Quiver: Consistent Object Sharing for Edge Services

Michael K. Reiter, *Senior Member*, *IEEE Computer Society*, and Asad Samar

Abstract—We present Quiver, a system that coordinates service proxies placed at the "edge" of the Internet to serve distributed clients accessing a service involving mutable objects. Quiver enables these proxies to perform consistent accesses to shared objects by migrating the objects to proxies performing operations on those objects. These migrations dramatically improve performance when operations involving an object exhibit geographic locality, since migrating this object into the vicinity of proxies hosting these operations will benefit all such operations. Other workloads benefit from Quiver, dispersing the computation load across the proxies and saving the costs of sending operation parameters over the wide area when these are large. Quiver also supports optimizations for single-object reads that do not involve migrating the object. We detail the protocols for implementing object operations and for accommodating the addition, involuntary disconnection, and voluntary departure of proxies. We also evaluate Quiver through experiments on PlanetLab. Finally, we discuss the use of Quiver to build an e-commerce application and a distributed network traffic modeling service.

Index Terms—Edge services, migration, serializability.

1 Introduction

Dynamic Web services are examples of Internet-scale applications that utilize mutable objects. Following the success of content distribution networks (CDNs) for static content (see [1] for a survey), numerous recent proposals have attempted to scale dynamic Web services by employing service proxies at the "edge" of the Internet (for example, see [2], [3], and the references therein). This approach has the potential of both distributing the operation processing load among the proxies and enabling clients to access the service by communicating with nearby proxies rather than a potentially distant centralized server.

A major challenge in this architecture, however, is to enable the (globally distributed) service proxies to efficiently access the mutable service objects for servicing client operations while ensuring *strong consistency* semantics for these object accesses. Consistent object sharing among the proxies enables them to export the same consistent view of the service to the clients in turn. However, achieving even just serializability [4], [5] for operations executed at these proxies using standard replication approaches (see Section 2) requires that a proxy involve either a centralized server or other (possibly distant) proxies on the critical path of each update operation. Strict serializability [4] via such techniques requires wide-area interactions for reads as well.

Here, we describe a system called Quiver that demonstrates an alternative approach to achieving consistent access to objects by edge proxies while retaining the proxies' load-dispersing and latency-reducing effects. Quiver organizes the proxies in a tree rooted at the server. The tree is structured so that geographically close proxies reside close to one another in the tree. To perform certain types of operations, a proxy uses the tree to *migrate* each involved object to itself and then performs the operation locally. Although this incurs the expense of object migration for some operations and is thus reasonable only if objects are not too large and operations involve only a few, it also promises performance benefits for two types of applications.

The first type is applications in which operations exhibit geographic locality: once an object has been migrated to a proxy, other operations (including updates) at that proxy involving this object can be performed locally in contrast to standard replication techniques. Even operations at nearby proxies benefit, since the object is already close and need not be migrated far. Our use of a tree, through which migrations occur, is the key to realizing this benefit. Given the diurnal pattern of application activity that is synchronized with the business day and the fact that the business day occupies different absolute times around the world, we believe that exploiting workload locality through migration can play an important role in optimizing global applications. The second type of applications that can benefit from Quiver are those for which migrating service objects to proxies and performing operations there is more efficient than performing all operations at a centralized server. One example is an application that involves large amounts of data that would be expensive to send to the server but for which objects remain small and, thus, can be migrated. Another is one for which processing load would overload a server, but because the operations involve diverse objects and are performed at different proxies, the load is naturally dispersed across proxies. We confirm the benefits for such applications via tests on PlanetLab.

Perhaps, the most obvious drawback of object migration is increased sensitivity to proxy disconnections: if a proxy disconnects while holding an object because the proxy fails,

- *M.K. Reiter is with the Department of Computer Science, University of North Carolina at Chapel Hill, Campus Box 3175, Sitterson Hall, Chapel Hill, NC 27599-3175. E-mail: reiter@cs.unc.edu.*
- *A. Samar is with Goldman Sachs International, 120 Fleet Street, London E14 3TU. E-mail: asad.samar@gs.com.*

Manuscript received 14 Nov. 2006; revised 21 July 2007; accepted 18 Sept. 2007; published online 12 Oct. 2007.
Recommended for acceptance by G. Agrawal.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0362-1106.
Digital Object Identifier no. 10.1109/TPDS.2007.70790.

it can no longer communicate with its parent, or its parent disconnects, then operations that it recently applied to the object may be lost. In Quiver, 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. 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 that it reflects become durable. Alternatively, an operation can be forced to be durable when performed but at additional cost.

For these durable operations, Quiver can implement either serializability [4], [5] or strict serializability [4]. The only difference in the two modes is in how single-object reads are handled. In either case, single-object reads do not require object migration, and if merely serializability suffices, then a proxy can perform a single-object read locally. Moreover, recall that strict serializability implies linearizability [7] for applications that employ only single-object operations. This case characterizes the main applications in which we are interested, and so, we will focus on this case in our empirical evaluations.

This paper proceeds as follows: We discuss related work in Section 2 and our goals in Section 3. We present our basic object management protocol in Section 4, how we use it to implement operations in Section 5, and how we manage the tree of proxies in Section 6. We evaluate the performance of Quiver in Section 7 and discuss the applications that we have built using Quiver in Section 8.

2 Related Work

Providing consistent and scalable access to shared objects is a topic with a rich research history. Approaches (of which we are aware) that do not use migration can be placed on a spectrum. On one end, all updates to an object are performed at one "primary" location. Updates or cache invalidations are then pushed out to (read-only) cached copies (for example, [8], [9], [10], [11], [12], [13], [14], and [15]). On the other end, objects are replicated across a set of proxies. Typically, any proxy can service updates or reads, and proxies are synchronized by propagating updates to all proxies via, for example, group multicast (for example, [16]) or epidemic (for example, [17]) algorithms. This approach is often referred to as the "update-anywhere" approach. Between these extremes lie other solutions. For example, in the update-anywhere scenario, synchronizing updates with only a quorum of proxies (for example, [18] employs quorums in the context of edge services) reduces the communication overhead. In the primary-site approach, using the primary only to order operations while processing the operations on other proxies reduces load on the primary [19], [20].

Our approach departs from these paradigms by migrating objects to proxies for use in updates. As discussed in Section 1, this enables processing load to be better dispersed across proxies in comparison to most primary-site-based approaches. It also provides communication savings in comparison to all the approaches above in cases where updates exhibit geographic locality. This is particularly true if strict serializability is required, since to implement this property with the above approaches, wide-area crossings occur on the critical path of all operations.

