NEW SYSTEMS AND ALGORITHMS FOR SCALABLE FAULT TOLERANCE

SIDDHARTHA SEN

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE
ADVISERS: PROFESSOR MICHAEL J. FREEDMAN
PROFESSOR ROBERT E. TARJAN

JUNE 2013

Abstract

Users of Internet services are increasingly intolerant of delays and outages, while demanding a consistent online experience. A website that is down or misbehaving is reported within seconds, often with an embarrassing screenshot that spreads through the news like wildfire. Among these failures, the most notorious are the ones that manifest arbitrary behavior, such as returning the wrong content to users or accidentally deleting their data. Unfortunately, protecting against such failures—whether due to misconfigurations, bugs, or even malice—is prohibitively expensive, because most existing solutions do not scale beyond a single server's performance. As a result, these solutions are not used for customer-facing services, where scalability is required to cope with large user populations.

This thesis describes new systems and algorithms for tolerating arbitrary failures in Internet services, inspired by real-world debacles. Unlike prior work, our solutions are highly scalable. Our approach integrates theoretical innovations into the later stages of system design, giving robust guarantees that are also practical. We begin with a real failure that occurred in the indexing technique used by a certain database provider, and explain theoretically why the technique failed. We remedy the technique by introducing a new class of tree data structures, called *relaxed trees*, with provably good properties. Our analysis of relaxed trees makes use of exponential potential functions.

Then, we describe a general system for tolerating arbitrary failures, called Prophecy, that delivers scalable performance on read-mostly workloads. With a modest trust assumption, Prophecy is practical for modern Internet services, as our evaluation confirms. Finally, we devise two techniques to scale this fault tolerance to very large-scale systems and general workloads. The first is an algorithm for securely composing many small replica groups, subject to an adversary that can coordinate faulty nodes across the groups dynamically. The second is a technique for improving the fault tolerance within each replica group, by adding small, trusted broadcast channels that mitigate the impact of faulty nodes.

Acknowledgements

In writing this thesis, I have come to realize just how much my taste in research reflects my personality. I have always played the role of the mediator, compromising between arguing viewpoints and trying to find the merit in each. In a way, I do the same thing with systems and theory. I thank my advisors, Mike Freedman and Bob Tarjan, for supporting an overly-optimistic, fresh-out-of-industry student's proposal of doing both during his PhD.

Bob was the reason I came to Princeton. His ideas and generosity have shaped the theoretician in me today. Bob has shared insights with me that have been stewing in his head for years, even decades. Despite our differences, I never once felt like I had asked him a dumb question (even though I had, repeatedly). Bob's way of mentoring is like an ideal parent: guiding, never pushing too hard, and taking a long view on life. It has changed the way I mentor all manner of students, from undergraduates to my own son.

Mike started his career the same year I started my PhD. Watching him over the years has been a thing of beauty, because he just keeps getting better at everything. Mike's emphasis on deep insights has permeated all of my systems work; his strong handle on practicality has made my work more relevant. Mike has shown an extraordinary level of care for my success along every step of the nonstandard path I took. I would not have the career options I have today if it weren't for him.

Many professors and mentors have lent me their wisdom and support over the years. Jen Rexford, Vivek Pai, and Bob Sedgewick served on my thesis committee at Princeton. Jen is the most constructive critiquer I know. My mentors at MSR—Jay Lorch, Jitu Padhye, Thomas Moscibroda, and Renato Werneck—have been great collaborators and fierce advocates of mine. Joseph Joy was my first boss at Microsoft (in Windows Server), and has followed and supported my research endeavors ever since. Charles Leiserson gave excellent advice on both research and life while I was at MIT. Frank Dabek has been a great source of reality checks as my Google fellowship mentor. Finally, Jinyang Li has been very generous with her advice and support during my final year jointly with NYU.

I would like to thank the many students and postdocs who have made this experience fun and educational. Each of you has contributed to this thesis in an important way. I especially thank Aravindan Vijayaraghavan and Aditya Bhaskara for being my theory gurus, and good friends, from the very beginning. Wyatt Lloyd and Jeff Terrace were my comrades in Mike's first batch of students. Dave Shue and his family have been fun collaborators both in and outside of work. Bernhard Haeupler holds a special place as my first academic collaborator and friend.

I owe a tremendous amount to my family, who have supported every endeavor of mine, whether it be running a hip-hop group out of our living room or pursuing a doctoral degree. My parents, Mala and Tilak, have been steadfast in their love and support, and have cultivated creativity, ambition, and humor in me. My sisters, Urvashi and Riya, have added laughter and variety to my life. They are both such unique and remarkable individuals.

My grandfather infected me with the academic bug with stories of his research and great mentors like S. N. Bose. He was a remarkable man, and I miss him dearly.

Finally, I want to thank Faye and Sammy for being my partners in life. Faye is the limit I would tend to if I continually became a better person. She enriches my life on so many different levels, and I am so lucky that Sammy has her as his mother. She is also a great comedian, second only to Sammy, who is hilarious and unstoppable. Sammy's laughter and uninhibited expression have brought so much joy to our lives and to everyone in the family. May he never become self-conscious, and never stop laughing.

This thesis has been generously supported by a Google PhD Fellowship in Fault Tolerant Computing, a US-Israel Binational Science Foundation grant (2006204), and a National Science Foundation CNS NeTS Award (0831374).

To Faye and Sammy

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Customers of Internet services today—whether end users browsing the web, or companies hosting their services on the cloud—have become increasingly intolerant of delays and outages, while demanding a consistent online experience. Every year, downtime costs North American businesses an estimated $26.5 billion [168]. The rise in cloud hosting has exacerbated this cost: failures in Amazon's EC2 cloud infrastructure, for example, have single-handedly taken down popular websites like Reddit, Netflix, and Edmodo [38, 39], resulting in public outcry.

The most notorious of these failures, and indeed the most damaging to the affected company and its customers, are the ones that result in incorrect service behavior. For example, Amazon S3 was taken offline because a single bit flip corrupted system state information and caused nearly all customer requests to fail [11]. A bug in Amazon EBS's clean-up software caused customer data to be incorrectly deleted [182]. Google posted safety warnings on every search result because someone mistyped a URL into its list of malware-suspected sites [71]. Facebook leaked source code due to one misconfigured server [167]. Flickr mixed up user images due to one faulty cache server [60].

In each of these examples, the affected service manifested arbitrary, or so-called *Byzantine*, behavior. In the case of Amazon S3, Facebook, and Flickr, the underlying error affected each server *independently*, causing just one or a handful of them to misbehave. In the case of Amazon EBS and Google, the underlying error had a *correlated* effect across the servers, affecting many of them simultaneously. Protecting against correlated failures is difficult in general because it is hard to predict their size and extent. Several existing approaches, such as relying on massive redundancy [78] or using history to predict failure patterns [174], have limited benefits in alleviating the true cost of correlated failures [123]. Thus, point solutions are often used in practice. We will see an example of a real-world correlated failure, and its solution, later in this thesis.

