Quiver on the Edge: Consistent Scalable Edge Services

ASAD SAMAR

August 2006

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

To Tehniat, my wife, my friend, my inspiration

Abstract

Hosting dynamic web services through proxies placed at the edge of the Internet is an upcoming trend that has the potential to scale these services to a very large number of geographically distributed clients. However, providing consistent access to shared mutable objects that make up the service, tolerating misbehaving proxies, and handling proxy disconnections, while still achieving the scalability and performance expected from such an architecture is a significant challenge.

This dissertation presents Quiver, a distributed object system that supports consistent (serializable or strictly serializable) operations on shared objects by service proxies in a wide-area setting, while reducing the client-perceived latency. We also present extensions to Quiver that detect compromised proxies attempting to violate Quiver's consistency properties, allow proxies to disconnect and efficiently reconnect without delaying operations from connected proxies, and optimize object access times by restructuring Quiver's communication network according to the workload.

Quiver proxies are arranged in a "location-aware" rooted tree. In order to perform an update or a multi-object operation involving certain service objects, a Quiver proxy *migrates* those objects through the tree, to itself. Object migrations for operation processing ensures a serial execution of update and multi-object operations involving the same objects, and enables Quiver to achieve the desired consistency semantics, while optimizing for the (typically, more frequent) single-object read operations. Furthermore, when operations involving an object exhibit geographic locality—e.g., during business hours on one continent (and non-business hours on others)—the

performance of these operations benefits from the object having been migrated to a nearby proxy. Other workloads benefit from Quiver dispersing the compute load across the proxies performing operations, and saving the costs of transmitting operation parameters over the WAN when these are large.

Guaranteed consistency semantics for shared objects via migrations through a tree of proxies requires that the proxies are well-behaved, i.e., follow the protocol specifications. We, therefore, present an extension to Quiver that relaxes this assumption by allowing honest proxies to efficiently detect a misbehaving proxy that attempts to compromise the consistency of object accesses.

Finally, this dissertation also discusses extensions to manage the rooted tree (an overlay) that connects the Quiver proxies, in order to improve the overall service availability and performance. In particular, we describe a distributed algorithm that constructs a fault-tolerant network on top of the tree, allowing proxies to efficiently reconnect to the primary partition (the partition containing the root), in case of proxy or link failures. This efficient reconnection algorithm reduces the "down-time" of Quiver proxies while avoiding some central point of reentry (e.g., the root) from being overloaded due to frequent reconnect requests. We also discuss extensions that heuristically restructure the tree to bring the proxies that frequently perform operations involving the same objects close to each other, guaranteeing an $O(\log n)$ (for n connected proxies) amortized object access cost for any workload.

This dissertation details the protocols for implementing consistent object operations; for accommodating the dynamic addition, involuntary disconnection and voluntary departure of Quiver proxies; for detecting misbehaving proxies; for the construction of a fault-tolerant network over the tree; and for restructuring the tree according to the workload to reduce access costs. These algorithms are evaluated using a combination of simulations and experiments performed on PlanetLab and isolated local clusters.

Acknowledgements

I would like to thank my advisor, Mike Reiter, for his help, support and encouragement through the years. The extent of what he has taught me reaches much further than just research. I will always be indebted to him. Thanks Mike!

I would also like to thank Chenxi Wang for all the enlightening discussions we have had over the years. My stay at Carnegie Mellon would not have been as pleasurable as it was, had it not been for friends like Alina, Florin and Charles—who give great advice on everything from new research ideas to life in Pittsburgh—and Scott Garriss who is always there to lend a hand.

I do not have the words to express my appreciation for my wife Tehniat, who I have so needed and who has always provided her unfaltering love and support. She has been all I could ever ask for. I would also like to thank my son Shahvaiz, who does not yet know why dad disappears for days on end, but he certainly misses me and has been as patient as you can expect from a two year old.

Finally, I would like to express great gratitude to my parents who have sacrificed so much over the years for my education. Their selfless devotion, untiring support, worldly advice and just pure hard work has enabled me to reach this stage in my life. Had I been with them, as I wished and they so deserved, this thesis would not have existed. Whatever its quality, it is indeed, a poor substitute.

Contents

Figures

Tables

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, numerous recent proposals attempt to scale dynamic web services by employing service proxies at the "edge" of the Internet (e.g., see Davis et al. [2004]; Tatemura et al. [2003] and the references therein). This approach has the potential to both distribute the operation processing load among the proxies, and to enable clients to access the service by communicating with nearby proxies, rather than a potentially distant centralized server.

A major challenge in realizing this architecture for *dynamic* web services, 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. Achieving even just serializability (Papadimitriou [1979]; Bernstein et al. [1987]) for operations executed at these proxies using standard replication approaches requires that a proxy involve either a centralized server or other (possibly distant) proxies on the critical path of each update operation. Strict serializability (Papadimitriou [1979]) via such techniques requires wide-area interactions for reads, as well.

This dissertation describes a system called Quiver that addresses this challenge and allows edge proxies to perform consistent operations on shared objects, without overloading some centralized coordination point, and without contacting distant proxies for each operation. We also present extensions to Quiver that provide a reliable and self-optimizing communication network for Quiver proxies. In this chapter, we briefly motivate the design of these protocols and summarize our contributions.

1.1 Consistent object access

This dissertation demonstrates an alternative to the traditional replication based approaches for 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 an update operation, or an operation involving multiple objects, a proxy uses the tree to *migrate* each involved object to itself and then performs the operation locally—thus serializing these operations and achieving the required consistency semantics. Though this approach incurs the expense of object migration for update and multi-object operations—and so is 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 are 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. Furthermore, 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 key to realizing this benefit. Given the well-known 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 the Quiver paradigm are those that involve either large amounts of data that would be expensive to send to the server or compute-intensive operations that would overload the server, since Quiver disperses the load induced by client operations across proxies rather than centralizing it in the server.

Perhaps the most obvious drawback of object migration is increased sensitivity to proxy disconnections: If a proxy disconnects while holding an object, either because the proxy fails, because it can no longer communicate with its parent, or because its parent disconnects, then operations that it recently applied to the object may be lost. 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 it reflects become durable.

For these durable operations, Quiver can implement either serializability (Papadimitriou [1979]; Bernstein et al. [1987]) or strict serializability (Papadimitriou [1979]). The only difference in the two modes is in how single-object reads are handled. In neither case do single-object reads 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 (Herlihy and Wing [1990]) for applications that employ only single-object operations.

Chapter 2 details these protocols and reports on an evaluation based on experiments performed on PlanetLab (Chun et al. [2003]) and a local isolated cluster. The PlanetLab experiments measured the inherent costs of Quiver through microbenchmarks employing up to 70 nodes in different continents. We also compare Quiver's performance against a centralized implementation and show the drastic improvement for workloads that involve either compute intensive operations or geographic locality of reference. For the experiments performed on the local cluster, we implemented a network traffic classification service on top of Quiver that computes traffic classifiers from distributed data sources. Computing these classifiers is a computationally expensive operation, one that is not feasible to run on the resource-starved

[1]We do not address the failure of the server; we presume it is rendered fault-tolerant using standard techniques (e.g., Budhiraja et al. [1993]).

PlanetLab nodes. We again compare the performance of this application implemented using Quiver against an implementation based on a centralized server. Quiver outperforms the centralized service by orders of magnitude under various workloads for both update and read-only operations.

1.2 Detecting misbehaving proxies

The protocol for consistent object access migrates objects to proxies performing operations on these objects through other service proxies, and so the correctness of this protocol depends on these *intermediate* proxies behaving according to their specifications. This is a rather strong assumption, considering these service proxies are geographically distributed, often not even under direct administrative control of the entity offering the service.

We, therefore, developed an extension to Quiver that allows the honest proxies to detect misbehaving intermediate proxies attempting to violate consistency semantics provided by Quiver[2]. In particular, when an intermediate proxy is compromised, one attack it can mount is a *rollback attack*, in which it suppresses some operations from reaching other proxies. Byzantine fault-tolerant replication of the intermediate proxies can detect (e.g., Shin and Ramanathan [1987]; Alvisi et al. [2001]; Buskens and R. P. Bianchini [1993]) or mask (e.g., Lamport [1978]; Schneider [1990]; Reiter and Birman [1994]; Castro and Liskov [2002]; Cachin and Poritz [2002]; Yin et al. [2003]; Abd-El-Malek et al. [2005]) such misbehavior, but generally introduces significant performance, management and hardware costs. In our setting, such costs are unacceptable and so the proxies are forced to rely upon untrusted intermediaries. Prevention of the rollback attack, thus, becomes impossible, and the best one can hope for is detection through *fork consistency* (Mazières and Shasha [2002]; Li et al. [2004]).

In a nutshell, fork consistency ensures that if (the result of) an operation *op* is observed by two honest proxies, then these proxies perceive the same sequence of operations to have been performed to that point, i.e., up to

[2]Defending against other types of malicious behavior by proxies, e.g., corrupting service objects or denial of service is not addressed in this thesis.

and including *op*. The utility of this property is perhaps more clear when it is stated in the contrapositive: if the compromised intermediaries permit an operation to be visible to one proxy but suppress it from another, then subsequently these two proxies will never see any operation in common; these proxies are said to be "forked". Out-of-band communication between these proxies then enables them to detect that they are forked. For example, one proxy can apply an operation and the other proxy tests if it can view the effects of this operation. If not, the proxies confirm that they are forked and can initiate action to, e.g., identify the misbehaving intermediary.

This thesis explores an alternative formulation of fork consistency that offers qualitatively similar properties against the rollback attack, with substantially lower overhead. Implicit in the definition of fork consistency is that when the intermediaries are not misbehaving, operations are synchronized to yield a serial order of all operations, independent of the objects they involve. Our relaxation of fork consistency enforces a serial order on only operations on the same object, and as such, it permits operations that involve distinct objects to proceed with the full concurrency that would otherwise be allowed by Quiver's consistency protocols. At the same time, it remains that proxies whose views of an object are forked can easily detect if they are, simply by seeing if one can modify the object in a way that the other can see the modification. (On the other hand, proxies who are unsuspectingly forked by an intermediary on one object might be permitted to interact normally via another object, and so a fork might persist longer in our model without detection.) In addition, the cryptographic mechanisms that we employ in our implementation are substantially simpler and less expensive than those utilized in previous implementations. The fork consistency implementation of SUNDR (Li et al. [2004]; Mazières and Shasha [2002])—the seminal and most comprehensive treatment of fork consistency to date—involves a digital signature per operation, along with overhead per operation that is linear in the total number of participants. In contrast, our solution has neither of these as characteristics, and in fact employs collision-resistant hashing as its common-case cryptographic operation.

Chapter 3 provides the detailed algorithms employed by Quiver to guarantee fork consistency. Our approach is very generic in that it is not tied to other mechanisms in Quiver and is equally applicable to other domains. In particular, we show that our formulation of fork consistency permits the implementation of a *file service* that offers qualitatively similar defense to rollback as SUNDR but with substantially better performance. We use analysis and trace-driven simulations to show that the costs per operation are significantly reduced by our approach. We additionally describe how our approach can be integrated into other distributed object sharing protocols, such as peer-to-peer directory and mutual exclusion protocols (Demmer and Herlihy [1998]; Raymond [1989]; Helary et al. [1994]; Naimi et al. [1996]).

1.3 Recovering from partitions

Quiver employs a rooted tree as the communication network between the service proxies. Using a tree has several advantages: First it allows the design of very simple protocols for locating and retrieving (together referred to as migrating) service objects, and for serializing these migrations. Second, an overlay tree that preserves the geographic distance between proxies—i.e., places geographically nearby proxies close to each other in the tree—enables the migration protocols to exploit locality of reference. Finally, a rooted tree naturally defines a primary partition—the one containing the root. This allows the object management protocols to define proxies in the primary partition as "connected"—and so these must be able to access service objects—and proxies in other partitions as "disconnected"—in which case the object access requests initiated and the operations performed by these proxies can be ignored until they reconnect. Thus, the server always has control over the service objects, even when some proxies disconnect, partitioning the network.

The downside of using a tree structure, however, is its vulnerability to proxy and link failures. In particular, a single proxy or link failure can partition the tree and make the service objects unavailable to the proxies that end up in partitions not containing the root. We address this issue through an extension that efficiently builds a logical fault-tolerant communication network overlayed on top of Quiver's distributed tree.

The overlay network constructed by our distributed algorithm is an *expander*. Expanders are an important class of graphs that have found applications in the construction of error correcting codes (e.g., Sipser and Spielman [1996]), de-randomization (e.g., Ajtai et al. [1983]), and in the design of fault-tolerant switching networks (e.g., Pippenger and Lin [1992]). The fault tolerance of expanders (Goerdt [1998]; Bagchi et al. [2004]) is precisely what motivated their use in Quiver. Our algorithm starts with proxies connected in a tree and proceeds to add edges to achieve an expander. Since explicit constructions of expanders are generally very complex, we present a construction that "approximates" a d-regular random graph, i.e., a random graph in which every proxy has almost d neighbors. A d-regular random graph is, with an overwhelming probability, a good expander (Friedman [1991]). We prove that our approximation achieves comparable expansion.

The contributions of this work rest primarily in three features. First, our algorithm is completely distributed. Though expander graphs have been studied extensively, distributed construction of expander networks remains a challenging problem. Our algorithms use only local information at each proxy that consists of the identities of the proxy's neighbors in the tree. A direct consequence of this is scalability—our algorithm is capable of generating expanders efficiently even with a large number of proxies. We bootstrap this algorithm using a novel technique that allows a proxy to sample other proxies uniformly at random from the tree with low message complexity.

Second, our algorithm adapts to joins, leaves and failures of proxies. Previous attempts at distributed construction of random expanders (Law and Siu [2003]; Pandurangan et al. [2003]; Gkantsidis et al. [2004]) try to construct d-regular random graphs where every node (a proxy, in our setting) has exactly d neighbors. Such graphs are difficult to construct and maintain in a dynamic distributed setting; e.g., most of these constructions require nodes to propagate their state to other nodes before leaving the network. We follow a more pragmatic approach, in that we only require that proxies have "close" to d neighbors. In doing so we define a new class of random graphs which we call (d, ϵ)-regular random graphs. These graphs give us more flexibility in dealing with the dynamic nature of our network, while still achieving fault tolerance.

Finally, we present a novel distributed algorithm that uses the overlay expander to keep the underlying tree connected in the presence of faults. This algorithm works on a "best-effort" basis—in most cases the algorithm is able to successfully patch the tree when proxies fail, however, in the unlikely event of a large fraction of proxies failing simultaneously, the algorithm might not succeed. In these cases we require some of the proxies to re-join the tree using the default mechanism, e.g., by contacting the root.

Chapter 4 details these algorithms. We report simulation results that show the effectiveness of the overlay expander in tolerating failures, and the cost of our algorithm in terms of its message complexity.

1.4 Restructuring for performance

Since proxies locate and retrieve objects through the tree, the worst case performance of these algorithms is proportional to the diameter (longest path between two proxies) of the tree. The trivial solution to make the tree "flat" (every proxy is a child of the server) does not scale well—the server becomes a bottleneck. Therefore, the performance of operations in Quiver that involve migrating or copying objects through the tree, can benefit from a distributed mechanism that would restructure the tree to reduce its diameter, while keeping a low fixed degree and while preserving the location-aware structure in the tree.

We, therefore, discuss a final extension to Quiver that employs a novel distributed algorithm, called *flattening*, that improves the performance of object migration and copying protocols. In particular, flattening achieves three properties. First, it brings proxies frequently accessing the same objects—and thus frequently accessing each other for migration or copying of objects—closer to each other in the tree. Workloads in several applications are known to exhibit locality, in the sense that proxies that have communicated in the past are likely to communicate again in the future;

such applications can benefit greatly from flattening. Note that in a degree-constrained tree (e.g., a k-ary tree), optimizing the access between a pair of proxies (by bringing them close to one another), could conflict with optimizing for another pair of proxies. This situation is further complicated due to the distributed nature of our algorithm: each proxy is only aware of its neighbors in the tree, and has no information about the remaining tree topology. Flattening employs a distributed algorithm that utilizes only local information at each proxy, and finds a balance among conflicting optimization decisions by restructuring for a particular pair of proxies while at the same time preserving the effects of recent restructuring decisions made for other pairs.

Second, flattening has a tendency to reduce the diameter of the tree, without ever explicitly balancing the tree. In particular, it reduces the diameter of the component of the tree that spans proxies involved in recent operations. Therefore, if the workload shows no locality—e.g., if each proxy accesses a proxy chosen uniformly at random from all proxies in the tree—then flattening reduces the diameter of the whole tree, since in this case the component containing frequently accessed proxies would span most of the tree.

Finally, the restructuring steps are all local, i.e., each restructuring step at a proxy involves either only direct neighbors or at most neighbors of neighbors (proxies two hops away from each other) in the tree. This allows simple implementation of local policies at each proxy, e.g., a subtree containing proxies geographically close to each other could enforce a policy that prevents a geographically distant proxy from entering this subtree, as the tree is restructured. Furthermore, this local restructuring enables proxies to easily update their routing information (used for object migration), to reflect the new tree topology.

The restructuring algorithm is discussed in Chapter 5. We analytically prove that flattening incurs a worst-case $O(\log n)$ amortized cost per flattening operation. Since the cost of this restructuring is directly tied to the cost of accessing another proxy—restructuring is performed along the path between the two proxies—the worst case cost of proxy accesses closely follows the $O(\log n)$ amortized performance of restructuring. We also report empirical results from tests performed on PlanetLab that validate this analysis. We further implemented a flood-based access mechanism that runs on a tree, and allows proxies to access other proxies in the tree. We present results that demonstrate the performance of this flood-based protocol using our self-optimizing tree, and compare them to those obtained by running the same protocol on a randomly generated static tree over the same set of proxies. The flood-based protocol shows significant performance gains when utilizing the flattening algorithm, and shows the generality of our scheme and its potential applications to several different protocols.

1.5 Structure of this document

This thesis addresses issues that are relevant to the deployment of consistent and scalable edge services. However, the different problems addressed here have their origins in several different domains like consistency protocols, systems research, database theory and graph theory. This diversity requires an independent treatment of the background and related research for each of these problems. In addition, our solutions need to be evaluated against other solutions developed in the particular research area. For each issue addressed, we therefore treat the corresponding related work and the evaluation independently along with the description of the specific problem and our proposed solution.

The remainder of this thesis is organized as follows: Chapter 2 describes and evaluates the protocols employed by proxies for performing consistent operations on service objects, and for the availability of service objects even when some proxies disconnect. Chapter 3 discusses the extensions that can be used by honest proxies to detect misbehaving intermediate proxies that attempt to violate consistency of object accesses, discusses the application of our approach in settings other than Quiver, and compares the performance costs incurred by our approach against existing solutions. Chapter 4 details our extensions that reduce the downtime experienced by disconnected proxies by constructing a fault-tolerant network on top of the overlay tree, that can be used by proxies for efficient reconnection. The algorithm to heuristically restructure the tree for performance gains in workloads that exhibit locality is described and evaluated in Chapter 5. We finally conclude in Chapter 6.

2 Consistent Object Sharing

This chapter discusses the object management protocols that allow Quiver proxies to perform consistent operations on service objects, achieving either serializability or strict serializability. We also present protocols that ensure continuous access to service objects by connected proxies, even when other proxies disconnect or voluntarily leave the service. An extensive evaluation of these protocols is presented through experiments performed on PlanetLab and a local cluster.

2.1 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 (e.g., Luo et al. [2002]; Li et al. [2003]; Amiri et al. [2003]; Li and Dong [1994]; Plattner and Alonso [2004]; Olston et al. [2005]; Rabinovich et al. [2003]). 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, e.g., group multicast (e.g., Amir et al. [2002]) or epidemic (e.g., Holliday et al. [2003]) 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 (e.g., Gao et al. [2005] 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 (e.g., Bernstein and Goodman [1981]; Özsu and Valduriez [1996]).

Our approach departs from these paradigms by migrating objects to proxies for use in updates. As discussed in Section 1.1, 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 circumstances 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 spans decades, e.g., Nuttall [1994]; Milojičić et al. [2000] offer useful surveys. Many previous studies in object migration have drawn from motivation similar to ours, namely co-locating processing and data resources. However, to our knowledge, the approaches in Quiver for managing migration and object reads, and for recovering from disconnections, are novel. The only work of which we are aware that applies object migration to dynamic web services (Sivasubramanian et al. [2005]) 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 Raymond [1989]; Naimi et al. [1996]; Demmer and Herlihy [1998]. These protocols allow nodes arranged in a tree to retrieve shared objects and perform operations atomically. While these approaches achieve scalability and consistency, they do not address failures. Our approach also enables consistent multi-object operations and optimizations for single-object reads that are not possible in these prior algorithms.

2.2 System model 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, e.g., 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 discussion, we treat the disconnection of a proxy as permanent, or more specifically, a disconnected proxy may re-join the service but with a re-initialized state. In an execution, a proxy that joins but does not disconnect (though 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 with several fields, including an *identifier* field $o.\mathsf{id}$ and a *version* field $o.\mathsf{ver}$. We refer to object instances with the same identifier as versions of the same object. Any operation that produces an object instance o as output takes as input the previous version, i.e., an instance o' such that $o'.\mathsf{id} = o.\mathsf{id}$ and $o'.\mathsf{ver} + 1 = o.\mathsf{ver}$.

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 connected processes perceive the operations in Durable (and no others) to be executed sequentially. More precisely, we present two variations of our algorithm. One enforces *serializability* (Papadimitriou [1979]; Bernstein et al. [1987]): all connected processes perceive the operations in Durable to be executed in the same sequential order. The other enforces an even stronger property, *strict serializability* (Papadimitriou [1979]): the same sequential order perceived by processes preserves the real-time order between operations.

2.3 Object management

We begin by describing a high-level abstraction in Section 2.3.1 that enables our solution, and then discuss the implementation of that abstraction in Section 2.3.2. Sections 2.4 and 2.5 describe how this implementation enables Quiver proxies to perform service operations.

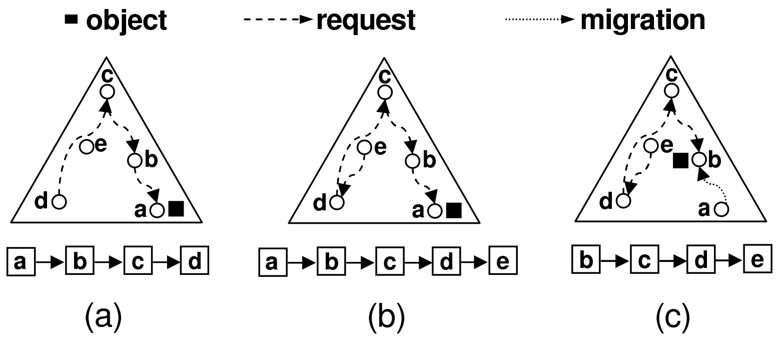


Figure 2.1. (a) distQ with processes a, b, c and d. (b) e appends itself to distQ by sending a retrieve request to d. (c) When a completes its operation, it migrates the object to b and drops off distQ.

2.3.1 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 2.1-(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 2.1-(c). This distributed queue ensures that the object is accessed sequentially.

A process performs an operation involving multiple objects by migrating each involved object via its distQ to itself. 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.

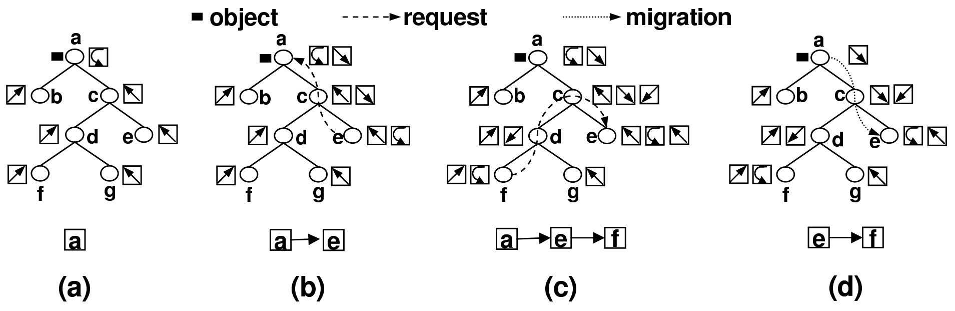


Figure 2.2. Squares at a process represent its localQ; left-most square is the head. Initially a has the object. e requests from a, f requests from e, and a migrates the object to e.

2.3.2 distQ implementation

The core of our algorithm implements 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 in the tree. 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 (Figure 2.2-(a)).

When a process p receives a retrieve request for the object with identifier id from its neighbor q, it forwards the request to the tail of p.localQ$[id]$ and adds q to the end of p.localQ$[id]$ as the new tail. Thus, the tail of p.localQ$[id]$ now points in the direction of the new tail of distQ$[id]$, which must be in the direction of q since the latest retrieve request came from q; see Figures 2.2-(b) and 2.2-(c). When a process p receives a migrate message containing the object, it removes the current head of p.localQ$[id]$ and forwards the object to the new head of p.localQ$[id]$. This ensures that the head of p.localQ$[id]$ points in the direction of the new head of distQ$[id]$, see Figure 2.2-(d).

Pseudocode for this algorithm is shown in Figure 2.3. 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 2.3; we describe 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 other proxies are added to the tree. For each object identifier id, p initializes a local queue p.localQ$[id]$ by enqueuing p if p is the server and p.parent otherwise. In addition, for each object identifier id, the server p initializes its copy of the object, p.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.parent. Variable names without the "p." prefix represent state local to its thread. In order to synchronize these threads, the pseudocode of process p employs a semaphore[1] p.sem$[id]$ per object identifier id, used to prevent the migration of object p.objs$[id]$ to another process before p is done using it. p.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.

2.3.3 Migrating one object

The routing of retrieve requests for objects is handled by the doRetrieveRequest function shown in Figure 2.3. When p executes

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

```
doRetrieveRequest($from, id, prog$)                      /* Invoked locally on request by $from$ */
1.  $\langle q, prog' \rangle \leftarrow p$.localQ[$id$].tail              /* $q$ made the last request for this object */
2.  $p$.localQ[$id$].addToTail($\langle from, prog \rangle$)          /* 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 migrate to requesting process */
6.  else                                                 /* If I did not last request this object ... */
7.      send (retrieveRequest : $p, id$) to $q$                /* ...then forward to who last requested it */

doMigrate($id$)                                          /* Invoked locally for handling migration */
8.  $p$.localQ[$id$].removeFromHead()                        /* Owner not towards current head now */
9.  $\langle q, prog \rangle \leftarrow p$.localQ[$id$].head                /* Being migrated towards $q$, the new head */
10. if $q = p$                                            /* If I requested this object ... */
11.     $prog$                                              /* ...then execute program registered earlier */
12. else if $q = p$.parent                                 /* If parent requested this object ... */
13.     $IDs \leftarrow \{id' : id \overset{p.\mathsf{Deps}}{\Longrightarrow} id'\}$                       /* ...then find objects this one depends on */
14.     $Objs \leftarrow \{p.\mathsf{objs}[id'] : id' \in IDs\}$             /* ...collect all these objects */
15.     $DepSet \leftarrow p.\mathsf{Deps} \cap (IDs \times IDs)$             /* ...and their dependency relations */
16.     send (migrate : $p$.objs[$id$], $Objs, DepSet$) to $q$  /* ...send everything to parent */
17.     $p.\mathsf{Deps} \leftarrow p.\mathsf{Deps} \setminus DepSet$                   /* ...remove dependencies for future */
18. else                                                 /* If a child requested this object... */
19.     send (migrate : $p$.objs[$id$], $\emptyset, \emptyset$) to $q$           /* ...then just migrate this object */

Upon receiving (retrieveRequest : $from, id$)            /* Request for $id$ received from $from \neq p$ */
20. doRetrieveRequest($from, id, \perp$)                    /* Invoke doRetrieveRequest on $from$'s behalf */

Upon receiving (migrate : $o, Objs, DepSet$)             /* $o$ is migrated and depends on $Objs$ */
21. $p$.objs[$o$.id] $\leftarrow o$                                /* Save the migrated object $o$ */
22. foreach $o' \in Objs$                                  /* For each *copied* object ... */
23.     $p$.objs[$o'$.id] $\leftarrow o'$                           /* ... save the copied object */
24. $p.\mathsf{Deps} \leftarrow p.\mathsf{Deps} \cup DepSet$                        /* Update the local dependency relation */
25. doMigrate($o$.id)                                     /* Invoke doMigrate for $id$ */
```

Figure 2.3. Object management pseudocode for process p

doRetrieveRequest($from, id, prog$), it adds $\langle from, prog \rangle$ to the tail of p.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 2.4.) 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 (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 migration begins. Note that p invokes doRetrieveRequest not only when it receives a retrieve request from another process (line 20), but also to migrate the object for itself.

Migrating an object with identifier id is handled by the doMigrate function. Since the head of $p.\mathsf{localQ}[id]$ should point toward the current location of the object, p must remove its now-stale head (line 8), and identify the new head q to which it should migrate the object to reach its future destination (line 9). If that future destination is p itself, then p runs the program $prog$ (line 11) that was stored when p requested the object by invoking $\mathsf{doRetrieveRequest}(p, id, prog)$; again, we defer discussion of $prog$ to Section 2.4. 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 2.3.4.

2.3.4 Object dependencies

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

It is not necessary for each process p to track $\Rightarrow$ between all object instances in order to compute the appropriate identifier set IDs. Rather, each process maintains a binary relation $p.\mathsf{Deps}$ between object identifiers, initialized to $\emptyset$. If p performs an update operation op such that an output $p.\mathsf{objs}[id] \overset{op}{\Rightarrow} p.\mathsf{objs}[id']$, then p adds (id, id') to $p.\mathsf{Deps}$. In order to perform $\mathsf{doMigrate}(id)$ to $p.\mathsf{parent}$, p determines the identifier set IDs as those indices reachable from id by following edges (relations) in $p.\mathsf{Deps}$—reachability is denoted $\overset{p.\mathsf{Deps}}{\Longrightarrow}$ in line 13 of Figure 2.3—and copies both $\mathit{Objs} = \{p.\mathsf{objs}[id']\}_{id' \in \mathit{IDs}}$ (line 14) and $\mathit{DepSet} = p.\mathsf{Deps} \cap (\mathit{IDs} \times \mathit{IDs})$ (line 15) along with the migrating object (line 16). Finally, p updates $p.\mathsf{Deps} \leftarrow p.\mathsf{Deps} \setminus \mathit{DepSet}$ (line 17), i.e., to remove these dependencies for future migrations upward.

Upon receiving a migration from a child with copied objects Objs and dependencies DepSet, p saves Objs in $p.\mathsf{objs}$ (lines 22–23) and adds DepSet to $p.\mathsf{Deps}$ (line 24). Note that the server (root of the tree) need not maintain any dependencies, since it always migrates or copies the objects downwards in the tree.

2.4 Update and multi-object operations

In order to achieve our desired consistency semantics, for each object we enforce sequential execution of all update and multi-object operations 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 2.5) that dominate many workloads.

2.4.1 Invoking operations

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

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

Process p then executes $prog_k$. Note that $prog_k$ requests id_k and, once that is migrated, $prog_{k-1}$ is executed (at line 11 of Figure 2.3). This, in turn, requests id_{k-1}, and so forth. Once id_1 has been migrated, $prog_0$ is executed. This performs op and then updates the dependency relation $p.\mathsf{Deps}$ (see Section 2.3.4) with the new dependencies introduced by op. Finally, $prog_0$ executes a V operation on the semaphore for each object, permitting it to migrate. Viewing the semaphores $p.\mathsf{sem}[id_1]$, ..., $p.\mathsf{sem}[id_k]$ as locks, $prog_k$ can be viewed as implementing strict two-phase locking (Bernstein et al. [1987]). So, to prevent deadlock, $id_1, \ldots, id_k$ must be arranged in a canonical order.

2.4.2 Update durability

A proxy 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 2.3.4) to the server, allowing each process p on the path to save o if $p.\mathsf{objs}[o.\mathsf{id}].\mathsf{ver} < o.\mathsf{ver}$. That said, doing so per update would impose a significant load on the system, and so our goals (Section 2.2) do not require this. Rather, our goals require only that a proxy forces its updates to be durable when it leaves the tree (Section 2.6.2), so that operations by a proxy that remains connected until it leaves are durable. Operations performed at the server are durable because our model assumes that the server never fails.

2.5 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 multi-object operations from Section 2.4. We provide correctness arguments for both versions of our protocols in Section 2.7.

2.5.1 Serializability

Due to the serial execution of update and multi-object operations (Section 2.4), single-object reads so as to achieve serializability (Bernstein et al. [1987]) can be implemented with local reads—i.e., a process p performs a read involving a single object with identifier id by simply returning $p.\mathsf{objs}[id]$.

2.5.2 Strict Serializability

Recall that all update and multi-object read operations involving the same object are performed serially (Section 2.4). 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 multi-object operations in $\mathsf{distQ}[id]$. However, this would require op to wait for the completion of the concurrent update and multi-object operations (those performed by processes preceding p in $\mathsf{distQ}[id]$).

A more efficient solution is to request the latest version from the process at the head of $\mathsf{distQ}[id]$—the process that is the current "owner" of the object with identifier id. Our algorithms already provide a way to route to the head of distQ$[id]$, using localQ$[id]$.head at each process. Thus a read request for id follows p.localQ$[id]$.head at each process p until it reaches a process p' such that either p'.localQ$[id]$.head $= p'$ (i.e., p' holds the latest object version), or p'.localQ$[id]$.head $= p''$ is the process that forwarded this read request to p'. In the latter case, p' forwarded p'.objs$[id]$ to p'' in a migration concurrently with p'' forwarding this read request to p' (since p''.localQ$[id]$.head $= p'$ when p'' did so), and so it is safe for p' to serve the read request with p'.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 2.6.1), we require that the object be passed through the tree to the highest process in the path from p' to p, i.e., the lowest common ancestor p'' of the initiator and responder of the read request. After receiving the object in response to the read request, p'' directly sends the object to p (the initiator) outside the tree. Note that since the requested object is copied upwards in the tree from p' to p'' (unless $p' = p''$), any objects that the requested object depends upon, must also be copied along using the techniques described in Section 2.3.4.

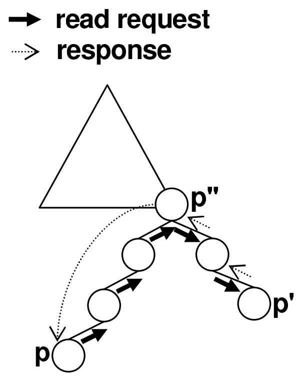


Figure 2.4. p initiates a single object read request that reaches p'. p' sends the response through the tree to the highest process p'' in the path. p'' then copies the requested object directly to p outside the tree.

2.6 Object availability in dynamic conditions

Our algorithms make no assumptions about how proxies join the tree, and this mechanism can be tailored to application needs—e.g., in our experiments (Section 2.9) we construct a minimum spanning tree of proxies based on network latencies. Here we detail how to adapt our algorithm to address disconnections (Section 2.6.1) and proxies leaving voluntarily (Section 2.6.2).

2.6.1 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), informs its subtree of the disconnection, to enable proxies to reconnect (after reinitializing) if desired. Of concern here, however, is that 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 $\mathsf{distQ}[id]$. In this case, it must be ensured that the connected processes preceded by a disconnected proxy in $\mathsf{distQ}[id]$ continue to make progress. To this end, all occurrences of the disconnected proxies in $\mathsf{distQ}[id]$ are replaced with the parent p of the uppermost disconnected proxy q, see Figure 2.5.