Migration is a staple of distributed computing. Work in this area is too voluminous to cover thoroughly here, though [21] and [22], for example, offer useful surveys. Many previous studies in object migration have drawn from motivation similar to ours, namely, colocating processing and data resources. However, to our knowledge, the approaches in Quiver for managing migration and object reads, as well as for recovering from disconnections, are novel. The only work (of which we are aware) that applies object migration to dynamic Web services [23] does not handle failure of proxies, supports only single-object operations, and provides weak consistency semantics. Quiver improves on all of these aspects.

Our approach to migration was most directly influenced by distributed mutual exclusion protocols, notably [24], [25], and [26]. These protocols allow nodes arranged in a tree to retrieve shared objects and perform operations atomically. Although these approaches achieve scalability and consistency, they do not address failures. Quiver also enables consistent multiobject operations and optimizations for single-object reads that are not possible in these prior algorithms.

3 Terminology and Goals

Our system implements a service with a designated *server* and an unbounded number of *proxies*. We generically refer to the server and the proxies as *processes*. To support the service, a proxy *joins* the service. In doing so, it is positioned within a tree rooted at the server. A proxy can also voluntarily *leave* the service.

If a process loses contact with one of its children, for example, due to the failure of the child or of the communication link to the child, then the child and all other proxies in the subtree rooted at the child are said to *disconnect*. To simplify the discussion, we treat the disconnection of a proxy as permanent, or more specifically, a disconnected proxy may rejoin the service but with a reinitialized state. A proxy that joins but does not disconnect (although it might leave voluntarily) is called *connected*.

The service enables proxies (on behalf of 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 including an *identifier* field $o.\mathrm{id}$ and a *version* field $o.\mathrm{version}$. 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, that is, an instance o' such that $o'.\mathrm{id} = o.\mathrm{id}$ and $o'.\mathrm{version} + 1 = o.\mathrm{version}$.

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 proxies that disconnect) such that the

1. We do not address the failure of the server. We presume that it is rendered fault tolerant, using standard techniques (for example, [6]).

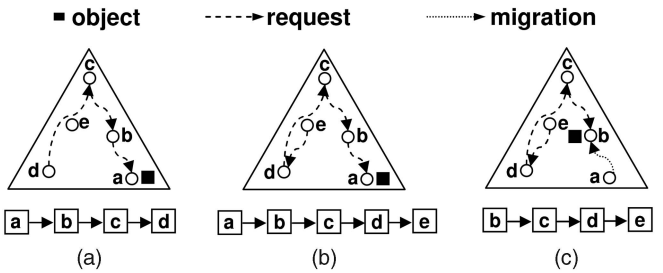

Fig. 1. (a) distQ with processes a, b, c, and d. (b) e appends itself to distQ by sending a retrieve request to d. (c) When a completes its operation, it migrates the object to b and drops off distQ.

connected processes perceive the operations in Durable (and no other) to be executed sequentially. More precisely, we present two variations of our algorithm. One enforces *serializability* [4], [5]: all connected processes perceive the operations in Durable to be executed in the same sequential order. The other enforces an even stronger property, that is, *strict serializability* [4]: the same sequential order perceived by processes preserves the real-time order between operations.

4 Object Management

We begin by describing a high-level abstraction in Section 4.1, which enables our solution, and then discuss the implementation of that abstraction in Section 4.2. Section 5 describes how Quiver proxies use this implementation to perform operations.

4.1 distQ Abstraction

For each object, processes that 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 Fig. 1b). 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 Fig. 1c). 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.

4.2 distQ Implementation

distQ for the object with identifier id (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 proxy p is initialized with a single entry, p's parent, the parent being in the direction of the server (see Fig. 2a). (Further details involved with object creation are discussed in Section 5.3.)

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 to 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 Figs. 2b and 2c). 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 to the direction of the new head of distQ[id] (see Fig. 2d).

The pseudocode for this algorithm is shown in Fig. 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 ith 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. The initialization of a process upon joining the tree is not shown in the pseudocode in Fig. 3; we describe the initialization here. When a process p joins the tree, it is initialized with a parent p.parent ($\perp$ if p is the server). Each process also maintains a set p.children that is initially empty but that grows as the tree is formed; see Section 4.1. When p first receives an operation invocation or message pertaining to object identifier id, p allocates and initializes a local

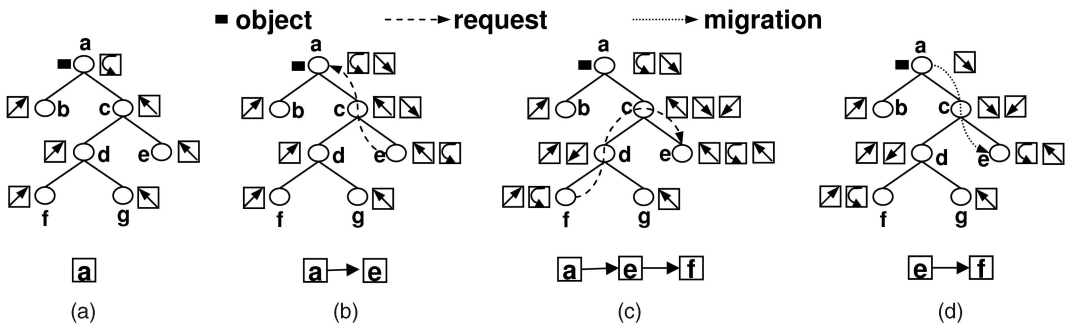

Fig. 2. Squares at a process represent its localQ. The leftmost square is the head. Initially, a has the object. e requests from a, f requests from e, and a migrates the object to e.

```
doRetrieveRequest(from, id, prog)                    /* Invoked locally on request by from (from could be p) */
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)           /* Object 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 : o'.version > p.objs[o'.id].version   /* For each one of the copied objects that is more recent than the local copy... */
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 */
```

Fig. 3. Object management pseudocode for process p.

queue $p.\text{localQ}[id]$ by enqueuing p if p is the server and $p.\text{parent}$ otherwise. In addition, if p is the server, it initializes its copy of the object, $p.\text{objs}[id]$, 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, for example, $p.\text{parent}$. Variable names without the "$p.$" prefix represent the state local to its thread. In order to synchronize these threads, the pseudocode of process p employs a semaphore $p.\text{sem}[id]$ per object identifier id, which is used to prevent the migration of object $p.\text{objs}[id]$ to another process before p is done using it. $p.\text{sem}[id]$ is initialized to one at the server and zero elsewhere. Our pseudocode assumes that any thread executes in isolation until it completes or blocks on a semaphore.