In contrast, if failures are independent, or if they can be made independent—for example, using techniques like opportunistic N-version programming [17, 32], where distinct implementations of the same service (e.g., a database) are run on different machines—then one can reasonably assume a fixed bound on the number of simultaneous failures, because the probability of such an occurrence drops exponentially. In this case, general solutions are possible, such as Byzantine-fault-tolerant (BFT) replicated state machines [29, 150], in which system state and functionality are replicated and executed across multiple machines. These protocols ensure the correctness and availability of the service when fewer than $1/3$ of the replicas are faulty. Unfortunately, despite considerable effort [31, 73, 100, 101, 170, 176, 178, 180], state-of-the-art BFT protocols [29, 73, 100] still scale poorly with system size, because they require each replica to participate in every request, more than $2/3$ of the replicas to maintain the service state [180], and multiple rounds of quadratic communication per request. Even services that already replicate their data, such as the Google File System [69], would see their throughput drop significantly when using BFT agreement. Thus, the application of these protocols has been limited to small sets of nodes within larger systems, e.g., as a lock service [36].

This thesis describes *scalable* systems and algorithms for tolerating Byzantine failures, by both *fixing* the causes of failures and *masking* their effect from customers. We first describe a specific solution that fixes a real-world correlated failure. Then, we show how to avoid such correlations, and describe general solutions for masking independent failures. Our main technique for achieving scalability is *avoiding work*, but doing so in a principled manner. We use a combination of systems and theory techniques to achieve this. In some cases, due to existing lower bounds, we must relax the guarantees of our protocols, or rely on trusted primitives, in exchange for added. However, we show that the guarantees are still strong enough for most applications, and any additional assumptions are both minimal and practical on commodity machines.

1.1 Contributions and outline

We begin by describing a real-world failure that befell a certain database provider. The provider was trying to optimize the tree index structure of a particular database, but ended up triggering a bug that brought down all of the database's replicas. We present a new class of search trees, called *relaxed trees*, that fixes the cause of the failure (Chapter 2). This incident sets the stage for the remainder of the thesis, by illustrating the difficulties imposed by Byzantine failures and correlated failures when designing reliable, scalable services.

Fortunately, there are well-known techniques for avoiding correlations in certain replicated systems, such as databases and file systems, which allows us to use BFT protocols to mask the effect of failures. We present a system called Prophecy that interposes itself between clients and any BFT group to perform fast, load-balanced reads when results are historically consistent, while only slightly weakening the group's semantics to what we term *delay-once* semantics (Chapter 3). Intuitively, delay-once consistency implies that faulty nodes can at worst return slightly stale (not arbitrary) data.

To overcome Prophecy's scalability limitations on write-heavy workloads, we present commensal cuckoo, an algorithm that securely composes many small BFT groups, despite a dynamic adversary that can coordinate faulty nodes across groups (Chapter 4). Finally, we improve the scalability of both commensal cuckoo and Prophecy by increasing the fault resilience within a BFT group, by introducing small broadcast channels that prevent faulty nodes from sending contradictory messages (Chapter 5).

We end with some conclusions (Chapter 6).

Most of the results in this thesis previously appeared in several conference and journal proceedings [87, 152, 153, 155].

Chapter 2

Relaxed Trees: A Data Structural Solution to a Real-World Failure

Here is the true story that motivated this work, fictionalized to protect the parties involved. A database provider was contracted to build a real-time database to store customer information, to be queried and updated on a regular basis. The provider decided to use a red-black tree [74] to index the database, but implemented rebalancing only after insertions, not after deletions. As a safety check, a limit of 80 was placed on the allowed height of the tree. This limit would allow storage of 2^{40} records in a valid red-black tree, far exceeding the anticipated number. Exceeding the height bound was interpreted as an error and triggered a recovery process intended to restore the database. For reliability and scalability, the database was replicated across several machines: updates were sent to all replicas, and queries were load-balanced to individual replicas. Sometime after the database was deployed, an update caused the height bound of the tree to be exceeded on one replica, triggering the recovery. This process, too, caused the height bound to be exceeded, and this cycle repeated, taking down the replica. Since all replicas received the same update (and all were running identical code), each suffered the same fate in turn, causing an extended service outage.

The above outage occurred for two reasons: there were bugs in the tree maintenance and recovery code, and the same code ran on all replicas—i.e., the failures were correlated. Thus, there are two ways in which we can address the outage. First, we can fix the bugs that caused the failure; this would improve the quality of the database code and prevent the same failure from recurring. Second, we can run distinct implementations of the database to avoid correlated failures, and use a BFT protocol to *mask* the effect of future failures. This chapter focuses on the first approach; the next chapter discusses the second approach.

Avoiding deletion rebalancing in search trees

Ultimately, the real cause of the database outage was the fact that the red-black tree used by the provider became very unbalanced and exceeded a predefined height limit. This raises an interesting theoretical question: can one maintain balance in a search tree by rebalancing only after insertions, not after deletions? Before considering this question, we review some of the literature concerning deletion in balanced trees. Such a review provides insight into how the event described above came about.

The original paper on balanced search trees [3], which introduced AVL trees to the world, is only four pages long. It describes how to rebalance an n-node AVL tree after an insertion by doing one or two rotations and updating height information in $\mathrm{O}(\log n)$ nodes. An algorithm for rebalancing after a deletion appeared several years later, in a technical report by a different author [61]; deletion rebalancing requires $\mathrm{O}(\log n)$ rotations rather than $\mathrm{O}(1)$. For all existing forms of balanced trees, of which there are many (e.g.,, [13, 22, 23, 74, 79, 80, 124, 131, 151]), deletion is at least a little more complicated than insertion, although for some kinds of balanced search trees, notably red-black trees [74] and the recently introduced weak AVL (wavl) trees [79, 80], rebalancing after a deletion can be done in $\mathrm{O}(1)$ rotations. Many textbooks describe algorithms for insertion but not deletion. If operations on the search tree occur concurrently, as in many database systems that use some form of B-tree as the underlying data structure, the synchronization necessary to do rebalancing on deletion reduces the available concurrency [72]. Several database systems, including Berkeley DB [132, 133], use a B$^+$ tree with underfilled nodes that is rebalanced after insertions but not deletions. Thus it was perhaps natural to try something similar for red-black trees. But disaster ensued.

A more precise version of our question is this: can one maintain a search tree so that search time is logarithmic but rebalancing is done only after insertions, not after deletions? To answer this question, we need to ask, "logarithmic in what parameter?" If there is no rebalancing after deletions (and none after accesses, which excludes self-adjusting structures such as splay trees [161]), then the tree can evolve to have arbitrary structure, which means that the search time can become $\Theta(n)$. But such an evolution may take many deletions, and it is still possible that the tree height, and hence the search time, could remain logarithmic in m, the number of insertions. We answer this question affirmatively.