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 child q. Second, as we will see below, in our algorithm p need only take local actions to replace the disconnected proxies; as such, this is a very efficient solution. Third, in case the head of $\mathsf{distQ}[id]$ 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 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 Sec-

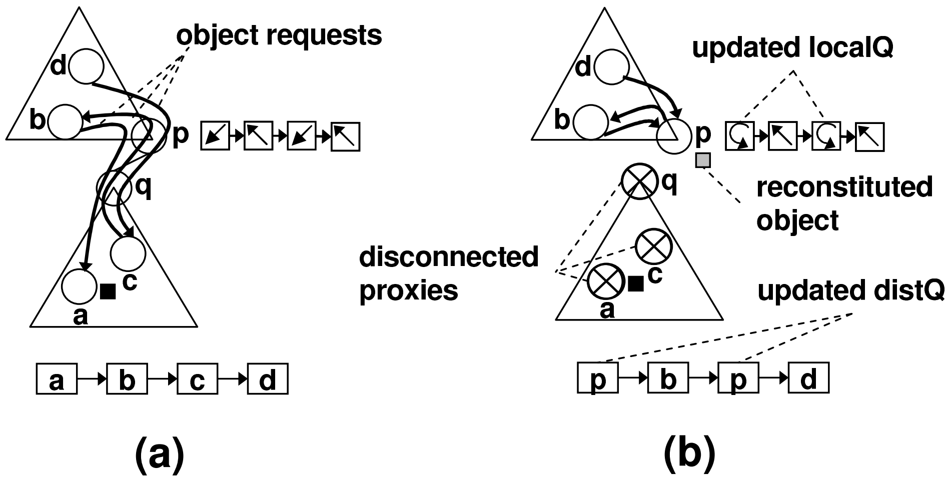


Figure 2.5. q loses contact with parent p and its subtree disconnects. p replaces disconnected proxies in distQ and reconstitutes the object so b and d can make progress.

tion 2.5). Note that in case of multiple simultaneous disconnections, only one connected process—that which has the object in its disconnected child's subtree—will reconstitute the object from its local copy, becoming the new head of distQ$[id]$.

The pseudocode that p executes when its child q disconnects is the childDisconnected(q) routine in Figure 2.6. 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 proxy is now blocked at p, see Figure 2.5-(b). For each of these requests that are now blocked at p, p creates and run-enables a new thread (lines 10–11 of Figure 2.6) 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). p also responds to any strictly-serializable single-object read requests initiated by a still-connected process and forwarded by p to q, and for which p has not observed a response (not shown in Figure 2.6).

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

Qreplace($id, q$)                                        /* Invoked locally by $p$ */
7.  foreach $i = 1, \ldots, p$.localQ[$id$].size $- 1$         /* For each element of localQ, except tail */
8.      if $p$.localQ[$id$].elmt[$i$] $= \langle q, * \rangle$          /* If it points to $q$ ("$*$" is wild-card)... */
9.          $p$.localQ[$id$].elmt[$i$] $\leftarrow \langle p, \|V(p$.sem[$id$])$\| \rangle$   /* ...change it to point to myself */
10.         $t \leftarrow$ new thread($\|P(p$.sem[$id$]); doMigrate($id$)$\|$) /* ...wait for object and then migrate it */
11.         $t$.enable()                                   /* ...run-enable this thread */
12. if $p$.localQ[$id$].tail $= \langle q, * \rangle$                  /* If the tail is the disconnected child... */
13.     $p$.localQ[$id$].tail $\leftarrow \langle p, \|V(p$.sem[$id$])$\| \rangle$      /* ...then just replace it by myself */
```

Figure 2.6. Disconnection-handling at p

2.6.2 Leaves

In order to voluntarily leave the tree, a proxy p must ensure that any objects in the subtree rooted at p are still accessible to connected processes, once p leaves. Furthermore, outstanding retrieve and (strictly serializable) read requests forwarded through p must not block as a result of p leaving the tree.

If p is a leaf node, then it serves any retrieve and strictly serializable read requests blocked on it, migrates any objects held at p to its parent (Section 2.3.2), forces its updates to be durable (Section 2.4.2), and departs. If p is an internal node then it forces its updates to be durable, serves any strictly serializable single-object read requests, and then chooses one of its children q to *promote*. The promotion updates q's state according to the state at p, and notifies other 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.parent, p.children, p.localQ[ ], p.objs[ ] (or, rather, only those object versions that q does not yet have) and p.Deps. When q receives these, it updates its parent, children, objects and object dependencies according to p's state.

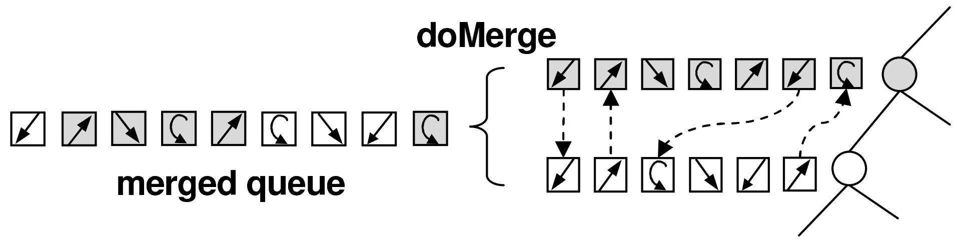


Figure 2.7. Queue merge. Shaded and unshaded elements are in parent's and child's localQ, respectively. Dashed arrows are from a skipped element to the element added next.

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 2.7. Figure 2.8 presents the pseudocode used by a promoted child q to merge q.localQ$[id]$ with its parent p's p.localQ$[id]$ for each identifier id, as the parent voluntarily leaves the service. In order 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 2.8), 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 2.6.

At this point, any outstanding retrieve requests that were initiated by p (represented by instances of p in p.localQ$[id]$) now appear as initiated

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

doMerge(localQ, localQ$'$, $p, p'$, $mergedQ$)        /* Invoked locally; merges the two queues */
14. while localQ.hasElements()                      /* If the first queue is not empty... */
15.   $\langle r, prog \rangle$ $\leftarrow$ localQ.removeFromHead()   /* ...then remove its head */
16.   if $r \neq p'$                                  /* If head does not point to other process... */
17.     $mergedQ$.addToTail($\langle r, prog \rangle$)        /* ...then add it to the tail of $mergedQ$ */
18.   else                                          /* If head points to the other process... */
19.     doMerge(localQ$'$, localQ, $p', p$, $mergedQ$)  /* ...then skip it and recurse with other queue */
```

Figure 2.8. Pseudocode at q for its promotion

by q since all instances of p from p.localQ[id] are copied to $mergedQ$ and then replaced by q. Retrieve requests forwarded through p but not q now appear as forwarded through q, as all elements in p.localQ[id] are added to $mergedQ$, except instances of q. Retrieve requests forwarded through q and not p appear as before since q.localQ[id] elements are all added to $mergedQ$, except instances of p. Finally, requests forwarded through both p and q now appear as forwarded through only q, due to skipping elements in p.localQ[id] that point to q and vice-versa.

2.7 Correctness

This section proves the correctness of Quiver's protocols for achieving serializability and strict serializability.

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

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

Proof. Among the connected processes, the localQ.tail pointers implement the Arrow protocol (Demmer and Herlihy [1998]), augmented to account for disconnections as described in Section 2.6.1. This protocol ensures that per object identifier, migrations among connected processes occur serially. We do not recount the proof of this fact here; interested readers are referred to, e.g., Demmer and Herlihy [1998]; Herlihy et al. [2001]; Kuhn and Wattenhofer [2004]. This fact implies that there is a unique object instance bearing a particular identifier and version number that is migrated by connected processes.

As a result, the existence of two object instances o_i and o_j with the same object identifier and version number implies that at least one of op_i and op_j, say op_i, was performed by a proxy that disconnects. Moreover, the proxy that performs op_i must disconnect prior to migrating o_i (or having it copied due to the migration, Section 2.4, or the strictly serializable read, Section 2.5.2, of an object instance that depends on o_i) out of the subtree that disconnects. Otherwise, the lowest connected ancestor in the tree, who reconstitutes the object following the disconnection, would reconstitute o_i or a later version (see Section 2.6.1). So, o_i is never visible in the connected component containing the server. This also implies that for each object instance o such that $o \Rightarrow o_i$ (o depends on o_i), o is not visible in the connected component: if o is migrated (or copied) up to the connected component, then o_i (or a later version) must be copied along with it (see Section 2.3.4). Therefore, none of the other object instances produced by op_i are visible in the connected component, as each of these instances depends on o_i. As a consequence, none of the instances produced by op_i is ever read by a connected process and so $op_i \overset{\text{rf}}{\not\rightarrow}_* op_k$. □

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

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

(*1*) *If* op_i *was executed at a connected process, then* $op_i \in$ Durable.

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

Below we prove that the operations in Durable are serializable when the updates and multi-object reads are implemented as in Section 2.4 and single object reads are implemented as in Section 2.5.1. Furthermore operations in Durable are strictly serializable for the other incarnation of our system, i.e., when the updates and multi-object reads are implemented as in Section 2.4 and single object reads are implemented as in Section 2.5.2. Note that in either case, operations in Durable are in fact durable, since "losing" an update could violate serializability or strict serializability. Finally, note that Lemma 1 holds for either incarnation of our system.

2.7.1 Proof of serializability

Multi-version Serializability theory Our system maintains multiple versions of the same object at the same time (although not at the same process), therefore we argue the serializability of our algorithms using multi-version serializability theory (Bernstein et al. [1987]). 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 (version precedence, $\xrightarrow{\text{ver}}$). *The version precedence relation, denoted $\xrightarrow{\text{ver}}$, is defined for operations as follows: For distinct operations op_i, op_j and op_k, let op_k read an object instance o_j produced by op_j and op_i produce an object instance o_i such that $o_i.\mathsf{id} = o_j.\mathsf{id}$. If $o_i.\mathsf{ver} < o_j.\mathsf{ver}$ then $op_i \xrightarrow{\text{ver}} op_j$, otherwise $op_k \xrightarrow{\text{ver}} op_i$.*

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

In order to prove that the set S of operations is serializable, it is both necessary and sufficient to prove that $MVSG(S)$ is acyclic [Bernstein et al., 1987, Theorem 5.4].

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

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

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

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

Then assign the timestamp to *op* as follows:

$$\mathit{ts}(op) \leftarrow \mathit{maxTs}(op) + 1$$

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

$$\forall o \in \mathit{input}(op) \cup \mathit{output}(op) : \ \mathit{ts\_recent}(o) \leftarrow \mathit{maxTs}(op) + 1$$

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

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

Proof. Since updates and multi-object operations migrate instances with the same identifier serially and there is a unique sequence of instances with the same identifier (Lemma 1), p_j cannot migrate o_j before p_i invokes a $V(p_i.\mathsf{sem}[id])$. This $V(p_i.\mathsf{sem}[id])$ is performed only after p_i completes op_i (last statement of prog_0, see Section 2.4) and therefore, only after assigning $\mathit{ts\_recent}(o_i') \leftarrow \mathit{ts}(op_i)$, where o_i' is either o_i or its newer version in case op_i updates o_i. Since *ts_recent* can only grow and op_j is assigned a timestamp greater than *ts_recent* of all instances in $\mathit{input}(op_j)$, $\mathit{ts}(op_j) \geq \mathit{ts}(op_i) + 1$. □

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

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

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

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

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

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

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

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

Lemma 5. *Adding single-object read operations as described in Section 2.5.1 (and the corresponding* $\xrightarrow{\text{rf}}$ *and* $\xrightarrow{\text{ver}}$ *edges) from* $\mathsf{Durable}$ *to the acyclic* $MVSG(\mathsf{Durable}')$ *does not introduce any cycles.*

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

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

Theorem 2. 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 (Bernstein et al. [1987]). □

2.7.2 Proof of strict serializability

In order to achieve strict serializability, the updates and multi-object reads are performed in the same way as for the serializable version of our protocols, i.e., updates and multi-object reads involving the same objects are serialized, see Section 2.4. However, single object reads are performed as in Section 2.5.2, instead of reading the local copy of the object as in the serializable algorithm. Strict serializability requires that all (connected) processes perceive the operations to be in the same sequential order (serializability) and furthermore, this sequential order must preserve the real-time order between operations, i.e., if op_i completes before op_j is invoked, then op_i must precede op_j in the sequential order perceived by the processes. We first prove that the subset $\mathsf{Durable}'$ of $\mathsf{Durable}$ containing all the updates and multi-object reads in $\mathsf{Durable}$, and no other operations, is strictly serializable. We then prove that the single object reads implemented as described in Section 2.5.2 do not violate strict serializability.

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

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

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

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

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

Proof. Assume for contradiction that op_j performed by process p_j completes before op_i performed by process p_i completes. Then there must exist an identifier $id \in ID_{in}(op_i) \cap ID_{in}(op_j)$ such that p_j migrates an instance $o_j : o_j.\mathsf{id} = id$ for input to op_j before p_i migrates an instance $o_i : o_i.\mathsf{id} = id$

for input to op_i: otherwise, if p_i migrates o_i before p_j migrates o_j, then p_j must wait for p_i to release the object with identifier id, which is done via a $V(p_i.\mathsf{sem}[id])$ only after p_i completes op_i (last statement of $prog_0$, see Section 2.4). Since p_j migrates instance $o_j : o_j.\mathsf{id} = id$ for input to op_j before p_i migrates instance $o_i : o_i.\mathsf{id} = id$ for input to op_i, it must be the case that $ts(op_j) < ts(op_i)$ (Lemma 2), a contradiction. □

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

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

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

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

Theorem 3. *$MVSSG$(*Durable′*) is acyclic.*

Proof. We know that $MVSG$(Durable′) is acyclic (Theorem 1). Assume for contradiction that $MVSSG$(Durable′) has a cycle. Now construct $MVSSG$(Durable′) by adding each real-time order edge to $MVSG$(Durable′) one by one. Let $op_i \xrightarrow{\text{rt}} op_j : op_i, op_j \in$ Durable′ be the first edge that creates a cycle during this construction of $MVSSG$(Durable′). This cycle is possible only if there already existed a path from op_j to op_i before adding

$op_i \xrightarrow{\text{rt}} op_j$ to the graph being constructed. This path consists of $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ver}}$ edges from $MVSG(\mathsf{Durable}')$, and any $\xrightarrow{\text{rt}}$ edges added to the graph before adding $op_i \xrightarrow{\text{rt}} op_j$. We can therefore state the relation between op_j and op_i as $op_j \xrightarrow{\text{rf,ver,rt}}_* op_i$. This implies that op_j completes before op_i completes (Corollary 1), and therefore, $op_i \overset{\text{rt}}{\not\rightarrow} op_j$, a contradiction. □

The remaining part of the proof deals with the single object read operations as implemented in Section 2.5.2, and proves that these operations do not introduce any cycles when added to the acyclic $MVSSG(\mathsf{Durable}')$.

Lemma 8. *Let $op_i \in \mathsf{Durable} \setminus \mathsf{Durable}'$ be a single object read of an object with identifier id implemented as described in Section 2.5.2. Let $op_j \in \mathsf{Durable} : id \in ID_{out}(op_j)$ be the most recent such update operation to complete before op_i is invoked, and let $o_j : o_j.\mathsf{id} = id$ be an instance produced by op_j. Then op_i either reads o_j or an instance $o_k : o_k.\mathsf{id} = id, o_k.\mathsf{ver} > o_j.\mathsf{ver}$.*

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

Corollary 2. *Let $op_i \in \mathsf{Durable} \setminus \mathsf{Durable}'$ be a single object read of an object with identifier id implemented as described in Section 2.5.2, and let $op_j \in \mathsf{Durable}$. If $op_i \xrightarrow{\text{ver}} op_j$, then op_j completes after op_i is invoked, i.e., $op_j \overset{\text{rt}}{\not\rightarrow} op_i$.*

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

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

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

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

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

Case 2 ($op_{i-1} \xrightarrow{\text{rf}} op_i \xrightarrow{\text{ver}} op_{i+1}$)**:** In this case, both op_{i-1} and op_{i+1} produce an instance of the object read by op_i, therefore $op_{i-1}, op_{i+1} \in$

$\mathsf{Durable}'$. Furthermore, $\mathit{ID}_{in}(op_i) \in \mathit{ID}_{in}(op_{i-1}) \cap \mathit{ID}_{in}(op_{i+1})$. Let id be the identifier of the object read by op_i. Then op_{i-1} produces an instance $o_{i-1} : o_{i-1}.\mathsf{id} = id$ that is read by op_i, and op_{i+1} produces an instance $o_{i+1} : o_{i+1}.\mathsf{id} = id$ and $o_{i+1}.\mathsf{ver} > o_{i-1}.\mathsf{ver}$ (hence the relation $op_i \xrightarrow{\text{ver}} op_{i+1}$). Therefore, the process performing op_{i-1} must have migrated an instance with identifier id for op_{i-1} before an instance with identifier id was migrated by the process performing op_{i+1}, and so $ts(op_{i-1}) < ts(op_{i+1})$ (Lemma 2). So op_{i-1} completes before op_{i+1} completes (Lemma 6).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.8 Online bookstore on the edge

This section describes the use of Quiver in an example setting—that of an online bookstore. We model our online bookstore application according to the TPC-W benchmark (TPC [2002])—an industry standard benchmark representing an e-commerce workload, specifically on an online bookstore. We first give an overview of TPC-W discussing the state required and the operations supported. We then discuss how this state can be divided into objects, and how these objects may be shared by edge proxies.

2.8.1 TPC-W overview

TPC-W is a transaction processing benchmark specifically for an online bookstore. Here we present a brief overview of its relevant aspects.

TPC-W defines different interactions with the bookstore. These include administrative tasks like adding or removing books; customer registration; searching for books according to different keys such as author, title, subject etc; listing detailed information about a particular book; creating and querying a customer's shopping cart; purchasing items in the shopping cart, resulting in the generation of orders; querying a customer's orders; maintaining and displaying the list of bestsellers and new products for different subjects.

The relevant state in the bookstore mainly consists of the following data structures.

– **Item table:** The item table maintains information about books in the bookstore. Each row of the table represents one item storing information such as the title, publisher, author, price, quantity in the stock etc. Rows in the item table are added and removed by the administrative interactions defined in the workload. Once added, elements in the row typically remain unchanged except for some dynamic state like stock quantity, availability and price etc. The item table is accessed during a number of read-only (e.g., searches) and read-write (e.g., order placement) interactions.

- **Customer table:** The customer table contains information about the registered customers. It is updated when a new customer registers with the bookstore.

- **Order table:** The order table contains a list of orders submitted by the users. Each row contains the order identifier, customer identifier that placed this order, the total cost for the order, shipping and billing addresses etc. A row is added to the order table when a user purchases items in her shopping cart.

- **Shopping cart:** A shopping cart data structure is created for a registered customer on request. The customer can then add items to the shopping cart, view her shopping cart at a later time and purchase items in the shopping cart.

- **Bestseller lists:** A bestseller list is maintained for each subject. This list consists of the 50 top items based on the volume sold in the last 3,333 orders.

- **New products lists:** This is a list of a number of newly added items to the bookstore.

TPC-W also specifies different workload mixes, ranging from a *browsing mix* consisting of 95% read-only interactions (e.g., searching for books, displaying bestseller lists etc) to an *ordering mix* where 50% of the interactions involve updates to some part of the state (e.g., creating a new shopping cart or placing an order).

2.8.2 Object definitions

The first step in porting the TPC-W online bookstore to the Quiver setting is dividing its state into objects, such that objects are not too large, each operation involves a small fraction of objects, and objects exhibit locality. Keeping these conditions in mind, we divide the state into the following objects; each object has a unique identifier:

– Item_S: An Item_S object is created whenever a new item is added to the bookstore by the administrator. This object maintains static information pertaining to the item, e.g., the title, subject, author and publisher.

– Item_D: An Item_D object is created whenever a new item is added to the bookstore. An Item_D object maintains dynamic information pertaining to a bookstore item, e.g., its availability or more specifically the number of items left in the stock. An Item_S object contains the identifier of the corresponding Item_D object.

– Customer: A Customer object is created whenever a new customer registers with the bookstore. A Customer object maintains information about a registered customer, including the name, address and other profile information. A Customer object also contains the identifiers of the customer's SCart and Order objects.

– SCart: An SCart object is created whenever a registered customer performs a "shopping cart interaction". An SCart object maintains a customer's shopping cart. It keeps information about the items added to the shopping cart, the total price of the items in the cart, shipping and billing addresses etc.

– Order: An Order object is created whenever a registered customer purchases items in her shopping cart. An Order object maintains information about a customer's order(s), e.g., the status of the order, the items purchased and total price.

- BSeller: A BSeller object per subject is created by the administrator; and is initially empty. It maintains the list of 50 best selling books on the subject in the last 3,333 sales.
- NProduct: A NProduct object per subject is created by the administrator and maintains a list of newly added books on this subject.
- MObject: A MObject object per "search key type", is a meta object that contains a list of item identifiers that can be searched using the particular key type, e.g., "author" is a possible key type, and so an "author MObject" contains the list of all authors, and for each author the list of item identifiers written by this author. Similar MObject objects may be defined for publisher, title, coarse price ranges, etc.

2.8.3 Bookstore interactions

We first categorize the objects according to the types of operations they support and the parties performing those operations, and then describe how these operations are to be performed while maintaining the required consistency guarantees, low client-perceived latency and distributing load across proxies.

Centrally updated objects

All objects that are created and updated by the administrator (at the server) and are only read by the proxies, fall in this category. These include the Item_S, NProduct and MObject objects. We note that Quiver is not optimized for such a workload, and therefore recommend other mechanisms for operations on these objects. There are well known techniques to share objects in a master/slave fashion, where the master performs the updates and propagates the changes to the slaves. The updates may be dispersed through gossip protocols, pushed out by the server to the proxies, or pulled by the proxies on demand. We focus more on the objects that can be updated by any proxy at any time, and therefore better suit the Quiver paradigm.

Update-anywhere objects

All objects that can be updated as a result of client interactions fall in this category. An Item_D object requires an update to the "quantity in stock" field after every sale of the corresponding item. A Customer object is created whenever a new customer registers, and can be read and updated later by the customer. An SCart object is created when a registered customer creates a shopping cart, and can later be read and modified by the customer. An Order object is created when a customer places an order, and can later be read by the customer. A BSeller object for a subject needs to be updated (ideally) after each sale of an item belonging to that subject. All of these objects and the operations can be implemented through Quiver.

Discussion

We have divided most of the update-anywhere objects on a per-customer basis, e.g., the Customer, SCart and Order objects all belong to the same customer. Therefore, operations on these objects are expected to exhibit strong locality—unless the customer travels to a different location or the redirection mechanism redirects the customer to a distant proxy (e.g., due to a misconfiguration)—and will benefit from Quiver migrating these objects to the corresponding proxies.

The consistency requirements for the BSeller object are not too strong, and therefore it can be updated outside of the critical path of the client's order placement operation. Also note that since the BSeller object is created on a per-subject basis, operations that result in its update may exhibit geographic locality. Although not specified in the TPC-W benchmark, division of bookstore state according to the language—e.g., a BSeller object for books in Chinese—could further increase the locality of reference in the workload.

The Item_D object contains, among other fields, the quantity of the item left in the stock. Ideally, this field should be updated with each client operation, and to maintain strong consistency, the update should be in the critical path of the client's order placement operation. However, note that this update is critical only when the quantity in the stock is low. Therefore, the proxies estimate the current quantity in the stock from their locally cached copy and the time elapsed since the local copy was updated, and migrate the $\mathsf{Item}_{\mathsf{D}}$ object in the critical path of client's order operation only if the quantity in stock is below some threshold.

Since Quiver does not guarantee durability of operations in the event of proxy disconnections, the proxies should copy the Order object along the path to the server, after one is created (see Section 2.4.2). This will ensure that an order placed by a customer never gets lost. Note however, that since the Order object is never updated after creation, the server need not perform any operations on the object, but rather acts as a persistent repository for these objects.

Lastly, we note that applications other than an online bookstore that use Quiver for performing operations on some objects, but use other techniques for other objects (e.g., for objects that are only updated at the server), must take further steps to ensure the atomicity of multi-object operations. In particular, if a multi-object operation involves objects that are managed using different techniques, then care must be taken to ensure the atomicity across these techniques. As an example, consider an object management strategy that keeps all centrally updated objects at the server. Then in order to perform a multi-object operation involving some of these centrally updated objects as well as objects managed by Quiver, the server should migrate the Quiver managed objects to itself and perform the operation locally.

2.9 Evaluation

We evaluated the performance of Quiver in two types of experiments. First, we performed experiments on PlanetLab (Chun et al. [2003]), involving a trivial service. These experiments include baseline tests to illustrate the inherent costs of our implementation (Section 2.9.2), and include experiments performed with specific workloads that better suit Quiver's paradigm, i.e., workloads that either have geographic locality of reference (Section 2.9.4) or compute intensive operations (Section 2.9.3): a small amount of computation was induced with each update operation. We also compare Quiver's performance against a centralized server for these specific workloads.

We then performed a second evaluation using an application that partially motivated Quiver's design. This service enables the construction of network traffic models from distributed data sources, and involves operations that are computationally too intensive to be run on resource-starved PlanetLab nodes. Therefore, we performed these experiments on a 70-node cluster. Again we compare the performance of Quiver against a centralized server implementing the same service, see Section 2.9.5.

2.9.1 Experimental setup

Our system is implemented in Java 2 (Standard Edition 5.0), and is relatively un-optimized. It does, however, employ the following optimizations: First, nontrivial objects (objects in the application discussed in Section 2.9.5 approach a few hundred kilobytes) are stored and transmitted in compressed form. The memory and bandwidth savings due to compression far outweigh the computation costs. We use the LZO compression library[2], invoked from Java via the Java Native Interface. Second, processes make a local copy of an object before updating it, so they can serve reads while the object is being modified. This improves the performance of reads when updates are computationally expensive. If an object is not being modified, reads are served directly from the object. Finally, objects are kept in memory in their compressed forms and are decompressed when needed to perform operations.

When testing Quiver, the server and proxies were organized in a minimum spanning tree across the participating nodes, with the `ping` latency between two nodes (averaged in both directions) being their "distance". The node with the minimum median latency to all other nodes was selected as the root (server). In addition to these nodes, we used a *monitor* to control the experiments and measure the system performance. The monitor ran on a dedicated machine and communicated with all nodes in an experiment. In each experiment, each proxy notified the monitor upon joining the tree and

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

then awaited a command from the monitor to begin reads and updates. Upon receiving this command, the proxy performed operations for 100 seconds sequentially, i.e., beginning the next operation after the previous completed. Each operation was chosen to be a read with a probability specified by the monitor, and was an update otherwise; in 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, for each operation, the objects to be read or updated were chosen uniformly at random from all objects in the experiment. Also unless stated otherwise, there were 50 objects in an experiment, and the proxies performed single-object operations. The PlanetLab experiments used trivial objects, specifically integer counters that support increment (update) and read operations. For the compute intensive workloads in these experiments, we induced a small computation with each increment operation. After 100 seconds, each proxy reported to the monitor its average read latency, average update latency, the total number of operations it completed, and the time taken in different phases of the operation processing: For read operations we measured the time taken by the read request to reach the target process, and the time taken for the read response to reach the requestor from this target process. For updates, we measured the time taken by the update request to reach the target process, the time this request was blocked at this target process (this approximates the time spent by the requesting process in distQ), and the time taken to transfer the object from the current owner to the requestor. These measurements assumed that the clocks at the processes were somewhat synchronized, which was true in our experimental setup. Each experiment was repeated five times.

For the experiments where we compare Quiver against a centralized server, the centralized server 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 strict serializability. Achieving serializability requires only local reads, and so we do not include these in our experiments.

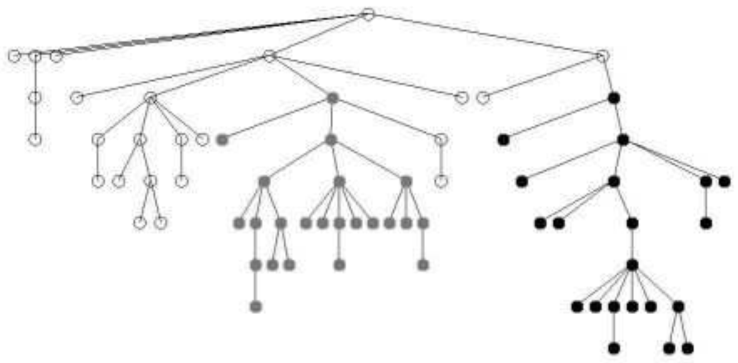


Figure 2.9. The overlay tree topology constructed as a minimum spanning tree of 70 PlanetLab nodes in North America. Black ovals represent west-coast proxies, white ovals represent mid-west proxies and gray ovals represent proxies on the east coast.

2.9.2 Baseline tests

The microbenchmarks employed 70 nodes (one server and 69 proxies) spread across North America, arranged in a minimum spanning tree, see Figure 2.9. We conducted several types of tests to evaluate Quiver's baseline performance. For each of these microbenchmarks, we report the latency and throughput of the operations performed by the proxies, as well as a breakdown of the time spent during various phases: e.g., the time spent by an update request to reach its target proxy, the time spent by this request waiting at the target proxy (denoted as the "update queue time" in the figures, since this approximates the time spent by the requestor proxy in distQ) and the time spent to transfer the object from its current owner to the requestor.

The first test varied the fraction of reads in the workload from 0 (only updates) to 1 (read-only workload). Update latency, read latency and overall throughput—number of operations (updates or reads) per second—are reported in Figure 2.10. The case with no reads can be viewed as indicative of the performance of updates in a configuration offering serializability only, i.e., where reads are performed locally by 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 2.5.2). 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 when moving up the tree, resulting in the sudden increase in read latency. This phenomenon is evident from the increase in the read transfer time (see the plot with the breakdown for read operations) as the fraction of reads in the workload reduces from 1 to 0.9.

The second test was performed by varying the number of objects, and the results are reported in Figure 2.11. A small number of objects resulted in higher contention and therefore higher update latency (as updates on the same object are serialized). This contention is visible in the breakdown graph for updates: the "update queue time" increased and made up a bigger percentage of the overall update time, as the number of objects decreased. For read latencies, note that the read requests always took somewhat longer compared to the read responses. This is due to sending the read responses outside the tree, at least part of the way back to the requestor.

The third test reported in Figure 2.12 was performed by varying the number of objects involved in each operation. All operations in this experiment are updates, since multi-object updates and reads are handled using the same algorithm. For each update, the proxy selects either one, two or three objects (depending on the experiment), out of the 50 total objects, uniformly at random, migrates these objects to itself (one by one, as described in Section 2.4) and then performs its operation, i.e., increments the counters. The latency of multi-object operations increases substantially with an increase in the number of objects involved in each operation, mainly due to the increased queue time (see the breakdown in Figure 2.12). This increase in the queue time is a direct consequence of the two-phase locking type approach used by our algorithms, and shows that Quiver's performance is best suited to workloads that are dominated by single-object operations.

The fourth microbenchmarks reported in Figure 2.13 vary the number of proxies involved in the experiment. For each experiment, the required number of proxies are chosen in a breadth-first order from the minimum spanning tree constructed initially: e.g., for an experiment involving thirty proxies, the top-most thirty proxies (proxies at the same level in the tree are selected from left to right) from the tree are selected and arranged in the same way as in the original 70 node minimum spanning tree. Note that in these experiments the load, i.e., the number of outstanding requests in Quiver at any point in time, varies. Therefore, we do not connect the data points reported for different numbers of proxies via lines (and use bar-charts instead), as these are not directly comparable: e.g., 70 proxies induce more load than 10 proxies, and so the corresponding throughputs in the two experiments are not comparable. These tests show the ability of Quiver to scale—a larger number of proxies can indeed handle more load, with an almost linear increase in throughput.

Our final microbenchmark evaluated the performance of Quiver during changes to the tree composition. Because accommodating leaves and joins is more involved than recovering from disconnections (which is a purely local algorithm, see Section 2.6.1), inducing leaves and joins yields a more conservative evaluation of our protocols when the tree is dynamic. In these tests, each proxy, after completing an operation and before starting the next, chose instead to leave the tree with probability Pr(leave). If it chose to not leave, then it commenced its next operation. Otherwise, it initiated the leave protocol described in Section 2.6.2. When a proxy left, it notified the monitor which then commands another proxy to join in its place. The new proxy commenced its operations as soon as it joined.

We calculate latency as before, though we modify the way in which we calculate throughput, because the time a proxy spent leaving or joining in place of another proxy should not count toward the throughput calculation. As such, during each run of the experiment, we calculate the time each proxy spent in the tree (actively performing operations), “pausing” this measurement when the proxy initiates a leave and “resuming” it when another proxy completes a join in its place. Then, we calculate throughput as the ratio of the total number of operations completed to the *average* time proxies spent in the tree.

These tests were performed using 40 PlanetLab nodes, actively performing operations at a time. The other 30 proxies from our set of 70 PlanetLab nodes were used as replacement for the leaving proxies. Results from these tests are reported in Figure 2.14. The results show that leaves and joins impact the latency and throughput numbers modestly. For example, setting Pr(leave) = .001, which induced between 16 and 20 leaves and joins (a large fraction of the total 40 proxies in the tree), resulted in latency increases of approximately 15% and 27% for single-object updates and reads, respectively, in a system with 50 objects. This was accompanied by a 13% decrease in throughput.

2.9.3 Compute-intensive workloads

Quiver should offer better performance than a centralized implementation case of compute intensive workloads, due to better dispersing compute load across proxies. To test this hypothesis, we artificially induced computation per update that, on a 1.4 GHz Pentium IV, took 22 ms on average.[3]

Figure 2.15 compares the centralized and Quiver implementations in this case, as the fraction of reads in the workload is varied. These results suggest that Quiver outperforms the centralized implementation for virtually all fractions of reads. The latencies and throughputs of the two implementations converge only once there are no updates in the system (i.e., the read fraction is 1), in which case obviously the computational cost of updates is of no consequence.

2.9.4 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 tests capture scenarios in which different objects are more popular in different regions, whereas the second tests capture scenarios in which different regions are active in different time intervals.

[3]This computation was the *Sieve of Eratosthenes* benchmark, repeated 40 times, each time finding all primes between 2 and 16384.