4.3 Retrieving One Object

The routing of retrieve requests for objects is handled by the doRetrieveRequest function shown in Fig. 3. When p executes $\text{doRetrieveRequest}(from, id, prog)$, it adds $\langle from, prog\rangle$ to the tail of $p.\text{localQ}[id]$ (line 2), since $from$ denotes the process from which p received the request for id. ($prog$ has been elided from the 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 $\text{doMigrate}(id)$ (line 5, discussed in the following). If the previous tail was another process q, then p sends $(\text{retrieveRequest} : p, id)$ to q (line 7). When received at q, q will perform $\text{doRetrieveRequest}(p, id, \perp)$ similarly (line 20). This way, a retrieve request is routed to the tail of $\text{distQ}[id]$, where it is blocked until the object migration begins. 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 $p.\text{localQ}[id]$ 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 $\text{doRetrieveRequest}(p, id, prog)$. Again, we defer the discussion of $prog$ to Section 5. Otherwise, p migrates the object toward that destination (line 16 or line 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), as detailed in Section 4.4.

4.4 Object Dependencies

There is a natural "depends-on" relation $\Rightarrow$ 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 $\Rightarrow \overset{\text{def}}{=} \bigcup_{op} \overset{op}{\Rightarrow}$, that is, $o \Rightarrow o'$, if and only if there is some op such that $o \overset{op}{\Rightarrow} o'$. Intuitively, a process p should pass an object instance o to $p.\text{parent}$ only if all object instances on which o depends are already recorded at $p.\text{parent}$. Otherwise, $p.\text{parent}$ might receive only o before p disconnects, in which case the atomicity of the operation that produced o cannot be guaranteed. Thus,

to pass o to $p.\text{parent}$, p must *copy* along all object instances on which o depends. Note that copying is different from migrating, in particular copying does not transfer the "ownership" of the object.

Because each process holds only the latest version that 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 $o \Rightarrow o'$, as o' may have been "overwritten" at p, that is, $p.\text{objs}[o'.\text{id}].\text{version} > o'.\text{version}$. In this case, it would suffice to copy $p.\text{objs}[o'.\text{id}]$ in lieu of o' if each o'' such that $p.\text{objs}[o'.\text{id}] \Rightarrow o''$ was 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 by the following rules until no more indices can be added to IDs: 1) initialize IDs to $\{o.\text{id}\}$ and 2) if $id \in IDs$ and $p.\text{objs}[id] \Rightarrow o'$, then add $o'.\text{id}$ to IDs. p then copies $\{p.\text{objs}[id]\}_{id \in IDs}$ 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 $p.\text{Deps}$ between object identifiers, initialized to $\emptyset$. If p performs an update operation op such that an output $p.\text{objs}[id] \stackrel{op}{\Rightarrow} p.\text{objs}[id']$, then p adds (id, id') to $p.\text{Deps}$. In order to perform $\text{doMigrate}(id)$ to $p.\text{parent}$, p determines the identifier set IDs as those indices reachable from id by following edges (relations) in $p.\text{Deps}$ (reachability is denoted $\stackrel{p.\text{Deps}}{\Longrightarrow}$ in line 13 in Fig. 3) and copies both $Objs = \{p.\text{objs}[id']\}_{id' \in IDs}$ (line 14) and $DepSet = p.\text{Deps} \cap (IDs \times IDs)$ (line 15), along with the migrating object (line 16). Finally, p updates $p.\text{Deps} \leftarrow p.\text{Deps} \setminus DepSet$ (line 17), that is, to remove these dependencies for future migrations upward.

Upon receiving a migration with copied objects $Objs$ and dependencies $DepSet$, p saves each $o \in Objs$ with a higher version than $p.\text{objs}[o.\text{id}]$ (lines 22 and 23) and adds $DepSet$ to $p.\text{Deps}$ (line 24).

5 Operation Implementation

To achieve our desired consistency semantics, for each object, we enforce a sequential execution of all update and multiobject 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 multiobject operations enables significant optimizations for single-object reads (Section 5.2) that dominate many workloads. Object creation (Section 5.3) is performed as a special case of an update operation.

5.1 Update and Multiobject Operations

Invoking operations. Let $id_1, \ldots, id_k$ denote distinct identifiers of the objects involved (read or updated) in an update or multiobject 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:

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

Process p then executes $prog_k$, which requests id_k, and once that is retrieved, it executes $prog_{k-1}$ (line 11 in Fig. 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 $p.\text{Deps}$ (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 $p.\text{sem}[id_1], \ldots, p.\text{sem}[id_k]$ as locks, $prog_k$ can be viewed as implementing strict two-phase locking [5]. Thus, to prevent deadlocks, $id_1, \ldots, id_k$ must be arranged 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 $p.\text{objs}[o.\text{id}].\text{version} < o.\text{version}$. 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 force its updates to be durable when it leaves the tree (Section 6.3) so that operations by a process that remains connected until it leaves are durable.

5.2 Single-Object Read Operations

We present two protocols implementing a single-object read. 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 multiobject operations from Section 5.1. (We provide correctness proofs in a companion document [27].)

Serializability. Due to the serial execution of update and multiobject operations (Section 5.1), single-object reads so as to achieve serializability [5] can be implemented with local reads; that is, a process p performs a read involving a single object with identifier id by simply returning $p.\text{objs}[id]$.

Strict serializability. Recall that all update and multiobject read operations involving the same object are performed serially (Section 5.1). Therefore, in order to guarantee strict serializability, it suffices 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. This could be achieved by serializing op with the update and multiobject operations in $\text{distQ}[id]$. However, this would require op to wait for the completion of the concurrent update and multiobject operations (those performed by processes preceding p in $\text{distQ}[id]$).

A more efficient solution is to request the latest version from the process at the head of $\text{distQ}[id]$, the process that is the current "owner" of the object with identifier id. Our algorithms already provide a way of routing to the head of $\text{distQ}[id]$ by using $\text{localQ}[id].\text{head}$ at each process. Thus, a read request for id follows $p.\text{localQ}[id].\text{head}$ at each

process p until it reaches a process p' such that either $p'.\mathrm{localQ}[id].\mathrm{head} = p'$ (that is, p' holds the latest object version) or $p'.\mathrm{localQ}[id].\mathrm{head} = p''$ is the process that forwarded this read request to p'. In the latter case, p' forwarded $p'.\mathrm{objs}[id]$ to p'' in a migration concurrently with p'' forwarding this read request to p' (since $p''.\mathrm{localQ}[id].\mathrm{head} = p'$ when p'' did so), and so, it is safe for p' to serve the read request with $p'.\mathrm{objs}[id]$.

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 (a mechanism discussed in Section 6.2), we require that the object be passed through the tree to the highest process in the path from p' to p, that is, 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. Note that since the requested object is copied upward in the tree from p' to p'' (unless $p' = p''$), any objects that the requested object depends upon must also be copied along by using the techniques described in Section 4.4.