Results and outline

We introduce a new kind of binary tree, the *ravl tree* (relaxed AVL tree), which is rebalanced only after insertions, not after deletions, and whose height is at most $\log_\phi m$, where ϕ is the golden ratio. This bound is the same as that for an ordinary AVL tree without deletions. Indeed, without deletions a ravl tree is exactly an AVL tree. Furthermore, rebalancing affects nodes exponentially infrequently in their heights, which means that the amortized rebalancing time per insertion is $\mathrm{O}(1)$ and most of the rebalancing occurs deep in the tree. Mehlhorn and Tsakalidis [116], proved the latter property for standard AVL trees if only insertions are allowed, not deletions. They used a multilevel credit method to obtain their result. Our analyses use exponential potential functions, a tool that unifies and simplifies the multilevel credit method, and which we also used [79, 80] to analyze wavl trees. Our results hold for bottom-up rebalancing; we extend them to top-down rebalancing with finite look-ahead as well. Perhaps surprisingly, we obtain better constant factors for many of our bounds than the corresponding bounds for wavl trees. Thus not only does rebalancing after deletions complicate the implementation, it makes the performance of the data structure worse in some ways.

It is natural to ask whether one can obtain similar results for multiway trees, in particular B trees or B^+ trees. The answer is yes, and indeed B^+ trees with underfilled nodes have the desired properties, as we show in companion work [154, 156]. Red-black trees can be viewed as a special case of B-trees, and our ideas apply to them as well.

The price we pay for our results on binary trees is that each node in the tree must store $\lg \lg m + 1$ bits of balance information (or $\lg \lg n + \mathrm{O}(1)$ with periodic rebuilding)[1], rather than the one bit per node needed in AVL [3], wavl [79, 80], and red-black trees [74]. Indeed, we provide evidence to suggest that $\mathrm{O}(1)$ bits suffice only if one does cascading swaps of items between nodes during deletions. (We leave rigorous resolution of this question as an open problem.) We conclude that the approach used by the unfortunate database provider to keep red-black trees balanced without rebalancing on deletion was theoretically doomed. That this would manifest itself in practice is a wonder to a theoretician.

The body of this chapter consists of nine sections. Section 2.1 contains our tree terminology. Section 2.2 defines ravl trees and describes bottom-up rebalancing after an insertion; the rebalancing algorithm is that of AVL trees, extended to ravl trees. Section 2.3 analyzes the amortized efficiency of bottom-up rebalancing. Section 2.4 describes and analyzes top-down rebalancing with fixed look-ahead, an alternative rebalancing method that improves concurrency. Section 2.5 applies the ideas in Sections 2.2-2.4 to red-black trees. Section 2.6 describes a way to rebuild the trees efficiently if they becomes unbalanced. Section 2.7 examines other ways of handling insertions and deletions, including doing deletions lazily using the "tombstone" method, and gives examples showing that natural eager methods that use one balance bit per node fail. Section 2.8 explores the pros and cons of rebalancing after deletions. Section 2.9 contains final remarks.

[1]We denote by $\lg$ the base-two logarithm.

2.1 Tree Terminology

Our tree terminology is the same as in [79, 80]. We repeat it here (almost verbatim) for completeness. A *binary tree* is an ordered tree in which each node x has a *left child* $left(x)$ and a *right child* $right(x)$, either or both of which may be missing. Missing nodes are *external*; non-missing nodes are *internal*. Each node is the *parent* of its children. We denote the parent of a node x by $p(x)$. The *root* is the unique node with no parent. A *leaf* is a node with both children missing. The *ancestor*, respectively *descendant* relationship is the reflexive, transitive closure of the parent, respectively child relationship. If x is a node, its *left*, respectively *right* subtree is the binary tree containing all descendants of $left(x)$, respectively $right(x)$. The *left*, respectively *right spine* of a binary tree is the path from the root down through left, respectively right children to a missing node. The *height* $h(x)$ of a node x is defined recursively by $h(x) = 0$ if x is a leaf, $h(x) = \max\{h(left(x)), h(right(x))\} + 1$ otherwise. The height h of a tree is the height of its root.

We are most interested in binary trees as search trees. A binary search tree stores a set of *items*, each of which has a *key* selected from a totally ordered universe. We shall assume that each item has a distinct key; if not, we break ties by item identifier. In an *internal binary search tree*, each node contains an item and the items are arranged in *symmetric order*: the key of the item in a node x is greater, respectively less than those of all items in its left, respectively right subtree. Given such a tree and a key, we can search for the item having that key by comparing the key with that of the item in the root. If they are equal, we have found the desired item. If the search key is less, respectively greater than that of the root, we search recursively in the left, respectively right subtree of the root. Each key comparison is a *step* of the search; the *current node* is the one whose item's key is compared with the search key. Eventually the search either locates the desired item or reaches a missing node, the left or right child of the last node reached by the search in the tree.

To insert a new item into such a tree, we first do a search on its key. When the search reaches a missing node, we replace this node with a node containing the new item. Deletion is a little harder. First we find node x containing the item to be deleted by doing a search on its key. If neither child of x is missing, we find either the next item or the previous item, by walking down through left, respectively right children of the right, respectively left child of x until reaching a node y with a missing left, respectively right child. We swap the items in x and y. Now the node containing the item to be deleted is either a leaf or has one missing child. In the former case, we replace it by a missing node; in the latter case, we replace it by its non-missing child. If each node has pointers to its children, an access, insertion, or deletion takes $O(h+1)$ time in the worst case, where h is the tree height.

An alternative kind of search tree is an *external binary search tree*: the external nodes contain the items, the internal nodes contain keys but no items, and all the keys are in symmetric order. Henceforth, unless we explicitly state otherwise, by a binary tree we mean an internal binary search tree. Our results extend to external binary search trees and to other binary tree data structures. We denote by n, m, and d, respectively the current number of nodes, the number of insertions, and the number of deletions in a sequence of intermixed searches, insertions, and deletions; $n = m - d$.

2.2 Relaxed AVL Trees

We define balance in a binary tree by giving each node a rank and imposing a rank rule that constrains the ranks. For a general discussion of this approach, which captures all forms of height balance of which we are aware, see [80].

A *ranked binary tree* is a binary tree in which each node x has an integer *rank* $r(x)$. Missing nodes have rank -1. The *rank difference* of a node x with parent $p(x)$ is $r(p(x)) - r(x)$. An *i-child* is a node of rank difference i; an *i, j-node* is a node whose children have rank differences i and j. The latter definition does not distinguish between left and right

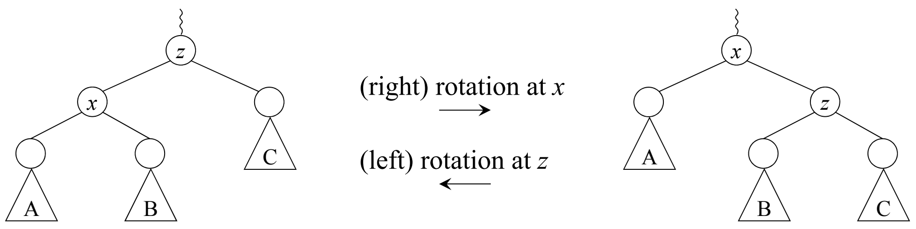


Figure 2.1: Rotation. Triangles denote subtrees.

children. An *AVL tree* is a ranked binary tree in which every node is a 1,1-node or a 1,2-node. The leaves of an AVL tree are 1,1-nodes of rank zero. A *relaxed AVL tree*, or *ravl*[2] *tree*, is a ranked binary tree that obeys the following *rank rule*: every rank difference is positive.