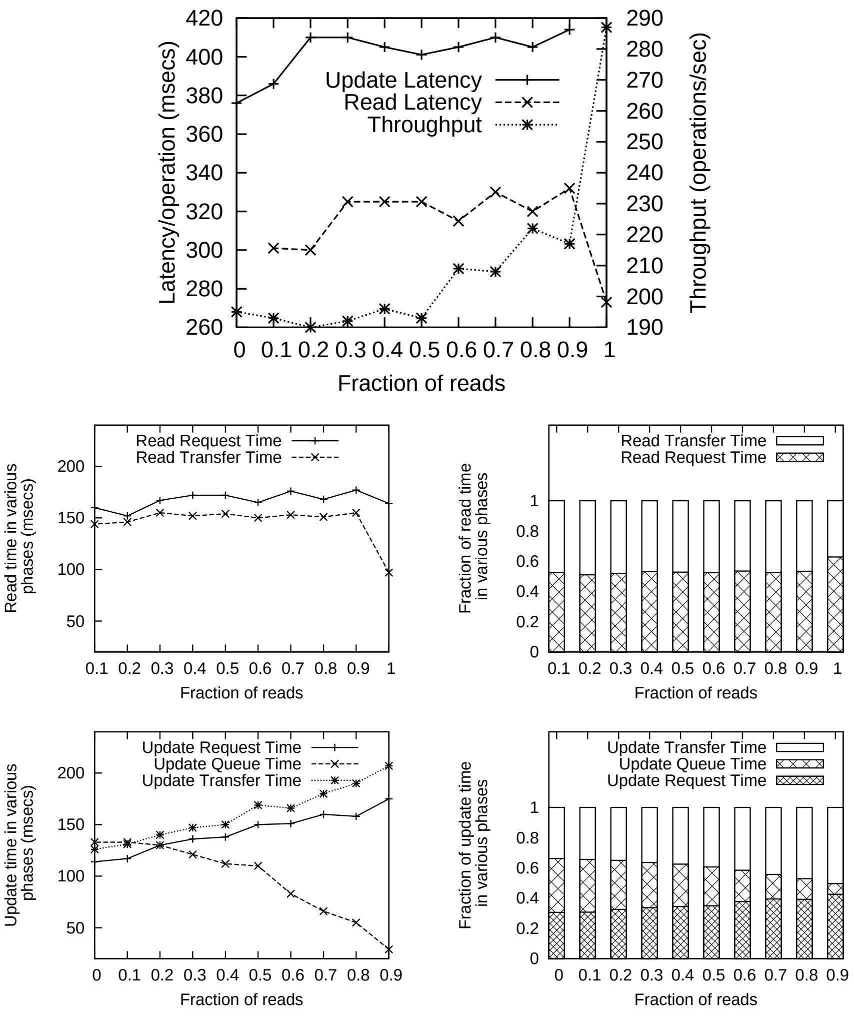


Figure 2.10. Microbenchmark results: Latency and throughput with varying fraction of reads in the workload. The time spent for reads and updates is broken down into the absolute and fraction spent in each phase (request and transfer for reads, request, queue and transfer for updates) of the operation.

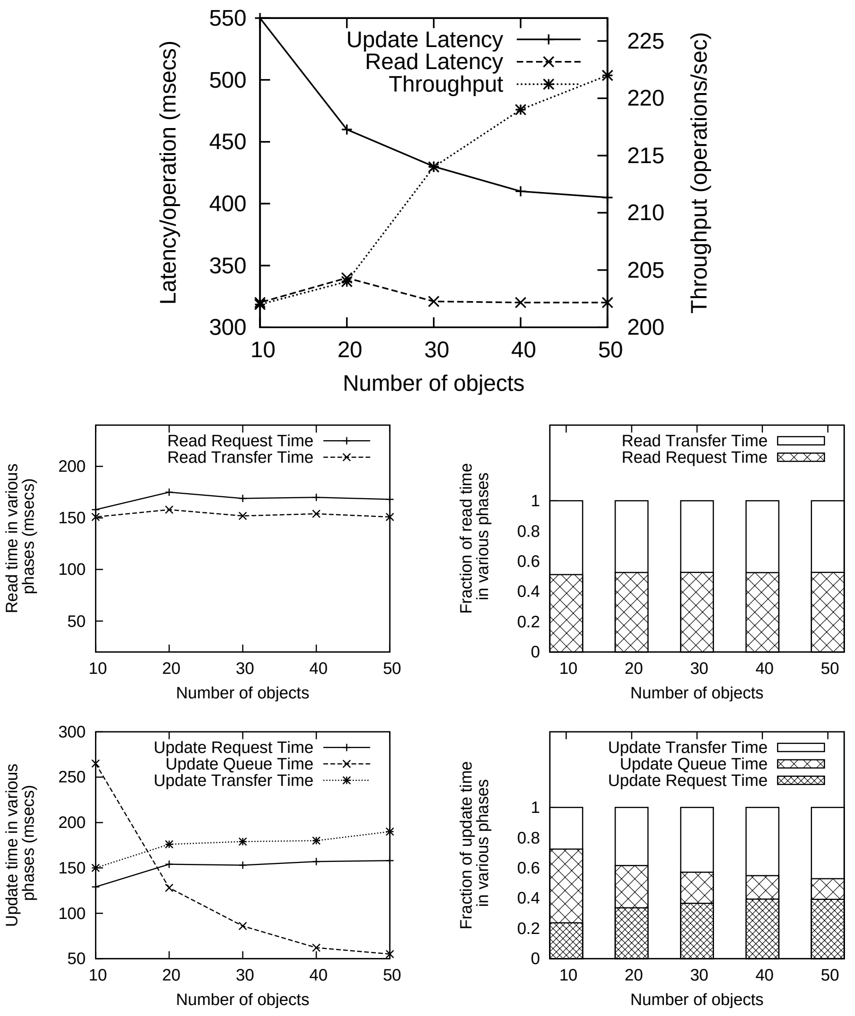


Figure 2.11. Microbenchmark results: Latency and throughput with varying number of objects shared by proxies. The time spent for reads and updates is broken down into the absolute and fraction spent in each phase (request and transfer for reads, request, queue and transfer for updates) of the operation.

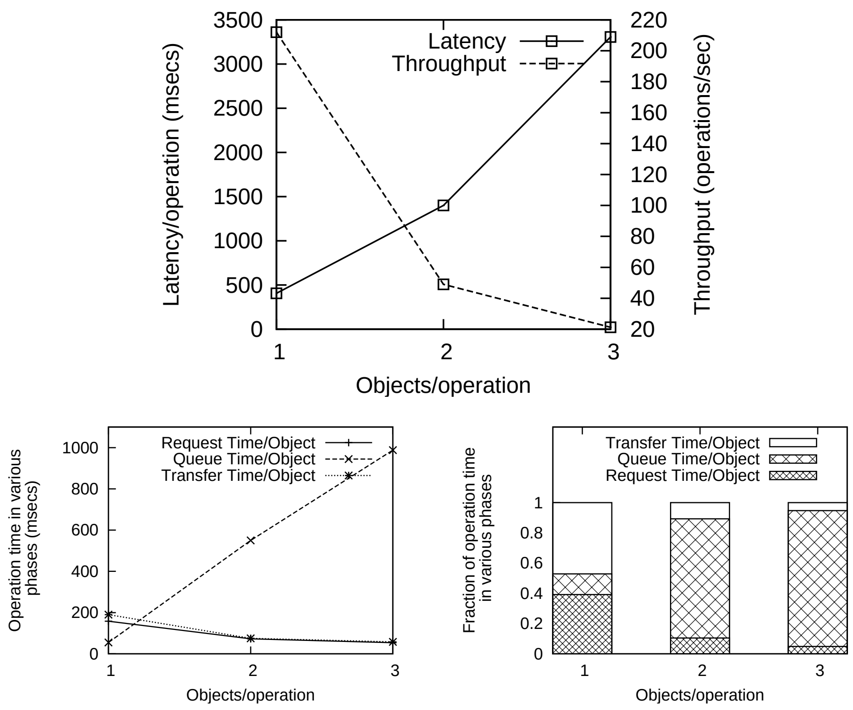


Figure 2.12. Microbenchmark results: Latency and throughput with varying number of objects involved per operation. The time spent for each operation is broken down into the absolute and fraction spent in each of the request, queue and transfer phases.

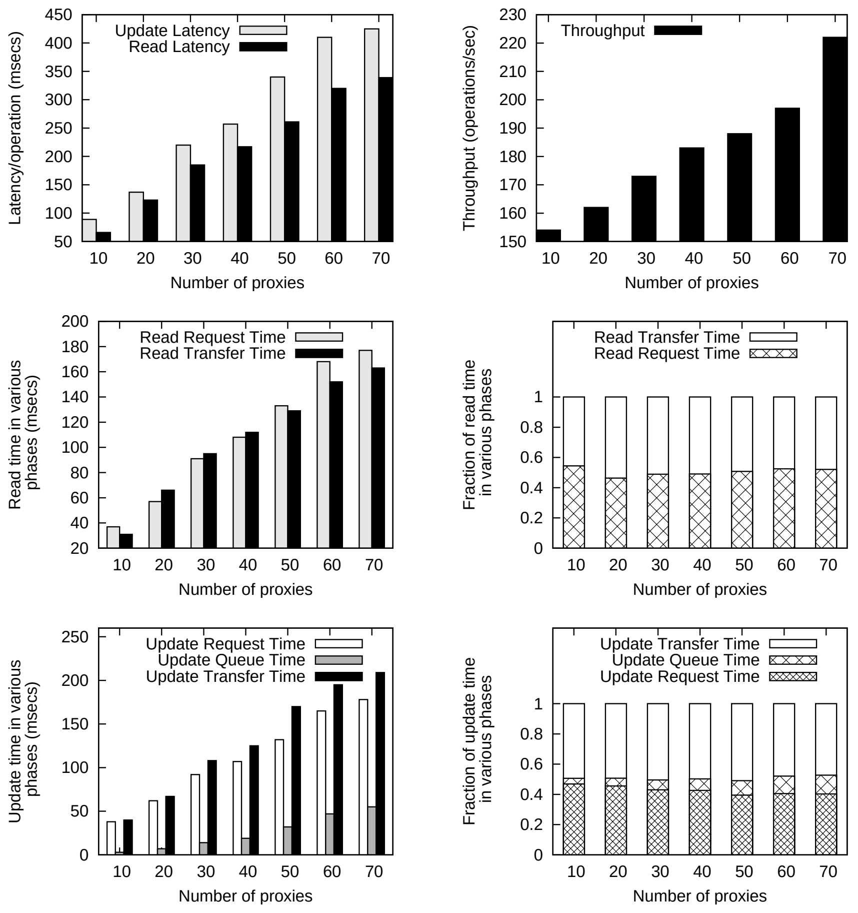


Figure 2.13. Microbenchmark results: Latency and throughput with varying number of proxies. The time spent for reads and updates is broken down into the absolute and fraction spent in each phase (request and transfer for reads, request, queue and transfer for updates) of the operation.

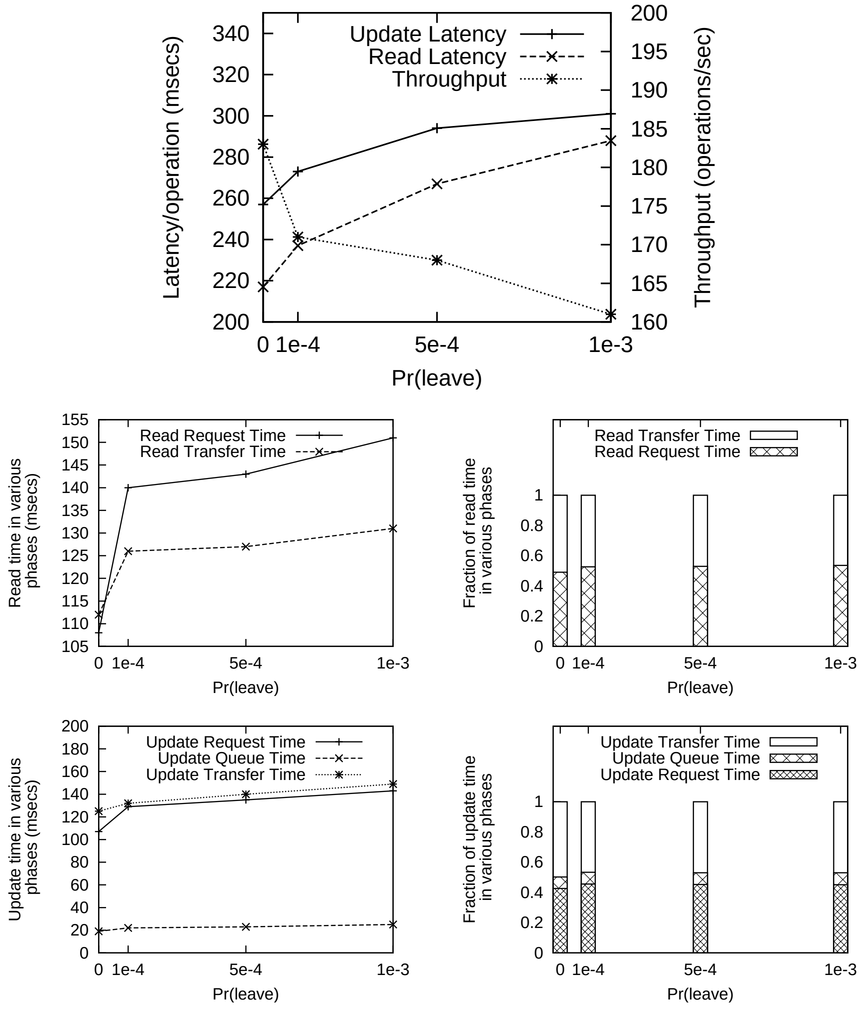


Figure 2.14. Baseline test results: Latency and throughput with varying probability with which each proxy leaves after completing each operation. Pr(leave) = .0001 resulted in 1–3 proxies leaving, Pr(leave) = .0005 resulted in 7–10 proxies leaving and Pr(leave) = .001 resulted in 16–20 proxies leaving in our tests, out of a total of 40 proxies. A new proxy joined in place of every departing proxy. The time spent for reads and updates is broken down into the absolute and fraction spent in each phase (request and transfer for reads, request, queue and transfer for updates) of the operation.

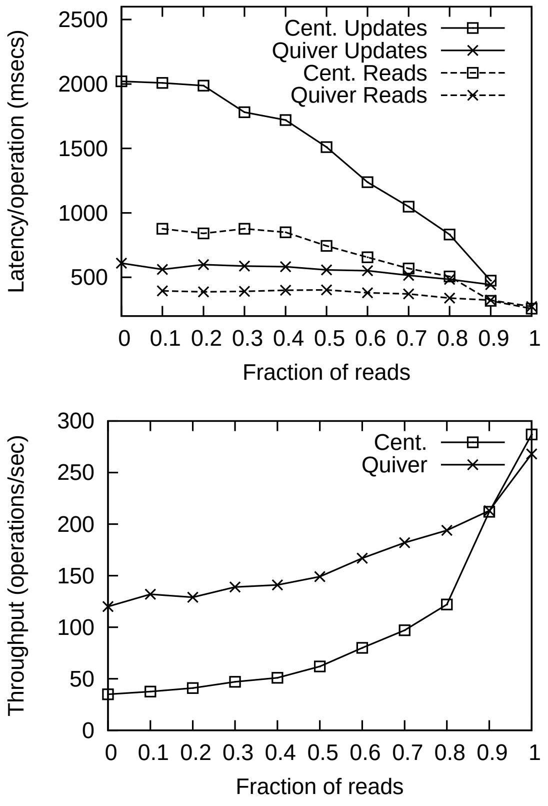


Figure 2.15. Compute intensive workload: Quiver's latency and throughput compared against the centralized server.

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 "focussed on" different objects. Specifically, the set of 50 objects were partitioned into $n = 5$ disjoint sets $\mathit{Objs}_0, \ldots, \mathit{Objs}_{n-1}$, each of size 10 objects. We defined permutations on $\{0, \ldots, n-1\}$ by

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

and distributions D_{east}, D_{west} and D_{central} satisfying $D_{\text{east}}(\pi_{\text{east}}(i)) = D_{\text{west}}(\pi_{\text{west}}(i)) = D_{\text{central}}(\pi_{\text{central}}(i)) = Z(i)$. Here, $Z : \{0, \ldots, n-1\} \to [0, 1]$ was a Zipfian 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_{central}.

The second test used 50 PlanetLab nodes divided into four roughly equal-sized groups located in China, Europe, and North American east and west coasts, see Figure 2.17. 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-second intervals of activity at times T, $T+\Delta$, $T+2\Delta$ and $T+3\Delta$, respectively, where $\Delta \in [0 \text{ seconds}, 100 \text{ seconds}]$. So, when $\Delta = 0$ seconds, the intervals completely overlapped, but when $\Delta = 100$ seconds, the intervals were disjoint. Note that in these tests, the overall load on the system fluctuated during the test as node groups "woke" and "slept". As such, the throughput numbers we report are the average of the throughput observed by the four regions during their "awake" intervals.

The results of these two tests demonstrate that as the popularity bias grows in the first tests (Figure 2.16) and as the offset Δ grows in the second tests (Figures 2.18)—in each case increasing the geographic locality of requests per object—Quiver surpasses a centralized implementation in both latency and throughput. The second tests further reveal a practical optimization enabled by Quiver's design: since each proxy typically only communicates with a small number of neighbor proxies in the tree (we saw a maximum degree of 5 in our experiments), 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 it serves. This cost is negligible for links with smaller round-trip times, e.g., the maximum round-trip time among the North American nodes was 59 ms. But, this cost was more profound on long-haul links: 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 tests even when all nodes were "awake" at the same time ($\Delta = 0$).

2.9.5 Network traffic classification service

Today, network traffic characterization is an area of active research, including techniques to classify traffic as that of a particular application (e.g., see Moore and Zuev [2005]; Karagiannis et al. [2005] and the references therein) or as anomalous and thus indicative of an attack (e.g., Lee et al. [1999]; Zanero and Savaresi [2004]). Much work suggests that models for performing this classification can be built more effectively by aggregating contributions from many networks (e.g., Yegneswaran et al. [2004]; Bailey et al. [2005]; Jiang and Xu [2004]). We are thus building a service through which networks can contribute traffic records toward the construction of classifiers for network traffic. In this application, the server is run by some coordination center, the proxies are at the various networks that contribute records, and the shared objects are the classifiers. Our application supports an arbitrary number of classifiers, e.g., parameterized by application (port),

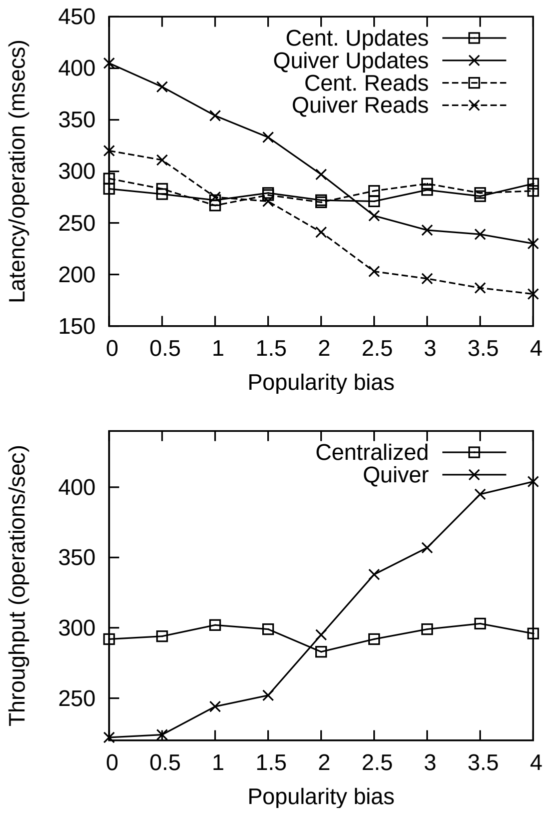


Figure 2.16. Object popularity bias workload: Latency and throughput for Quiver and the centralized server.

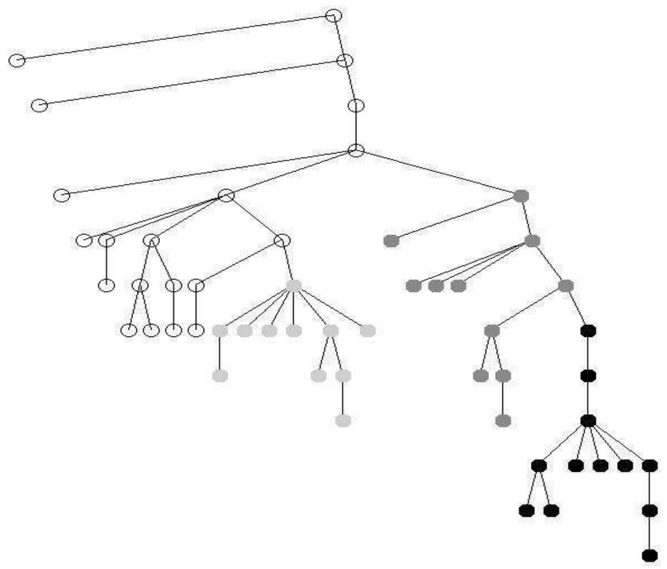


Figure 2.17. The overlay tree topology constructed as a minimum spanning tree of 50 PlanetLab nodes spread across different continents. Light gray, white, dark gray and black ovals represent proxies in China, Europe, and the North American east and west coasts respectively.

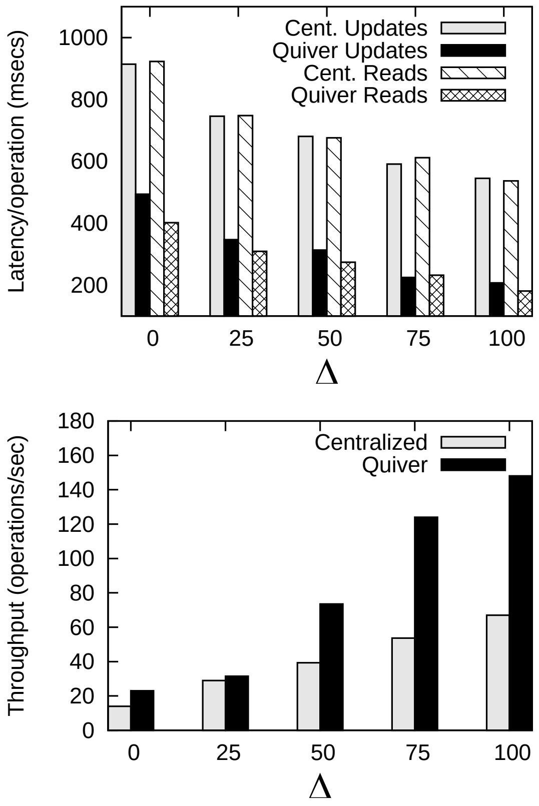


Figure 2.18. Regional activity workload: Quiver's latency and throughput compared against the centralized server.

attack attributes ("attack" vs. "normal") or other characteristics. Strictly serializable semantics ensure reads see the latest classifiers, in addition to offering atomic updates.

The classifiers that our service presently implements are support vector machines (SVMs) (Cortes and Vapnik [1995]), a popular learning mechanism used for classification and regression and that is particularly well-suited to data with many features. More specifically, we use a variant of traditional SVMs called incremental SVMs (Ralaivola and d'Alché-Buc [2001]; Cauwenberghs and Poggio [2001]; Fung and Mangasarian [2002]) that allow the models to be constructed incrementally as new contributions are received. SVMs have previously been used to characterize network traffic (Eskin et al. [2002]; Honig et al. [2002]; Mukkamala and Sung [2003]), though not in a distributed setting. Our implementation uses the LIBSVM library[4] to construct SVM models from raw data.

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 raw data consisted of pre-recorded connection records, each consisting of 41 features related to the connection including the application protocol, the transport protocol, protocol flags, connection length etc. Each update operation updated a classifier with 500 new records. Figure 2.19 shows the CDFs of the sizes of the resulting models (compressed and uncompressed) and the time required for updates on a 1.4 GHz Pentium IV.

Due to the compute intensive nature of these experiments obvious from Figure 2.19, these experiments could not be performed on PlanetLab. Instead, we conducted these experiments on a local isolated cluster with (up to) 70 nodes, each with an Intel P-IV 2.8GHz processor, 1GB of memory and an Intel PRO/1000 network interface card. The machines were connected with an HP ProCurve Switch 4140gl specified with a maximum throughput of 18.3Gbps.

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

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

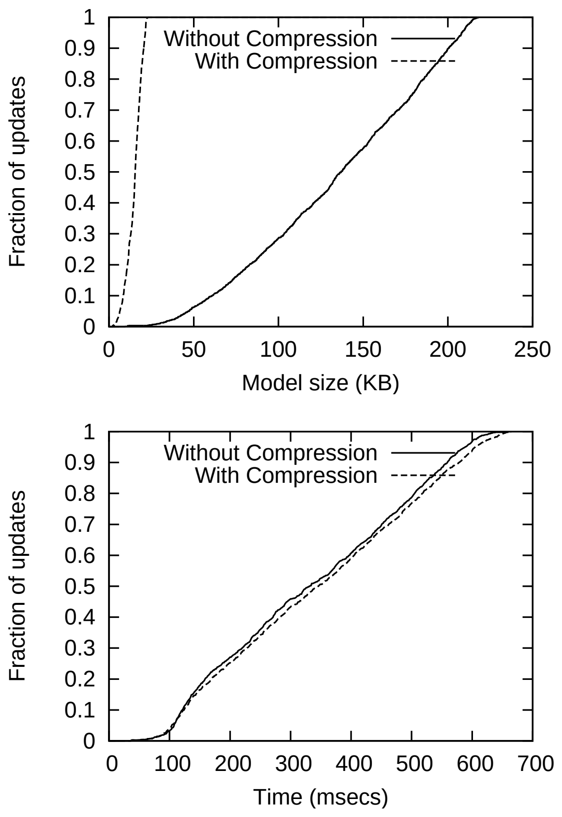
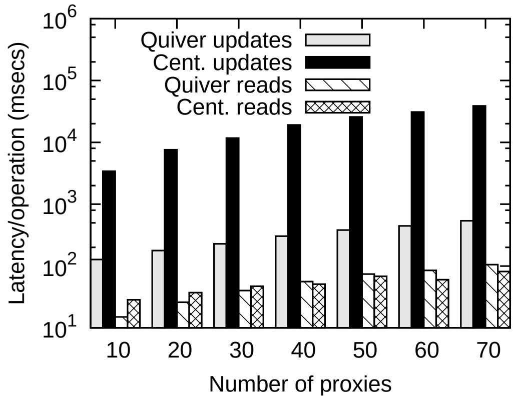


Figure 2.19. CDF of model sizes and update times

We compared the performance of Quiver 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 proxy sent 500 connection records to the server who updated the corresponding classifier with this data. The server responded to a read operation by sending the requested classifier back to the proxy. The optimizations discussed for the Quiver implementation—compressing objects, serving reads from copies and keeping objects in memory—were preserved in this implementation.

Our first experiment evaluated single-object operations. Figure 2.20 plots the results; note that the vertical axes of these graphs are log scale. Our experiments showed that Quiver's update latency and throughput were dramatically superior to those of the centralized server, by roughly an order of magnitude or more in all cases. Moreover, the trends suggest that as the number of proxies increases past our ability to test, the performance difference for updates might become even more pronounced since, e.g., the update throughput is trending downward for the centralized server but upward for Quiver. The performance improvement for updates yielded by Quiver were the result of harnessing proxy cycles to contribute to the computation. Moreover, the read performance of each implementation was comparable.

For the types of traffic models we envision, we expect single-object operations to be the norm in this application. However, multi-object updates could naturally arise, e.g., to incorporate the same traffic records into distinct but related models (e.g., a model for BitTorrent and a model for all file sharing protocols in aggregate). Thus, in our experiments, a multi-object update incorporates the same data into multiple models. Our experiments with multi-object operations are illustrated in Figure 2.21. As the number of objects per operation increased, the greater contention for migrating objects impacted Quiver's throughput; Nevertheless, Quiver still achieved much better update latency than the centralized server; the centralized server's processor was the bottleneck in this case. On the other hand, the centralized server's multi-object reads outperformed Quiver's multi-object reads, since



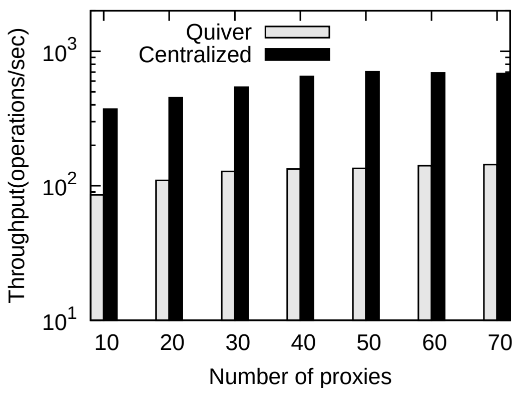


Figure 2.20. Building traffic models: Quiver's latency and throughput against the centralized server.

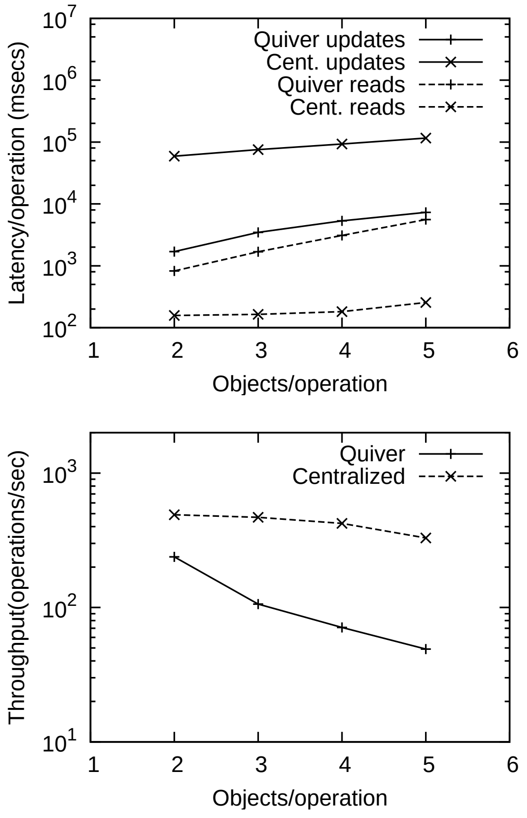


Figure 2.21. Building traffic models: Quiver's latency and throughput against the centralized server for multi-object operations.

in the multi-object case, Quiver implements reads using the same protocol as updates. So, these reads contend with the updates for migrating the relevant objects.

3 Rollback Attacks and Detection

The global distribution of service proxies and the possibly different administrative domains hosting these proxies, makes them vulnerable to compromise and misconfiguration. However, the design of edge service infrastructures typically assumes the correct behavior of service proxies. Indeed, efficiently tolerating misbehaving proxies while allowing them to serve clients' operations involving reads as well as updates of the service state, so as to reduce the client-perceived latency, is a challenging problem. Traditional solutions involving Byzantine fault-tolerant replication to detect (e.g., Shin and Ramanathan [1987]; Alvisi et al. [2001]; Buskens and R. P. Bianchini [1993]) or mask (e.g., Lamport [1978]; Schneider [1990]; Reiter and Birman [1994]; Castro and Liskov [2002]; Cachin and Poritz [2002]; Yin et al. [2003]; Abd-El-Malek et al. [2005]) proxy misbehavior, would either require interaction with other globally distributed proxies, or require significant management and hardware costs (in case each proxy is run as a cluster).

This chapter describes an extension to Quiver that addresses a particular attack on Quiver's consistency protocols (as stated in Chapter 1, there are other attacks on Quiver not addressed in this thesis); we refer to this attack as the *rollback attack*. In the Quiver paradigm processes (proxies and the server) act as "intermediaries" when forwarding read and update requests and object migrations on behalf of other processes. In a rollback attack, malicious or misconfigured intermediaries hide operations on the shared objects from some "honest" processes, while these operations are visible to others, resulting in a violation of consistency semantics. We develop an efficient mechanism that allows honest processes to detect such misbehavior by intermediate processes through *fork consistency* (Mazières and Shasha [2002]; Li et al. [2004]).

We present a new formulation of fork consistency that better suits our model, and in fact allows more efficient implementations and a broader application than the existing formulation (Mazières and Shasha [2002]). We then discuss an implementation of fork consistency and also show how it can be employed in settings other than Quiver. In particular, most work in this domain has been focussed on detecting or preventing rollback attacks by remote untrusted storage accessed by honest clients. We therefore, also discuss how our solution can be mapped to the context of a distributed file system, so as to compare the security properties achieved, and the performance costs incurred by our mechanism, against existing approaches.

3.1 Related work

There has been much research on masking and detecting failures using replication techniques and Byzantine fault-tolerant algorithms, as mentioned earlier. Some of these techniques have also been applied to distributed file systems (e.g., Castro and Liskov [2002]; Adya et al. [2002]; Rhea et al. [2003]). These systems provide strong security and availability guarantees, but require additional resources for replication, and typically incur performance penalties due to the use of Byzantine fault-tolerant coordination protocols.

Numerous systems have been proposed to deal with malicious intermediaries (that have not been replicated), e.g., Dabek et al. [2001]; Fu et al. [2002]; Kallahalla et al. [2003]; Muthitacharoen et al. [2002]; Fu [1999]; Zadok et al. [1998]. These systems typically provide strong integrity, privacy, and scalable key management, or a combination of these properties. The integrity properties of these systems guarantee that a malicious intermediary, e.g., a corrupt remote file server, cannot modify files, or corrupt file data or metadata in some other way. None of these systems, however, provide a way to detect or prevent against a rollback attack, where the file server hides some file updates from some file readers.

SUNDR (Li et al. [2004]; Mazières and Shasha [2002]) was the first work that formalized the notion of a rollback attack, introduced the fork consistency property, and proposed a protocol to guarantee this property in the context of a distributed file system. SUNDR allows clients to store and retrieve data from untrusted storage, while detecting unauthorized modifications or rollback of the data. In the SUNDR protocol, each client keeps a version structure containing the client's current timestamp (a logical counter value) and its estimate of the timestamp of each other client in the system. Each client updates its version structure with each operation (on any object), e.g., by incrementing its own timestamp, and sends the signed updated version structure to the server. When a client retrieves a file object from the server, the server also sends the signed version structures of all the clients in the system, along with the file object. The client then verifies signatures on all the version structures, and orders them according to the timestamps stored inside. An inconsistency is detected if the version structures cannot be totally ordered. In order to prevent the detection of a rollback of one honest client compared to the other, the server must fork the clients' views by never showing them each other's operations and thus each others updated version structures. This enables simple out-of-band mechanisms to detect the misbehavior. We suggest a variation of the fork consistency property defined and achieved by SUNDR, and detail a more efficient implementation of this new property that uses only collision resistant hash functions (as opposed to public key signatures used in SUNDR), and requires a smaller amount of state to be stored and transmitted for each operation than SUNDR. A more detailed cost-comparison with the SUNDR protocol is presented in Section 3.8.

SiRiUS (Goh et al. [2003]) is a secure file system that provides integrity verification, access control, key management and freshness guarantees for files stored on remote untrusted storage. SiRiUS achieves the freshness guarantees through the use of two hash trees per client, one for the client's data files (data files that the client can write to) and one for the client's metadata files. Each client then periodically computes the root for each hash tree, and signs the root along with the current timestamp value. When reading a file, a client verifies its freshness using the root of the hash tree of the file's writer, and comparing timestamps to check the freshness of the root itself. However, rollbacks within the validity interval for the signed root will go undetected. In addition, the clients need to be time-synchronized; otherwise the untrusted server could return an old hash tree root for the writer, and the reader will not be able to detect this just based on the timestamp value.

Finally, PastWatch (Chen [2004]) is a distributed version control system that allows clients to detect forks created by malicious intermediaries. However, it requires each client to store all versions of the repository (or at least the "diffs" between versions). This may be manageable for version control systems with infrequent "commit" operations, but could be impractical for other domains, like edge services or file systems that involve frequent update operations and a large number of objects.

3.2 System model extensions for rollback attacks

Here we present extensions to the system model of Section 2.2 that are required for the description of the rollback attack and the detection mechanisms against this attack.