5.3 Object Creation

As discussed in Section 4.2, when a proxy p first receives an operation invocation locally or a message (from another proxy) pertaining to the object identifier id, p allocates and initializes a local queue $p.\mathrm{localQ}[id]$ by enqueuing p if p is the server and $p.\mathrm{parent}$ otherwise. In addition, if p is the server, it initializes its copy of the object $p.\mathrm{objs}[id]$ to a default initial state.

These initialization steps support object creation as a natural consequence, with no additional mechanism. That is, to create an object with identifier id, a proxy only needs to perform an update operation on id. This update operation will allocate these local structures and then attempt to retrieve this object, as described in Section 5.1. Upon receiving the retrieveRequest message for id, a proxy will allocate its local structures for id and proceed with the protocol. When the request reaches the server, the server does similarly and initiates the migration of the newly created object to p. By virtue of reaching the server, object creation is a durable operation (see Section 5.1).

6 Tree Management

Here, we detail how we construct and maintain the tree (Section 6.1) and how we adapt our algorithm to address disconnections (Section 6.2) and proxies leaving voluntarily (Section 6.3).

6.1 Construction

Because communication occurs between neighboring processes in the tree, Quiver performs best if the proxies are arranged in an approximately minimum spanning tree rooted at the server. Many distributed algorithms exist by which nodes can generate a minimum-weight (or approximately minimum-weight) tree spanning them. For example, see [28], [29], and [30].

For simplicity, however, we utilize a more centralized approach in Quiver to construct a spanning tree among the proxies. This is done by first measuring the round-trip delay between each pair of processes. We denote the round-trip delay between processes p and p' as $d(p, p')$. These delays are sent to the server, which computes the minimum spanning tree by using Kruskal's algorithm [31], with the following exceptions. First, the server is chosen as the root of the tree. Second, before adding a proxy p as a child of a proxy p' (according to Kruskal's algorithm), we first check if $d(p, p'') - d(p, p') < \gamma$, where p'' is the parent of p', and γ is a predefined threshold. If the difference is within this threshold, then p is inserted as a child of p'' instead. This is done to avoid increasing the depth of the tree unnecessarily while still approximating the minimum spanning tree. Finally, we restrict the number of children for each node in the tree to a predefined constant η. In particular, if a node already has η children, then we remove any other edges to this node from the set of edges that have not been inserted in the tree yet.

Strict maintenance of the spanning tree in the event of proxies joining and leaving would require recomputing and reconstructing the tree (although not necessarily from scratch). In case of joins, we avoid this cost by making the joining proxy a child of the closest (in terms of the round-trip delay) existing process that has less than η children. This could result in a tree that is far from the theoretical optimal, but nevertheless, this is one that preserves enough network proximity to allow our algorithms to exploit it. The mechanisms that deal with proxy disconnections (Section 6.2) and voluntary departures (Section 6.3) result in similarly pragmatic solutions.

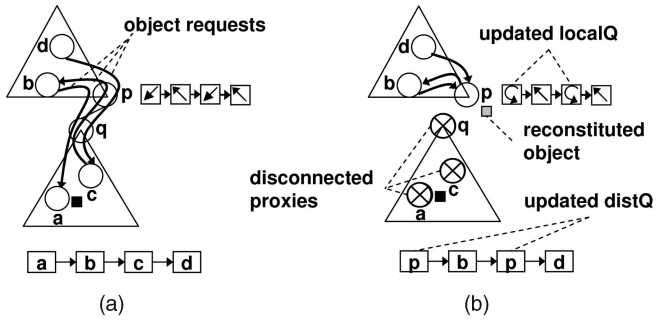

Fig. 4. q loses contact with parent p, and its subtree disconnects. p replaces disconnected proxies in distQ and reconstitutes the object, so b and d can make progress.

6.2 Disconnections

Recall that when a process loses contact with a child, all proxies in the subtree rooted at that child are said to *disconnect*. The child (or, if the child failed, each uppermost surviving proxy in the subtree) can inform its subtree of the disconnection. The subtree can stay put until the disconnection heals, or they might reconnect via the server. Either way, some of these disconnected proxies may have earlier issued retrieve requests for objects, and for each such object with identifier id, the disconnected proxy may appear in $\mathrm{distQ}[id]$. In this case, it must be ensured that the connected processes preceded by a disconnected process in $\mathrm{distQ}[id]$ continue making progress. To this end, all occurrences of the disconnected proxies in $\mathrm{distQ}[id]$ are replaced with the parent p of the uppermost disconnected proxy q (see Fig. 4).

Choosing p to replace the disconnected proxies is motivated by several factors: First, p is in the best position to detect the disconnection of the subtree rooted at its

```
childDisconnected(q)
1.  p.children ← p.children \ {q}
2.  foreach id
3.      q' ← p.localQ[id].head
4.      Qreplace(id, q)
5.      if q' = q
6.          V(p.sem[id])

Qreplace(id, q)
7.  foreach i = 1, ..., p.localQ[id].size − 1
8.      if p.localQ[id].elmt[i] = ⟨q, ∗⟩
9.          p.localQ[id].elmt[i] ← ⟨p, ||V(p.sem[id])||⟩
10.         t ← new thread(||P(p.sem[id]); doMigrate(id)||)
11.         t.enable()
12. if p.localQ[id].tail = ⟨q, ∗⟩
13.     p.localQ[id].tail ← ⟨p, ||V(p.sem[id])||⟩
```

Fig. 5. Disconnection handling at p.

child q. Second, as we will see in the following, in our algorithm, p only needs to take local actions to replace the disconnected proxies. As such, this is a very efficient solution. Third, in case the head of $\mathrm{distQ}[id]$ is one of the disconnected proxies, 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 the now-disconnected proxies 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). Even with 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 $\mathrm{distQ}[id]$.

The pseudocode that p executes when its child q disconnects is the $\mathrm{childDisconnected}(q)$ routine in Fig. 5. Specifically, p replaces all instances of q in $p.\mathrm{localQ}[id]$ with itself and a "no-op" operation to execute once p obtains the object (lines 8 and 9, and 12 and 13). As such, any retrieve request that was initiated at a connected process and blocked at a disconnected proxy is now blocked at p (see Fig. 4b). For each of these requests that are now blocked at p, p creates and run-enables a new thread (lines 10 and 11 in Fig. 5) to initiate the migration of $p.\mathrm{objs}[id]$ to the neighbor following (this instance of) p in $p.\mathrm{localQ}[id]$ once p has the object. If the disconnected child was at the head of $p.\mathrm{localQ}[id]$, then p reconstitutes the object simply by making its local copy (which is the latest at any connected process) available (lines 5 and 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 Fig. 5).