Lemma 2.2.1. *In a ravl tree, each node has height no greater than its rank.*

Proof. Every node has rank greater than the maximum of the ranks of its children. Since missing nodes have rank -1, leaves have non-negative rank. The lemma follows by induction on the node rank. □

Any binary tree can be made into a ravl tree by a suitable choice of node ranks; indeed, there are always many ways to do it. The efficiency of ravl trees comes not from their static structure but from the implementation of insertions and deletions and how this affects the tree structure over time. We consider ravl trees built from the empty tree by a sequence of intermixed insertions of leaves and deletions of arbitrary nodes. A new leaf x replaces a missing node and has a rank of zero. If the parent $p(x)$ of x was itself a leaf before the insertion, x is a 0-child and violates the rank rule. We restore the rank rule by promoting and demoting nodes and doing rotations. A *promotion* increases the rank of a node by one, a *demotion* decreases it by one. A *rotation* at a left child x with parent y makes y the right child of x while preserving symmetric order; a rotation at a right child is symmetric. (See Figure 2.1.) The insertion rebalancing algorithm is as follows (see Figure 2.2):

[2]One meaning of "ravel" is "to undo the intricacies of". Ravl trees undo the intricacies of deletions.

Insertion rebalancing:

While $p(x) \neq$ null and $p(x)$ is 0,1, repeat the following step:

> *Promote*: Promote $p(x)$; replace x by $p(x)$.

Now either the rank rule holds or x is a 0-child whose sibling is an i-child with $i > 1$. In the latter case, proceed as follows. Assume x is the left child of $z = p(x)$; the other possibility is symmetric. Let y be the right child of x; y may be missing. Do the appropriate one of the following two steps:

> *Rotate*: If node y is missing or a 2-child, rotate at x and demote z.
>
> *Double Rotate*: Otherwise (node y is a 1-child), rotate at y twice, making x its left child and z its right child; promote y and demote x and z.

During rebalancing, there is exactly one violation of the rank rule: x is a 0-child. A rotate or double rotate step restores the rank rule and terminates rebalancing, as does a promote step that promotes the root or results in the new x being an i-child with $i > 0$. In the first rebalancing step, x is a leaf of rank zero and hence a 1,1-node; in each rebalancing step after the first, x is a 1,2-node. The *rank* of a rebalancing step is the rank of $p(x)$ just before the step. Each step has rank one higher than that of the previous step. The *rank* of an insertion is the rank of the last rebalancing step, or zero if there is no rebalancing.

To delete an item in a leaf in a ravl tree, we replace the leaf by a missing node. To delete an item in a node with one child, we remove the node and replace it by its child; this child becomes the left or right child of the old parent of the deleted node if the deleted node was a left or right child, respectively. To delete an item in a node with two children, we swap the item with its symmetric-order predecessor or successor, thereby moving it to a leaf or a

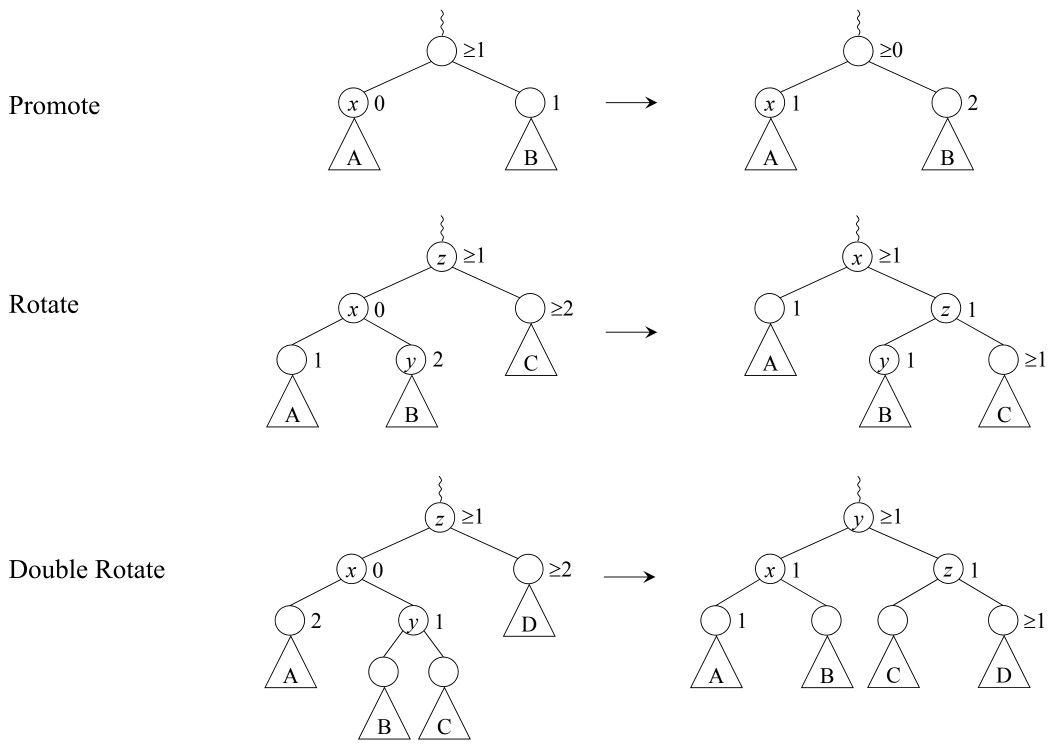


Figure 2.2: Bottom-up rebalancing after an insertion in ravl trees. Numbers are rank differences. The first case is possibly non-terminating.

node with one child, and proceed as above. In a deletion, no rotations occur and *no ranks change*.

As long as there are no deletions, all nodes remain 1,1- or 1,2-nodes (except in the middle of rebalancing), so the tree remains an AVL tree. Indeed, the rebalancing algorithm is just the standard bottom-up rebalancing algorithm for AVL trees. All the results we shall derive for bottom-up rebalancing hold as a special case for AVL trees built by insertions only. Deletions can create nodes of arbitrary positive rank difference, however, and thus can create trees that are not AVL trees. Indeed, deletions can produce trees of arbitrary structure.