We assume the existence of n processes (one server and $n-1$ proxies), and denote the set of all processes as P, i.e., $|P| = n$; we specify n only to simplify analysis. Processes that follow the protocol description are called *honest*. Processes that are not honest are called *corrupt* and may fail in any way possible, limited only by the assumptions stated below or those implicit in the cryptographic primitives employed. Processes communicate with each other through messages sent over communication channels. We assume that the out-of-band communication for detecting forks between processes is conducted via authenticated channels. However, this is not required for the messages sent during normal protocol execution.

Processes perform read and update *operations* on service objects as usual, but here we only consider single-object operations for simplicity—fork consistency, and our protocols for implementing fork consistency extend to multi-object operations by treating each object involved in such an operation, individually. Each operation consists of an *invocation* and a *response* event. These events are considered instantaneous. A *history* is a sequence of invocation and response events of read and update operations. A *well-formed* history is one where for each invocation event in the history, the corresponding response event is also in the history, i.e., the history only consists of "complete" operations. We only consider well-formed histories. A *sequential history* is a history where each invocation event is immediately followed by the corresponding response event. A sequential history is *legal* if each read operation on an object, returns the value written to the object by the most recent update operation. A *serialization* of a history H, is a legal sequential history that contains all operations in H and no other operations. Note that a serialization S of a history H induces a total order on all operations in H, denoted as $\rightarrow^{\mathsf{S}}$, i.e., $op_1 \rightarrow^{\mathsf{S}} op_2$ if op_1 precedes op_2 in S. Let $S_{p,H}$ denote a serialization of a subset of operations in H including all operations performed by a process p and possibly some operations performed by other processes. Then a *serialization set* for a history H is the set $\{S_{p,H} : p \in P\}$, i.e., an $S_{p,H}$ for each process $p \in P$.

Rollback attacks target operations that need to be serialized, e.g., only update and multi-object operations in Quiver. In this chapter, however, we do not concern ourselves with the types of operations involved (reads vs updates, single-object vs multi-object); from here on we talk about any operations that can be a target of rollback attacks. These operations could also involve single-object read operations in systems other than Quiver depending on the protocols employed and the consistency semantics achieved.

In this chapter, we denote an object instance o, with $o.\mathsf{ver} = i$ as o_i, for brevity. Note that o_i—the instance with version i—may not be unique, e.g., in case of a rollback attack, one honest process p may have a different i^{th} instance than another honest process forked from p. However, the i^{th} instance of the object seen at a particular process p, denoted $o_{i,p}$, is unique. In this chapter, we describe the protocol as executed by a particular process p, and thus use the compact notation o_i to mean $o_{i,p}$.

Each operation op, considered in this chapter, takes an instance of an object as input, and produces a new instance, e.g., $o_{i+1} \leftarrow op(o_i)$. Note that if *op* is a read operation (e.g., if the application requires serializing single-object reads as well), then the application performing *op* may not produce a new object instance. One way to overcome this is to embed the "application object" (handled by the application) in a "protocol object" (handled by our protocol) that also contains a "nonce". The nonce may be re-generated with each operation resulting in a new protocol object instance, even if the application performs a read-only operation. For simplicity, however, we do not make the distinction between application and protocol objects, and simply assume that each operation (that our protocol deals with, i.e., that requires serialization) produces a distinct object instance.

3.3 Properties

Here we define fork consistency and how it can be used to detect a rollback attack. We then present our formulation of this property, and compare it with the existing formulation.

3.3.1 FORK_S: System-wide fork consistency (Mazières and Shasha [2002])

Fork consistency, denoted FORK_S, is the strongest consistency notion possible for a set of *honest* parties (e.g., honest clients in a distributed file system, or honest processes in an edge service infrastructure) communicating with each other through malicious intermediaries (e.g., a corrupt file server, or corrupt intermediate processes in Quiver). In such a setting it is impossible to prevent an intermediary from conducting a rollback attack, i.e., prevent a malicious intermediary from "hiding" an operation performed by a process p from another process p'—e.g., the read performed by p' on an object may not reflect a preceding update of the same object by p. Fork consistency guarantees that either process (p or p') will detect this misbehavior upon "seeing" any subsequent operation performed by the other, even if through a third process. Therefore, a misbehaving intermediary is forced to divide the set of honest processes into two groups, such that neither group can see the operations performed by the other, and as such their views of the system state are "forked".

This situation is then very easily detectible through any out-of-band communication, e.g., a communication from one honest process to the other of the form "Have you seen the following update I performed on the object with identifier *id*". The goal of fork consistency is, therefore, to force the intermediary conducting a rollback attack to bring the system in a state where detection via out-of-band methods becomes convenient.

We now formally define the $\mathsf{FORK_S}$ notion (this formal definition is borrowed from Oprea and Reiter [2006]):

Definition 7. *A* fork serialization set *for a history H is a serialization set $\{S_{p,H} : p \in P\}$, such that for any two distinct processes $p, p' \in P$, if $op_2 \in S_{p,H} \cap S_{p',H}$ and $op_1 \rightarrow^{S_{p,H}} op_2$, then $op_1 \in S_{p',H}$ and $op_1 \rightarrow^{S_{p',H}} op_2$.*

Definition 8. *A history H is* $\mathsf{FORK_S}$ consistent*, if there exists a fork serialization set for H.*

The intuition behind this definition is that the serializations of the histories seen at different processes form a forking tree, with each node in the tree containing a serialization. Thus, the serializations of the histories seen at different processes may have a common prefix (a common ancestor in the forking tree), but once the histories diverge (fork) they never see the same operation again and thus never converge (due to the "treeness").

3.3.2 $\mathsf{FORK_O}$: Object-based fork consistency

$\mathsf{FORK_S}$ is a very useful property, however, it is restrictive in the sense that if the views of p and p' are forked on an object with identifier id, either process is required to detect this situation even upon seeing an operation performed by the other process on an object with identifier $id' \neq id$, e.g., a different file in case of a file system.

An implementation of this property necessarily requires synchronizing operations across objects. Indeed, the implementation presented by Mazières and Shasha (Mazières and Shasha [2002]; Li et al. [2004]) either requires all operations performed on all objects to be serialized, or in case concurrent operations are allowed, requires additional communication, storage and processing for synchronization. In addition $\mathsf{FORK_S}$ would require each process to store and process state encoding the latest operation performed by every other process on any object, and thus the performance suffers with an increase in the number of processes. Finally, $\mathsf{FORK_S}$ and its implementation are developed in a file system setting, and as such make certain implicit assumptions about the system model, e.g., the file server is used as a coordination site in SUNDR (Li et al. [2004]). It is not clear how the existing implementation can be mapped to a setting where proxies communicate with the server through other, possibly malicious, proxies, i.e., the proxies themselves are the intermediaries and can conduct rollback attacks on other proxies.

We present a variation of fork consistency, denoted $\mathsf{FORK_O}$, that provides guarantees on a per-object basis. $\mathsf{FORK_O}$ guarantees that if two honest processes are shown different instances of an object with identifier *id* by a malicious intermediary (or possibly multiple colluding intermediaries), then either process will be able to detect the attack upon seeing an operation on the object with identifier *id* (as opposed to any object, as in $\mathsf{FORK_S}$) from the other process, and therefore, must be forked on this particular object (as opposed to all objects, as in $\mathsf{FORK_S}$) by the intermediary to prevent detection. A direct consequence of this new definition is that an out-of-band communication of the form "Have you seen the following update I performed to the object with identifier *id*", can only detect a forked view of the processes for the object with identifier *id*, and not for any other object. Thus $\mathsf{FORK_O}$ is a strictly weaker property than $\mathsf{FORK_S}$ and requires out-of-band communication for each relevant object shared between the two honest processes, in order to detect a rollback attack for that object.

We now formally define the $\mathsf{FORK_O}$ notion:

Definition 9. *The* object subhistory $H|id$ *of a history H, is the subsequence of H containing all operations (their invocation and response events, to be exact) in H performed on the object with identifier id, and no other operations.*

Definition 10. *A history H is* $\mathsf{FORK_O}$ consistent*, if for each unique object identifier id (recall that there is a unique identifier for each object in the* *system), there exists a fork serialization set for $H|id$.*

Intuitively, $\mathsf{FORK_O}$ requires a forking tree per object, where nodes are serializations of object subhistories seen at processes, rather than the whole histories. Therefore, it is possible that two processes have forked object subhistories for one object (and thus they will never converge in the forking tree for this object), but see the same operations on a different object.

Although weaker than $\mathsf{FORK_S}$, $\mathsf{FORK_O}$ provides the flexibility to optimize for performance and scalability, and maps well to different system models: First, $\mathsf{FORK_O}$ does not require any synchronization of operations across objects and naturally allows for more concurrency in the system, without the overhead of additional communication and processing. Second, processes are not required to keep (and process) state for each other process in the system, as in $\mathsf{FORK_S}$, rather state required for each object can be efficiently encoded inside that object (as in our implementation, see Section 3.5), allowing the system to scale to a very large number of processes without performance penalties. Finally, $\mathsf{FORK_O}$ allows for implementations that map well to different system models where participants share abstract objects. These participants may be arranged in any communication network, with any number of colluding corrupt intermediaries. $\mathsf{FORK_O}$ is guaranteed for any two honest participants in this setting.

For simplicity we describe our algorithms assuming a single shared object with identifier id. We define the *instance history* IH of the history $H|id$ (which is the same as H because we consider a single object) to be the sequence of all the instances with identifier id produced by operations in $H|id$. Since each operation produces a new instance in our model, there is an object instance in IH corresponding to each operation in $H|id$, and thus there is a unique instance history corresponding to each history $H|id$.

3.4 Overview of $\mathsf{FORK_O}$ implementation

At a high level, we obtain $\mathsf{FORK_O}$ by efficiently embedding the current instance history of the object within each instance. The instance history encodes all instances of the object produced so far as well as the relative ordering between these instances. In order to perform an operation op on the object, a process p migrates the latest object instance (with the instance history embedded)—this instance may not be the latest in case of a rollback attack; performs op; updates the instance history with the new instance produced by op; embeds this updated instance history in the instance produced by op; and finally stores a copy of this updated instance history locally. Then upon migrating a later instance (of the same object) for a subsequent operation, p ensures that the new instance history received with the recently migrated instance is *reachable* from the locally stored instance history. Reachability is defined such that forked instance histories are not reachable from each other, and thus p detects a rollback attack if the instance history embedded in the migrated object instance is not reachable from the instance history stored locally at p, i.e., the received instance history contains an instance produced by a process p', such that p' was shown different instances than p by a malicious intermediary.

$o_i.\mathsf{IH}$ denotes the instance history embedded in the instance o_i (as seen at a particular process), encoding the instances up to and including o_i. We denote the version number of the last object instance seen at a process p as ver_p.

Definition 11. $\mathsf{IH}' \leftarrow \mathsf{add}(\mathsf{IH}, o_i)$ *adds all the contents of* o_i*, except for* $o_i.\mathsf{IH}$*, to the instance history* IH *such that* o_i *is ordered after all the instances that exist in* IH*, and outputs the updated instance history* IH'*. (The exact implementation of* add *depends on the way the instance histories are encoded, see Section 3.5).*

Definition 12. *We say an instance history* IH' *is* reachable *from an instance history* IH *if there exists a sequence of object instances* $\{o_{j+1}, o_{j+2} \ldots o_i\}$*, such that iteratively adding these instances to* IH*, i.e.,* $\mathsf{IH}'' \leftarrow \mathsf{add}((\ldots \mathsf{add}(\mathsf{add}(\mathsf{IH},\ o_{j+1}), o_{j+2}) \ldots), o_i)$*, results in* $\mathsf{IH}'' = \mathsf{IH}'$*.*

Definition 13. $\mathsf{isForked}(\mathsf{IH}, \mathsf{IH}', \mathsf{ver}, \mathsf{ver}')$ *detects if the instance histories* IH *and* IH' *encoding instances up to and including instances with version number* ver *and* ver' *respectively, are forked or not and returns a boolean. (The* *exact implementation of* $\mathsf{isForked}$ *depends on the way the instance histories are encoded, see Section 3.5).*

Each process p is initialized when it migrates its first object instance o_k, containing the embedded instance history $o_k.\mathsf{IH}$. Each process p maintains a local instance history IH_p representing the object instances seen at p. IH_p is initialized to $o_k.\mathsf{IH}$ and is updated to reflect the instance history embedded with the latest instance seen at p. Let $\mathsf{ver}_p = j$, i.e., o_j be the latest object instance seen at p, then $\mathsf{IH}_p = o_j.\mathsf{IH}$. Now in order to perform a new operation op, p receives a tuple $\{o_i, \mathsf{aux}_{j,i}\}$, where o_i is the most recent object instance (except in the case of a rollback attack) and $\mathsf{aux}_{j,i}$ contains auxiliary information (detailed in Section 3.5) required to verify if $o_i.\mathsf{IH}$ is reachable from IH_p ($o_j.\mathsf{IH}$). p then performs the following steps:

- **Reachability verification:** p first verifies if $o_i.\mathsf{IH}$ is reachable from the locally stored instance history IH_p using $\mathsf{aux}_{j,i}$, and detects a fork (rollback attack) if $o_i.\mathsf{IH}$ is not reachable.
- **Instance history extension:** If reachability verification succeeds, then p performs its operation $o_{i+1} \leftarrow op(o_i)$, extends the instance history and embeds it in the new instance $o_{i+1}.\mathsf{IH} \leftarrow \mathsf{add}(o_i.\mathsf{IH}, o_{i+1})$. p finally updates the local instance history $\mathsf{IH}_p \leftarrow o_{i+1}.\mathsf{IH}$ and the version number $\mathsf{ver}_p \leftarrow i + 1$.

p then sends the tuple $\{o_{i+1}, \mathsf{aux}_{l,i+1}\}$, possibly through malicious intermediaries, to any process p' that needs to perform an operation, where $\mathsf{ver}_{p'} = l$. In order to detect if the instance histories at two honest processes p and p' are forked or not (using out-of-band communication):

- **Fork detection:** Without loss of generality, assume $\mathsf{ver}_p \leq \mathsf{ver}_{p'}$. Then p sends IH_p and ver_p to p', and p' invokes $\mathsf{isForked}(\mathsf{IH}_p, \mathsf{IH}_{p'}, \mathsf{ver}_p, \mathsf{ver}_{p'})$.

Note that fork detection reduces to reachability verification if the auxiliary information is available. However, out-of-band fork detection may be much more infrequent than operation processing, and keeping all auxiliary information used by reachability verification may not be practical over long periods of execution. Therefore, we assume that this auxiliary information is not available for fork detection.

3.5 Iterative hashing based encoding

In this section we describe how the instance histories are encoded in our system, and describe how to perform reachability verification, instance history extension and fork detection for the encoding.

A candidate instance history encoding must be a representation of the object instances and their relative ordering. In addition it should allow efficient history extension, reachability verification (which should fail if the instances encoded in the two histories are different due to a rollback attack), and fork detection without any auxiliary information. Finally the instance history encoding must be space efficient. A naive encoding would be a sequence keeping all the object instances (or their hashes) produced so far. Such an encoding allows for history extension—simply add the new instance to the end of the sequence; reachability verification—add the intermediate hashed instances to the sequence and compare with the sequence embedded in the received object; and fork detection—check if one instance history is a prefix of the other. However, the encoding is not space efficient: it requires keeping some state for each instance produced (requiring $O(\mathsf{ver}_p)$ state at each process p), which is linear in the number of operations, and is therefore prohibitive.

We now present an encoding scheme that is extensible, allows reachability verification and fork detection while incurring $O(1)$ storage cost. In fact, the instance history is encoded as a single hash value. This scheme iteratively applies a one-way collision resistant hash function [Menezes et al., 1996, Chapter 9] $h : \mathcal{M} \rightarrow \{0,1\}^m$, to the object instances, and encodes the instance history as the resulting hash value.

Extension: add is invoked as $\mathsf{add}(\mathsf{IH}, h(o_i))$ and is implemented as $\mathsf{IH}' \leftarrow h(h(o_i) \mid\mid \mathsf{IH})$, where h is the hash function and "$||$" denotes concatenation. Note that if $h(o_i) \in \{0,1\}^m, \mathsf{IH} \in \{0,1\}^m$, then $\mathsf{IH}' \in \{0,1\}^m$. Thus, an instance history is always encoded as a single hash value (an m-bit binary string), regardless of the number of instances encoded within, or the number of operations performed on the object.

Reachability verification: A process p with $\mathsf{ver}_p = j$ receives the tuple $\{o_i, \mathsf{aux}_{j,i}\}$ for performing its operation op, where $\mathsf{aux}_{j,i} = \{h(o_{j+1}), h(o_{j+2}), \ldots, h(o_{i-1})\}$, and o_i is either the latest object instance, or a rolled back instance in case of an attack. p then starts with its local instance history IH_p, and iteratively adds the received instances to it: $\mathsf{IH}' \leftarrow \mathsf{add}((\ldots \mathsf{add}(\mathsf{add}(\mathsf{IH}_p, h(o_{j+1})), h(o_{j+2}))\ldots), h(o_i))$, which translates to $\mathsf{IH}' \leftarrow h(o_i||(\ldots h(o_{j+2}||h(o_{j+1}||\mathsf{IH}_p))\ldots))$—substituting add with its implementation. p then verifies reachability by comparing IH'—the instance history resulting from the iterative hashing—to $o_i.\mathsf{IH}$—the instance history embedded in the received object. If reachability verification succeeds, i.e., $\mathsf{IH}' = o_i.\mathsf{IH}$, then p performs its own operation, and adds the resulting instance o_{i+1} to the instance history, $\mathsf{IH}'' \leftarrow \mathsf{add}(o_i.\mathsf{IH}, o_{i+1})$ and updates $\mathsf{IH}_p \leftarrow \mathsf{IH}''$. Finally p embeds $o_{i+1}.\mathsf{IH} \leftarrow \mathsf{IH}''$ and sends the new object instance to the process that next requests it. Note that if an intermediary conducts a rollback attack and shows different instances to two honest processes, their local instance histories will be forked, and the reachability verification at either process will fail upon seeing an instance produced by the other, see Figure 3.1.

Fork detection: The challenge then is to allow fork detection, i.e., given $\mathsf{IH}_p, \mathsf{ver}_p, \mathsf{IH}_{p'}$ and $\mathsf{ver}_{p'}$, processes p and p' should be able to employ some out-of-band protocol to decide if their instance histories are forked or not. We develop a simple technique to perform fork detection using an out-of-band protocol between p and p' in which these processes perform *synchronized operations.*

p first performs a "dummy operation" resulting in an updated IH_p. p then sends IH_p to p'. Finally p' performs its dummy operation, receiving the latest instance o_i and some auxiliary information. During reachability verification, p' compares each intermediate instance history produced with IH_p. If none of the instance histories match IH_p then p' decides that p and

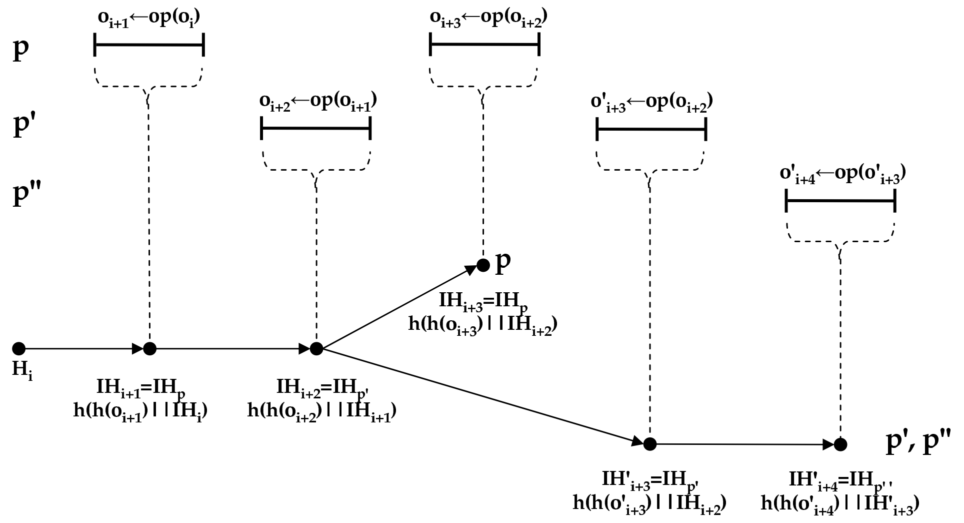

p
p'
p"
o_{i+1}←op(o_i)
o_{i+2}←op(o_{i+1})
o_{i+3}←op(o_{i+2})
o'_{i+3}←op(o_{i+2})
o'_{i+4}←op(o'_{i+3})
H_i
IH_{i+1}=IH_p h(h(o_{i+1})||IH_i)
IH_{i+2}=IH_{p'} h(h(o_{i+2})||IH_{i+1})
IH_{i+3}=IH_p h(h(o_{i+3})||IH_{i+2})
IH'_{i+3}=IH_{p'} h(h(o'_{i+3})||IH_{i+2})
IH'_{i+4}=IH_{p''} h(h(o'_{i+4})||IH'_{i+3})
p', p"

Figure 3.1. p, p', p'' are honest and not forked. p then performs an operation and produces instance o_{i+1} and instance history $\mathsf{IH}_{i+1} = \mathsf{IH}_p$. p' receives o_{i+1} (with IH_{i+1} embedded) and produces o_{i+2} and $\mathsf{IH}_{i+2} = \mathsf{IH}_{p'}$. At this point, processes are not forked. Now p produces o_{i+3} and $\mathsf{IH}_{i+3} = \mathsf{IH}_p$. p''s next operation receives the older instance o_{i+2} from a malicious intermediary hiding p's operation, resulting in a fork from p. Finally, p'' receives the instance produced by p', so they lie in the same branch. Reachability verification at p will now fail upon seeing a instance produced by p' or p'', and vice versa.

p' are forked on the object o. If some instance history computed during the iterative hashing matches with IH_p and the reachability verification succeeds then p' decides that it has the same view of o as p.

3.5.1 Discussion

The costs associated with bandwidth—the size of the tuple $\{o_i, \mathsf{aux}_{j,i}\}$—and processing—the number of hash computations required for reachability verification—depend on the workload. In particular, if a process p performs an operation producing instance o_j and subsequently migrates the instance o_i for its next operation, then $\mathsf{aux}_{j,i}$ contains $i - j$ hashes, and p needs to perform $(i - j)$ hash computations (for constructing intermediate instance histories) in order to verify reachability. We refer this quantity $i - j$ at process p for an operation op that inputs instance o_i as $\mathsf{myGap}_p(op)$ (as opposed to $\mathsf{neighborGap}$ defined in Section 3.7.1). Thus, if the workload is such that myGap is small for (most) operations, i.e., two successive operations performed by p on a particular object o are "spaced" by only a small number of operations performed at other processes on o, then the bandwidth and processing costs are small. These costs are discussed in more detail in the context of a distributed file system in Section 3.8.

3.5.2 Summary

Our protocol achieves $\mathsf{FORK_O}$ through iteratively applying a collision resistant hash function to efficiently encode the object instance history, and embedding this encoded instance history within the object itself. If a malicious intermediary conducts a rollback attack introducing inconsistencies in the views of two honest processes for a shared object o, then either process detects the attack upon subsequently seeing an operation from the other on o, as the reachability verification fails. Therefore, the intermediary has to keep the honest processes forked by not showing them each other's operations to prevent detection, and in this case the fork on o can be easily detected in an out-of-band protocol between the two honest processes.

The out-of-band protocol for detecting forks is an interactive approach where processes synchronize to perform one operation each, one after the other. The process that performs the second operation looks for the instance history produced by the first process, during reachability verification. If it does not find the instance history or if the reachability verification fails, a fork is detected.

3.6 Security

Our algorithm guarantees that if the instance histories seen at two processes diverge, then they never see the same object instance again. This maps directly to the object subhistories, i.e., if the object subhistories seen at the two processes diverge, i.e., one is not a prefix of the other, then the two processes never see each other's operations again.

The security of the iterative hashing scheme depends on the security of the reachability verification mechanism. Reachability verification ensures that given two honest processes p and p' that have seen different object instances due to a rollback attack, either process will detect the attack upon seeing an instance (and embedded history) produced by the other process, i.e., their instance histories will never coincide again. We prove that breaking reachability verification is at least as hard as finding a collition in h, where $h : \mathcal{M} \rightarrow \{0,1\}^m$ is a collision-resistant hash function.

Definition 14. *For an adversary algorithm $\mathcal{A}$, the advantage of $\mathcal{A}$ in breaking the collision resistance of hash function h is defined as:*

$$\begin{aligned}\mathsf{Adv}_h^{\mathsf{CR}}(\mathcal{A}) = \mathsf{Pr}[< m, m' >\leftarrow \mathcal{A}() : \\ (m \neq m') \wedge (h(m) = h(m'))]\end{aligned}$$

$\mathsf{Adv}_h^{\mathsf{CR}}(t)$ *denotes the maximum advantage* $\mathsf{Adv}_h^{\mathsf{CR}}(\mathcal{A})$ *for all adversaries* $\mathcal{A}$ *taking time* t.

Let IH and IH' be two instance histories. We say $\mathsf{IH} \subseteq \mathsf{IH}'$, if the sequence of object instances encoded in IH is a prefix of the sequence of object instances encoded in IH'. Note that $\mathsf{IH} = \mathsf{IH}'$ if and only if $\mathsf{IH} \subseteq \mathsf{IH}'$ and $\mathsf{IH}' \subseteq \mathsf{IH}$.

Definition 15. *For an adversary algorithm $\mathcal{A}$, the advantage of $\mathcal{A}$ in breaking the reachability verification of the iterative hashing scheme is defined as:*

$$\begin{aligned}\mathsf{Adv}_{\mathsf{IH}}^{\mathsf{RV}}(\mathcal{A}) = \mathsf{Pr}[\mathsf{IH}_p \not\subseteq \mathsf{IH}_{p'}, \mathsf{IH}_{p'} \not\subseteq \mathsf{IH}_p, < o_m, o_{m'} >\leftarrow \mathcal{A}(\mathsf{IH}_p, \mathsf{IH}_{p'}) : \\ h(h(o_m) \mid\mid \mathsf{IH}_p) = h(h(o_{m'}) \mid\mid \mathsf{IH}_{p'})]\end{aligned}$$

$\mathsf{Adv}_{\mathsf{IH}}^{\mathsf{RV}}(t)$ *denotes the maximum advantage* $\mathsf{Adv}_{\mathsf{IH}}^{\mathsf{RV}}(\mathcal{A})$ *for all adversaries* $\mathcal{A}$ *taking time* t.

Intuitively, the goal of the adversary is to compute two object instances o_m and $o_{m'}$ that when added to forked histories IH_p and $\mathsf{IH}_{p'}$ respectively, result in the same history, i.e., $\mathsf{IH}_m \leftarrow h(h(o_m) \mid\mid \mathsf{IH}_p) = h(h(o_{m'}) \mid\mid \mathsf{IH}_{p'})$.

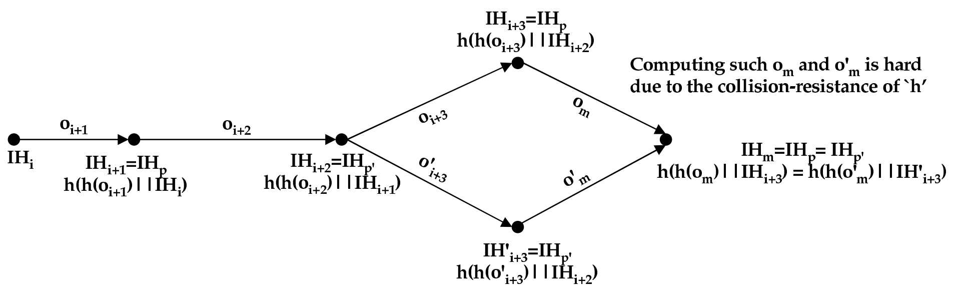

IH$_{i+3}$=IH$_p$
h(h(o$_{i+3}$)||IH$_{i+2}$)
Computing such o$_m$ and o'$_m$ is hard due to the collision-resistance of `h'
o$_{i+1}$
o$_{i+2}$
o$_{i+3}$
o$_m$
IH$_i$
IH$_{i+1}$=IH$_p$
h(h(o$_{i+1}$)||IH$_i$)
IH$_{i+2}$=IH$_{p'}$
h(h(o$_{i+2}$)||IH$_{i+1}$)
o'$_{i+3}$
o'$_m$
IH$_m$=IH$_p$= IH$_{p'}$
h(h(o$_m$)||IH$_{i+3}$) = h(h(o'$_m$)||IH'$_{i+3}$)
IH'$_{i+3}$=IH$_{p'}$
h(h(o'$_{i+3}$)||IH$_{i+2}$)

Figure 3.2. p and p' perform operations in the same way as in Figure 3.1. Adversary's goal is to compute object instances o_m and $o_{m'}$ that when hashed concatenated with IH_p and $\mathsf{IH}_{p'}$ result in the same history (some IH_m in this case). This is considered hard if h is a collision resistant hash function.

The adversary can then request the object instances from p and p' with embedded histories IH_p and $\mathsf{IH}_{p'}$ respectively, compute o_m and $o_{m'}$ and send o_m to p and $o_{m'}$ to p'. Both processes perform reachability verification which succeeds—IH_m is an extension of both IH_p and $\mathsf{IH}_{p'}$, see Figure 3.2. We prove that this is at least as hard as finding a collision for h.

Theorem 5. *If h is a collision resistance hash function, then the reachability verification of the iterative hashing scheme is secure:* $\mathsf{Adv}^{\mathsf{RV}}_{\mathsf{IH}}(t) \leq \mathsf{Adv}^{\mathsf{CR}}_{h}(t)$.

Proof. Assume there is an adversary for reachability verification of the iterative hashing scheme with advantage $\mathsf{Adv}^{\mathsf{RV}}_{\mathsf{IH}}(\mathcal{A}_{\mathsf{IH}})$. We construct an adversary $\mathcal{A}_h$ for the collision resistance of the hash function. $\mathcal{A}_h$ runs the adversary $\mathcal{A}_{\mathsf{IH}}$ on some forked instance histories IH_p and $\mathsf{IH}_{p'}$. $\mathcal{A}_{\mathsf{IH}}$ outputs $< o_m, o_{m'} >$ and $\mathcal{A}_h$ outputs $m \leftarrow h(o_m) \mid\mid \mathsf{IH}_p$ and $m' \leftarrow h(o_{m'}) \mid\mid \mathsf{IH}_{p'}$. Since $\mathsf{IH}_p \neq \mathsf{IH}_{p'}$, $m \neq m'$ but if $\mathcal{A}_{\mathsf{IH}}$ succeeds then $h(m) = h(h(o_m) \mid\mid \mathsf{IH}_p) = h(h(o_{m'}) \mid\mid \mathsf{IH}_{p'}) = h(m')$. Thus $\mathcal{A}_h$ succeeds every time $\mathcal{A}_{\mathsf{IH}}$ succeeds, and so $\mathsf{Adv}^{\mathsf{RV}}_{\mathsf{IH}}(\mathcal{A}_{\mathsf{IH}}) \leq \mathsf{Adv}^{\mathsf{CR}}_{h}(\mathcal{A}_h)$. □

3.7 Other considerations

3.7.1 Denial of service

We denote the set of neighbors of a process p in the communication network as $\mathsf{N}(p)$, e.g., in case of Quiver, $\mathsf{N}(p)$ are the neighbors of a process in the tree, whereas for a distributed file system, the neighbor set for each client only contains the server, while the server's neighbor set contains all the clients sharing the object.

In order to implement the $\mathsf{FORK_O}$ protocol of Section 3.5, a process p when sending the latest object instance o_i to a process $p' \in \mathsf{N}(p)$ with $\mathsf{ver}_{p'} = j$, must also include the hashed object instances $h(o_{j+1}), h(o_{j+2}), \ldots, h(o_{i-1})$ so that p' may perform reachability verification. These hashed instances must be available at p when sending this message to p'. Thus any process p must maintain $\mathsf{minVersion}_p = \min_{p' \in \mathsf{N}(p)} \mathsf{ver}_{p'}$, and keep all the hashed object instances $o_j, \forall j > \mathsf{minVersion}_p$. We denote this quantity as $\mathsf{neighborGap}_p \leftarrow \mathsf{ver}_p - \mathsf{minVersion}_p$, which denotes the number of hashed object instances that need to be stored at p. Let $p' \in \mathsf{N}(p)$ be the neighbor such that $\mathsf{minVersion}_p = \mathsf{ver}_{p'}$. When p sends the latest object instance to p', p recomputes $\mathsf{minVersion}_p$ and garbage-collects all the hashed object instances less than the updated $\mathsf{minVersion}_p$. Note that each process only keeps this state for its neighbors and not for any other processes in the system.

In case a neighbor $p' \in \mathsf{N}(p)$ does not perform an operation for a long time and thus does not request the object, or p' is malicious and intentionally does not request the object, then $\mathsf{minVersion}_p = \mathsf{ver}_{p'}$ would be much older than the current instance. This would require p to store a large number of hashed object instances, and could result in a denial of service by filling up the memory available at p. Thus as a counter-measure, each process p "pushes" the hashed object instances older than a certain threshold to all neighbors that have previously performed an operation on the object, and garbage-collects this stale state. A different remedy for p would be to send the hashed object instances to a persistent server known to all the processes, so p's neighbors can retrieve the hashed instances required for reachability verification from this server. Note that the persistent server need not be trusted to provide integrity because in this model, it acts as another intermediary and can at best fork honest processes by returning incorrect hashed instances.

Note that p need not keep any state for a neighbor p' that has not yet performed its first operation on the object. When p' performs its first operation, p simply sends the current object instance to p': this instance initializes the object state at p'. From the perspective of p', it does not need to perform reachability verification for the first instance received: if a malicious intermediary conducts a rollback attack on p' from the onset, p''s instance history will be initialized differently from some honest process p, will remain forked due to the mechanisms described previously, and can be detected using the out-of-band techniques developed earlier.

3.7.2 Authenticated operations

Although our protocol achieves $\mathsf{FORK_O}$ using only collision resistant hash functions, several existing applications that could benefit from $\mathsf{FORK_O}$ employ digital signatures in order to authenticate processes that operate on the shared objects. In these cases, each new object instance is digitally signed by the process that computes this instance, i.e., each hashed object instance $h(o_i)$ is accompanied by a signature σ_i on this hash, that can be verified using the public key K_{p_i} of the process p_i that computes the instance o_i. In such a setting, we can exploit trust relationships between processes to reduce the amount of bandwidth consumed by our protocol by reducing the number of hashed object instances exchanged for reachability verification.