6.3 Leaves

To voluntarily leave the tree, a proxy p must ensure that any objects in the subtree rooted at p are still accessible to connected nodes once p leaves. In addition, outstanding retrieve requests forwarded through p must not block due to p leaving.

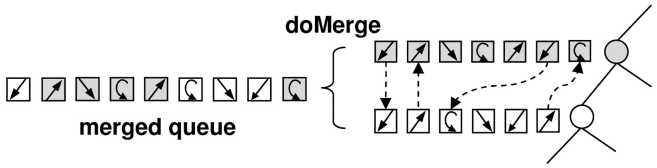

Fig. 6. Queue merge. Shaded and unshaded elements are in parent's and child's localQ, respectively. Dashed arrows are from a skipped element to the element added next.

If p is a leaf node, 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 chooses one of its children q to *promote*. The promotion updates q's state according to the state at p and notifies neighbors of p about q's promotion.

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 $p.\mathrm{parent}$, $p.\mathrm{children}$, $p.\mathrm{localQ}[\,]$, $p.\mathrm{objs}[\,]$ (or, rather, only those object versions that q does not yet have), and $p.\mathrm{Deps}$. When q receives these, it updates its parent, children, objects, and object dependencies according to p's state.

The interesting part of q's promotion is how it *merges* $q.\mathrm{localQ}[id]$ with $p.\mathrm{localQ}[id]$ 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 Fig. 6. Fig. 7 presents the pseudocode used by a promoted child q to merge $q.\mathrm{localQ}[id]$ with its parent p's $p.\mathrm{localQ}[id]$ for each identifier id as the parent voluntarily leaves the service. In order to merge $p.\mathrm{localQ}[id]$ and

```
Upon receiving (promote :gParent, siblings, parentQ[ ],
                        parentObjs[ ], parentDeps)
1.  foreach id
2.      if parentObjs[id].version > q.objs[id].version
3.          q.objs[id] ← parentObjs[id]
4.      mergedQ[id] ← ∅
5.      if q.localQ[id].head = ⟨q.parent, ∗⟩
6.          doMerge(q.localQ[id], parentQ[id],
                    q, q.parent, mergedQ[id])
7.      else
8.          doMerge(parentQ[id], q.localQ[id],
                    q.parent, q, mergedQ[id])
9.      q.localQ[id] ← mergedQ[id]
10.     Qreplace(id, q.parent)
11. q.parent ← gParent
12. q.children ← (q.children ∪ siblings) \ {q}
13. q.Deps ← q.Deps ∪ parentDeps

doMerge(localQ, localQ', p, p', mergedQ)
14. while localQ.hasElements()
15.     ⟨r, prog⟩ ← localQ.removeFromHead()
16.     if r ≠ p'
17.         mergedQ.addToTail(⟨r, prog⟩)
18.     else
19.         doMerge(localQ', localQ, p', p, mergedQ)
```

Fig. 7. Pseudocode at q for its promotion.

$q.\text{localQ}[id]$, q begins with $q.\text{localQ}[id]$ if its head points to p; otherwise, it begins with $p.\text{localQ}[id]$. q adds elements from the chosen queue, say, $p.\text{localQ}[id]$, to a newly created $mergedQ$ until an instance of q is reached (line 19 in Fig. 7), say, at the $i\text{th}$ index, that is, $p.\text{localQ}[id].\text{elmt}[i] = q$. The merge algorithm then skips this $i\text{th}$ element and begins adding elements from $q.\text{localQ}[id]$ until an instance of p is found. This element is skipped and the algorithm switches back to $p.\text{localQ}[id]$ adding elements starting from the $(i+1)\text{st}$ 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 $\text{Qreplace}(id, p)$ defined in Fig. 5.

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

7 Evaluation

We evaluated the performance of Quiver through experiments performed on PlanetLab. Our system is implemented in Java 2 (Standard Edition 5.0). As discussed in Section 1, the main uses that we are pursuing for Quiver involve single-object updates and reads only, in which case strict serializability equates to linearizability [7]. Due to space limitations, here, we primarily focus our evaluation to this configuration of our system, that is, linearizable single-object operations in a static tree of proxies. The cost of multiobject operations and the impact of leaves and disconnections on performance are evaluated in a companion document [27].

When testing Quiver, the server and proxies were organized in an approximate minimum spanning tree (see Section 6.1). The node with the minimum median latency to all other nodes was selected as the root (server), and each node had a maximum of $\eta = 4$ children. The parameter γ from Section 6.1 (the threshold that decides whether the depth of the tree should be increased when a new node joins) was chosen as 2 percent of the smallest distance (latency) between nodes in the experiment. In addition to these nodes, we used a *monitor* to control the experiments and measure performance. The monitor ran on a dedicated machine and communicated with all nodes (the server and proxies) in an experiment. In each experiment, each proxy notified the monitor upon joining the tree and then awaited a command from the monitor to begin reads and updates. Upon receiving this command, the proxy performed operations sequentially for 100 seconds, that is, beginning the next operation after the previous completed. Each operation was chosen to be a read with a probability specified by the monitor; otherwise, this was an update. This way, the monitor dictated the percentage of reads in the workload. Unless stated otherwise, each operation was a read with probability 0.8 and involved a single object chosen uniformly at random from all objects in the experiment. In addition, unless stated otherwise, there were 50 objects in an experiment, and to isolate the costs attributable to Quiver, these objects were simple integer counters that support increment (update) and read operations. After 100 seconds, each proxy reported its average read and update latencies, as well as the number of operations that it completed, to the monitor. Each experiment was repeated five times.

We provide results for three types of tests. The first was baseline tests of the performance of Quiver in a wide-area network. The second and third types of tests highlighted workloads for which Quiver is suited, namely, those in which operations for each object exhibit geographic locality or in which updates are computationally intensive. For the latter two workloads, we also compared Quiver to an implementation using a centralized server, which was chosen to be the same node as the server (root of the tree) in the corresponding Quiver tests. In the centralized tests, each proxy sent its operations to the centralized server to be performed, awaiting the server's response to each before sending the next. We note that involving the centralized server in reads (versus reading from a local copy) is necessary to achieve linearizability and, more generally, strict serializability.

Although we compare Quiver only to a centralized implementation, we believe that some trends that this comparison reveals should hold in comparison to quorum-based solutions as well. In particular, like a centralized solution, quorum-based solutions cannot significantly improve network delays for clients to access the necessary replicas: Lin [32, Theorem 3] proves that a centralized approach optimizes the average network delays observed by clients to within a factor of two of *any* quorum-based implementation. Moreover, an empirical analysis of quorum placement algorithms on wide-area topologies [33] confirmed that centralized solutions yield better network access delays than quorum implementations do. While quorum-based solutions can disperse processing load across replicas, this goal is at odds with minimizing network delays: dispersing processing load may require an operation to bypass a nearby but heavily loaded quorum, instead landing at a more distant quorum and thus incurring more network delays. For this reason, exploiting this load-dispersing property of quorums to lower client response times is hard both in theory [34] and in practice [33].