We represent a ravl tree by storing with each node its rank and pointers to its left and right children. An alternative is to store ranks in difference form: the root stores its rank, and every child stores its rank difference. This only works if access to the tree is always via the root, and it requires computing node ranks during an insertion by summing rank differences along the path from the root to the new leaf. In an AVL tree, rank differences are one or two, so one bit per node suffices to store rank differences. But in a ravl tree, rank differences can become arbitrarily large, and storing ranks in difference form offers no advantages and at least one disadvantage. Thus we prefer to store ranks explicitly.

The rebalancing process after an insertion needs access to the affected nodes on the search path. There are several ways to provide such access, as we have discussed previously [80]. One way is to add parent pointers, which requires three pointers per node instead of two and increases the cost of rotations. By using an alternative representation that saves space but costs time, this can be reduced to two pointers per node [64].

Instead of adding or modifying pointers to support parental access, we can store the search path during the search from the root for the insertion position, either in a separate stack or by reversing child pointers along the path.

A third method is to maintain a *safe node* during the search. This node is the topmost node that will be affected by rebalancing. Metzger [117] and Samadi [148] used safe nodes to limit the amount of locking in a concurrent B-tree. We apply this idea to binary trees and use it for a slightly different purpose: to avoid the need for parent pointers or a stack to do rebalancing. During an insertion, the safe node is either the root or the parent of the nearest ancestor of the current node that is not a 1-child and not a 1,1-node. A simpler alternative is to define the safe node to be the parent of the nearest ancestor of the current node that is not a 1,1-node, or the root if there is no such node. The latter definition gives the same node as the former, or its parent. We initialize the safe node to be the root and change it to the parent of the current node each time the current node is not a 1,1-node (or not a 1-child, if we are using the former definition). Once the search reaches the bottom of the tree, we do rebalancing steps (modified appropriately) top-down from the safe node to the new leaf. One advantage of this method is that it extends naturally to support top-down rebalancing with fixed look-ahead, as we discuss in Section 2.4.

2.3 Analysis of Bottom-Up Rebalancing

A search in a ravl tree takes $\mathrm{O}(h+1)$ time, where h is the tree height. A deletion takes $\mathrm{O}(h+1)$ time to find the item to be deleted and the node containing its replacement, if any, plus $\mathrm{O}(1)$ time to do the deletion. An insertion takes $\mathrm{O}(h+1)$ time to find the location of the new leaf, plus at most two rotations and $\mathrm{O}(h+1)$ rebalancing steps. All these bounds are worst-case. To obtain better bounds for rebalancing and to bound the height of the tree, we use the potential method of amortized analysis [165]. To each state of the data structure we assign a non-negative *potential*, zero for an empty structure. We define the *amortized cost* of an operation to be its actual cost plus the net increase in potential it causes. Then, for any sequence of operations on an initially empty structure, the total amortized cost of the operations is an upper bound on their total actual cost.

Our first, simple amortization argument shows that each insertion takes $\mathrm{O}(1)$ amortized promote steps. We define the potential of a tree to be the number of 0,1-nodes plus the number of 1,1-nodes of positive rank. We define the cost of an insertion to be the number of promotion steps done during rebalancing. A deletion cannot increase the potential since it cannot create a 0,1-node or a 1,1-node. Consider an insertion. Adding a new leaf increases the potential by at most one, by creating a 0,1-node or a 1,1-node of positive rank. A non-terminal promote step decreases the potential by one: the promoted node changes from a 0,1-node to a 1,2-node; the potential of its parent does not change. Thus such a step has an amortized cost of zero. This is also true of a promote step that promotes the root. A terminal promote step that promotes a node other than the root can leave the potential unchanged (if the parent of the promoted node becomes a 1,1-node), and thus has an amortized cost of one. A rotate or double rotate step can increase the potential by at most two, by creating two 1,1-nodes of positive rank. We conclude that the amortized cost of an insertion is at most three: one for the increase in potential caused by inserting a new leaf, plus zero for each non-terminal promote step, plus at most two for the last rebalancing step. This gives the following theorem:

Theorem 2.3.1. *Starting with an empty ravl tree, a sequence of m insertions with bottom-up rebalancing intermixed with arbitrary deletions does at most $3m$ promote steps.*

An exponential potential function of the kind first used by us to analyze wavl trees [79, 80] gives our most important result: a ravl tree built from an empty tree has height logarithmic in the number of insertions, even if deletions are intermixed arbitrarily. Recall the definition of the Fibonacci numbers F_k and of the golden ratio ϕ: $F_0 = 0, F_1 = 1, F_k = F_{k-1} + F_{k-2}$ for $k > 1$; $\phi = (1+\sqrt{5})/2$. The inequality $F_{k+2} \geq \phi^k$ is well-known [98]. We define the potential of a node of rank k to be F_{k+2} if it is a 0,1-node, F_{k+1} if it has a 0-child but is not a 0,1-node, F_k if it is a 1,1-node, and zero otherwise. We define the potential of a tree to be the sum of the potentials of its nodes. We call this the *Fibonacci potential*.

A deletion cannot increase the potential. Adding a new leaf increases the potential by at most one, either by creating a new 1,1-node of rank one or by creating a new 0,1-node of rank zero, since $F_1 = F_2 = 1$. Consider a rebalancing step of rank k. A promote step that promotes $z = p(x)$ and makes $p(z)$ a 0,1-node does not change the potential: z and $p(z)$, respectively, have potentials F_{k+2} and F_{k+1} before the step and zero and $F_{k+3} = F_{k+2} + F_{k+1}$ after. A promote step that promotes $z = p(x)$ and makes $p(z)$ a $0, i$-node with $i > 1$ also does not change the potential: z and $p(z)$, respectively, have potentials F_{k+2} and zero before the step and zero and F_{k+2} after. Likewise, a rotate step does not increase the potential: nodes x and $z = p(x)$, respectively, have potentials zero and $F_{k+1} = F_k + F_{k-1}$ before the step and F_k and at most F_{k-1} after. Neither does a double rotate step: if $y = right(x)$ is a 1,1-node before the step, the total potential of x, y, and $z = p(x)$ is $F_{k-1} + F_{k+1}$ before the step and at most $F_{k-1} + F_k + F_{k-1}$ after; if y is not a 1,1-node before the step, the total potential of x, y, and z is F_{k+1} before the step and at most $F_k + F_{k-1}$ after. The final possibility is a terminal promote step. If the promotion of $z = p(x)$ makes $p(z)$ a 1,1-node, it does not change the potential: z and $p(z)$, respectively, have potentials F_{k+2} and zero before the step and zero and F_{k+2} after the step. If the promotion of z does not make $p(z)$ a 1,1-node, in particular if z is the root, the step decreases the potential by F_{k+2}.