Let $\mathsf{T}(p)$ be the set of processes trusted by p. When p sends an object request to a process $p' \in \mathsf{N}(p)$, it includes $\mathsf{T}(p)$ in the request. p' then sends the latest object instance o_i along with $\mathsf{aux}_{j,i}$ as before, where $j = \mathsf{ver}_p$. Let $\{< h(o_{j+1}), \sigma_{j+1} >, < h(o_{j+2}), \sigma_{j+2} >, \ldots, < h(o_{i-1}), \sigma_{i-1} >\}$ be the sequence of hash-signature tuples for all the instances between o_j and o_i. Let $p'' \in \mathsf{T}(p)$ be a process trusted by p, and let $< h(o_k), \sigma_k >$ and $< h(o_l), \sigma_l >$, $(j+1) \leq k < l \leq (i-1)$, be two of the hash-signature tuples in the sequence, that are computed and signed by p'', i.e., σ_k and

σ_l can be verified on the corresponding hashes using $K_{p''}$. Then p' sends to p only $\mathsf{aux}_{j,i} = \{< h(o_{j+1}), \sigma_{j+1} >, \ldots, < h(o_k), \sigma_k >, < h(o_l), \sigma_l >, \ldots, < h(o_{i-1}), \sigma_{i-1} >\}$. As such, p' does not include any hash-signature tuples for instances that fall between two instances constructed by a process that is trusted by p, in this case p''. The rationale behind this optimization is that if p trusts p'', then p can "short circuit" its reachability verification by relying on p'' to have performed its own reachability verification correctly. Note that by trusting p'', p is implicitly trusting all processes in $\mathsf{T}(p'')$ as well and the processes trusted by each process in $\mathsf{T}(p'')$ and so on. Thus, in this model, trust is defined by *equivalence classes*. Depending on the number of processes trusted by each process p, this optimization can result in significant bandwidth savings. Furthermore, if a process p maintains $\mathsf{T}(p')$ for each process $p' \in \mathsf{N}(p)$, then p can garbage-collect hashed object instances based on the knowledge of $\mathsf{T}(p')$ more aggressively, i.e., p may garbage-collect an instance $< h(o_i), \sigma_i >$ either if $\forall p' \in \mathsf{N}(p), \mathsf{ver}_{p'} \geq i$, or for each process $p'' \in \mathsf{N}(p)$ that has not seen this instance yet, i.e., $\mathsf{ver}_{p''} < i$, this instance lies between two instances computed and signed by a process in $\mathsf{T}(p'')$.

3.8 Application to distributed file systems

The $\mathsf{FORK_S}$ notion was initially developed and implemented in the context of a distributed file system with an untrusted server, and some possibly malicious clients (Li et al. [2004]; Mazières and Shasha [2002]). In this section we map our $\mathsf{FORK_O}$ notion and implementation to such a distributed file system, and quantify the computation, storage and bandwidth costs using actual file system traces.

A typical distributed file system comprises of a file server and clients. The server manages all the data and meta-data and synchronizes client operations to present a consistent view of the file system to the clients. The clients request meta-data and data from the server, perform their operations and send any updated state back to the server. The exact implementation of this functionality (and in some cases the delegation of some of these roles) varies from one file system to the other. Our protocol treats each piece of the file system state that can be read or updated by clients as a *file object*. Thus a file object could be a very large data file or a small directory object containing meta information. Each file object is treated as a shared object of our protocol. Clients perform their operations on these objects as described in the earlier sections.

We quantify the costs related to our protocol based on distributed file system traces that were collected from an NFS server at the Computer Science Department at Harvard University (Ellard and Seltzer [2003]). These traces were collected over a one week period between February 17, 2003 and February 21, 2003. The server was used to serve home directories and shared data for the users. When doing the analysis on these traces, we mainly looked at the operations relating to file objects, i.e., the objects read and updated by the clients, e.g., data files, meta data, etc. As such we ignored all operations that were specific to the file system implementation and did not affect the shared objects, e.g., FSINFO, FSSTAT, NULL and LOOKUP operations. Note that the actual number of operations that reach the server in a distributed file system may depend on the actual file system used and the locally defined policies, e.g., policies pertaining to caching, etc. Nevertheless, the client behavior depicted in these traces is expected to be typical of most distributed file systems, and the amount of sharing seen is typical of most workloads in a research/educational environment.

We first introduce some notation useful for discussing the associated costs. Let n be the total number of clients of the distributed file system, and $\mathsf{SharedObjs}$ be the subset of the total file objects (all files in the system) consisting of file objects that are shared among at least two clients, i.e., read or updated by more than one client. We use $\mathsf{Clients}_o$ to denote the set of all clients sharing an object $o \in \mathsf{SharedObjs}$. Finally let $\mathsf{ver}_{p,o}$ denote the latest instance of the object o seen at a client $p \in \mathsf{Clients}_o$.

3.8.1 Storage costs

Each client is required to store a single hash value for each object it actively shares with other clients. The hash value is the latest instance history IH_p of this object seen by the client p. Thus the storage cost of our protocol for each client p is $O(|\{o : p \in \mathsf{Clients}_o\}|)$, i.e., a function of the number of objects p shares with other clients; the constant is just the size of a hash output, e.g., 160 bits for a SHA-1 hash.. Figure 3.3 shows the amount of sharing seen in the trace. In particular, the clients performed their operations on a total of 367755 file objects, and 97% of these file objects were not shared. Of the 3% (10733 shared objects) objects that were shared, most (94% of the shared objects), were shared only between two clients. As such, we saw an average of ≈ 200 shared objects (most of these meta objects) per client, and thus the expected storage cost of our protocol incurred at each client is $200 * 20 = 4$ KBytes (negligible for typical clients); note that this is independent of the size of the actual objects being shared, or the number of operations performed on the objects.

In case of a file system, the neighbor set $\mathsf{N}(p)$ for each client p just contains the server (as the clients communicate with each other through the server), so the clients need not store any intermediate hashed instances for reachability verification performed at other clients. The server, denoted s, however has all the clients as its neighbors, i.e., $|\mathsf{N}(s)| = n$. Thus, the server needs to keep hashed object instances for each shared object $o \in \mathsf{SharedObjs}$, in order to allow the clients sharing this object to perform their reachability verification. We define $\mathsf{neighborGap}_s(o)$ for an object o, as $\max_{p,p' \in \mathsf{Clients}_o} |\mathsf{ver}_{p,o} - \mathsf{ver}_{p',o}|$, i.e., the maximum gap between the latest instances of this object known to the clients that share this object; see Section 3.7.1. Then the server needs to keep $O(\mathsf{neighborGap}_s(o))$ state for each shared object o. In the traces, we saw an average $\mathsf{neighborGap}_s(o)$ of 89, i.e., the maximum $\mathsf{neighborGap}$ ever seen between two clients sharing the same object, averaged over all the shared objects is 89. Therefore, in the unlikely worst case scenario, when the $\mathsf{neighborGap}$ of all the shared objects is 89 at the same time, the server would need $89 * 10733 * 20 = 19$MBytes, to store all the hashed object instances. This storage cost is acceptable considering it need not be kept in the server memory and the very large amount of disk space available to the file servers. Finally employing the optimizations from Sections 3.7.1 and 3.7.2 will reduce this state.

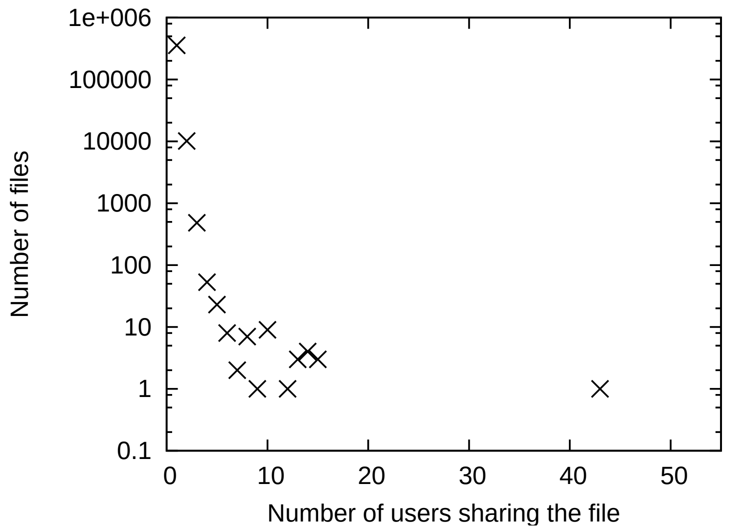


Figure 3.3. Most files are either not shared, or shared by a small number of users (< 3). A small amount of meta-data is shared by a large number of users.

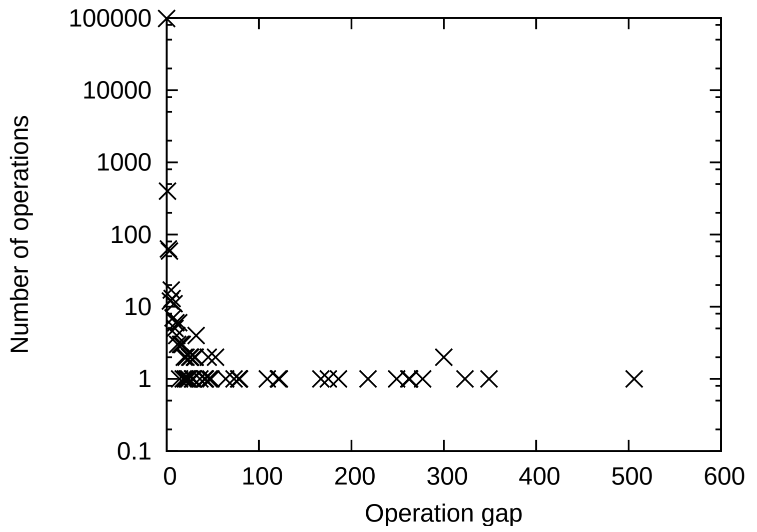


Figure 3.4. myGap (Section 3.5.1) is computed for each operation. As shown, for most operations this value is either zero, or very small (< 10).

3.8.2 Bandwidth and computation costs

Next we use the traces to quantify the approximate bandwidth and computation costs per operation. Recall from Section 3.5 that the per-operation bandwidth and computation costs of our protocol depend on the quantity myGap—if op_i and op_{i+1} are two successive operations performed by p on an object o, then $\mathsf{myGap}_p(op_{i+1})$ for op_{i+1} is the number of operations by other processes on o since op_i, and is zero if op_{i+1} is actually the first operation performed by p on this object.

The bandwidth and computation costs are dictated by the number of hashed object instances transmitted and hashes computed per operation for reachability verification, respectively. This number is a function of myGap for this operation, see Section 3.5. Figure 3.4 plots the quantity myGap seen in the NFS server trace for operations on objects shared by at least two clients. Of all the operations performed on shared objects, 99.3% had a myGap of zero, i.e., the operations were performed successively by the same client, and 99.9% of operations had myGap of 7 or less. This means that for 99.9% of operations on shared objects, the bandwidth cost associated with our protocol is transmitting just $20 * 7 = 140$Bytes per operation from the server to the client, and the compute cost required is only 7 SHA-1 hash computations at the client. Note that the server need not perform any hash computations.

3.8.3 Cost comparison with SUNDR

Table 3.1 compares the costs of the iterative hashing scheme with the corresponding costs in SUNDR (Li et al. [2004]). SUNDR keeps a timestamp vector at each client, containing the client's estimate of the timestamp of every other client in the system, along with other information in a structure called the *version structure*, hence the $O(n)$ storage at each client. Clients send their signed version structures (containing their timestamp vectors) to the server, which manages the updated version structure of each client, hence the $O(n^2)$ storage cost at the server. When a client receives a file object from the server, it also receives the latest version structures of all the clients in the system and hence the $O(n^2)$ bandwidth cost per operation. Finally, when the client performs its operations, it verifies signatures on all the version structures and updates and signs its own version structure, hence the $O(n)$ computation cost per operation. Since myGap is small and sharing is rare for typical file system workloads (Figures 3.3 and 3.4), we expect significant bandwidth, computation and storage savings for the iterative hashing scheme over SUNDR, specially as the number of clients grow.

	SUNDR	**IterHash**
Consistency property	$\mathsf{FORK_S}$	$\mathsf{FORK_O}$
Client storage	$O(n)$	$O(\lvert\{o : p \in \mathsf{Clients}_o\}\rvert)$
Server storage	$O(n^2)$	$O(\mathsf{neighborGap} * \lvert\mathsf{SharedObjs}\rvert)$
Bandwidth/operation	$O(n^2)$	$O(\mathsf{myGap}(op))$
Computation/operation	$O(n)$	$O(\mathsf{myGap}(op))$

Table 3.1. Cost comparison between SUNDR and iterative hashing. The computation/op of SUNDR is $O(n)$, and requires signature verification and computation, compared to just hash computations in iterative hashing. myGap, $\mathsf{neighborGap}$ and $\lvert\mathsf{SharedObjs}\rvert$ are small for typical file system workloads, see Figures 3.3 and 3.4.

4 Distributed Fault-Tolerant Trees

Quiver arranges its proxies in a rooted tree structure, that is constructed so that geographically close proxies are close to each other in the tree. This "location-aware" tree allows Quiver's migration algorithms to exploit geographic locality in the workload—an extremely useful property for optimizing performance of global scale applications. The tree structure also enables simple consistency protocols based on well-studied "path-reversal" mechanisms that exploit the existence of unique paths between any two proxies. Finally the hierarchy in the tree maps naturally to the fork consistency based detection mechanisms.

The use of a tree, however, has its drawbacks, as well. The foremost being its vulnerability to partitions due to proxy or link failures. Upon disconnection from the primary partition—the partition containing the root of the tree—a proxy is either not able to access service objects (in case these objects are not in its partition) or can access service objects but its operations on these objects are not visible to the connected proxies (in case the objects are in its partition, but have been reconstituted in the primary partition after the disconnection), and so these operations are "ignored" by Quiver. A simple multicast by the root of each disconnected partition can be used to notify proxies of their disconnection, so they may initiate their reconnection.

This chapter describes an extension to Quiver that allows disconnected proxies to reconnect to the primary partition with minimal "downtime". This mechanism is based on constructing a fault-tolerant network—an *expander*—on top of the tree structure. Disconnected Quiver proxies then use their neighbors in this expander network to connect back to the primary partition, e.g., by connecting to the geographically closest expander neighbor that is in the primary partition itself. Here we detail this algorithm and the proofs and simulation results that show the effectiveness of the expander constructed by our algorithm in tolerating failures.

4.1 Related work

Fault-tolerant tree structures were first introduced in the context of multiprocessor computer architectures such as X-Trees (Despain and Patterson [1978]) and Hypertrees (Goodman and Carlo [1981]). Fault tolerance was not the primary goal of this research. As a result, these structures impose other constraints that may not be reasonable in our target applications, e.g., X-tree (Despain and Patterson [1978]) assumes a complete binary tree and tolerates only a single node failure. Furthermore, distributed constructions of X-tree and Hypertree are not known.

Expander graphs are a well studied design for fault-tolerant networks. Both randomized (Steger and Wormald [1999]; Kim and Vu [2001]) and explicit (Margulis [1973]; Gabber and Galil [1981]) constructions of expanders have been known for some time. However, little has been done to construct expander networks in a distributed setting.

Law and Siu (Law and Siu [2003]) presented a distributed construction of expander graphs based on $2d$-regular graphs composed of d Hamiltonian cycles. However, to sustain expansion of the graph in the event of nodes leaving the system, they require that a leaving node send its state to some other node in the expander. Therefore, this approach cannot tolerate node failures. Furthermore, their algorithm requires obtaining global locks on the Hamiltonian cycles when new nodes join, which can be impractical in a large distributed system. Finally, they revert to employing either a centralized approach or using broadcast when the number of nodes is small since their mechanism can only sample uniformly from a sufficiently large number of nodes.

Gkantsidis, Mihail and Saberi (Gkantsidis et al. [2004]) extend the mechanisms presented by Law and Siu (Law and Siu [2003]) to construct expanders more efficiently. However, their approach uses d processes in a $2d$-regular graph, called "daemons". These daemons move around in the topology. Every joining node must be able to find and query a daemon. Thus, as noted in Gkantsidis et al. [2004], this system is only "weakly decentralized". In addition, node departures are handled as in Law and Siu [2003], requiring special messages to be sent by nodes leaving the system.

Pandurangan, Raghavan and Upfal (Pandurangan et al. [2003]) present a distributed solution to constructing constant-degree low-diameter peer-to-peer networks that share many properties with the graphs we construct here. However, their proposal employs a centralized server, known to all nodes in the system, that helps nodes pick random neighbors.

Loguinov et al. (Loguinov et al. [2003]) present a distributed construction of fault resilient networks based on de Bruijn graphs that achieve good expansion. However, they also require nodes leaving the system to contact and transfer state to existing nodes and thus cannot tolerate failures.

4.2 Background material

In this section we present some known results from the theory of random regular graphs and random walks. These concepts are used in the subsequent sections.

4.2.1 Random regular graphs

Let $\mathcal{G}_{n,d}$ denote the set of all d-regular graphs on n nodes and $G_{n,d}$ be a graph sampled from $\mathcal{G}_{n,d}$ uniformly at random. Then $G_{n,d}$ is a *random regular graph*. It is known that random regular graphs have asymptotically optimal *expansion* (we formally define expansion in Section 4.3) with high probability (Friedman [1991]).

Configuration model (Bollobás [1980]) is the standard method for generating random d-regular graphs on n nodes $v_1, v_2, ..., v_n$, though not in a distributed setting. In this model each vertex is represented as a *cluster* containing d elements, called *points*, resulting in n such clusters $\gamma(v_1), \gamma(v_2), ..., \gamma(v_n)$. A *perfect matching* of these nd points is a set of $\frac{nd}{2}$ pairs of points such that every point appears in exactly one pair. Assuming nd is even, many perfect matchings exist for these points. A *uniform random perfect matching* is a perfect matching chosen uniformly at random from the set of all possible perfect matchings. To construct a random d-regular graph on n vertices, a uniform random perfect matching on these nd points is computed and an edge is inserted in the graph between vertices v_i and v_j if and only if the perfect matching pairs a point in $\gamma(v_i)$ to a point in $\gamma(v_j)$. This model allows self loops (pairing points from the same cluster) and parallel edges (more than one pair from the same two clusters) and is very inefficient if the goal is to construct a *simple graph*, i.e., one without self loops and parallel edges. A refinement (Steger and Wormald [1999]) of this model constructs random d-regular simple graphs by pairing points, one pair at a time, from the uniform distribution over all available pairs, i.e., those that do not result in self loops and parallel edges. Graphs generated using this approach are asymptotically uniform for any $d \leq n^{1/3-\epsilon}$, for any positive constant ϵ (Kim and Vu [2001]). In Section 4.4, we extend this model to the distributed setting for building expander graphs.

4.2.2 Uniform sampling using random walks

A random walk on a graph can be modeled as a Markov chain. For a graph containing n nodes, the *probability transition matrix* M of the random walk is an $n \times n$ matrix where each element M_{ij} specifies the probability with which the random walk moves from node i to node j in one step. Let π_t be a vector such that $\pi_t[i]$ is the probability with which the random walk visits vertex i at step t. Then $\pi_{t+1} = \pi_t M = \pi_0 M^{t+1}$. A vector π is called the *stationary distribution* of the random walk if $\pi = \pi M$, i.e., the stationary distribution remains the same after the random walk takes a step, or any number of steps for that matter. It is known that a random walk on a connected undirected graph with an odd cycle has a unique stationary distribution (Lovasz [1993]). *Mixing time* is the time required for the random walk to reach its stationary distribution and it depends on the expansion of the graph: the walk reaches the stationary distribution quickly if the graph is a good expander. A random walk on a graph can be used to sample nodes from the walk's stationary distribution if the walk is run long enough to mix properly.

Let $\Gamma_G(x)$ denote the set of neighbors of node x in graph G. Then a *simple random walk* is a walk which, at each step, moves from a node x in G to one of its neighbors in $\Gamma_G(x)$ with probability $1/|\Gamma_G(x)|$. The stationary distribution of a simple random walk on a regular graph is uniform, i.e, $\pi = \frac{1}{n}[1, 1, ..., 1]$. In case the graph is not regular, the stationary distribution of a simple random walk is a function of the nodes' degrees. One of the known ways (recently also discussed in Awan et al. [2004]; Boyd et al. [2004]) to sample uniformly at random from an irregular graph G with maximum degree $d_{\max}$ is to run a random walk on G that takes a step from node x to node y with probability:

$$P_{xy} = \begin{cases} \frac{1}{d_{\max}} & \text{if } y \neq x \text{ and } y \in \Gamma_G(x) \\ 1 - \frac{|\Gamma_G(x)|}{d_{\max}} & \text{if } y = x \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

We call such a random walk a *maximum degree random walk* and denote it as MDwalk. An MDwalk has a uniform stationary distribution even on irregular graphs but it suffers from two main issues: First in a dynamic distributed system it is often difficult to estimate the maximum degree of the graph. Second, low degree nodes imply higher self transition probabilities (see Equation 4.1) which result in longer mixing times for MDwalks. If MDwalks are not run long enough to achieve sufficient mixing, they are biased towards low-degree nodes.

4.3 System model and goals

Here we present extensions to the system model of Sections 2.2 and 3.2 that are required for the description of the algorithm to construct an expander network on top of the tree structure.

We denote the rooted tree of processes (the server and the proxies) as $T = (V, E_T)$. The vertex set of the tree and the overlay expander that we construct, is the same but their edge sets differ, hence the subscript. For any subset $S \subset V$ we define the set of *neighbors* of S in T as $\Gamma_T(S) = \{y \in V \mid \exists x \in S, (x,y) \in E_T\}$. As usual, processes are initialized only with the identities of their neighbors and do not have access to any central database containing information about T.

Proxies are allowed to join and leave the tree. We further allow proxies to experience fail stop (Schlichting and Schneider [1983]) failures; thus, failure of a proxy can be detected by other processes in the system. Our algorithms are designed independent of a particular fault distribution, however, our experiments use a random distribution of faults. See Goerdt [1998]; Bagchi et al. [2004] for a detailed analysis of how expanders behave under different fault distributions.

We present some notation used to define expander graphs.

Definition 16. *Given a graph $G = (V, E_G)$, the vertex boundary $\partial_G(S)$ of a set $S \subset V$ is $\partial_G(S) = \{y \in V \setminus S : \exists x \in S,\ (x,y) \in E_G\}$.*

Definition 17. *A graph $G = (V, E_G)$ is an (α, β)-expander if for every subset $S \subset V$ of size $|S| \leq \beta|V|$, $|\partial_G(S)| \geq \alpha|S|$.*

Our goals can be summarized as follows: Construct an expander graph with the vertex set V (that consists of all the processes) using a distributed algorithm that scales well. New proxies should be able to join the expander with a low messaging cost even when the expander is very large. In the event of proxy failures, the expander should "self heal" to regain its fault tolerance and the partitioned underlying tree should be patched to a single connected component.

4.4 Distributed expander construction

Our approach is to construct a random graph among vertices in V (processes in the tree) such that processes in the graph have degrees "close to" some constant d. Such a graph is much easier to construct and maintain in a distributed system with dynamic membership than a d-regular random graph, while still achieving comparable expansion.

4.4.1 Random almost-regular graphs

We say a graph is (d, ϵ)*-regular* if the degrees of all vertices in the graph are in the range $[d-\epsilon, d]$. Let $\mathcal{G}_{n,d,\epsilon}$ denote the set of all (d, ϵ)-regular graphs on n vertices, and $G_{n,d,\epsilon}$ be a graph sampled from $\mathcal{G}_{n,d,\epsilon}$ uniformly at random. Then, $G_{n,d,\epsilon}$ is a (d, ϵ)*-regular random graph.* Section 4.4.6 shows that large sets of vertices expand well in a (d, ϵ)-regular random graph when ϵ is small compared to d.

Our distributed construction builds (d, ϵ)-regular random graphs according to the refinement (Steger and Wormald [1999]) of the configuration model (see Section 4.2.1) as shown in Figure 4.1. $\Gamma_G(x)$ is the set containing x's neighbors in the overlay expander. Processes are sampled from the tree (line 4)—using mechanisms discussed later—and added to this set, maintaining a maximum of d neighbors (line 3 and lines 9–12). We avoid self-loops and parallel edges (lines 6 and 7). Upon detecting the failure or departure of an expander neighbor, x removes this proxy from $\Gamma_G(x)$ (line 16).

Using (d, ϵ)-regular random graphs allows us to avoid complicated mechanisms that synchronize the state of departing proxies with processes in the network in an attempt to maintain exactly d neighbors. Instead, we allow proxies to leave without announcing their departure and ignore periods where some processes may have less than d neighbors. A large number of simultaneous failures can result in some processes having degrees even less than $d - \epsilon$, but the fault tolerance of the expander will ensure that most processes remain connected in a component that has high expansion. This allows processes with low degrees to recover “quickly”. We present results related to the convergence rate of the expander under different conditions in Section 4.6.

These mechanisms reduce the problem of constructing $G_{n,d,\epsilon}$ to that of a process $x \in V$ choosing another process uniformly at random from the

```
Every process $x \in V$ executes the following:
Initialization:
1.  $\Gamma_G(x) \leftarrow \emptyset$                      /* Start with no expander neighbors */

Main:
2.  repeat forever
3.      if $|\Gamma_G(x)| < d$                              /* If I have less than $d$ neighbors... */
4.          uniformly sample process $y$ from $V$           /* ...then sample a new process */
5.          send (add : $y$) to $x$                         /* ...and add it as a neighbor */

Upon receiving (add : $y$):                                 /* Sent locally or from another process */
6.  if $y = x$ or $y \in \Gamma_G(x)$                       /* If a self-loop or parallel edge... */
7.      do nothing                                          /* ...then ignore it */
8.  else                                                    /* If not self or existing neighbor... */
9.      if $|\Gamma_G(x)| = d$                              /* ...and if already found $d$ neighbors */
10.         pick $z$ from $\Gamma_G(x)$ at random           /* ...then choose an existing neighbor */
11.         remove $z$ from $\Gamma_G(x)$                   /* ...remove it from neighbor set */
12.         send (Remove : $x$) to $z$                      /* ...and notify removed neighbor */
13.     add $y$ to $\Gamma_G(x)$                            /* ...add sampled process as neighbor */
14.     send (add : $x$) to $y$                             /* ...notify the newly added process */

Upon receiving (Remove : $y$):                              /* If a process removed me as neighbor... */
15. remove $y$ from $\Gamma_G(x)$                           /* ...then remove it from my set as well */

Upon receiving (Failed : $y$)                               /* Received from a failure detector */
16. remove $y$ from $\Gamma_G(x)$                           /* Remove edge to failed proxy */
```

Figure 4.1. Algorithm to generate (d, ϵ)-regular random graph

tree (line 4), i.e., with probability $1/|V|$. Such a sampling procedure could be used by processes to construct and maintain $G_{n,d,\epsilon}$ as described above.

4.4.2 Biased irreversible random walks

For a process, choosing another process uniformly at random from the tree is challenging because the structure is a tree (and not a random graph for example) and because each process only knows about its neighbors in the tree.

We approach this problem by assuming that every process x knows about the number of processes in the tree in the direction of each of its neighbors (we relax this assumption in Section 4.4.3): x knows the size of the subtree rooted at each of its children and x knows the number of processes in the tree that are not in the subtree rooted at x—this is the number of processes in the direction of x's parent. Then, to choose a process uniformly at random from the tree, x starts a *biased irreversible random walk*, BIwalk. At each step, the BIwalk either (i) moves from a process to one of its neighbors in the tree, except the neighbor where it came from—and hence, it is irreversible—or (ii) picks the current process. In case (i), the probability of choosing a neighbor is directly proportional to the number of processes in the tree in the direction of that neighbor—and hence, it is biased. In case (ii), we say the BIwalk *terminates*. The process where the BIwalk terminates adds x to its neighbor set and notifies x. Upon receiving this notification, x also adds the sampled process to its neighbor set, thus forming an undirected edge. We prove that a BIwalk samples processes uniformly at random from the tree when the tree is static.

Let (x, y) be an edge in E_T (E_T is the edge set of T) and $F(V, E_T \setminus \{(x, y)\})$ be the forest containing two components formed by removing (x, y) from T. Then, we define $C(x \triangleleft y)$ to be the component of F that contains process y. The '$\triangleleft$' notation captures the intuition that this is x's view of the tree in the direction of its neighbor y. Let V' denote the vertex set of $C(x \triangleleft y)$; then let $W(x \triangleleft y) = |V'|$. Intuitively, $W(x \triangleleft y)$ represents x's view of the "weight" of the tree in the direction of its neighbor y, i.e., the number of processes in the tree in the direction of y. For convenience, we define $W(x \triangleleft y) = |V|$ if $x \notin V$ and $y \in V$ (the view from outside the tree), and $W(x \triangleleft y) = 1$ if $x = y$ (the view when x looks down at itself).

We denote a BIwalk as a sequence of random variables $X_1, X_2, ..., Y$, where each X_i represents the process that initiates the i^{th} step of the BIwalk (X_1 starts the BIwalk) before the BIwalk terminates at process Y. Note that by definition a BIwalk terminates if and only if it picks the same process twice, i.e., $X_j = X_{j+1}$ and in this case we denote $Y = X_{j+1}$. For notational convenience we define $X_0 = x_0 \notin V$, so for any $x \in V, W(x_0 \triangleleft x) = |V|$. Note that there is a unique BIwalk between every pair of processes in V, since there is a unique path between every pair of processes in the tree and the BIwalk only travels over edges in the tree.

Say the BIwalk moves from process z to a process $x \neq z$ at the $(i-1)^{st}$

step, i.e., $X_{i-1} = z$ and $X_i = x$. Then the probability that the Blwalk moves to a process $y \in V$ at the i^{th} step is given as:

$$\Pr[X_{i+1} = y \mid X_i = x, X_{i-1} = z] = \begin{cases} \frac{W(x \triangleleft y)}{W(z \triangleleft x)} & \text{if } y \in (\Gamma_T(x) \cup \{x\}) \setminus \{z\} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

If $y = x$, i.e., x chooses itself, then by definition the Blwalk terminates at x and $Y = x$. It is easy to see from Equation 4.2 that the Blwalk takes a maximum of $t_{\max}$ steps to terminate, where $t_{\max}$ is the diameter of T. We now prove that the Blwalk samples vertices from V (processes in the tree T) uniformly at random.

Theorem 6. *For every* Blwalk, $\Pr[Y = x_{last}] = 1/|V|$ *for all* $x_{last} \in V$.

Proof. We prove this claim by induction on the size of the tree, $|V|$. For the base case $|V| = 1$, the claim holds trivially since x_{last} is the only process in the tree (by assumption) and so $\Pr[Y = x_{\text{last}}] = 1$.

Assume the claim holds for all trees of size up to k, i.e., for all trees $T = (V, E_T)$ such that $|V| \leq k$. We prove that it holds for $|V| = k + 1$. Say the Blwalk starts at some process $x_1 \in V$, i.e., $X_1 = x_1$. Then there are two possible cases:

(1) $x_1 = x_{\text{last}}$. From Equation 4.2 the probability that the Blwalk terminates at x_1 given that it starts at x_1 is $\Pr[Y = x_1 \mid X_1 = x_1, X_0 = x_0 \notin V] = 1/|V|$, since by definition $W(x_1 \triangleleft x_1) = 1$ and $W(x_0 \triangleleft x_1) = |V|$.

(2) $x_1 \neq x_{\text{last}}$. Let y be the neighbor of x_1 such that x_{last} is in the component $C(x_1 \triangleleft y)$. Then from Equation 4.2 and the definition $W(x_0 \triangleleft x_1) = |V|$, the probability that the Blwalk enters the component $C(x_1 \triangleleft y)$, i.e., steps from x_1 to y is given by:

$$\Pr[X_2 = y \mid X_1 = x_1, X_0 = x_0 \notin V] = \frac{W(x_1 \triangleleft y)}{|V|} \tag{4.3}$$

Note that $C(x_1 \lhd y)$ is a tree of size at most k, since $|V| = k+1$, $x_1 \in V$ and x_1 is not contained in $C(x_1 \lhd y)$. So by assumption once the BIwalk enters the component $C(x_1 \lhd y)$, it terminates at x_{last} with probability

$$\Pr[Y = x_{\text{last}} \mid \mathsf{BIwalk\ reaches}\ y] = \frac{1}{W(x_1 \lhd y)} \tag{4.4}$$

Process y is in the path from x_1 to x_{last} and there is a unique BIwalk between every pair of processes. Therefore, the probability that the BIwalk terminates at x_{last} when $x_1 \neq x_{\text{last}}$ and x_{last} is in the component $C(x_1 \lhd y)$ for some $y \in \Gamma_T(x_1)$, is given by:

$$\begin{aligned}\Pr[Y = x_{\text{last}}] &= \Pr[Y = x_{\text{last}} \mid \mathsf{BIwalk\ reaches}\ y] \times \\ &\quad \Pr[\mathsf{BIwalk\ reaches}\ y] \\ &= \frac{1}{W(x_1 \lhd y)} \times \frac{W(x_1 \lhd y)}{|V|} = \frac{1}{|V|}\end{aligned}$$

□

4.4.3 Reducing message complexity

The mechanism described in Section 4.4.2 assumes that each process x in the tree T knows the weight $W(x \lhd y)$ for each neighbor $y \in \Gamma_T(x)$. At the start of the execution, this can be achieved by an initial messaging round. However, once all the weights are known, the addition or removal of a proxy would require multicasting this information to keep the weights updated at all processes. This is not acceptable due to the large messaging costs this would induce. Furthermore, if multicast is being employed then a trivial solution to uniform sampling from the tree exists: the joining proxy multicasts its arrival and all existing processes reply with their identities allowing the new proxy to choose neighbors uniformly at random.

Our goal is to sample processes uniformly from the tree using an algorithm that requires a much lower messaging cost than multicast. To achieve this we modify the mechanism described in Section 4.4.2 as follows: To choose a process uniformly at random from the tree, a process x first sends a request called BIrequest to the server (the root of the tree). The server then starts a BIwalk on behalf of x. As before, if this BIwalk terminates on a process y, then y adds x to $\Gamma_G(y)$ and x adds y to $\Gamma_G(x)$. Theorem 6 proves that irrespective of where this BIwalk originates (from x or from root), it chooses y uniformly at random.

To understand the effects of this minor change, we first note that Equation 4.2 can also be expressed as:

$$\Pr[X_{i+1} = y \mid X_i = x, X_{i-1} = z] \\ = \begin{cases} \frac{W(x \lhd y)}{1+\sum_{u\in\Gamma_T(x), u\neq z} W(x \lhd u)} & \text{if } y \in (\Gamma_T(x) \cup \{x\}) \setminus \{z\} \\ 0 & \text{otherwise} \end{cases}$$

Thus to compute the transition probabilities, a process x that is currently hosting a BIwalk needs to know the weights of all of its neighbors $u \in \Gamma_T(x)$ except the neighbor z where the BIwalk came from. In the context of the new mechanism this implies that each process only needs to know the weights of its children and not the parent, since the BIwalk always comes from the parent—the BIwalk originates at the root and is irreversible. Therefore, a join or leave operation at process x, i.e., a proxy joins as a child of x or some child of x leaves the tree, now requires updating the weights only at processes that are in the path from x to the root. This takes only $O(\log n)$ messages assuming a balanced tree, a substantial improvement to the multicast required earlier.

4.4.4 Load balancing

The optimization described in Section 4.4.3 reduces message complexity considerably for each update but increases the load on the root, as every BIwalk originates at the root. We reduce this load by interleaving BIwalks with MDwalks (see Section 4.2.2) that run on the expander. Our algorithm constructs the expander incrementally, initially consisting of a small set of processes and growing in size as new processes join the expander by sampling enough neighbors from the tree. We say a process x is an *expander* *process* if $|\Gamma_G(x)| \geq d - \epsilon$. Once an expander is constructed, MDwalks can be used to sample from the set of expander processes.

MDwalks are a good match to our setting because they have a uniform stationary distribution even on irregular graphs (our expander is an irregular graph), the maximum degree of the expander graph is known and the mixing time is small due to high expansion. For our application, MDwalks mix sufficiently in $5\log(m)$ steps, where m is the number of expander processes; a detailed analysis of mixing times on different graphs appears in Boyd et al. [2004]. Processes can estimate the logarithm of expander size using only local information through mechanisms described in Horowitz and Malkhi [2003]. The main assumption in Horowitz and Malkhi [2003] is that a new node joining the network (in our case the graph G) has a randomly chosen existing node as its first contact point. This fits well with our construction as the expander neighbors are chosen uniformly at random.