Baseline tests. The baseline experiments employed 70 nodes (one server and 69 proxies) spread across North America. We conducted three types of tests to evaluate Quiver's baseline performance. The first test varied the fraction of reads in the workload from 0 (only updates) to 1 (read-only workload). The update latency, read latency, and overall throughput, that is, the number of operations (updates or reads) per second, are reported in Fig. 8. The case without reads can be viewed as indicative of the performance of updates in a configuration offering serializability only, that is, where reads are performed locally by

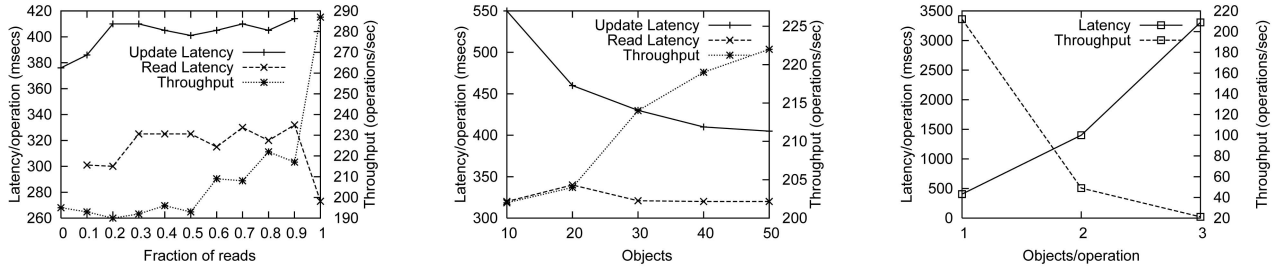

Fig. 8. Baseline tests: latency and throughput with varying fractions of reads in the workload and numbers of shared objects and objects per operation.

proxies and, hence, with negligible costs. Note that in a read-only workload, the objects are not migrated and remain at the root. The read requests are therefore served by the root, which sends the object outside the tree directly to the requester (see Section 5.2). With an introduction of updates in the workload, however, the objects are migrated to proxies, and the response to a read request may go through the tree, at least partly, resulting in a sudden increase in read latency.

The second test was performed by varying the number of objects. A small number of objects resulted in higher contention and therefore increased the update latency (updates are serialized per object).

The final baseline test varied the number of objects involved per operation. All operations in this experiment were updates, since multiobject reads and updates are handled using the same algorithm. For each update, the proxy selected one, two, or three objects (depending on the experiment), out of the 50 total objects, uniformly at random, migrated these objects to itself (one by one, as described in Section 5.1), and then performed its operation. The latency of multiobject operations increased substantially with an increase in the number of objects involved in each operation, mainly due to the use of the two-phase locking approach in our algorithms, and shows that Quiver's performance is best suited to workloads that are dominated by single-object operations. Due to space limitations, we omit baseline results involving nontrivial objects and varying numbers of proxies. We refer the reader to our companion document [27] for these.

Workloads with operation locality. We performed two types of tests to validate our hypothesis that workloads in which operations per object exhibit geographic locality will benefit from Quiver. The first test captured scenarios in which different objects are more popular in different regions, whereas the second test captured scenarios in which different regions are active in different time intervals.

For the first test, we divided the 70 North American PlanetLab nodes into three groups of roughly equal size, consisting of East Coast nodes, West Coast nodes, and others ("central" nodes). Each group selected objects in operations according to a different distribution so that different groups "focus on" different objects. Specifically, the set of 50 objects were partitioned into $n = 5$ disjoint sets $Objs_0, \ldots, Objs_{n-1}$, each of size 10 objects. We defined permutations on $\{0, \ldots, n-1\}$ by

$$\begin{aligned} \pi_{\text{east}}(i) &= i \\ \pi_{\text{west}}(i) &= n - 1 - i \\ \pi_{\text{cent}}(i) &= \lfloor n/2 \rfloor + \lceil i/2 \rceil \times (-1)^{i \bmod 2} \end{aligned}$$

and distributions D_{east}, D_{west}, and D_{cent}, satisfying $D_{\text{east}}(\pi_{\text{east}}(i)) = D_{\text{west}}(\pi_{\text{west}}(i)) = D_{\text{cent}}(\pi_{\text{cent}}(i)) = Z(i)$. Here, $Z : \{0, \ldots, n-1\} \to [0, 1]$ was a Zipf distribution such that $Z(i) \propto 1/(i+1)^{\alpha}$ and $\sum_{i=0}^{n-1} Z(i) = 1$. α is called the *popularity bias*. When an East Coast node initiated an operation, it selected the object on which to do so by first selecting an object set index i according to the distribution D_{east} and then selecting from $Objs_i$ uniformly at random. West Coast and central nodes did similarly, using their respective distributions D_{west} and D_{cent}.

The second test used 50 PlanetLab nodes divided into four groups of roughly equal size located in China, Europe, and North American East and West Coasts. Each group chose objects uniformly at random from the set of all objects but was "awake" during different time intervals. Specifically, the monitor instructed the Chinese, European, East Coast and West Coast nodes to initiate their 100-seconds intervals of activity at times T, $T + \Delta$, $T + 2\Delta$, and $T + 3\Delta$, respectively, where $\Delta \in [0 \text{ seconds}, 100 \text{ seconds}]$. Thus, when $\Delta = 0$ seconds, the intervals completely overlapped, but when $\Delta = 100$ seconds, the intervals were disjoint. In these tests, unlike the others that we have reported, the overall load on the system fluctuated during the test as node groups "woke" and "slept." As such, the throughput that we report is the average of the throughputs observed by the four regions during their "awake" intervals.

The results demonstrate that as the popularity bias grows in the first test (Fig. 9) and as the offset Δ grows in the second test (Fig. 10), in each case increasing the geographic locality of requests per object, Quiver surpasses a centralized implementation in both latency and throughput. The second test further reveals a practical optimization

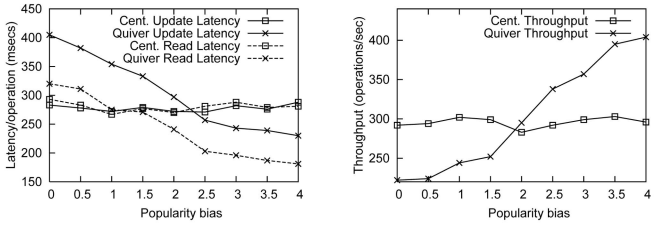

Fig. 9. Object popularity bias workload.