Theorem 2.3.2. *If a ravl tree is built from an empty tree by a sequence of m insertions with bottom-up rebalancing intermixed with arbitrary deletions, $m \geq F_{h+3} - 1 \geq \phi^h$. Thus $h \leq \log_\phi m$.*

Proof. Let the potential be the Fibonacci potential. The first insertion leaves the potential at zero. Each subsequent insertion increases the potential by at most one, not counting decreases resulting from terminal promote steps. If the rank of the root is r, there was a terminal promote step of rank i that promoted the root, for each i from 0 to $r-1$, inclusive. The total decrease in potential caused by these promotions is $\sum_{i=2}^{r+1} F_i = F_{r+3} - 2$. Since the potential is always non-negative, $m - 1 \geq F_{r+3} - 2$. By Lemma 2.2.1, $h \leq r$. Thus $m \geq F_{h+3} - 1 \geq F_{h+2} \geq \phi^h$. $\square$

By truncating the Fibonacci potential, we can show that rebalancing steps of rank k occur exponentially infrequently in k.

Theorem 2.3.3. *Starting from an initially empty tree, a sequence of insertions with bottom-up rebalancing intermixed with arbitrary deletions does at most $(m-1)/F_k \leq (m-1)/\phi^{k-2}$ rebalancing steps of rank k, for any $k > 0$.*

Proof. Fix $k > 0$. Let the potential of a node be its Fibonacci potential if its rank is less than k, F_{k+1} if its rank is k, it has a 0-child, and it is not a 0,1-node, and zero otherwise. Let the potential of a tree be the sum of the potentials of its nodes. The effect of a rebalancing step on the potential is the same as discussed above, with the following exceptions. A promote step of rank k or higher does not change the potential. A promote step of rank $j < k$ that promotes a node $z = p(x)$ whose parent $p(z)$ has rank at least k decreases the potential by F_{j+2} unless $j = k-1$, $p(z)$ has rank k, and $p(z)$ is not a 1,1-node before the step, in which case it does not change the potential. A rotate or double rotate of rank greater than k does not change the potential. A rotate or double rotate of rank k decreases the potential by at least $F_{k+1} - F_{k-1} = F_k$. Thus no rebalancing step increases the potential. Furthermore a rebalancing step of rank k either decreases the potential by at least F_k (if it is a rotate or double rotate) or is preceded by a promote step of rank $k-1$ that reduces the potential by F_{k+1}. Thus the potential decreases by at least F_k for every rebalancing step of rank k. $\square$

2.4 Top-Down Rebalancing

Rather than rebalance bottom-up after a new leaf is added, we can rebalance top-down before the leaf is added. Indeed, the safe node method described at the end of Section 2.2 does rebalancing top-down once it reaches the bottom of the tree. We can modify this method to rebalance more eagerly and thereby keep the look-ahead fixed; that is, keep the safe node within $\mathrm{O}(1)$ nodes of the current node of the search. This improves the worst-case concurrency of the tree, because the critical section of an insertion encompasses only $\mathrm{O}(1)$ nodes at any time. The idea is to force a reset of the safe node after a sufficiently large number of search steps that do not do a reset. A reset occurs at the next search step unless the current node is a 1,1-node. If the current node is a 1,1-node but not a 1-child, or both it and its parent are 1,1-nodes, we can force a reset by promoting the current node and rebalancing from the safe node top-down. This gives us the following top-down insertion algorithm, which we describe in complete detail to make its operation crystal-clear. If the tree is empty, create a new node of rank zero containing the item to be inserted and make it the root, completing the insertion. Otherwise, initialize t and z to be the root, and promote the root if it is 1,1. This establishes the invariant for the main loop of the algorithm: z is a non-null node that is not a 1,1-node; t is the parent of z unless z is the root, in which case $t = z$. Repeat the following step until the item is inserted:

Top-down insertion step:

From z, take one step down the search path, to x. If x is null, replace it by a new node of rank zero containing the item to be inserted, completing the insertion: the new node cannot be a 0-child since z was not a 1,1-node and hence has positive rank. In the remaining cases, x is not null. If x is not a 1,1-node, replace t by z and z by x, completing the step. If x is a 1,1-node but not a 1-child, promote x and replace t by z and z by x, completing the step. In the remaining cases, x is a 1,1-node and a 1-child. From x take one step down the search path, to y. If y is null, replace it by a new node of rank zero containing the item to be inserted; if the new node is a 0-child, promote x and do a single or double rotate step to make all rank differences positive. This completes the insertion. The remaining possibility is y non-null. If y is not a 1,1-node, replace t by x and z by y. Otherwise, promote x and y, making x a 0-child, and do a single or double rotate step to make all rank differences positive. If a single rotation is done, replace t by x and z by y. If a double rotation is done, replace t by y and z by the new node one step down from x along the search path (either x or z). This completes the step.

If the first insertion step does a single or double rotation, it changes the root. Subsequent steps do not affect the root and have $t \neq z$. Each insertion step either finishes the insertion or replaces z by a node of smaller rank, so the number of insertion steps is at most one plus the rank of the root. We define the *rank* of an insertion step to be the rank of x, or zero if x is null. Every insertion step of positive rank is non-terminal, since in such a step x is non-null and is either a 1,1-node, in which case y must be non-null, or not a 1,1-node, in which case the step finishes without descending to y. We call an insertion step *rebalancing* if it is terminal or it does at least one promotion or rotation; the non-rebalancing steps merely traverse the search path without changing the tree.

We can show by a simple potential argument that the number of rebalancing steps is $\mathrm{O}(m)$. Let the potential of a tree be twice the number of 1,1-nodes of positive rank plus the number of $1, i$-nodes with $i > 1$. An insertion into an empty tree does not increase the potential, nor does promoting z in the initialization. An examination of the remaining cases shows that a non-terminal rebalancing step decreases the potential by at least one, and a terminal rebalancing step increases it by at most one. This gives us the following theorem:

Theorem 2.4.1. *Starting with an empty ravl tree, a sequence of m top-down insertions intermixed with arbitrary deletions takes at most $2m$ rebalancing steps.*

To obtain analogues of Theorems 2.3.2 and 2.3.3, we use an exponential potential function that grows more slowly than the Fibonacci potential. We define the potential of a node of rank $k > 0$ to be $2^{(k+1)/2}$ if it is a 1,1-node, $2^{(k-1)/2}$ if it is a $1, i$-node with $i > 1$, and zero otherwise; we define the potential of a node of rank zero to be zero; and we define the potential of a tree to be the sum of the potentials of its nodes. An insertion into an empty tree does not increase the potential, nor does promoting z in the initialization. Consider a restructuring insertion step of rank k that is not the last insertion step. If x is a 1,1-node that is not a 1-child, it is promoted. This increases the potential by at most $2^{(k+1)/2} - 2^{(k+1)/2} \leq 0$, since the potential of x decreases from $2^{(k+1)/2}$ to zero, and the potential of z can only increase if it has rank $k+2$, in which case it increases by $2^{(k+1)/2}$. If x is a 1,1-node that is a 1-child, and y is also a 1,1-node, x and y are promoted and a single or double rotation is done. In the former case, the potentials of x, y, and z, respectively, are $2^{(k+1)/2}$, $2^{k/2}$, and $2^{k/2}$ before the step and $2^{(k+2)/2}$, 0, and at most $2^{(k+1)/2}$ after, resulting in no increase in the potential of the tree. In the latter case, the potentials of x, y, and z, respectively, are $2^{(k-1)/2}$, $2^{(k+2)/2}$, and at most $2^{(k-1)/2}$ after the step, again resulting in no increase in the potential of the tree. Consider a terminal insertion step. If the step replaces x by an empty node, it increases the potential of z, and hence of the tree, by at most one. Suppose the step replaces y by an empty node. If it does a single rotation, it increases the potential of x from zero to two and decreases that of z from 1 to zero, for a net increase of one. If it does a double rotation, it increases the potential of y from zero to two and decreases that of z from one to zero, again for a net increase of one. We conclude that a terminal insertion step increases the potential by at most one.

