

[47] 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.