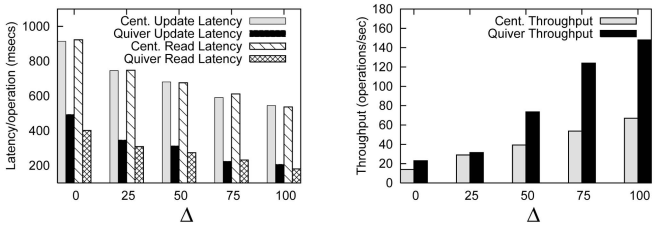

Fig. 10. Regional activity workload.

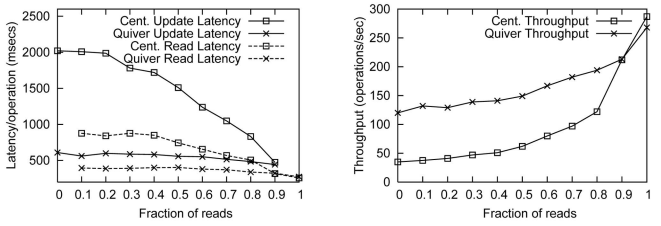

Fig. 11. Computationally intensive workload.

enabled by Quiver's design: since each proxy typically communicates with only a small number of neighbor proxies in the tree, a proxy can afford to maintain long-lived TCP connections to its neighbor proxies, avoiding the cost of a TCP handshake for messaging between neighbors. A centralized server, however, cannot keep long-lived connections to an unbounded number of proxies and, therefore, incurs this cost for each request that it serves. This cost is negligible for links with smaller round-trip times. For example, the maximum round-trip time among the North American nodes was 59 ms. However, this cost was more profound on long-haul links: the maximum round-trip time in this experiment was 289 ms between nodes in China and the North American East Coast. This explains Quiver's better performance over the centralized server in the second test, even when all nodes were "awake" at the same time ($\Delta = 0$).

Computationally intensive workloads. Our final tests involved a computationally intensive workload on the 70 North American nodes. As discussed in Section 1, Quiver should offer better performance than a centralized server in this case due to better dispersing the computation load across proxies. To test this, we artificially induced computation per update that, on a 1.4-GHz Pentium IV, took 22 ms on the average.[2]

Fig. 11 compares the centralized and Quiver implementations in this case as the fraction of reads in the workload is varied. Fig. 11 suggests that Quiver outperforms the centralized implementation for virtually all fractions of reads. The latencies and throughputs of the two implementations converge only once that there are no updates in the system (that is, the read fraction is 1), in which case, obviously, the computational cost of updates is of no consequence.

8 Applications

Here, we summarize the use of Quiver in two example applications. We refer the reader to our companion report [27] for further details on the use of Quiver in these applications.

8.1 e-Commerce with Edge Proxies

The first application is an online bookstore that maintains state about books, customers, orders, etc., and allows client interactions to read and update parts of the state. We model our online bookstore application according to the TPC-W benchmark [35], an industry standard benchmark representing an e-commerce workload, specifically on an online bookstore. TPC-W specifies the bookstore state as relational tables and related data structures and defines the semantics of the bookstore interactions that read or update this state.

In order to map this to the Quiver setting, we first divide this state into objects such that objects are small in size and each interaction involves few objects, typically one. Furthermore, the objects are defined so that interactions involving these objects exhibit locality. For example, some of the most frequently accessed objects encapsulate the state that relates to a single client such as the objects related to that client's registration, shopping cart, and orders. These objects will typically be involved in operations at the proxy closest to the client, and therefore, the client-perceived latency will decrease as Quiver migrates these objects to the corresponding proxy.

In addition to improving the performance through migration, Quiver provides facilities to support other features needed for such an e-commerce site. For example, order placement is an operation that should be durable. Quiver supports this as described in Section 5.1. In addition, although certain bookstore objects can be more efficiently managed without migration, namely, objects that are updated at the server and only read by the proxies such as the objects encapsulating a book's description and price, the server can nevertheless perform operations involving such objects and others using Quiver's multiobject operation support.

8.2 Distributed Network Traffic Modeling

We have also used Quiver in an application that constructs network traffic classifiers from traffic logs collected at geographically distributed networks (see [36], [37], and [38] for recent research in this area). These classifiers are then employed to characterize future traffic as belonging to a certain application, say, to identify prohibited traffic on nonstandard ports. In the Quiver setting, each contributing network includes a proxy. A coordination site acts as the server (root of the tree), and the traffic classifiers being constructed are the objects. The proxies perform update and read operations on the traffic classifiers. To update a particular classifier using new records, the contributing network's proxy migrates the classifier to itself and updates the classifier using its data. The algorithms for incrementally updating the classifier require the updates to be serialized, an important property achieved through Quiver. Furthermore, this strategy distributes the computational load of updating the classifiers to all the networks and does not require any network to reveal its raw data to any other entity (except to the extent that this data is revealed by the

2. This computation was the *Sieve of Eratosthenes* benchmark, repeated 40 times, each time finding all primes between 2 and 16,384.

classifier, which is typically much less than the traffic log itself). Experimental results [27] show that Quiver outperforms a centralized implementation of this application by orders of magnitude.

9 Conclusion

We presented a system called Quiver for implementing highly scalable object sharing with strong consistency semantics (serializability or strict serializability). Quiver is well suited to workloads where operations typically access few objects and where operations involving each object exhibit geographic locality or are computationally intensive. Quiver migrates objects to proxies in order to perform update or multiobject operations while supporting more efficient single-object reads. Proxies may join, leave, or disconnect from the service. In case of disconnects, the service recovers objects whose latest versions are left unreachable. We evaluated Quiver over PlanetLab to confirm the performance improvements that it offers for various workloads.

References

[1] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen, "Replication for Web Hosting Systems," *ACM Computing Surveys,* vol. 36, no. 3, pp. 291-334, Sept. 2004.

[2] A. Davis, J. Parikh, and W.E. Weihl, "Edgecomputing: Extending Enterprise Applications to the Edge of the Internet," *Proc. 12th Int'l World Wide Web Conf. (WWW),* 2004.

[3] J. Tatemura, W. Hsiung, and W. Li, "Acceleration of Web Service Workflow Execution through Edge Computing," *Proc. 13th Int'l World Wide Web Conf. (WWW),* 2003.

[4] C. Papadimitriou, "The Serializability of Concurrent Database Updates," *J. ACM,* vol. 26, no. 4, pp. 631-653, Oct. 1979.

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

[6] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg, "Chapter 8: The Primary-Backup Approach," *Distributed Systems,* second ed., S. Mullender, ed., 1993.

[7] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Languages and Systems,* vol. 12, no. 3, pp. 463-492, 1990.