Theorem 2.4.2. *A ravl tree built from an empty tree by a sequence of m top-down insertions intermixed with arbitrary deletions has height at most $2 \lg m$.*

Proof. Deletions do not increase the potential. By the discussion above, each insertion other than the first increases the potential by at most one. If the root has rank $k > 0$ and it is promoted, the potential decreases by $2^{(k+1)/2}$. For the root to have height h, it must have rank at least h, which means that root promotions have decreased the potential by at least $\sum_{i=1}^{h-1} 2^{(i+1)/2} = 2^{(h+1)/2}/(\sqrt{2}-1)$. Since the total decrease is at most m, this gives $2^{h/2} \leq m$. $\square$

Theorem 2.4.3. *Starting from an empty tree, a sequence of m top-down insertions intermixed with arbitrary deletions does at most $m/2^{k/2}$ rebalancing steps of rank k.*

Proof. The lemma is immediate for $k = 0$. Fix $k > 0$. Redefine the exponential potential function used to prove Theorem 2.4.2 to be zero for all nodes of rank greater than k. It is still true that no insertion step increases the potential, and that each terminal step increases it by at most one. A rebalancing step of rank k is non-terminal. If it does not do a single or double rotation, it decreases the potential by $2^{(k+1)/2}$, by decreasing the potential of y from $2^{(k+1)/2}$ to zero. If it does a single rotation, it decreases the potential by at least $2^{k/2}$: the potentials of x, y, and z, respectively, are $2^{(k+1)/2}$, $2^{k/2}$, and zero before the step and zero, zero, and at most $2^{(k+1)/2}$ after. If it does a double rotation, it also decreases the potential by at least $2^{k/2}$: the potentials of x, y, and z, respectively, are $2^{(k-1)/2}$, zero, and at most $2^{(k-1)/2}$ after the step. The theorem follows. $\square$

By increasing the amount of look-ahead in top-down insertion, we can improve the constants in Theorems 2.4.2 and 2.4.3. Specifically, we force a reset after traversing k consecutive 1,1-nodes, of which the top one is not a 1-child, or traversing $k+1$ consecutive

1,1-nodes of which the top one is a 1-child, by promoting the bottom 1,1-node and rebalancing appropriately. Here $k \geq 2$ is an appropriately large constant. To analyze this method, we define the potential of a 1,1-node of rank k to be b^k for some appropriate constant $b > 1$, that of any other node to be zero, and that of a tree to be the sum of the potentials of its nodes. If the parent of the top 1,1-node has rank k just before the forced reset, then the rebalancing increases the potential by at most $b^k - b^{k-2} - b^{k-3} - \ldots - 1$, whether or not the top 1,1-node is a 1-child. By choosing k sufficiently large, we can choose b arbitrarily close to ϕ while guaranteeing that forced resets do not increase the potential, giving an analogue of Theorem 2.4.2 with b in place of $\sqrt{2}$. By truncating the potential, we obtain an analogue of Theorem 2.4.3 with b in place of $\sqrt{2}$. Choosing $k = 3$ is sufficient to give $b > \sqrt{2}$. Interestingly, for minimum look-ahead ($k = 1$), this potential function is not useful; for $k > 2$, giving positive potential to the $1, i$-nodes for $i > 1$ makes the analysis worse, not better.

2.5 Relaxed Red-Black Trees

In this section we apply our ideas to red-black trees to obtain relaxed red-black trees. Although the results of this section follow from our results on relaxed B-trees [156], we sketch them here for completeness, to show that they can be derived directly, without using multiway trees as an intermediary, and to enlarge our study in Section 2.7 of what can go wrong with alternative deletion methods.

A *red-black* tree [74] is a binary tree is which each node is either red or black, with the node colors satisfying the following constraints:

Black Rule: Every path from the root to a missing node contains the same number of black nodes.

Red Rule: The parent of a red node is black.

Red-black trees are equivalent to 2,4-trees, which are multiway trees in which all leaves have the same depth, each internal node has 2, 3, or 4 children, each internal node contains one less item than its number of children, and each leaf contains 1, 2, or 3 items. To obtain the 2,4-tree equivalent of a given red-black tree, contract each red node into its parent. To obtain the red-black tree equivalent of a given 2,4-tree, split each node with two items into a black parent and a red child, each with one item, and split each node with three items into a black parent and two red children. Since there are two ways to split a node with two items, the latter mapping is one-to-many.

Red-black trees are also equivalent to ranked binary trees satisfying the *red-black rank rule*: every node has a non-negative rank, all rank differences are zero or one, every leaf has rank zero, and every 0-child has a parent that is not a 0-child [80]. Given a ranked binary tree satisfying the red-black rank rule, we color its nodes to satisfy the red and black rules, by coloring the root black and coloring each child black if it is a 1-child or red if it is a 0-child. Given a red-black tree, we assign ranks to the nodes to satisfy the red-black rank rule, by giving each node a rank equal to the number of black nodes on every path from it to a missing node.

Red-black trees were invented by Bayer [23], who called them *symmetric binary B-trees*, and popularized by Guibas and Sedgewick [74], who invented the red-black representation. A red-black tree of n nodes has height at most $2 \lg n$. Rebalancing a red-black tree after an insertion or deletion takes at most two rotations worst-case for an insertion, at most three rotations worst-case for a deletion, and $\mathrm{O}(1)$ amortized color flips for an insertion or deletion [164]. The insertion rebalancing algorithm is like that of AVL trees (and ravl trees): there are three rebalancing cases (ignoring symmetries); the first does only color flips but need not terminate, the second does a rotation and some color flips and terminates, and the third does two rotations and some color flips and terminates. For further results and discussion about red-black trees see [23, 74, 164, 166].