Using MDwalks in our system, however, raises two issues: First, MDwalks sample from a uniform distribution only if the expander is sufficiently large. Second, if the tree contains many processes that are not expander processes—e.g., if they just joined the tree or if several of their neighbors failed resulting in less than $d - \epsilon$ neighbors—then the MDwalks will only be sampling from a subset of processes, since MDwalks only sample from the expander processes. To address these issues, we develop a "throttling mechanism" shown in Figure 4.2 that results in more MDwalks as the tree becomes large and stable—a large, stable tree implies a large expander covering most processes in the tree. Proxies send BIrequests along the path towards the root so the root can start a BIwalk on their behalf, as described in Section 4.4.3. However, upon receiving a BIrequest from its child, an expander process forwards this request towards the root only with probability p (lines 7 and 8). With probability $1 - p$, the expander process starts an MDwalk (lines 9 and 10) on behalf of the process that initiated the BIrequest. An MDwalk stepping on a process that is not an expander process (i.e., one that does not have at least $d - \epsilon$ expander neighbors) implies that there might be a non-negligible fraction of such processes in the tree. Hence, in this case the MDwalk is interrupted (lines 11 and 12) and a special request BIrequest$'$

is deterministically sent to the root that results in a BIwalk (lines 19–22). When the tree is large, there are more processes in the path to the root and thus a higher probability of starting an MDwalk (lines 7–10). When the tree is stable most processes are expander processes and so MDwalks are not interrupted (lines 11 and 12).

We note that our algorithm cannot add or remove undirected edges to the expander graph instantaneously due to the distributed setting. This could be done using some global locking mechanism but at a considerable performance cost, and is therefore avoided. As a result the expander has some directed edges, e.g., process x has added y to $\Gamma_G(x)$ but y has not yet added x to $\Gamma_G(y)$. The results concerning uniform sampling by MDwalks discussed in Section 4.2.2 relate to undirected graphs only. Therefore, when an MDwalk reaches a process y from a process x such that $x \notin \Gamma_G(y)$, y sends the MDwalk back to x and x chooses another neighbor from the set $\Gamma_G(x) \setminus \{y\}$ according to the transition probabilities in Equation 4.1. This ensures that MDwalks effectively only step from a process to another process if there is an undirected edge between them.

4.4.5 Summary

Our construction of an expander from a tree can be summarized as follows:

– We construct (d, ϵ)-regular random graphs from a tree. Each process uniformly samples processes from the tree and adds them to its neighbor set, maintaining a maximum of d neighbors.

– We use BIwalks to sample processes uniformly at random from the tree. All BIwalks are started from the root as this requires low message complexity for each update.

– As the expander grows, we can reduce load on the root by using MDwalks. MDwalks step across edges of the expander. Our algorithm results in more MDwalks as the tree grows in size and becomes relatively stable.

```
Every process $x \in V$ executes the following:
Initialization (addendum to Figure 4.1):
1.  set parent to $x$'s parent in $T$                /* Initialize parent */

Upon receiving (BIrequest : $u$):                   /* BIwalk request initiated by $u$ */
2.  if $x$ is root                                  /* If I am the root... */
3.      send (BIwalk : $u$) to $y$ chosen using Eq. 4.2    /* ...then initiate BIwalk */
4.  else if $|\Gamma_G(x)| < d - \epsilon$          /* If am not root, and not in expander yet... */
5.      send (BIrequest : $u$) to parent            /* ...then forward BIwalk request to parent */
6.  else                                            /* If am not root, but am in expander... */
7.      with probability $p$                        /* ...then flip a $p$-biased coin, if heads... */
8.          send (BIrequest : $u$) to parent        /* ...then, forward request to parent */
9.      with probability $1 - p$                    /* ...if tails... */
10.         send (MDwalk : $u$) to $y$ chosen using Eq. 4.1/* ...then start an MDwalk on behalf of $u$ */

Upon receiving (MDwalk : $u$):                      /* I am the current step of MDwalk for $u$ */
11. if $|\Gamma_G(x)| < d - \epsilon$               /* If I am not part of the expander... */
12.     send (BIrequest$'$ : $u$) to parent         /* ...then send request for BIwalk towards root */
13. else                                            /* If I am part of the expander... */
14.     choose $y$ using Eq. 4.1                    /* Decide next step of MDwalk */
15.     if $y = x$                                  /* If I am chosen as the next step... */
16.         send (add : $u$) to $x$                 /* ...then add the initiator as a neighbor */
17.     else                                        /* If I am not the next step... */
18.         send (MDwalk : $u$) to $y$              /* ...then send MDwalk to whoever is */

Upon receiving (BIrequest$'$ : $u$):                /* Direct request going to root */
19. if $x$ is root                                  /* If I am the root... */
20.     send (BIwalk : $u$) to $y$ chosen using Eq. 4.2    /* ...then start a BIwalk on behalf of $u$ */
21. else                                            /* If I am not the root... */
22.     send (BIrequest$'$ : $u$) to parent         /* ...then send the request towards root */
```

Figure 4.2. Using MDwalks with BIwalks to reduce root load

4.4.6 Proof of expansion

On an intuitive level, the constructed graph is a good expander because each process keeps at least $d - \epsilon$ neighbors, chosen uniformly at random from the set of all processes; ϵ is a positive constant chosen according to the dynamic conditions in the network, e.g., failure rate of the proxies or network links, etc, and is assumed to be small compared to d.

The empirical results presented in Section 4.6 show that the graphs constructed by our algorithms preserve good expansion and connectivity properties even during very dynamic periods, e.g., when a large number of proxies join or leave the network simultaneously. However, for the analytical results presented in this section, we consider the graph constructed by our algorithm as seen at any particular instant in a stable period, i.e., we take a "snapshot" of the graph at a time when proxies have not recently joined or left the network. This allows us to safely assume that all proxies in the tree have between $d - \epsilon$ and d expander neighbors: if a new proxy joins the network, we give it enough time to sample at least $d - \epsilon$ neighbors, and if some proxies leave the network, we give their expander neighbors enough time to sample replacements so that they have at least $d - \epsilon$ neighbors. Furthermore, we assume that d and ϵ are chosen such that $d - \epsilon \geq 3$, this almost surely results in a connected graph (Wormald [1999]).

Let $V = \{v_1, v_2, \ldots, v_n\}$ denote the n vertices of the graph G as seen with a snapshot taken during some stable period. Let $d_{v_1}, d_{v_2}, \ldots, d_{v_n}$ denote the degrees of the n vertices in this graph, where $d - \epsilon \leq d_{v_i} \leq d$, for $1 \leq i \leq n$. Then the construction of this graph can be modeled as follows (this is similar to the configuration approach described in Section 4.2.1): Each vertex v_i is represented as a *cluster* C_{v_i} of *points*, such that the cluster has cardinality d_{v_i}. The total number of points is $D = \sum_{v_i \in V} d_{v_i}$. We then compute a uniform random perfect matching of these D points. The constructed graph then contains an edge between vertices v_i and v_j, if a point in the cluster C_{v_i} is matched to a point in the cluster C_{v_j}. Note that a graph constructed using this model may not be simple, i.e., it may contain self-loops and parallel edges, whereas the graphs constructed using our algorithm are necessarily simple graphs. However, it is much easier to analyze the graphs constructed in the configurational model, without the restriction of being simple. For the sake of simplicity, our analysis allows self-loops and parallel edges in the graph; we point to the following lemma for proof that our analysis carries over to simple graphs:

Lemma 11 (Molloy and Reed [1999]). *If a random configuration F on a particular degree sequence with constant maximum degree has a property P, then a random simple graph G on the same degree sequence has P. Moreover, the probability that G does not have P, is at most a constant multiple of the probability that F does not have P.*

In order to prove the expansion of our graph G, we utilize the following lemma that states a different but related property, called *conductance*, of a graph constructed using the same random configuration model as used in our construction:

Lemma 12 (Gkantsidis et al. [2003]). *Let $\vec{d} = d_1 \geq d_2 \geq \ldots \geq d_n$ be a sequence of integers with $d_n \geq 3$ and $\sum_{i=1}^{n} d_i = O(n)$. Let $G = (V, E)$ be a graph generated according to the configurational random graph model, such that each cluster C_{v_i} contains d_i points. Then, the conductance of $G = (V, E)$:*

$$\min_{S \subset V, D_S \leq D_V/2} \frac{|\delta(S)|}{D_S} \geq \Omega(1)$$

with probability $1 - o(1)$, where $\delta(S) = \{(v_i, v_j) \in E : v_i \in S, v_j \in V \setminus S\}$, $D_V = \sum_{v \in V} d_v$, and $D_S = \sum_{u \in S} d_u$.

Note that our graphs satisfy $\sum_{i=1}^{n} d_i = D_V = O(n)$, as each process's degree is between d and $d - \epsilon$ for some constants d and ϵ. Let $s = |S|$ and $d_{\mathsf{avg}}(S) = \frac{1}{s} \sum_{u \in S} d_u$, i.e., $d_{\mathsf{avg}}(S)$ is the average degree of processes in the set S, then $D_S = s d_{\mathsf{avg}}(S)$, and $D_V = n d_{\mathsf{avg}}(V)$. We can now restate the conductance bound from Lemma 12 as follows:

$$\mathsf{Pr}\left[\min_{S \subset V, s \leq \frac{n}{2} \frac{d_{\mathsf{avg}}(V)}{d_{\mathsf{avg}}(S)}} |\delta(S)| \geq \alpha' s d_{\mathsf{avg}}(S)\right] \geq 1 - o(1) \tag{4.5}$$

for some $\alpha' = \Omega(1)$. We now state the main theorem for the expansion of our graphs.

Theorem 7. *Let $\vec{d} = d_1 \geq d_2 \geq \ldots \geq d_n$ be a sequence of integers with $d - \epsilon \leq d_i \leq d, d - \epsilon \geq 3$ and $\sum_{i=1}^{n} d_i = O(n)$. Let $G = (V, E)$ be a graph generated according to the configurational random graph model. Then there is a positive constant α (the expansion factor) such that:*

$$\mathsf{Pr}\left[\min_{S \subset V, s \leq \beta n} |\partial(S)| \geq \alpha s\right] \geq 1 - o(1)$$

where $\beta = \frac{d_{\mathsf{avg}}(V)}{2d_{\mathsf{avg}}(S)}$ and $\partial(S) = \{y \in V \setminus S : \exists x \in S,\ (x, y) \in E\}$.

Proof. First note that Equation 4.5 implies that for any set S of size $|S| = s \leq \beta n$, at least a fraction of all edges incident on the processes in S, have the other endpoint in $V \setminus S$. This allows us to directly give a lower bound on the number of distinct neighbors chosen by these edges between S and $V \setminus S$. In particular, if there are $\alpha' s d_{\mathsf{avg}}(S)$ edges coming out of the set S, then

$$\begin{aligned} |\partial(S)| &\geq \frac{\alpha' s d_{\mathsf{avg}}(S)}{d} && (4.6) \\ &\geq \frac{\alpha' s (d - \epsilon)}{d} && (4.7) \end{aligned}$$

Equation 4.6 is due to the fact that out of the $\alpha' s d_{\mathsf{avg}}(S)$ edges coming out of S, at most d can choose the same vertex in $V \setminus S$. Equation 4.7 follows since $d_{\mathsf{avg}}(S) \geq d - \epsilon$. We can therefore, use the bound in Equation 4.5 and state the bound for the graph expansion as:

$$\mathsf{Pr}\left[\min_{S \subset V, s \leq \beta n} |\partial(S)| \geq \frac{\alpha' s (d - \epsilon)}{d}\right] \geq 1 - o(1)$$

This proves the theorem for $\alpha = \frac{\alpha'(d-\epsilon)}{d}$. □

4.5 Tree reconstruction after failures

Keeping the tree connected is essential for applications that need to communicate across tree edges, e.g., Quiver's object migration protocol. It is also desirable in the construction of the expander, especially in dynamic scenarios when we need to use BIwalks that run across tree edges. Here we present a distributed algorithm that uses the fault-tolerant expander to "patch" the tree in the event of proxy failures.

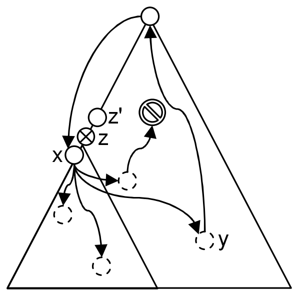


Figure 4.3. Tree maintenance using the expander. Triangle denotes the tree. Small triangle denotes the subtree rooted at the failed proxy z. Curved arrows show tokens sent by z's child x to x's expander neighbors denoted by dashed circles.

When a proxy z fails, the parent z' of z simply removes its failed child from $\Gamma_T(z')$ and sends the updated weight to its own parent (except when z' is root), similar to the case of a proxy joining. It would seem that a child x of z also only needs to remove z from $\Gamma_T(x)$ and connect itself as a child of some randomly chosen expander neighbor in $\Gamma_G(x)$. However, if this randomly chosen neighbor is in the subtree rooted at the failed proxy z, i.e., in the component $C(z' \triangleleft z)$ (shown by the small triangle in Figure 4.3), then connecting x (and any other children of z) to this proxy would still leave the tree partitioned. Therefore, x must find an expander neighbor $y \in \Gamma_G(x)$ (x's expander neighbors are shown as dashed circles in Figure 4.3) such that $y \in C(z \triangleleft z')$, i.e., y is in the component that contains the root of the tree.

Our approach to find such a process is to send "tokens" from x to its expander neighbors. Upon receiving such a token, a proxy forwards the token to its parent in an attempt to reach the root. If such a token does in fact reach the root, it implies that the corresponding neighbor in $\Gamma_G(x)$ (y in Figure 4.3) is in the component containing the root. The root then sends the token back to x and x attaches itself as a child of y.

```
Every process $x \in V$ executes the following:
Upon receiving (Failed : parent):               /* I am disconnected from primary partition */
1.  parent $\leftarrow \perp$                    /* Old parent has failed */
2.  start timer                                  /* Start a timer */
3.  for each $y \in \Gamma_G(x)$                 /* Pick each expander neighbor one by one... */
4.      send (Tok : $x, y$) to $y$               /* ...and send a Tok to it */

Upon receiving (TimerExpired : ):                /* If the timer expires... */
5.  re-join the tree, set parent to new parent   /* ...then reconnect using default procedure */

Upon receiving (Tok : $x', y'$):                 /* Received from another process */
6.  if $x$ is root                               /* If I am root... */
7.      send (Tok : $x', y'$) to $x'$            /* ...then send the token back to initiator */
8.  else if $x = x'$ and (Tok : $x', y'$) is sent by root /* If I initiated this token coming from root... */
9.      if parent $= \perp$                      /* ...and I haven't reconnected yet... */
10.         parent $\leftarrow y'$, update weight at $y'$ /* ...then reconnect through expander neighbor */
11.         stop timer                           /* ...and stop the timer */
12. else                                         /* If I am not the root or the initiator... */
13.     send (Tok : $x', y'$) to parent          /* ...then forward Tok towards root */
```

Figure 4.4. Tree maintenance in the presence of proxy failures

Figure 4.4 shows the distributed algorithm run by proxy x in case it detects that its parent has failed. The mechanism described above provides only a probabilistic guarantee to find a process in the component containing the server. Therefore, x starts a timer (line 2) before sending the tokens. If a suitable candidate for the new parent is not found within the specified timeout period, e.g., because a large fraction of proxies failed simultaneously, then x re-joins the tree using the default mechanism (line 5); in Quiver, this default mechanism is to contact the root of the tree. We presume that the default joining mechanism is more costly, e.g., because it involves manual intervention or a central coordination point, as in Quiver.

The token (Tok : x, y) sent by x to its expander neighbor y (line 4) is forwarded along the path from y to the root (line 13), which finally returns the token back to x (lines 6 and 7). x sets y as its parent, unless the parent has already been set to another process, e.g., because the timer expired or because a different token was received from the root earlier (lines 8–11). To avoid complex scenarios that could result in the formation of cycles, a process x must discard tokens of the form $(\mathsf{Tok} : x, y)$, if the token is forwarded to x by a child. In addition, "nonces" should be used to distinguish between tokens sent across different runs of the protocol. We omit these details from the pseudo-code for brevity.

4.6 Simulation results

We present simulation results measuring graph expansion and connectivity under different conditions. These results validate our expander construction and prove that the resulting graph is tolerant to proxy failures. We also show that using the mechanisms described in Section 4.4.4, the load is better distributed among processes in the tree during stable periods, since most BIrequests result in MDwalks. When the tree is more dynamic thus causing more BIwalks, the load on the server is higher than the load on the proxies, roughly by a constant amount, even as the number of proxies increases. These two results provide evidence that our algorithm scales well.

Verifying if a graph is an expander is co-NP-complete (Blum et al. [1981]) since it requires verifying the expansion of an exponentially large number of subsets of vertices. However, we can estimate a graph's expansion by computing the second smallest eigenvalue λ of the graph's *Laplacian matrix*: a graph is a $\frac{2\lambda}{2\lambda+\Delta}$-expander, where Δ is the maximum degree of the graph (Alon [1986]), in our case $\Delta = d$. We use Kleitman's algorithm (Kleitman [1969]) to find the vertex connectivity of the expander. We note that the theory behind these well known results deals only with undirected graphs. Thus, we ignore any directed edges in the expander network when computing its expansion and connectivity. Therefore, the results reported in this section are pessimistic in the sense that our graphs actually have more edges which are not represented here.

We developed a round-based simulator in Java. The simulator sets up an initial topology by constructing a random tree containing $d + 1$ processes.

Processes construct (d, ϵ)-regular random graph with $\epsilon = d/2$ (a pessimistic value, simulating highly dynamic conditions in the tree) overlayed on this random tree using mechanisms described in earlier sections. In what follows, n denotes the upper bound on the number of processes (one server and $n-1$ proxies) used in the experiment; i.e., if joins are being simulated then proxies are added until the total number of processes is n, whereas if failures are being simulated then proxies are removed starting with an initial set of n processes. To simulate proxies joining the tree, n_{add} proxies are added to the random tree after every T_{add} rounds until the total number of processes in the tree becomes n. Each of the n_{add} proxies is added as a child to an existing process chosen from a distribution that picks more recently added processes with a higher probability. This is done so that the experiments measuring the load on different processes are not affected due to a process having many more children than other processes. To simulate proxy failures, we remove n_{remove} proxies, chosen uniformly at random from the tree, after every T_{remove} rounds. Proxies in the tree send BIrequests to their parents every T_{walk} rounds if they have less than d neighbors. T_{walk} simulates latency and other factors in real networks. We vary this parameter in some experiments to see the effect of these delays on the convergence rate of our algorithm. BIrequests are forwarded by expander processes to their parents with probability $p = 0.5$ (the choice is arbitrary, smaller p will obviously reduce load on the root). Expander processes initiate MDwalks with probability $1 - p$. We specify values for $n, n_{\text{add}}, n_{\text{remove}}, d, T_{\text{add}}, T_{\text{remove}}$ and T_{walk} for different experiments as these are all tunable parameters.

Figure 4.5 plots the graph expansion and connectivity for different values of $\frac{T_{\text{walk}}}{T_{\text{add}}}$. In this experiment we use $n = 200$, $d = 20$, $n_{\text{add}} = 20$, $T_{\text{add}} = 200$ and values $10, 20, 30$ and 50 for T_{walk}. We compute the expansion and connectivity of the graph every single round. The plot shows expansion and connectivity after the first 100 proxies have already been added (only to make the figure more visible). Each point in the plot is a mean of 30 tests, each starting from a new random tree. Expansion and connectivity are computed for all processes in the tree, not just the expander processes so that we can see the time it takes for the new proxies to be added to the expander.

When new proxies are added to the tree, expansion and connectivity go down to zero since the new proxies do not have any neighbors in the expander yet. A larger $\frac{T_{\text{walk}}}{T_{\text{add}}}$ ratio implies that processes look for expander neighbors slowly while the graph is changing fast. This results in the graph taking a longer time to achieve better expansion and connectivity as shown in the figure.

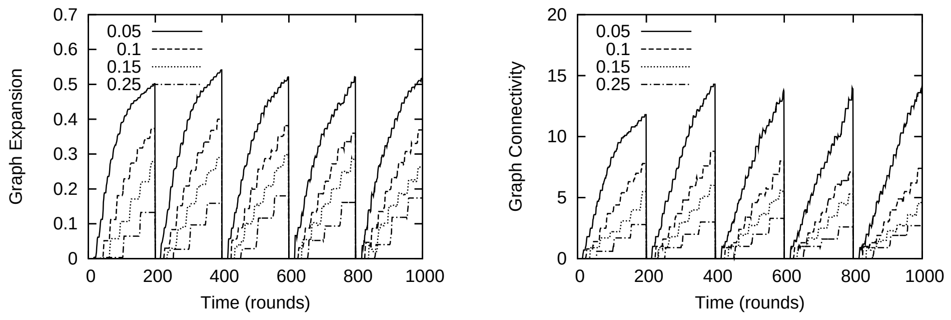


Figure 4.5. Expansion and Connectivity for various values of $\frac{T_{\text{walk}}}{T_{\text{add}}}$. 20 proxies are added every 200 rounds. Shows how quickly new proxies join the expander.

Figure 4.6 shows network behavior in the presence of proxy failures. Proxies run the algorithm from Section 4.5 to re-connect the tree after their neighbors fail so they can still run Blwalks. For this experiment we use $n = 150, d = 20, n_{\text{remove}} = 10, T_{\text{walk}} = 10$ and $T_{\text{remove}} = 300$. All 150 proxies are first added to the tree and we wait 1000 rounds for the expander to be constructed (this period is not shown in the plot). We then start measuring the expansion and connectivity every single round and remove 10 proxies after every 300 rounds until we are left with 100 processes. Each point in the plot is a mean of 15 tests, each test starting from a different random tree. The figure shows that proxy failures affect the graph expansion and connectivity slightly and the expander attempts to regain any lost fault tolerance during stable periods.

Figure 4.7 plots graph expansion and connectivity against different values of d. We use $n = 200$, $n_{\text{add}} = 20$, $T_{\text{add}} = 100$, $T_{\text{walk}} = 10$ and varied $d = 6, 10, 15, 20, 25, 29, 33$ and 38. For each value of d, we waited 1500 rounds after adding all proxies to the graph to give enough time to construct the

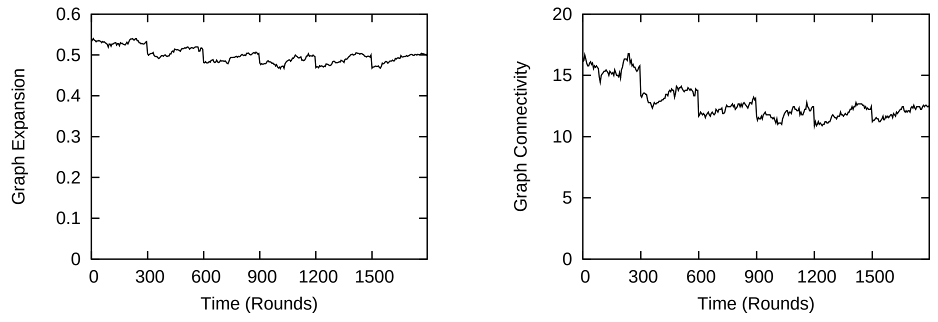


Figure 4.6. Expansion and Connectivity as proxies fail. 10 proxies fail every 300 rounds. Failures have a minor effect and any lost expansion and connectivity is regained in stable periods.

expander and then measured expansion and connectivity. For each d, we repeated this process 30 times on different random trees and plot the average expansion and connectivity of these 30 results. As shown in the figure, expansion and connectivity increase with d. Our graphs achieve reasonable fault tolerance even for small values of d like $d = 10$.

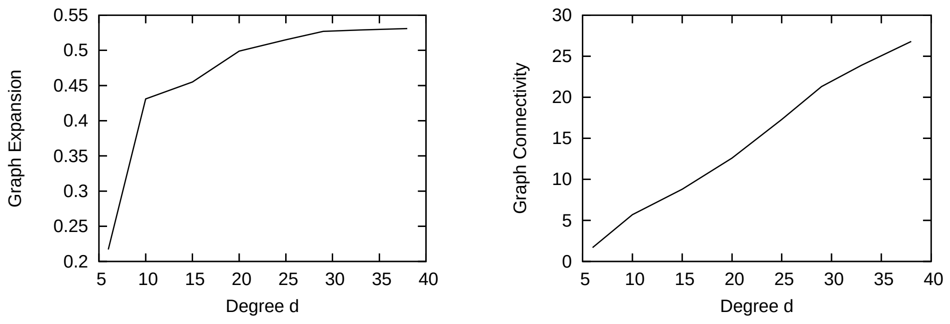
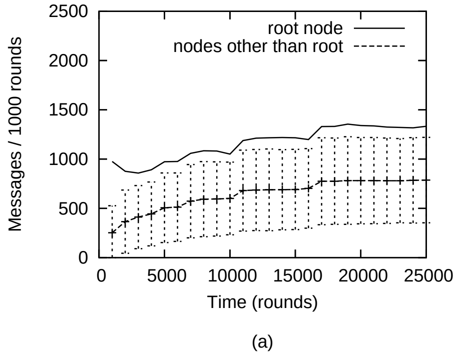


Figure 4.7. Expansion and Connectivity for various values of d.

Figure 4.8-(a) compares the load (number of messages handled) on the server (the root) with the mean load on proxies (and the standard deviation) in the tree. For this experiment we used $n = 2500, d = 20, n_{\text{add}} = 10, T_{\text{add}} = 100$ and $T_{\text{walk}} = 10$. We start measuring the load from the first round by counting the number of messages received by each process every 1000 rounds. We stop the experiment when all proxies are added to the tree, i.e., after

25000 rounds. Each point in the plot is a mean of 30 tests with each test starting from a different random tree. The dashed curve plots the mean load seen by all proxies along with the standard deviation. This standard deviation is high since proxies closer to the root have a higher load than proxies closer to the leaves. The plot shows a slight increase in the load on all processes as the number of processes in the graph increases. This is because the MDwalks run longer as the number of processes increases—we use MDwalks of length $5 \log(m)$, where m is the number of processes in the expander (see Section 4.4.4). However, this effect becomes less visible when the number of processes is large. Also note that the load on the server is higher than the load on the proxies only by a constant amount, even as the number of processes increases. This constant can also be controlled using the parameter p (see lines 7–10 in Figure 4.2). We use $p = 0.5$ in all experiments, a smaller value would reduce the constant difference between the load on the server and the proxies.

Figure 4.8-(b) plots the mean load per process against the level in the tree, with the server at level zero. We use the same values for all the parameters as in Figure 4.8-(a), except n_{add} which is varied $n_{\text{add}} = 10, 15, 20, 25$. Higher n_{add} implies a more dynamic tree and thus results in more BIwalks causing a higher load on the processes close to the root. Smaller n_{add} implies a less dynamic tree resulting in more MDwalks and a better distribution of load across all processes in the tree.



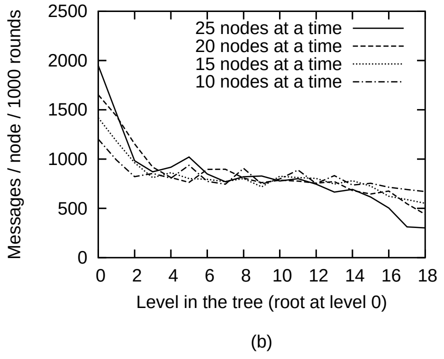


Figure 4.8. (a) Load on the server vs mean load on proxies. Server load is higher only by a constant amount. (b) Load at different levels is better distributed when tree is less dynamic.

5 Distributed Self-Optimizing Trees

In order to exploit locality in the workload, Quiver arranges the proxies in a location-aware tree. However, there are no constraints on the diameter of this tree, and in the worst case it can be completely degenerate depending on the proxies' join procedure. Even if the tree is initially constructed to have a low diameter, the disconnections and reconnections through the random expander graph can increase this diameter with time. A tree with a large diameter results in poor performance for updates and strictly serializable reads, as these have to send requests or migrate objects across more hops.

This chapter presents an extension to Quiver that reduces the "effective" diameter of the tree through a distributed algorithm that brings the processes frequently requesting the same objects, closer to each other in the tree. The algorithm is based on the heuristic that if two processes *access* each other, e.g., because one migrates or reads (strictly serializable version) the object from the other, then they are likely to access each other again in the future for performing more operations on the same or other "semantically related" objects. Thus, bringing these processes close to each other in the tree would reduce the messaging costs for future requests and migrations between them. We analytically prove the bounds on the cost of restructuring the tree, and evaluate the performance gains of employing this self-adjusting tree for a generic flood-based access protocol via experiments on PlanetLab.

5.1 Related work

Our work is inspired by work on balancing binary search trees (BSTs) in a centralized system (e.g., Adelson-Velskii and Landis [1962]; Sleator and

Tarjan [1985]; Bayer [1972]; Guibas and Sedgewick [1978]), particularly the work on *splay trees* (Sleator and Tarjan [1985]). A splay tree is an elegant BST that achieves $O(\log n)$ amortized cost per access. When a node in the tree is accessed, splaying brings the node to the root of the tree, while balancing the tree in the process.

Our work differs from splaying in two ways. First, flattening employs a different heuristic that brings an accessed proxy (the target of an object migration or strictly serializable read request) close to the process initiating the access, by restructuring along the path between these two processes—as opposed to bringing the accessed proxy close to the root by restructuring along the path to the root. In this way, our heuristic often enables more efficient implementations involving less restructuring than splaying. Indeed, the amount of restructuring performed by splay trees is a limitation in the centralized setting as well, and has been addressed previously; e.g., variants like *semi-splaying* (Sleator and Tarjan [1985]), *randomized splaying* Furer [1999]; Albers and Karpinski [2002] and *periodic splaying* Williams et al. [2001], all attempt to reduce restructuring.

Second, splaying uses non-local restructuring steps; this is mainly a result of restructuring in the context of binary search trees that require preserving the order of nodes in the tree. In particular, top-down splaying involves distant nodes in the original tree to form an edge with each other within a single restructuring step. This not only complicates the process of enforcing local policies (like preserving geographic locality in the tree), but can also make it difficult for an application's routing protocol, e.g., the object migration protocol in Quiver, to adjust routes (processes' local queues) according to the restructuring. Flattening improves on both of these aspects through local restructuring steps—an optimization enabled since our target setting does not require preserving the order of processes in the tree.

Apart from splay trees, other balanced tree structures (e.g., Bayer [1972]; Lehman and Yao [1981]; Adelson-Velskii and Landis [1962]; Guibas and Sedgewick [1978]) have also been proposed. Most of these proposals explicitly balance the tree with each insertion and deletion, incurring a high cost for these operations. Distributed implementations of some of these algorithms (but not splaying) have also been proposed (e.g., Gilon and Peleg [1991]; Johnson and Colbrook [1992]; Peleg [1990]; Jagadish et al. [2005]). Our approach is different in that it is less sensitive to insertions and deletions of proxies, and more sensitive to the actual workload. That is, our algorithm focuses its restructuring on the communication paths that are actually used, thereby yielding better performance for some workloads than even explicit balancing can achieve.

5.2 System model

We initially assume that the server and the proxies are arranged in a binary (though not necessarily complete) tree, and relax this assumption in Section 5.4.4. The algorithms presented here make no assumptions about proxies joining or leaving the tree, except that the tree remains connected, e.g., through the use of the expander, see Chapter 4. As usual, each proxy is initialized only with the identities of its neighbors in the tree, i.e., a parent pointer (the distinguished value "$\perp$" in the case of the root) and a set of child pointers of cardinality at most two.

We say a process p *accesses* a process p', when p sends a request to either migrate or simply copy an object (e.g., for a strictly serializable read operation) to itself, and p' is the process that serves this request, i.e., migrates or copies the requested object back to p (see Chapter 2). The process that initiates the access request is denoted as the *requestor*, and the process that is being accessed is denoted as the *target*.

5.3 Overview

On a high level, our algorithm works as follows: When a requestor process r accesses a target process t, flattening is performed along the path between t and r. In particular, *bottom-up flattening* (Section 5.4.1) is employed while moving up the tree and *top-down semi-flattening* (Section 5.4.2) is used while moving down the tree. When this restructuring completes, t and r are closer to each other than before, and the height (distance from the root) of all the proxies in the path between t and r is reduced, i.e., the smallest subtree containing both t and r is left more balanced. This restructuring of the tree is not performed in the "critical path" of the access protocol (or more specifically, of the update or strictly serializable read operation that initiated this request), but rather as a background process: our tree simply "observes" the workload, and then optimizes itself so future accesses may be performed more efficiently.

In order to avoid concurrent restructuring of the tree, which would require expensive locking of parts of the tree, processes share a special object, called *token*. The token is migrated through the same protocols used for migrating service objects, see Chapter 2. When a requestor r completes its access operation, it migrates the token to itself. After becoming the new (and only) owner of the token, r notifies t, and t in turn initiates restructuring along the path to r, using the local queues at each intermediate process to navigate the restructuring towards r—the owner of the token; this is similar to the way strictly serializable single-object read requests are navigated to the current owner of the object, see Section 2.5.2. An intermediate process x uses its local queue for the token to find the next process in the path from t to r (denoted as $\mathsf{nextProcess}(x, t, r)$ in our pseudocode). The local queue also allows x to find if x is the highest process in this path or not (denoted as $\mathsf{amHighProcess}(x, t, r)$ in the pseudocode), e.g., by observing which neighbor the restructuring is coming from, and the neighbor pointed to by $x.\mathsf{localQ}$. Note that as the tree is restructured, the local queues at the effected processes need to be adjusted accordingly to reflect the new tree topology. Since flattening works in local restructuring steps, adjusting these queues is feasible. We summarize the high level steps as follows:

(1) Requestor r accesses target t for migrating or copying an object using protocols described in Chapter 2.

(2) r migrates the token to itself, ensuring that no other process restructures the tree concurrently.

(3) r notifies t, and t initiates flattening, that is navigated from t to r using the head pointers of the local queues for the token, at each intermediate process.

(4) Bottom-up flattening and top-down semi-flattening algorithms are used when going up and down the tree, respectively, along the path from t to r.

(5) As flattening proceeds in local steps, the processes involved at each step adjust their local queues (for all the objects) to reflect the new topology.

Note that steps 2–5 are not in the critical path of the application workload (step 1), and in particular, r (or any other process) may initiate subsequent operations without waiting for the completion of steps 2–5. Furthermore, the migration of the token need not be synchronized with any of the service objects: as such this protocol minimizes the effects of flattening costs on the operations performed by Quiver proxies.