[8] Q. Luo, S. Drishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B.G. Lindsay, and J.F. Naughton, "Middle-Tier Database Caching for e-Business," *Proc. ACM SIGMOD '02,* June 2002.

[9] M. Altinel, C. Bornhvd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald, "Cache Tables: Paving the Way for an Adaptive Database Cache," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* Sept. 2003.

[10] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan, "DBProxy: A Dynamic Data Cache for Web Applications," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* Mar. 2003.

[11] W. Li, O. Po, W. Hsiung, K.S. Candan, D. Agrawal, Y. Akca, and K. Taniguchi, "CachePortal II: Acceleration of Very Large Scale Data Center-Hosted Database-Driven Web Applications," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* Sept. 2003.

[12] C. Plattner and G. Alonso, "Ganymed: Scalable Replication for Transactional Web Applications," *Proc. Fifth ACM/IFIP/Usenix Int'l Middleware Conf. (Middleware),* 2004.

[13] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B.M. Maggs, and T.C. Mowry, "A Scalability Service for Dynamic Web Applications," *Proc. Second Biennial Conf. Innovative Data Systems Research (CIDR),* 2005.

[14] M. Rabinovich, Z. Xiao, and A. Aggarwal, "Computing on the Edge: A Platform for Replicating Internet Applications," *Proc. Eighth Int'l Workshop Web Content Caching and Distribution (WCW),* 2003.

[15] B. Urgaonkar, A.G. Ninan, M.S. Raunak, P. Shenoy, and K. Ramamritham, "Maintaining Mutual Consistency for Cached Web Objects," *Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS '01),* Apr. 2001.

[16] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu, "Practical Wide-Area Database Replication," Technical Report CNDS-2002-1, Johns Hopkins Univ., 2002.

[17] J. Holliday, R. Steinke, D. Agrawal, and A. El-Abbadi, "Epidemic Algorithms for Replicated Databases," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 3, 2003.

[18] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar, "Dual-Quorum Replication for Edge Services," *Proc. Sixth ACM/IFIP/Usenix Int'l Middleware Conf. (Middleware),* 2005.

[19] A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys,* vol. 13, no. 2, pp. 185-221, 1981.

[20] M.T. Özsu and P. Valduriez, "Distributed and Parallel Database Systems," *ACM Computing Surveys,* vol. 28, no. 1, pp. 125-128, 1996.

[21] M. Nuttall, "A Brief Survey of Systems Providing Process or Object Migration Facilities," *ACM Operating Systems Rev.,* vol. 28, no. 4, pp. 64-80, Oct. 1994.

[22] D.S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," *ACM Computing Surveys,* vol. 32, no. 3, pp. 241-299, Sept. 2000.

[23] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen, "GlobeDB: Autonomic Data Replication for Web Applications," *Proc. 15th Int'l World Wide Web Conf. (WWW),* 2005.

[24] K. Raymond, "A Tree-Based Algorithm for Distributed Mutual Exclusion," *ACM Trans. Computer Systems,* vol. 7, no. 1, pp. 61-77, Feb. 1989.

[25] M. Naimi, M. Trehel, and A. Arnold, "A $\log(N)$ Distributed Mutual Exclusion Algorithm Based on Path Reversal," *J. Parallel and Distributed Computing,* vol. 34, no. 1, pp. 1-13, 1996.

[26] M.J. Demmer and M.P. Herlihy, "The Arrow Distributed Directory Protocol," *Proc. 12th Int'l Symp. Distributed Computing (DISC '98),* pp. 119-133, 1998.

[27] A. Samar, "Quiver on the Edge: Consistent, Scalable Edge Services," PhD dissertation, Carnegie Mellon Univ., Aug. 2006.

[28] R. Gallager, P. Humblet, and P. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. Programming Languages and Systems,* vol. 5, no. 1, pp. 66-77, 1983.

[29] B. Awerbuch, "Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems," *Proc. 19th ACM Symp. Theory of Computing (STOC '87),* pp. 230-240, 1987.

[30] M. Khan and G. Pandurangan, "A Fast Distributed Approximation Algorithm for Minimum Spanning Trees," *Proc. 20th Int'l Symp. Distributed Computing (DISC '06),* pp. 355-369, 2006.

[31] J. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Am. Math. Soc.,* vol. 7, pp. 48-50, 1956.

[32] X. Lin, "Delay Optimizations in Quorum Consensus," *Proc. 12th Int'l Symp. Algorithms and Computation (ISAAC '01),* pp. 576-586, 2001.

[33] F. Oprea and M.K. Reiter, "Minimizing Response Time for Quorum-System Protocols over Wide-Area Networks," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '07),* June 2007.

[34] A. Gupta, B.M. Maggs, F. Oprea, and M.K. Reiter, "Quorum Placement in Networks to Minimize Access Delays," *Proc. 24th ACM Symp. Principles of Distributed Computing (PODC '05),* pp. 87-96, July 2005.

[35] *TPC Benchmark W V1.8.* Transaction Processing Performance Council, http://www.tpc.org/, 2002.

[36] V. Yegneswaran, P. Barford, and S. Jha, "Global Intrusion Detection in the DOMINO Overlay System," *Proc. 11th Ann. Network and Distributed System Security Symp. (NDSS '04),* Feb. 2004.

[37] M. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, and D. Watson, "Data Reduction for the Scalable Automated Analysis of Distributed Darknet Traffic," *Proc. Internet Measurement Conf. (IMC '05),* Oct. 2005.

[38] X. Jiang and D. Xu, "Collapsar: A VM-Based Architecture for Network Attack Detention Center," *Proc. 13th Usenix Security Symp.,* Aug. 2004.

Michael K. Reiter received the BS degree in mathematical sciences from the University of North Carolina, Chapel Hill (UNC-CH) in 1989 and the MS and PhD degrees in computer science from Cornell University in 1991 and 1993, respectively. He is currently the Lawrence M. Slifkin Distinguished Professor in the Department of Computer Science, UNC-CH. He has held technical leadership positions in both the industry and academe. From 1998 to 2001, he was the director of Secure Systems Research, Bell Laboratories. From 2001 to 2007, he was a professor and the technical director of CyLab, Carnegie Mellon University. His research interests include computer and communications security and distributed computing. He is a senior member of the IEEE Computer Society.

Asad Samar received the BEng degree in computer engineering from the National University of Science and Technology, Pakistan, in 1999, and the MS and PhD degrees in electrical and computer engineering from Carnegie Mellon University in 2003 and 2006, respectively. He is currently a strategist at Goldman Sachs International. From 1999 to 2001, he was a research staff member at the California Institute of Technology. His research interests include secure distributed computing and large-scale data management. He has published several papers in related conference proceedings and journals.