5.4 Flattening algorithms

Binary search trees use the "move to front" heuristic and *rotate* accessed nodes close to the root, since all searches in a BST start from the root. In our setting however, a request may originate from any process in the tree. So a better heuristic is to move the targets close to the requestors. In order to achieve this, we rotate the requestors and the targets close to the root of the smallest subtree that contains them. This scheme minimizes restructuring in the tree (compared to some BSTs that rotate nodes all the way to the root of the tree) if the requestor and the target are already close to each other. We implement this scheme by restructuring along the path from the target to the requestor, employing both *bottom-up flattening* and *top-down semi-flattening* techniques, which we detail in Sections 5.4.1 and 5.4.2, respectively. This combination of "full" and "semi" flattening also allows our algorithm to adapt rather quickly to changing workloads

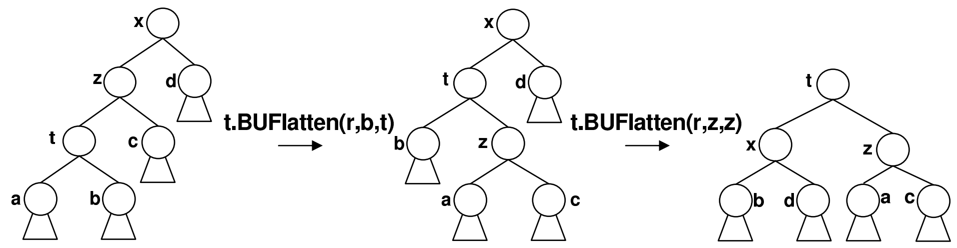


Figure 5.1. Bottom-up flattening: t first rotates over z and then x. Any child may be preferred for the first rotation (b preferred here). For subsequent rotations, preferred child is the one that t last rotated over (z here).

while still being conservative about the number of messages exchanged for restructuring purposes.

Most existing restructuring techniques (again used in the context of BSTs) employ *rotation* as the basic restructuring step. This is convenient as rotation preserves the order of nodes in the tree—a requirement for binary search trees. Since ordering of processes is irrelevant in our target protocols, we define and use new primitives that are better suited to our goals. Here we present these primitives and the bottom-up, top-down and hybrid (that combines bottom-up and top-down) flattening algorithms that use these primitives.

5.4.1 Bottom-up flattening

Our first algorithm is a bottom-up scheme that is employed when navigating up the tree from the target t to the requestor r. Bottom-up flattening starts from t and proceeds to the highest process in the path to r. In case t is this highest process, no bottom-up restructuring is performed. The result of bottom-up flattening is to bring t to the root of the subtree that contains r (except when this root is the server, and t is a proxy), while leaving the subtree containing t and r more balanced than before.

```
1.  $t$.BUFlatten($r, b, w$)                                /* $r$: requestor, $b$: pref. child, $w$: $t$.parent's child */
2.      $a \leftarrow$ elmt($t$.children $\setminus \{b\}$)             /* $a$ is the child not preferred */
3.      $z \leftarrow t$.parent                                 /* $z$ is the current parent */
4.      $t$.children $\leftarrow \{t$.children $\setminus \{a\}\} \cup \{z\}$     /* replace child $a$ with $z$ */
5.      [gParent, isHigh] $\leftarrow z$.rotEdge($t, r, w, a$)   /* $z$ replaces its child $w$ with $a$, sets $t$ as parent */
6.      $t$.parent $\leftarrow$ gParent                          /* set new parent to old grand-parent */
7.      $a$.setParent($z$)                                  /* $a$.parent now points to $z$ */
8.      if isHigh is true                                  /* if $z$ was the highest process in the path, then... */
9.          $t$.parent.replaceChild($z, t$)                  /* ...new parent replaces its child $z$ with me; stop */
10.     else $t$.BUFlatten($r, z, z$)                        /* otherwise, perform next rotation preferring $z$ */

11. $z$.rotEdge($t, r, w, a$)                               /* $t$: responder, $r$: requestor, replace child $w$ by $a$ */
12.     $x \leftarrow z$.parent                                 /* $x$ is my current parent */
13.     $z$.parent $\leftarrow t$                                 /* set $t$ as new parent */
14.     $z$.children $\leftarrow \{z$.children $\setminus \{w\}\} \cup \{a\}$     /* replace child $w$ with $a$ */
15.     return [$x$, amHighProcess($z, t, r$)]              /* return $x$ and if I am highest in this path */

16. $x$.replaceChild($z, t$)                                /* $z$: child to replace, $t$: new child */
17.     $x$.children $\leftarrow \{x$.children $\setminus \{z\}\} \cup \{t\}$     /*replace child $z$ with $t$ and return */

18. $a$.setParent($z$)                                      /* $z$: new parent */
19.     $a$.parent $\leftarrow z$                                 /* set parent to $z$ and return */
```

Figure 5.2. Bottom-up flattening. All processes implement all algorithms. A proxy never rotates over the root, i.e., the server. But this case is omitted from the pseudocode for brevity.

Preferred rotation primitive

We define a variation of the well-known rotation primitive, for bottom-up flattening. For each rotation performed by the target t over its parent z, t chooses one of its children as a *preferred child.* The rotation is performed such that t keeps the preferred child and "hands-off" the other child to z. We call this a *preferred rotation.* Preferred rotations are used in bottom-up flattening as shown in Figure 5.1. For the first rotation, t chooses either one of its children as the preferred child. For each subsequent rotation, the child that t just rotated over in the previous step (process z in Figure 5.1) is preferred. t performs these steps until it rotates over the highest process in the path to r, or until t becomes a child of the server: proxies do not rotate over the server.

Bottom-up flattening algorithm

Figure 5.2 shows the distributed algorithm that implements bottom-up flattening. We denote the variables encoding persistent state at a process y using the prefix "$y.$", e.g., $y.\mathsf{parent}$. Variable names without the prefix denote temporary state that is deleted once this invocation is over.

The target t initiates bottom-up flattening by invoking $t.\mathsf{BUFlatten}(r, b, t)$, where r is the requestor and b is t's preferred child. In line 2, $\mathsf{elmt}(S)$ simply returns the element of a singleton set S; this element is the non-preferred child of t. If t is initially a leaf then $b = \perp$ and $a = \perp$ (line 2). If t only has one child then b is that child and $a = \perp$. We assume that when there is a remote invocation on a $\perp$ process, the method returns (possibly with an error message) so the invoking process can carry on its execution. The $\mathsf{rotEdge}$ invocation (line 5) results in z setting $z.\mathsf{parent}$ to t (line 13) and adding t's non-preferred child a to $z.\mathsf{children}$, replacing t (line 14). Note that t's new parent after each preferred rotation (t's grand-parent before the rotation) need not be notified of its new child t, since t is going to rotate over this process anyway in the next step. Therefore, at each subsequent step after the first rotation, $t.\mathsf{parent}$ does not contain t in its $\mathsf{children}$ set but rather contains the process z that t

just rotated over in the previous step. After the last rotation, t.parent is notified of its new child (line 9). The RPCs in lines 5, 7 and 9 ensure that all restructuring is complete by the time the last rotation completes. Proxies do not rotate over the server, and in particular, if z in line 3 is the server, then t stops bottom-up flattening; this is omitted from the pseudocode for brevity and in the interest of generality of this algorithm to other applications.

5.4.2 Top-down semi-flattening

Our second algorithm is a top-down scheme that restructures the tree whenever navigating down the tree from t to r. Top-down semi-flattening starts at the highest process in the path from the target t to the requestor r and brings r part way up to this highest process. In case top-down semi-flattening is preceded by the bottom-up variant (as in hybrid flattening, Section 5.4.3), this highest process is, in fact, t, or the server (the root) in which case t is a child of the server.

Child swap primitive

Top-down semi-flattening is performed by repeating the step shown in Figure 5.3. y, x and a are in the path from t to r. Process y swaps its child c with x's child a. We call this step *child swap*. "+" represents the current process of the flattening operation, i.e., the next child swap is performed by a. Top-down semi-flattening is initiated by the highest process in the path between t and r, and terminates if r is the current process or a child of the current process.

Top-down semi-flattening algorithm

Figure 5.4 shows the distributed algorithm for this scheme. The algorithm is initiated by the highest process h in the path from t to r as h.TDSemiFlatten$(t, r, h.\text{parent})$. At each step, the current process y and its child x on the path from t to r swap y's child that is not in this path with

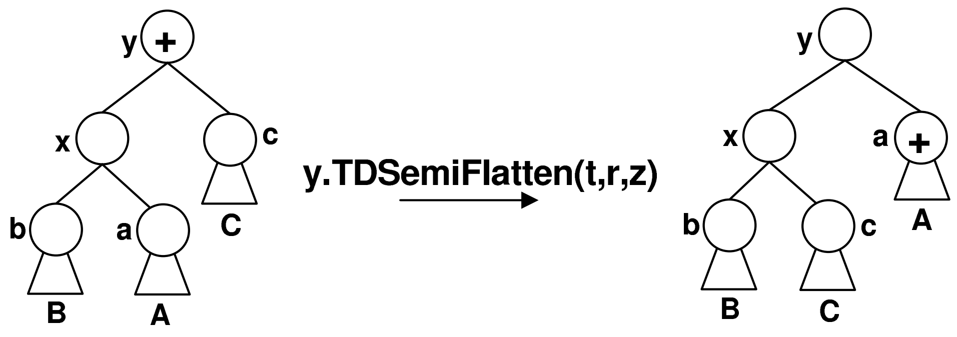


Figure 5.3. Top-down semi-flattening: y, x and a are in the path from t to r. z is y's parent (not shown). Next invocation is a.TDSemiFlatten(t, r, y).

x's child that is in this path (lines 9 and 13). The children are notified of their new parents (lines 7, 10).

Top-down semi-flattening approximately halves the depth of each proxy (relative to h) in the path from h to r. As a result, semi-flattening brings r closer to t.

5.4.3 Hybrid flattening

Our main algorithm combines bottom-up flattening with top-down semi-flattening to restructure along the path from the target t to the requestor r. Figure 5.5 shows the distributed algorithm for hybrid flattening. t performs bottom-up flattening if it is not the highest process (lines 2–7). This results in t either becoming the root of the subtree that contains r, or a child or this root, if the root is the server (and t is not). After bottom-up flattening is complete, if r is a child of t then no more restructuring is required (line 8). Otherwise, t initiates top-down semi-flattening (line 9) that continues until r is reached.

Hybrid flattening restructures along the whole path from t to r. Figure 5.6 shows an example tree where r accesses the target t. Hybrid flattening brings t and r close to each other and in process, balances the subtree containing t and r.

```
1.  $y$.TDSemiFlatten($t, r, z$)                    /* $t$: responder, $r$: requestor, $z$: my new parent */
2.    $y$.parent $\leftarrow z$                          /* set parent to $z$ */
3.    if $r \in \{y\} \cup y$.children                /* if I or my child is the requestor, then...*/
4.      stop                                        /* ...stop the restructuring */
5.    $x \leftarrow$ nextProcess($y, t, r$)               /* find the child that is in path from $t$ to $r$ */
6.    $c \leftarrow y$.children $\setminus \{x\}$             /* this is the child not in path */
7.    $c$.setParent($x$)                              /* $c$'s parent should now be $x$ */
8.    $a \leftarrow x$.childSwap($t, r, c$)              /* swap child at $x$ and get the grand-child in path */
9.    $y$.children $\leftarrow \{y$.children $\setminus \{c\}\} \cup \{a\}$ /* swap child with grand-child */
10.   $a$.TDSemiFlatten($t, r, y$)                    /* initiate next child swap; this is non-blocking */

11. $x$.childSwap($t, r, c$)                          /* $t$: responder, $r$: requestor, $c$: sibling not in path */
12.   $a \leftarrow$ nextProcess($x, t, r$)               /* find my child that is in path from $t$ to $r$ */
13.   $x$.children $\leftarrow \{x$.children $\setminus \{a\}\} \cup \{c\}$ /* swap child with parent's child */
14.   return $a$                                    /* return my child that has been swapped */
```

Figure 5.4. Top-down semi-flattening. All processes implement all algorithms.

```
1.  $t$.HybridFlatten($r$)                  /* $r$: requestor */
2.    if amHighProcess($t, t, r$) is false  /* BUFlatten if I am not the highest process */
3.      $\{a, b\} \leftarrow t$.children           /* $a$ and $b$ are $t$'s children, could be null */
4.      if $a = \perp$                        /* If $a$ is null... */
5.        prefChild $\leftarrow b$              /* ...choose the non-null child as the preferred child */
6.      else prefChild $\leftarrow a$           /* if both are null or both are non-null then choose any */
7.      $t$.BUFlatten($r$, prefChild, $t$)      /* do bottom-up flattening */
8.    if $r \notin t$.children                 /* if more than one hop away from $r$, then...*/
9.      $t$.TDSemiFlatten($t, r, t$.parent)   /*...do top-down semi-flattening */
```

Figure 5.5. Hybrid flattening algorithm.

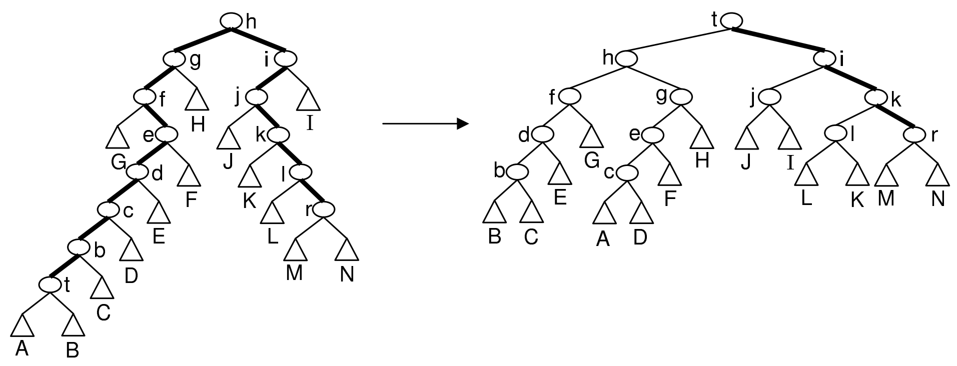


Figure 5.6. Hybrid flattening. Bold lines show the path between t and r. Root of A was the first preferred child.

5.4.4 K-ary trees

Our algorithms as described in the previous sections work only for a binary tree. However, extensions to k-ary trees are straightforward. In both bottom-up flattening and top-down semi-flattening, each step consists of a process replacing one of its children—let us denote this as the *least significant process*—with a process in the path to the requestor—denote it as the *most significant process*. In the first step of Figure 5.1, the root of subtree A is the least significant process and z is the most significant process whereas in Figure 5.3, c is the least significant process and a is the most significant process. In the case of a k-ary tree, the most significant process is still well-defined (the process in the path to the requestor) but the least significant process is not. A simple strategy to define the least significant process could be the following: If a process x in a k-ary tree has $k' < k$ children, then we say it has $k - k'$ *null* children. x prioritizes its children according to some heuristic, e.g., a least recently used (LRU) type algorithm that gives a higher priority to a child that was in the access path of the most recent access through x. The null children always get the lowest priority. Then x may choose the child with the lowest priority as the least significant process when restructuring.

5.4.5 Preserving geographic locality

Quiver arranges the proxies such that geographically nearby proxies are close to each other in the tree. When restructuring the tree however, proxies exchange neighbors and the tree topology may change arbitrarily, depending on the workload, thus destroying any location-aware structure in the tree.

Flattening, however, performs only local restructuring steps, i.e., each flattening step (a preferred rotation or a child swap) involves either only neighbors or at most involves neighbors of neighbors in the tree. Thus processes can set local policies and easily implement these policies during restructuring, so as to preserve the geographic locality in the tree.

In particular, in our implementation proxies implement the following policies to avoid restructuring across regional boundaries: When performing a preferred rotation for bottom-up flattening (Section 5.4.1), t rotates over z (Figure 5.1) only if z is in the same geographic region as t. If z is in a different region than t, then t simply "delegates" further restructuring to z, i.e., z then rotates over x (again only if x is in the same region as z). By the same token, when performing a child swap for top-down semi-flattening (Section 5.4.2), y exchanges its child c with x's child a (Figure 5.3) only if y and x are in the same region. Otherwise, y delegates further top-down semi-flattening to x. This policy ensures that a contiguous part of the tree containing proxies from the same geographic region is not "polluted" by geographically distant proxies, while still bringing the target t close to the requestor r, and balancing each such contiguous component of the tree independently. The performance effects of this policy are evaluated in our experiments (see Section 5.7). More complicated policies can also be implemented using similar mechanisms.

5.5 Restructuring cost analysis

Flattening minimizes the number of messages exchanged during each flattening step, while maximizing the effects of the restructuring in balancing the tree and bringing the requestors and their targets closer to each other. In particular, each preferred rotation (used in bottom-up flattening) requires only 4 messages—two messages for each of the two RPCs in lines 5 and 7 in Figure 5.2—except for the last rotation that requires 6 messages due to line 9 in Figure 5.2. Each child swap (used in top-down semi-flattening) requires 5 messages and moves two steps down the tree—two messages for each of the two RPCs in lines 7 and 8 in Figure 5.4 and an additional message for the RPC in line 10 in Figure 5.4 (this RPC may be non-blocking, and so we do not count its response against the latency of the child swap).

We perform a detailed analysis of the amortized cost of flattening using the potential method (Tarjan [1985]). We assign a real number called *potential* to each possible state of the tree. A *potential function* is a mapping from the tree states to the potential. The *expense* of an operation in the potential method is defined as the sum of the actual work of the operation and the net increase in the potential as a result of this operation. Using this definition, the total actual work of a sequence of m operations can be derived as:

$$\text{total actual work} = \text{total expense} + \text{net decrease in potential} \tag{5.1}$$

Our proof strategy to bound the total actual work of a sequence of operations is to bound the expense of the sequence of operations (Lemma 13 for top-down and Lemma 14 for bottom-up flattening) and the net decrease in potential (Lemma 15) resulting from the sequence of operations. For this proof, we assume that the tree contains a static set of n processes; note that this is only required to quantify the cost of flattening in terms of the size of the tree—our algorithm does not require a static set of processes or an upper bound on the number of processes in this set.

We begin by assigning a positive weight $w(x)$ to each process x that remains fixed throughout the execution. Then define the size $\mathsf{size}(x)$ of a process x to be the sum of weights of all processes in the subtree rooted at x. We define the rank $\mathbf{r}(x)$ of x as $\log(\mathsf{size}(x))$ (binary logarithms are used throughout). The potential function is just the sum of the ranks of all processes in the tree. As a measure of the actual work, we charge one work unit for each child swap and preferred rotation. Since each child swap and preferred rotation is performed with a fixed number of RPCs (described above), our analysis with work unit one suffices to yield an asymptotic bound (the constants can be immediately derived from the discussion above on the number of messages required in each step). We use size and size', $\mathbf{r}$ and $\mathbf{r}'$ to denote the sizes and ranks of processes just before and after a restructuring step, respectively.

Lemma 13. *The expense of top-down semi-flattening from a process t to a process r is at most $2(\mathbf{r}(t) - \mathbf{r}(r))$.*

Proof. Top-down semi-flattening consists of child swaps. The expense of top-down semi-flattening is the sum of the expense of all the child swaps from t to r. We claim that the expense of a single child swap with x being the parent of a and y being the parent of x (see Figure 5.3) is at most $2(\mathbf{r}(y) - \mathbf{r}'(a))$. The sum of these child swap expenses "telescopes" to $2(\mathbf{r}(t) - \mathbf{r}(r))$ if the path length between t and r is even and $2(\mathbf{r}(t) - \mathbf{r}(r'))$ if this length is odd, where r' is the parent of r. The Lemma holds in either case since $\mathbf{r}(r') \geq \mathbf{r}(r)$.

So we only need to prove the claim regarding the expense of each child swap. The child swap is as shown in Figure 5.3. The actual number of work units associated with a child swap is one so the expense is:

$$\begin{array}{lll} = & 1 + \text{net increase in potential} & \\ = & 1 + \mathbf{r}'(x) - \mathbf{r}(x) & \text{[since only } x\text{'s rank changes]} \\ \leq & 1 + \mathbf{r}'(x) - \mathbf{r}(a) & \text{[since } \mathbf{r}(x) \geq \mathbf{r}(a)] \end{array}$$

Now we need to prove that $1 + \mathbf{r}'(x) - \mathbf{r}(a) \leq 2(\mathbf{r}(y) - \mathbf{r}'(a))$, which we rearrange as follows, with each line being equivalent:

$$\begin{array}{rcll} 1 & \leq & 2\mathbf{r}(y) - 2\mathbf{r}'(a) + \mathbf{r}(a) - \mathbf{r}'(x) & \\ 1 & \leq & 2\mathbf{r}(y) - \mathbf{r}'(a) - \mathbf{r}'(x) & \text{[since } \mathbf{r}'(a) = \mathbf{r}(a)] \\ -1 & \geq & \mathbf{r}'(a) - \mathbf{r}(y) + \mathbf{r}'(x) - \mathbf{r}(y) & \\ -1 & \geq & \log(\frac{\mathsf{size}'(a)}{\mathsf{size}(y)}) + \log(\frac{\mathsf{size}'(x)}{\mathsf{size}(y)}) & \end{array}$$

This is true since $\mathsf{size}(y) \geq \mathsf{size}'(a) + \mathsf{size}'(x)$ and $\log b + \log c$ maximizes at -2 if $b + c \leq 1$ (convexity of log). □

Lemma 14. *The expense of bottom-up flattening from a process t to a process h is at most* $2(\mathbf{r}(h) - \mathbf{r}(t)) + 1$.

Proof. Bottom-up flattening consists only of preferred rotations. To see the effects of preferred rotations on the expense of bottom-up flattening, we need to analyze two preferred rotations at a time. Bottom-up flattening consists of these pairs of preferred rotations, possibly followed by a single preferred rotation at the end, in case the path between t and h is of odd length.

Let z be the parent of t and x be the parent of z as shown in Figure 5.1. t is the process that performs the preferred rotations. We claim that the amortized expense of a single preferred rotation is at most $2(\mathbf{r}'(t) - \mathbf{r}(t)) + 1$ and that of a pair of preferred rotations is at most $2(\mathbf{r}'(t) - \mathbf{r}(t))$. The sum of these expenses telescopes and proves the lemma. We now prove our claim.

The actual number of work units of a preferred rotation performed by t over z is one. The expense is:

$$\begin{array}{rcll} 1 + \mathbf{r}'(t) - \mathbf{r}(t) + \mathbf{r}'(z) - \mathbf{r}(z) & \leq & 1 + \mathbf{r}'(t) - \mathbf{r}(t) & [\text{since } \mathbf{r}'(z) \leq \mathbf{r}(z)] \\ & \leq & 1 + 2(\mathbf{r}'(t) - \mathbf{r}(t)) & [\text{since } \mathbf{r}'(t) \geq \mathbf{r}(t)] \end{array}$$

The actual number of work units of a pair of preferred rotations performed by t over z and then over x (see Figure 5.1) is two. The amortized expense is:

$$\begin{array}{rll} = & 2 + \mathbf{r}'(t) - \mathbf{r}(t) + \mathbf{r}'(z) - \mathbf{r}(z) + \mathbf{r}'(x) - \mathbf{r}(x) & \\ = & 2 - \mathbf{r}(t) + \mathbf{r}'(z) - \mathbf{r}(z) + \mathbf{r}'(x) & [\text{since } \mathbf{r}'(t) = \mathbf{r}(x)] \\ \leq & 2 + \mathbf{r}'(z) + \mathbf{r}'(x) - 2\mathbf{r}(t) & [\text{since } \mathbf{r}(t) \leq \mathbf{r}(z)] \end{array}$$

Now we need to prove that $2 + \mathbf{r}'(z) + \mathbf{r}'(x) - 2\mathbf{r}(t) \leq 2(\mathbf{r}'(t) - \mathbf{r}(t))$, which we rearrange as follows with each line being equivalent:

$$\begin{array}{rcl} 2 & \leq & 2\mathbf{r}'(t) - \mathbf{r}'(z) - \mathbf{r}'(x) \\ -2 & \geq & \mathbf{r}'(z) - \mathbf{r}'(t) + \mathbf{r}'(x) - \mathbf{r}'(t) \\ -2 & \geq & \log(\frac{\mathsf{size}'(z)}{\mathsf{size}'(t)}) + \log(\frac{\mathsf{size}'(x)}{\mathsf{size}'(t)}) \end{array}$$

This is true since $\mathsf{size}'(t) \geq \mathsf{size}'(z) + \mathsf{size}'(x)$ and $\log b + \log c$ maximizes at -2 if $b + c \leq 1$ (convexity of log). □

Lemma 15. *The net decrease in potential over any sequence of operations is at most* $\sum_{v \in V} \log(\frac{W}{w(v)})$, *where* $W = \sum_{v \in V} w(v)$.

Proof. The maximum size of a process v, for all $v \in V$, is W when v is the root of the tree and the minimum size is $w(v)$ when v is a leaf. Thus the net decrease in the rank of process v is at most $\log(W) - \log(w(v))$. Summing up over all processes proves the lemma. □

Theorem 8. *The total actual work done by a sequence of* m *top-down flattening operations is at most* $(2m + n) \log n$.

Proof. Assign a weight of $1/n$ to each process, and so $W = 1$. The total expense of the sequence is at most $m(2(\mathbf{r}(t) - \mathbf{r}(r))) \leq 2m \log n$ for any t and r, see Lemma 13. The net decrease in potential is at most $\sum_{v \in V} \log(\frac{W}{w(v)}) = n \log n$. Substituting these values in Equation 5.1 proves the result. □

Theorem 9. *The total actual work done by a sequence of* m *bottom-up flattening operations is at most* $m + (2m + n) \log n$.

Proof. Assign a weight of $1/n$ to each process. The total expense of the sequence is at most $m(1 + 2(\mathbf{r}(h) - \mathbf{r}(t))) \leq m + 2m \log n$ for any h and t, see Lemma 14. The net decrease in potential is at most $\sum_{v \in V} \log(\frac{W}{w(v)}) = n \log n$. Substituting these values in Equation 5.1 proves the result. □

Theorem 10. *The total actual work done by a sequence of* m *hybrid flattening operations is at most* $3m + (2m + n) \log n$.

Proof. Assign a weight of $1/n$ to each process. The total expense of the sequence is at most $m(1 + 2(\mathbf{r}(h) - \mathbf{r}(t)) + 2(\mathbf{r}''(t) - \mathbf{r}(r)))$ for any t, h and r, where $\mathbf{r}''(t)$ is the rank of t after bottom-up flattening, see Lemmas 13 and 14. Note that $r''(t)$ is the same as the rank of h before bottom-up flattening (the subtree contains the same processes), so the total expense of the sequence is at most $m(1+2(\mathbf{r}(h)-\mathbf{r}(t))+2(\mathbf{r}(h)-\mathbf{r}(r))) \leq m+2m \log 2n = 3m+2m \log n$ for any t, h and r. The net decrease in potential is at most $\sum_{v \in V} \log(\frac{W}{w(v)}) = n \log n$. Substituting these values in Equation 5.1 proves the result. □

5.6 Integration with Quiver's consistency protocols

Quiver processes maintain per-object state to route migration (for updates and multi-object operations) and copying (for strictly serializable single-object read operations) request and transfer messages for this object. As flattening restructures the tree, this state must be updated to reflect the new tree topology. Here we discuss mechanisms that allow efficiently updating this state so Quiver processes may continue to perform consistent operations on service objects.

Quiver processes employ a per-object local queue (denoted localQ, see Section 2.3.2) for routing object migration requests and transfers, and for routing strictly serializable single-object read requests. For the routing of strictly serializable single-object read transfers, processes use local state that allows them to copy the object through the tree while going upward, and directly outside the tree when moving downward in the tree, see Section 2.5.2. This latter state is simple to update with restructuring: if a process p is replaced by a process p' in the path of a read transfer, p simply sends its state regarding this read transfer to p'; this state consists of one entry per read request through p of the form "am I the highest process in the transfer path of this read or not". After receiving the corresponding read transfer, p' then decides whether to send the transfer to its parent—transfer goes through the tree when moving up—or to send it directly (outside the tree) to the process that initiated the read—transfers are sent outside the tree when moving downwards—depending on the state sent to p' by p. Processes also update their own state if their position in the read transfer path changes, e.g., if a previously highest process in the path is not the highest anymore, or vice-versa. We invest the rest of this section on the maintenance of the local queues, a more involved mechanism.

In order to update p.localQ for an object as a result of a flattening step, each process p obtains the localQ structures for the same object from each of its neighbors in the tree topology prior to the restructuring. Then using p.localQ, the queues obtained from the neighbors, and with the information about what restructuring step is performed (preferred rotation or child swap), p can update p.localQ to reflect the new topology. The local queues at p and its neighbors represent the outstanding migrate requests that were sent through this neighborhood—p's neighborhood includes p, its immediate neighbors and possibly some neighbors of neighbors. In order to update p.localQ, p first reconstructs the path taken by these migrate requests through p's neighborhood, and then updates p.localQ according to the path the requests would have taken in the topology constructed as a result of this flattening step.

We describe how a process can reconstruct the path taken by migrate requests through its neighborhood, given its own local queue and the local queues of its immediate neighbors, using an example scenario shown in Figure 5.7. Let t be the process trying to reconstruct these paths in the topology before the restructuring step. Then t starts with the process t.localQ.head, i.e., a in this case, and looks at the second element of a.localQ. This element is t itself, which implies that a received a request from t. t then looks at the corresponding element in t.localQ and finds that this request in fact came from z. Finally looking at z.localQ reveals that this request was sent to z by c. Thus, given t.localQ, a.localQ and z.localQ, t can establish that the first of the outstanding requests was sent by c to z; by z to t; and by t to a, thus reconstructing the path of this request in this neighborhood. For the next request, t starts with z.localQ—the local queue of the immediate neighbor where the last "trace" ended—and finds that z received a request from x; since x is not a direct neighbor of t, t ends this trace at z. t can observe that the next request was sent to z by t, and looking at t.localQ, this request was originated by t itself. Finally, for the last request t starts with t.localQ (since the previous trace ended at t) and finds that this last request was sent to t from a, and to a from a' (a child of a). Note that these requests can be traced using a simple iterative algorithm at t that scans the queues of its neighbors, starting the first trace at t.localQ.head, ending each trace when it leaves the neighborhood, and starting each subsequent trace from the process where the last trace ended, to reconstruct the path taken by the outstanding requests through t's neighborhood.

All that remains now is for t to update t.localQ such that the requests

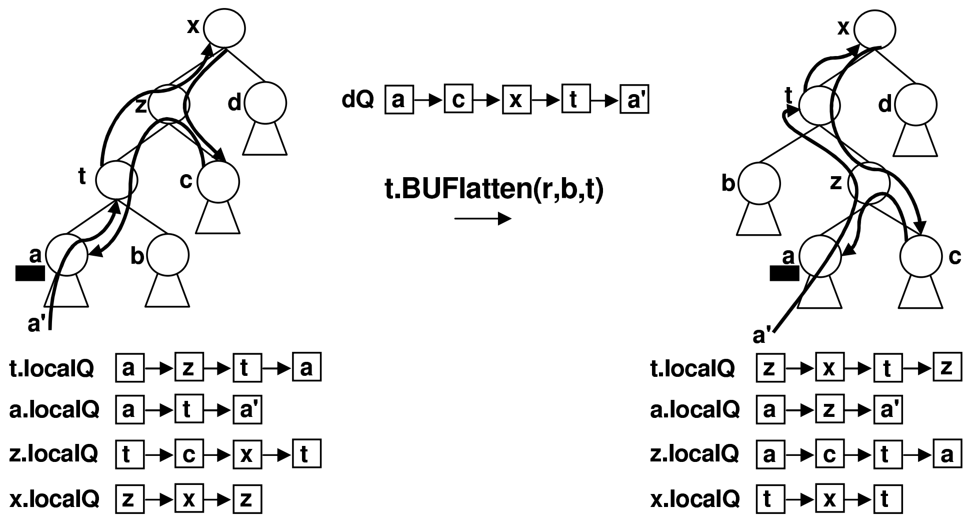


Figure 5.7. localQ maintenance with flattening. a has the object. c requests from a; x requests from c; t requests from x; a' (a's child) requests from t. Processes first trace the paths taken by these requests, and then update their queues according to the paths these requests would have taken in the new topology.

appear to have been routed through the new topology. This requires applying straightforward rules at each process involved in each flattening step. We detail these rules in Tables 5.1–5.8. Rows in these tables are made up of a request path traced as described earlier—the "Before" column—and the path this request would have taken in the new topology—the "After" column. Each path entry is of the form *from* $\rightarrow$ *self* $\rightarrow$ *to*, i.e., the process *self* received this request from the process *from* and sent it to the process *to*. Any entries at the neighbors of *self* required to update the path at *self* are also included in the "Before" column. The tables also show a boolean flag that answers the query amHighProcess for this request. This flag is true at the process *self* if $\mathit{from}, \mathit{to} \in \mathit{self}.\mathsf{children} \cup \mathit{self}$, i.e., *from* and *to* are either children of *self* or *self* itself, and the flag is false otherwise. As an example, consider the first row of Table 5.1: Before performing a bottom-up flattening step, such as the one shown in Figure 5.7, if the request arrived at t from a and was forwarded by t to z (t's "Before" column), and z then forwarded

Before $t.\mathsf{BUFlatten}(r, b, t)$ Figure 5.1:		After:
t	z	t
$a \rightarrow t \rightarrow z$	$t \rightarrow z \rightarrow x$	$z \rightarrow t \rightarrow x$, false
$a \rightarrow t \rightarrow z$	$t \rightarrow z \not\rightarrow x$	$\perp$
$t \rightarrow z$	$t \rightarrow z \rightarrow x$	$t \rightarrow x$, false
$t \rightarrow z$	$t \rightarrow z \not\rightarrow x$	$t \rightarrow z$, true
$b \rightarrow t \rightarrow z$	$t \rightarrow z \rightarrow x$	$b \rightarrow t \rightarrow x$, false
$b \rightarrow t \rightarrow z$	$t \rightarrow z \not\rightarrow x$	$b \rightarrow t \rightarrow z$, true
$t \rightarrow a$		$t \rightarrow z$, true
$b \rightarrow t \rightarrow a$		$b \rightarrow t \rightarrow z$, true
$z \rightarrow t \rightarrow a$	$x \rightarrow z \rightarrow t$	$x \rightarrow t \rightarrow z$, false
$z \rightarrow t \rightarrow a$	$x \not\rightarrow z \rightarrow t$	$\perp$
$a \rightarrow t \rightarrow b$		$z \rightarrow t \rightarrow b$, true
$z \rightarrow t \rightarrow b$	$x \rightarrow z \rightarrow t$	$x \rightarrow t \rightarrow b$, false
$z \rightarrow t \rightarrow b$	$x \not\rightarrow z \rightarrow t$	$z \rightarrow t \rightarrow b$, true
	$x \rightarrow z$	$x \rightarrow t \rightarrow z$, false
	$z \rightarrow x$	$z \rightarrow t \rightarrow x$, false
	$x \rightarrow z \rightarrow c$	$x \rightarrow t \rightarrow z$, false
	$c \rightarrow z \rightarrow x$	$z \rightarrow t \rightarrow x$, false

Table 5.1. Route changes at t during a bottom-up flattening operation in Figure 5.1. flag is omitted from the "Before" entries as it is not required.

it to x (z's "Before" column), then in the new topology, this request would have arrived at t from z and t would have forwarded this to x, and t would not be the highest process in this path. Finally, processes reconstruct their local queues based on these paths that the requests would have followed in the new topology.

5.7 Experiments

We have completed an implementation of the flattening algorithms described in the previous sections. Here we report results from an evaluation of the flattening algorithms on PlanetLab.

Before $t.\mathsf{BUFlatten}(r, b, t)$ Figure 5.1:		After:
z	t	z
$z \to x$		$z \to t$, false
$c \to z \to x$		$c \to z \to t$, false
$t \to z \to x$	$a \to t \to z$	$a \to z \to t$, false
$t \to z \to x$	$a \not\to t \to z$	$\perp$
$z \to t$	$z \to t \to a$	$z \to a$, true
$z \to t$	$z \to t \not\to a$	$z \to t$, false
$x \to z \to t$	$z \to t \to a$	$t \to z \to a$, false
$x \to z \to t$	$z \to t \not\to a$	$\perp$
$c \to z \to t$	$z \to t \to a$	$c \to z \to a$, true
$c \to z \to t$	$z \to t \not\to a$	$c \to z \to t$, false
$x \to z \to c$		$t \to z \to c$, false
$t \to z \to c$	$a \to t \to z$	$a \to z \to c$, true
$t \to z \to c$	$a \not\to t \to z$	$t \to z \to c$, false
	$a \to t$	$a \to z \to t$, false
	$t \to a$	$t \to z \to a$, false
	$a \to t \to b$	$a \to z \to t$, false
	$b \to t \to a$	$t \to z \to a$, false

Table 5.2. Route changes at z during a bottom-up flattening operation in Figure 5.1. flag is omitted from the "Before" entries as it is not required.

Before $t.\mathsf{BUFlatten}(r, b, t)$ Figure 5.1:		After:
x		x
$* \to x \to z$, flag		$* \to x \to t$, flag
$z \to x \to *$, flag		$t \to x \to *$, flag

Table 5.3. Route changes at x during a bottom-up flattening operation in Figure 5.1.

Before $t.\mathsf{BUFlatten}(r, b, t)$ Figure 5.1:		After:
a		a
$* \to a \to t, \mathsf{flag}$		$* \to a \to z, \mathsf{flag}$
$t \to a \to *, \mathsf{flag}$		$z \to a \to *, \mathsf{flag}$

Table 5.4. Route changes at a during a bottom-up flattening operation in Figure 5.1.

Before $y.\mathsf{TDSemiFlatten}(t, r, z)$ Figure 5.3:		After:
y	x	y
$c \to y \to z$		$x \to y \to z, \mathsf{false}$
$x \to y \to z$	$a \to x \to y$	$a \to y \to z, \mathsf{false}$
$y \to x$	$y \to x \to a$	$y \to a, \mathsf{true}$
$z \to y \to x$	$y \to x \to a$	$z \to y \to a, \mathsf{false}$
$c \to y \to x$	$y \to x \to a$	$x \to y \to a, \mathsf{true}$
$c \to y \to x$	$y \to x \not\to a$	$\perp$
$y \to c$		$y \to x, \mathsf{true}$
$z \to y \to c$		$z \to y \to x, \mathsf{false}$
$x \to y \to c$	$a \to x \to y$	$a \to y \to x, \mathsf{true}$
$x \to y \to c$	$a \not\to x \to y$	$\perp$
	$a \to x$	$a \to y \to x, \mathsf{true}$
	$x \to a$	$x \to y \to a, \mathsf{true}$
	$a \to x \to b$	$a \to y \to x, \mathsf{true}$
	$b \to x \to a$	$x \to y \to a, \mathsf{true}$

Table 5.5. Route changes at y during a top-down semi-flattening operation in Figure 5.3. flag is omitted from the “Before” entries as it is not required.

Before $y.\mathsf{TDSemiFlatten}(t, r, z)$ Figure 5.3:		After:
x	y	x
$x \to y$	$x \to y \to c$	$x \to c, \mathsf{true}$
$b \to x \to y$	$x \to y \to c$	$b \to x \to c, \mathsf{true}$
$a \to x \to y$	$x \to y \to c$	$y \to x \to c, \mathsf{false}$
$a \to x \to y$	$x \to y \not\to c$	$\perp$
$x \to a$		$x \to y, \mathsf{false}$
$b \to x \to a$		$b \to x \to y, \mathsf{false}$
$y \to x \to a$	$c \to y \to x$	$c \to x \to y, \mathsf{false}$
$y \to x \to a$	$c \not\to y \to x$	$\perp$
$a \to x \to b$		$y \to x \to b, \mathsf{false}$
$y \to x \to b$	$c \to y \to x$	$c \to x \to b, \mathsf{true}$
	$c \to y \not\to x$	$c \to x \to y, \mathsf{false}$
	$x \not\to y \to c$	$y \to x \to c, \mathsf{false}$

Table 5.6. Route changes at x during a top-down semi-flattening operation in Figure 5.3. flag is omitted from the "Before" entries if not required.

Before $y.\mathsf{TDSemiFlatten}(t, r, z)$ Figure 5.3:		After:
c		c
$* \to c \to y, \mathsf{flag}$		$* \to c \to x, \mathsf{flag}$
$y \to c \to *, \mathsf{flag}$		$x \to c \to *, \mathsf{flag}$

Table 5.7. Route changes at c during a top-down semi-flattening operation in Figure 5.3.

Before $y.\mathsf{TDSemiFlatten}(t, r, z)$ Figure 5.3:		After:
a		a
$* \to a \to x, \mathsf{flag}$		$* \to a \to y, \mathsf{flag}$
$x \to a \to *, \mathsf{flag}$		$y \to a \to *, \mathsf{flag}$

Table 5.8. Route changes at a during a top-down semi-flattening operation in Figure 5.3.

5.7.1 Experimental setup

Our experiments were run on PlanetLab Chun et al. [2003] using 40 nodes spread across North America. For each experiment, these 40 nodes were initially arranged in a binary tree; we chose a binary tree so as to maximize the tree diameters experienced with only 40 nodes. After initializing each node with information about its parent and children, nodes initiated an application workload. In order to emulate an application that uses the tree structure for communication between nodes, each node performed flood-based searches for other nodes in the tree. In each of these searches, the requestor node broadcast a packet to each of its neighbors with the identity of the target node. Each node (except the target) that received such a packet, forwarded it to each of its neighbors, except the neighbor where the packet came from. When the target node itself received this packet, it sent an acknowledgement directly (outside the tree) to the requestor. After receiving the acknowledgement from the target, the requestor measured the latency of the access and then retrieved the token and notified the target, which then initiated the flattening algorithm along the path to the requestor.

In order to control the sequence of requests (so we could construct worst cases and other distributions), we used one node external to the tree as a "monitor". The monitor exchanged control messages with all nodes, e.g., to have nodes initiate an access request or to pull information about how long an access operation took.

We performed three sets of experiments: the first employed a randomly constructed tree and a random workload, i.e., each node chose its target uniformly at random among all nodes. The second set again employed a randomly constructed tree, but used a workload where nodes were divided in groups such that nodes in a particular group accessed each other more frequently. The final set utilized a location-aware tree, i.e., geographically nearby nodes were placed close to each other in the tree, and a random workload. Each data point in our graphs is an average value over four runs.

5.7.2 Random tree, random workload

For these tests we constructed a random binary tree among 40 PlanetLab nodes, and arranged the nodes according to this random tree structure. In order to perform an access operation, a node chose another node uniformly at random from the set of all 39 other nodes in the tree, and initiated a flood-based access request for the chosen node. The top of Figure 5.8 presents the amortized latencies of the access operations with and without using flattening. Flattening significantly improved the performance of access operations, and it did so by reducing the diameter of the tree; see the bottom of Figure 5.8. This is further confirmed by comparing the tree topologies at the beginning and end of each experiment. An example is shown in Figure 5.9. Note that since nodes chose their targets at random, the set of recent requestors and targets span most of the tree, and so flattening has the effect of balancing the whole tree.

5.7.3 Random tree, group workload

In a second set of tests, we again arranged the 40 PlanetLab nodes in a random binary tree. In these tests, though, we partitioned the nodes into 10 non-intersecting groups of four nodes each. When selecting a node to access via the flooding algorithm, each node selected from within its group with probability 0.8, and selected from outside its group with probability 0.2. We hypothesized that with such a workload, our approach would tend to bring the members of each group closer to one another, thereby improving the latency of intra-group accesses.

The results from these tests are shown in Figure 5.10. This figure shows the average access latencies that resulted when flattening was or was not used. As the top portion shows, the access latency was dramatically improved through the use of flattening. The reason for this improvement is demonstrated in the bottom portion of the same figure, which shows the average tree diameter and the average group diameter, i.e., the hops between the furthest members in each group, averaged over all groups. The error bars in this bottom graph show the standard deviation of the group

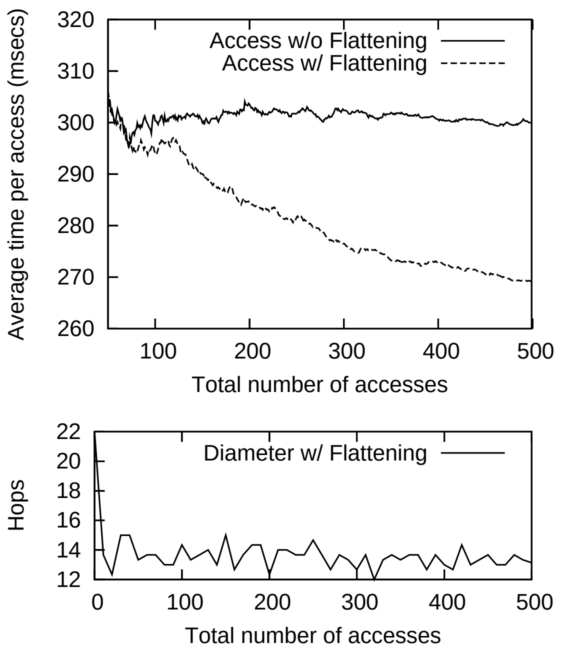


Figure 5.8. Amortized access latencies (top) for a random workload on a randomly constructed tree (Section 5.7.2), with and without flattening. Flattening improves the access costs by reducing the diameter of the tree (bottom).

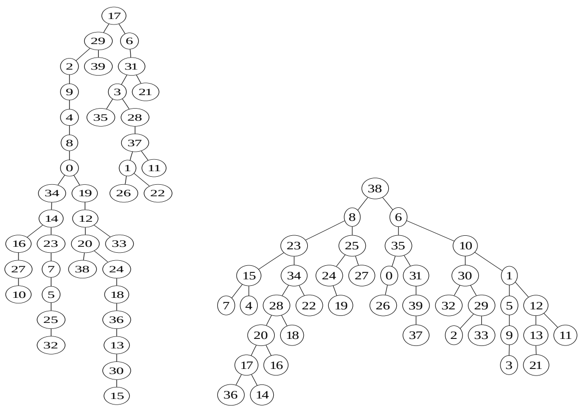


Figure 5.9. An example start (left) and end (right) topology from an experiment beginning with a random tree and running a random workload (Section 5.7.2).

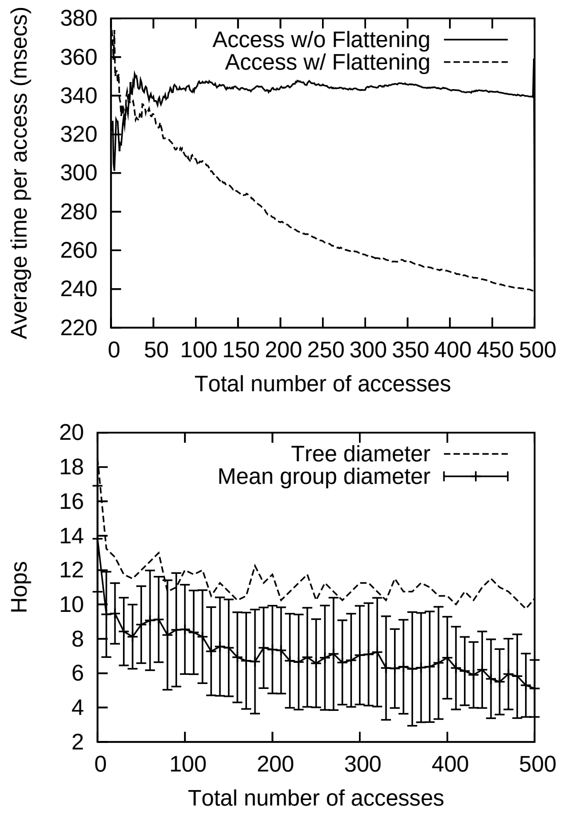


Figure 5.10. Latencies (top) and tree and group diameters (bottom) in group-biased workload (Section 5.7.3).

diameters. As this graph shows, the tree and group diameters drop rapidly at first, and then continue a slight downward trend for the duration of the workload. This, in turn, translates to significant latency savings for access (see top of Figure 5.7.3).

This conclusion is also supported by examining the topologies that resulted from our experiments. For example, Figure 5.11 shows an initial and an ending topology in one of our experiments. The label on each node indicates the group of which that node is a member. The right tree also shows the groups that end with the largest group diameter (in gray) and that end with the smallest group diameter (in black); these two groups are colored

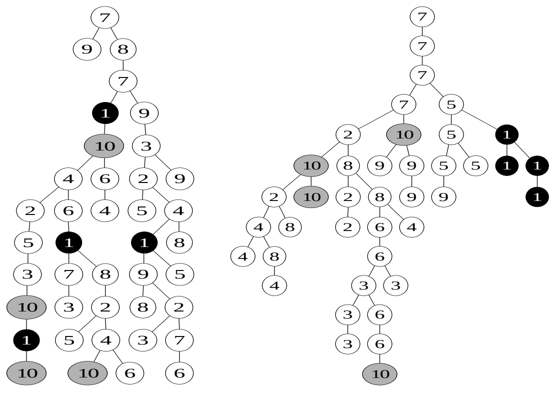


Figure 5.11. Example PlanetLab node topologies at the start (left) and end (right) of the group-bias experiment (Section 5.7.3). Circles with the same numbers represent nodes in the same group. The gray group (10) ends with the worst diameter, and the black group (1) ends with the best. Note that flattening reduces group diameters overall.

similarly in the left tree to show where these groups began in the initial topology. A careful examination of the various groups shows that the group diameters became smaller during the workload due to flattening.

5.7.4 Geographic tree, random workload

Our last experiments organized the 40 nodes geographically, by partitioning the nodes into "west coast", "east coast" and "central" nodes; each such group occupied a contiguous portion of the tree initially. We then performed random workloads in which each node, to perform an access, selected a node to access uniformly at random from among the 39 other nodes. In this context we explored three different restructuring regimes: no restructuring; universal flattening without attention to the geographic partitioning; and geographic flattening, i.e., flattening within each geographic region only. That is, a path between a requestor and a target that traversed multiple regions was restructured only on its contiguous subpaths within each region; connections between regions and the nodes they connected were left alone.

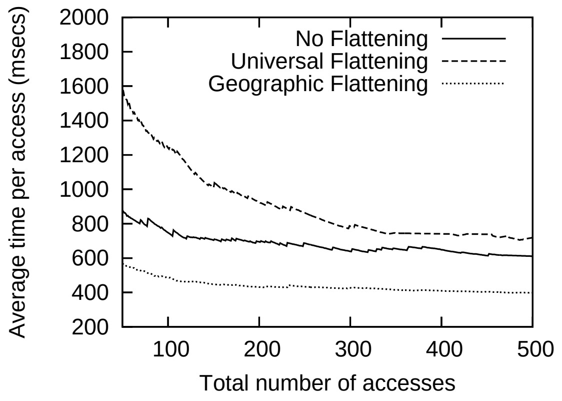


Figure 5.12. Latencies in geographic workload (Section 5.7.4).

We explored this workload to highlight a feature of flattening, namely that each restructuring step, being localized to the vicinity of the node executing it, can be applied in subtrees effectively and enables the node to apply localized policies that limit restructuring. The benefit that this offers is convincingly demonstrated in Figure 5.12. As shown, the restructuring that respected local geographic policies outperformed both no restructuring and universal restructuring, and in fact the universal restructuring performed no better than no restructuring due to its failure to account for geographic realities. An example topology produced by this geographic restructuring is shown in Figure 5.13.

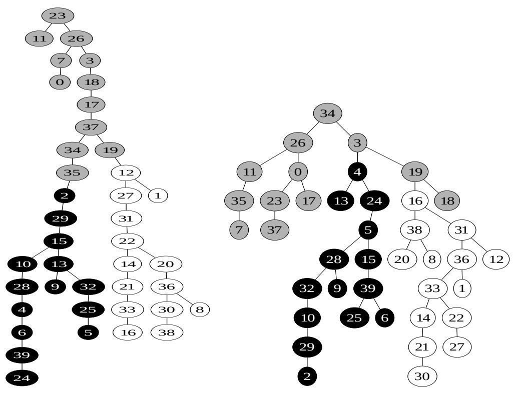


Figure 5.13. An example start (left) and end (right) topology from an experiment beginning with a geographically placed tree and running a random workload with geographic flattening (Section 5.7.4). Nodes are colored according to their regions.

6 Conclusions

Edge service infrastructures are playing a vital role in scaling services beyond what is achievable with centralized or cluster based architectures. However, in their current state, these infrastructures can either only support applications with static content, or support dynamic applications but with assumptions about the frequency of updates and application's consistency requirements.

This dissertation presented Quiver, an architecture that allows edge service infrastructures to support dynamic applications with strong consistency guarantees, while preserving the load balancing, and latency reducing effects of employing proxies at the edge. Edge service infrastructures reduce the client-perceived latency through serving client operations at edge proxies located geographically close to the client. Quiver preserves this property through locality-aware object migrations that allow proxies to perform consistent operations on service objects, while exploiting locality in the application workload, and minimizing wide-area communication in the critical path of the client's operation. Empirical evaluation through experiments on PlanetLab showed that migrating objects to proxies for operation execution is a viable strategy for global scale applications.

We have also presented extensions to Quiver that allow the detection of malicious or misconfigured edge proxies that violate Quiver's consistency protocols; tolerate failures in Quiver's communication infrastructure; and optimize this communication infrastructure dynamically to best suit the current application workload. Each of these extensions may also be of interest independently—we have discussed some applications of these extensions in other domains as well.

This thesis builds the foundations for a consistent and scalable edge service infrastructure, and demonstrates the feasibility of these mechanisms via PlanetLab experiments. However, there are several open problem areas.

First, Quiver does not attempt to preserve the durability of all operations performed by all proxies. In particular, operations performed by proxies that disconnect may not be durable. Quiver trades-off this durability with the overall load on the system; making an operation durable requires copying all the instances it produced along the path from this proxy to the root. An interesting point in this trade-off could be achieved using gossip-based protocols for propagating the new instances to other proxies. Such a gossip protocol could also employ the expander graph for fast convergence, and would give probabilistic guarantees for durability and consistency of operations. We have not explored these alternatives here.

Second, there is a trade-off between the cost of migrating service objects to a proxy, and the benefit of this migration, e.g., due to the locality of reference in the workload that allows the proxy to perform future operations without involving wide-area communication. This trade-off is more profound if the objects to be migrated are large in size: Quiver assumes small objects and locality in the workload, and so always migrates the objects to the requesting proxy. In practice, however, a decision on whether to migrate the object or not should depend on multiple factors, for example the size of the object; whether the operation history of this object has shown locality of reference or not; how many concurrent requests for this object exist in the system; and how far the object needs to be migrated, etc. Note that the expander neighbors of a proxy may act as "sensors" to monitor some of these factors. We hope to explore some of these optimizations in the future.

Finally, we have presented Quiver as a generic infrastructure for edge services, not tied to any particular application. We intend to explore applications of Quiver in different domains in the future.

Bibliography

Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. 2005. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. 59–74. 4, 71

Adelson-Velskii, G. M. and Landis, E. M. 1962. An algorithm for the organization of information. *Soviet Math. Dokl. 3*, 1259–1263. 123, 124

Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of Operating Systems Design and Implementation*. 72

Ajtai, M., Koml'os, J., and Szemer'edi, E. 1983. An $O(n \log n)$ sorting network. *Combinatorica 3,* 1, 1–19. 7

Albers, S. and Karpinski, M. 2002. Randomized splay trees: theoretical and experimental results. *Information Processing Letters 81,* 4, 213–221. 124

Alon, N. 1986. Eigenvalues and expanders. *Combinatorica 6,* 2, 83–96. 117

Alvisi, L., Malkhi, D., Pierce, E., and Reiter, M. K. 2001. Fault detection for Byzantine quorum systems. *IEEE Transactions on Parallel Distributed Systems 12,* 9 (Sept.). 4, 71

AMIR, Y., DANILOV, C., MISKIN-AMIR, M., STANTON, J., AND TUTU, C. 2002. Practical wide-area database replication. Tech. Rep. CNDS-2002-1, Johns Hopkins University. 13

AMIRI, K., PARK, S., TEWARI, R., AND PADMANABHAN, S. 2003. DBProxy: A dynamic data cache for Web applications. In *Proceedings of International Conference on Data Engineering.* 13

AWAN, A., FERREIRA, R. A., JAGANNATHAN, S., AND GRAMA, A. 2004. Distributed uniform sampling in real-world networks. In *Purdue University, CSD Technical Report (CSD-TR-04-029).* 101

BAGCHI, A., BHARGAVA, A., CHAUDHARY, A., EPPSTEIN, D., AND SCHEIDELER, C. 2004. The effects of faults on network expansion. In *Proceedings of 16th ACM Symposium on Parallel Algorithms and Architectures.* 7, 102

BAILEY, M., COOKE, E., JAHANIAN, F., PROVOS, N., ROSAEN, K., AND WATSON, D. 2005. Data reduction for the scalable automated analysis of distributed darknet traffic. In *Proceedings of the Internet Measurement Conference.* 62

BAYER, R. 1972. Symmetric binary B-Trees: data structure and maintenance algorithms. *Acta Informatica 1,* 290–306. 124

BERNSTEIN, A. AND GOODMAN, N. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR) 13,* 2, 185–221. 14

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. 1, 3, 15, 22, 23, 31, 32, 35

BLUM, M., KARP, R. M., VORNBERGER, O., PAPADIMITRIOU, C. H., AND YANNAKAKIS, M. 1981. The complexity of testing whether a graph is a superconcentrator. *Information Processing Letters 13,* 4/5, 164–167. 117

BOLLOBÁS, B. 1980. A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *European Journal of Combinatorics 1,* 4, 311–316. 99

BOYD, S., DIACONIS, P., AND XIAO, L. 2004. Fastest mixing markov chain on a graph. *SIAM Review 46,* 4, 667–689. 101, 109

BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F. B., AND TOUEG, S. 1993. The primary–backup approach. In *Distributed Systems*, second ed., S. Mullender, Ed. Addison-Wesley, Chapter 8, 199–216. 3

BUSKENS, R. W. AND R. P. BIANCHINI, J. 1993. Distributed on-line diagnosis in the presence of arbitrary faults. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing.* 470–479. 4, 71

CACHIN, C. AND PORITZ, J. A. 2002. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks.* 4, 71

CASTRO, M. AND LISKOV, B. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20,* 4 (Nov.). 4, 71, 72

CAUWENBERGHS, G. AND POGGIO, T. 2001. Incremental and decremental support vector machine learning. *Advances in Neural Information Processing Systems 13.* 66

CHEN, B. 2004. A serverless, wide-area, version control system. Ph.D. thesis, Massachusetts Institute of Technology. 74

CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. 2003. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review 33,* 3, 3–12. 3, 48, 147

CORTES, C. AND VAPNIK, V. 1995. Support vector networks. *Machine Learning 20*, 273–297. 66

DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2001. Wide-area cooperative storage with CFS. In *Proceedings of 18th ACM Symposium on Operating Systems Principles.* 72

DAVIS, A., PARIKH, J., AND WEIHL, W. E. 2004. Edgecomputing: Extending enterprise applications to the edge of the internet. In *Proceedings of International World Wide Web Conference.* 1

DEMMER, M. J. AND HERLIHY, M. P. 1998. The Arrow distributed directory protocol. In *Proceedings of 12th International Symposium of Distributed Computing.* 119–133. 6, 14, 30

DESPAIN, A. M. AND PATTERSON, A. D. 1978. X-tree: A tree structured multi-processor computer architecture. In *Proceedings of 5th Annual Symposium on Computer Architecture.* 98

ELLARD, D. AND SELTZER, M. 2003. New NFS tracing tools and techniques for system analysis. In *Large Installation Systems Administration Conference.* 91

ESKIN, E., ARNOLD, A., PRERAUA, M., PORTNOY, L., AND STOLFO, S. J. 2002. A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. *Applications of Data Mining in Computer Security*. 66

FRIEDMAN, J. 1991. On the second eigenvalue and random walks in random d-regular graphs. *Combinatorica 11,* 4, 331–362. 7, 99

FU, K. 1999. Group sharing and random access in cryptographic storage file systems. M.S. thesis, Massachusetts Institute of Technology. 72

FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. 2002. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems 20,* 1 (February), 1–24. 72

FUNG, G. AND MANGASARIAN, O. L. 2002. Incremental support vector machine classification. In *Proceedings of 2nd SIAM International Conference on Data Mining.* 247–260. 66

FURER, M. 1999. Randomized splay trees. In *Proceedings of of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms.* 124

GABBER, O. AND GALIL, Z. 1981. Explicit construction of linear-sized superconcentrators. *Journal of Computer and System Sciences 22,* 3, 407–420. 98

GAO, L., DAHLIN, M., ZHENG, J., ALVISI, L., AND IYENGAR, A. 2005. Dual-Quorum replication for edge services. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware).* 13

GILON, K. AND PELEG, D. 1991. Compact deterministic distributed dictionaries. In *Proceedings of of the Annual ACM Symposium on Principles of Distributed Computing.* 125

GKANTSIDIS, C., MIHAIL, M., AND SABERI, A. 2003. Conductance and congestion in power law graphs. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS).* 148–159. 113

GKANTSIDIS, C., MIHAIL, M., AND SABERI, A. 2004. Random walks in peer-to-peer networks. In *Proceedings of IEEE Conference on Computer Communications.* 7, 99

GOERDT, A. 1998. Random regular graphs with edge faults: Expansion through cores. In *Proceedings of 9th International Symposium on Algorithms and Computation.* 7, 102

GOH, E., SHACHAM, H., MODADUGU, N., AND BONEH, D. 2003. SiRiUS: Securing remote untrusted storage. In *Proceedings of 10th Network and Distributed Systems Security Symposium.* 73

GOODMAN, J. R. AND CARLO, H. S. 1981. Hypertree: A multiprocessor interconnection topology. *IEEE Transactions on Computers C-30,* 12, 923–933. 98

Guibas, L. J. and Sedgewick, R. 1978. A dichromatic framework for balanced trees. In *Proceedings of 19th IEEE Symposium on Foundations of Computer Science.* 124

Helary, J. M., Mostefaoui, A., and Raynal, M. 1994. A general scheme for token- and tree-based distributed mutual exclusion algorithms. *IEEE Transactions on Parallel and Distributed Systems 5,* 11, 1185–1196. 6

Herlihy, M., Tirthapura, S., and Wattenhofer, R. 2001. Competitive concurrent distributed queuing. In *Proceedings of ACM Symposium on Principles of Distributed Computing.* 127–133. 30

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

Holliday, J., Steinke, R., Agrawal, D., and El-Abbadi, A. 2003. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering 15,* 3. 13

Honig, A., Howard, A., Eskin, E., and Stolfo, S. 2002. Adaptive model generation: An architecture for the deployment of data mining-based intrusion detection systems. *Applications of Data Mining in Computer Security.* 66

Horowitz, K. and Malkhi, D. 2003. Estimating network size from local information. *Information Processing Letters 88,* 5, 237–243. 109

Jagadish, H. V., Ooi, B. C., and Vu, Q. H. 2005. BATON: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st VLDB Conference.* 125

Jiang, X. and Xu, D. 2004. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium.* 62

Johnson, T. and Colbrook, A. 1992. A distributed data-balanced dictionary based on the B-link tree. Tech. Rep. MIT/LCS/TR-530, Massachusetts Institute of Technology. 125

Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., and Fu, K. 2003. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of 2nd Conference on File and Storage Technologies.* 72

Karagiannis, T., Papagiannaki, K., and Faloutsos, M. 2005. BLINC: Multilevel traffic classification in the dark. *ACM SIGCOMM Computer Communication Review 35,* 4, 229–240. 62

Kim, J. H. and Vu, V. H. 2001. Generating random regular graphs. In *Proceedings of 35th ACM Symposium on Theory of Computing.* 98, 100

Kleitman, D. 1969. Methods for investigating connectivity of large graphs. *IEEE Transactions on Circuits and Systems 16,* 2, 232–233. 117

Kuhn, F. and Wattenhofer, R. 2004. Dynamic analysis of the arrow distributed protocol. In *Proceedings of 16th ACM Symposium on Parallelism in Algorithms and Architectures.* 294–301. 30

Lamport, L. 1978. The implementation of reliable distributed multiprocess systems. *Computer Networks 2*, 95–114. 4, 71

Law, C. and Siu, K.-Y. 2003. Distributed construction of random expander networks. In *Proceedings of IEEE Conference on Computer Communications.* 7, 98, 99

Lee, W., Stolfo, S. J., and Mok, K. W. 1999. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy.* 120–132. 62

Lehman, P. L. and Yao, S. B. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6,* 4, 650–670. 124

Li, J., Krohn, M. N., Mazières, D., and Shasha, D. 2004. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation.* 91–106. 4, 5, 72, 73, 77, 78, 90, 94

Li, Q. and Dong, G. Z. 1994. A framework for object migration in object-oriented databases. *Data and Knowledge Engineering 13*, 221–242. 13

Li, W., Po, O., Hsiung, W., Candan, K. S., Agrawal, D., Akca, Y., and Taniguchi, K. 2003. CachePortal II: Acceleration of very large scale data center-hosted database-driven web applications. In *Proceedings of International Conference on Very Large Data Bases.* 13

Loguinov, D., Kumar, A., Rai, V., and Ganesh, S. 2003. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of ACM Annual Conference of the Special Interest Group on Data Communication (SIGCOMM).* 99

Lovasz, L. 1993. Random walks on graphs: A survey. *Combinatorics, Paul Erdos is Eighty 2*, 1–46. 100

Luo, Q., Drishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B. G., and Naughton, J. F. 2002. Middle-tier database caching for e-business. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 13

Margulis, G. A. 1973. Explicit constructions of concentrators. *Problemy Peredachi Informatsii*, 71–80. 98

Mazières, D. and Shasha, D. 2002. Building secure file systems out of Byzantine storage. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing.* 108–117. 4, 5, 72, 73, 76, 77, 90

Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. 1996. *Handbook for Applied Cryptography.* CRC Press. 82

Milojičić, D. S., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. 2000. Process migration. *ACM Computing Surveys 32,* 3 (Sept.), 241–299. 14

Molloy, M. and Reed, B. 1999. Critical subgraphs of a random graph. *The Electronic Journal of Combinatorics 6.* 113

Moore, A. and Zuev, D. 2005. Internet traffic classification using Bayesian analysis techniques. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS).* 62

Mukkamala, S. and Sung, A. H. 2003. Identifying significant features for network forensic analysis using artificial intelligent techniques. *International Journal of Digital Evidence 1,* 4, 1–17. 66

Muthitacharoen, A., Morris, R., Gil, T., , and Chen, B. 2002. Ivy: A read/write peer-to-peer file system. In *Proceedings of Operating Systems Design and Implementation.* 72

Naimi, M., Trehel, M., and Arnold, A. 1996. A log(N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing 34,* 1, 1–13. 6, 14

Nuttall, M. 1994. A brief survey of systems providing process or object migration facilities. *ACM Operating Systems Review 28,* 4 (Oct.), 64–80. 14

Olston, C., Manjhi, A., Garrod, C., Ailamaki, A., Maggs, B. M., and Mowry, T. C. 2005. A scalability service for dynamic web applications. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 13

Oprea, A. and Reiter, M. K. 2006. On consistency of encrypted files. Tech. Rep. CMU-CS-06-113, Computer Science Department, Carnegie Mellon University. 77

Özsu, M. T. and Valduriez, P. 1996. Distributed and parallel database systems. *ACM Computing Surveys (CSUR) 28,* 1, 125–128. 14

Pandurangan, G., Raghavan, P., and Upfal, E. 2003. Building low-diameter p2p networks. In *Proceedings of 42nd IEEE Symposium on Foundations of Computer Science.* 7, 99

Papadimitriou, C. 1979. The serializability of concurrent database updates. *Journal of the ACM 26,* 4 (Oct.), 631–653. 1, 3, 15

Peleg, D. 1990. Distributed data structures: a complexity oriented view. In *Proceedings of 4th International Workshop on Distributed Algorithms.* 71–89. 125

Pippenger, N. and Lin, G. 1992. Fault-tolerant circuit-switching networks. In *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures.* 7

Plattner, C. and Alonso, G. 2004. Ganymed: Scalable replication for transactional web applications. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware).* 13

Rabinovich, M., Xiao, Z., and Aggarwal, A. 2003. Computing on the edge: A platform for replicating internet applications. In *Proceedings of of the 8th International Workshop on Web Content Caching and Distribution.* 13

Ralaivola, L. and d'Alché-Buc, F. 2001. Incremental support vector machine learning: A local approach. *Lecture Notes in Computer Science 2130*, 322–330. 66

Raymond, K. 1989. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems 7,* 1 (Feb.), 61–77. 6, 14

Reiter, M. K. and Birman, K. P. 1994. How to securely replicate services. *ACM Trans. Program. Lang. Syst. 16,* 3 (May), 986–1009. 4, 71

RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. 2003. Pond: the Oceanstore prototype. In *Proceedings of 2nd USENIX Conference on File and Storage Technologies.* 72

SCHLICHTING, R. D. AND SCHNEIDER, F. B. 1983. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems 1,* 3, 222–238. 102

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22,* 4 (Dec.), 299–319. 4, 71

SHIN, K. AND RAMANATHAN, P. 1987. Diagnosis of processors with Byzantine faults in a distributed computing system. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing.* 55–60. 4, 71

SIPSER, M. AND SPIELMAN, D. A. 1996. Expander codes. *IEEE Transactions on Information Theory 42,* 6, 1710–1722. 7

SIVASUBRAMANIAN, S., ALONSO, G., PIERRE, G., AND VAN STEEN, M. 2005. GlobeDB: autonomic data replication for web applications. In *Proceedings of International World Wide Web Conference.* 14

SLEATOR, D. D. AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *Journal of the ACM (JACM) 32,* 3, 652–686. 123, 124

STEGER, A. AND WORMALD, N. 1999. Generating random regular graphs quickly. *Combinatorics, Probability and Computing 8,* 4, 377–396. 98, 100, 103

TARJAN, R. E. 1985. Amortized computational complexity. *SIAM J. Appl. Discrete Math 6*, 306–318. 136

TATEMURA, J., HSIUNG, W., AND LI, W. 2003. Acceleration of web service workflow execution through edge computing. In *Proceedings of International World Wide Web Conference.* 1

TPC. 2002. TPC Benchmark W V1.8. `http://www.tpc.org/`. 43

Williams, H. E., Zobel, J., and Heinz, S. 2001. Self-adjusting trees in practice for large text collections. *Software, Practice and Experience 31,* 10, 925–939. 124

Wormald, N. C. 1999. Models of random regular graphs. *Surveys in Combinatorics 276*, 239–298. 112

Yegneswaran, V., Barford, P., and Jha, S. 2004. Global intrusion detection in the DOMINO overlay system. In *Proceedings of the Network and Distributed System Security Symposium.* 62

Yin, J., Martin, J., Venkataramani, A., Alvisi, L., and Dahlin, M. 2003. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of 19th ACM Symposium on Operating Systems Principles.* 4, 71

Zadok, E., Badulescu, I., and Shender, A. 1998. Cryptfs: A stackable vnode level encryption file system. Tech. Rep. CUCS-021-98, Computer Science Department, Columbia University. 72

Zanero, S. and Savaresi, S. 2004. Unsupervised learning techniques for an intrusion detection system. In *Proceedings of ACM Symposium on Applied Computing.* 62