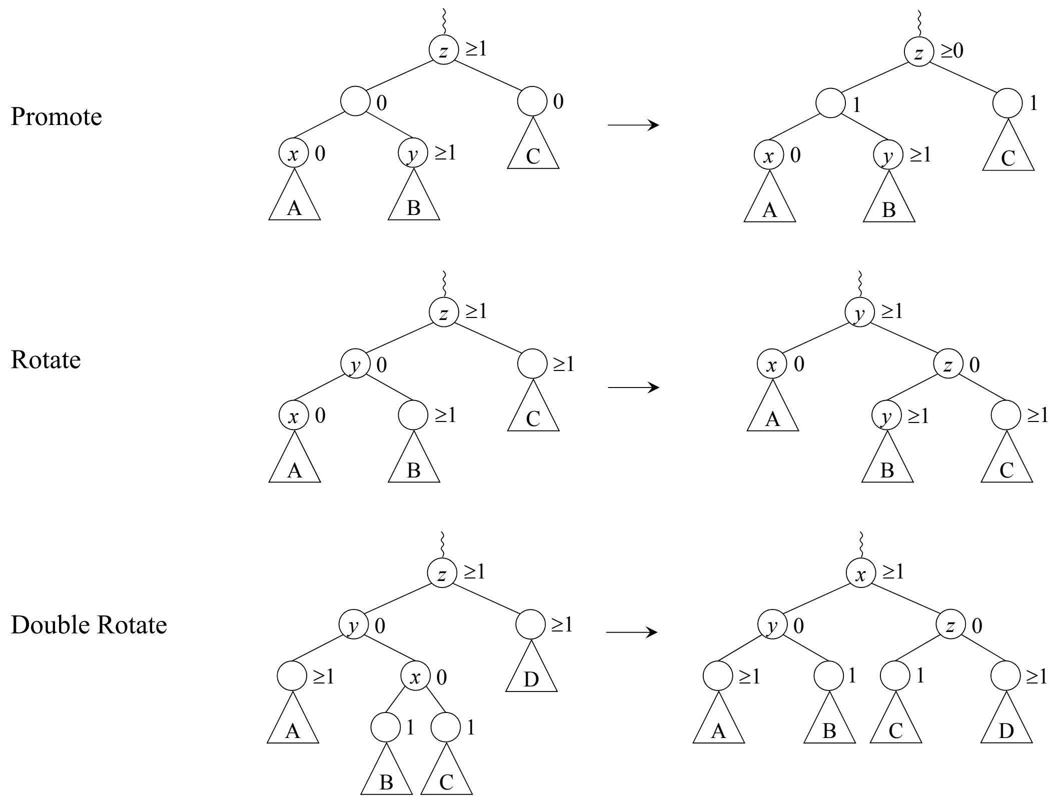


Figure 2.3: Bottom-up rebalancing after an insertion in relaxed red-black trees. Numbers are rank differences. The first case is possibly non-terminating.

We develop a relaxed version of red-black trees in which rebalancing occurs only during insertions, not deletions, with properties like those of ravl trees. To obtain such trees we relax the red-black rank rule to allow arbitrary positive ranks. A *relaxed red-black tree* is a ranked binary tree such that all ranks and rank differences are non-negative and no 0-child has a 0-child as a parent. To insert an item into a relaxed red-black tree using bottom-up rebalancing, follow the search path until reaching a missing node. Replace the missing node by a node x of rank zero containing the item to be inserted. Then rebalance as follows (see Figure 2.3):

Insertion rebalancing:

While x is a 0-child with a non-null grandparent z that is a 0,0-node, repeat the following step:

Promote: Promote z; replace x by z.

Now either the rank rule holds or x is a 0-child whose grandparent z is a $0, i$-node with $i > 0$. In the latter case, proceed as follows. Let y be the parent of x. Do the appropriate one of the following two steps:

Rotate: If nodes x and y are both left or both right children, rotate at y.

Double Rotate: Otherwise (node x is a left child and y a right child or vice-versa), rotate at x twice.

The *rank* of an insertion step is the rank of x just before the step.

To delete an item, we proceed exactly as in ravl trees: we find the node containing the item to be deleted and swap this item with its predecessor or successor if it is in a node with two non-null children. Now the item is in a leaf or a node with only one non-null child. If it is in a leaf, we replace the leaf by a missing node; if it is in a node with one non-null child, we replace this node by its non-null child. No nodes change ranks.

Instead of rebalancing bottom-up after an insertion, we can rebalance top-down during the search for the insertion position. If the tree is empty, create a new node of rank zero containing the item to be inserted and make it the root, completing the insertion. Otherwise, initialize t and z to be the root, and promote the root if it is 0,0. This establishes the invariant for the main loop of the algorithm: z is a non-null node that is not a 0,0-node and not a 0-child; t is the parent of z unless z is the root, in which case $t = z$. Repeat the following step until the item is inserted:

Top-down insertion step:

From z, take one step down along the search path, to y. If y is null, replace it by a new node of rank zero containing the item to be inserted. This completes the insertion: the new node may be a 0-child, but z is not. In the remaining cases, y is non-null. If y is a 0,0-node, promote y, replace y by t, and replace z by the child of y along the search path; this child cannot be null since it has non-negative rank. This completes the step. If y is a 1-child that is not a 0,0-node, replace t by z and z by y, completing the step. In the remaining cases y is a 0-child and hence a 1,1-node. From y take one step down the search path, to x. If x is null, replace x by a new node of rank zero containing the item to be inserted; if the new node is a 0-child, do a single or double rotate step to restore the rank rule. This completes the insertion. The remaining possibility is x non-null. If x is not a 0,0-node, replace y by y and z by z, completing the step. Otherwise (x is a 0,0-node), promote x and do a single or double rotate step to restore the rank rule. If a single rotation is done, replace t by x and z by the child of x along the search path; if a double rotation is done, replace t by whichever of y and z is along the search path from x after the rotations, and replace z by the child of the new t along the search path. This completes the step.

Each top-down insertion step either finishes the insertion or replaces z by a node of smaller rank, so the number of insertion steps is at most one plus the rank of the root. We define the *rank* of a top-down insertion step to be the rank of y, or zero if y is null. Every such step of positive rank is non-terminal, since if y is a 1,1-node of positive rank, both of its children are non-null. We call a top-down insertion step *rebalancing* if it is terminal or it does at least one promotion or rotation.

We can analyze both bottom-up and top-down rebalancing using the same potential function. To get an amortized constant bound on the number of rebalancing steps, we define the potential of a tree to be twice the number of 0,0-nodes plus the number of $0, i$-nodes with $i > 0$. Deletions do not increase the potential. Insertion of a new node increases the potential by at most one. Each promote step in bottom-up rebalancing and each non-terminal rebalancing step in top-down rebalancing decreases the potential by at least one.

Each single or double rotate step in bottom-up rebalancing, and each terminal insertion step in top-down rebalancing does not change the potential. This we obtain the following theorem:

Theorem 2.5.1. *In a relaxed red-black tree built by m insertions, each with either bottom-up or top-down rebalancing, intermixed with arbitrary deletions, the number of rebalancing steps is at most $2m$.*

To bound the root rank and hence the height of the tree, we use an exponential potential function. We define the potential of a node of rank k to be 2^k if it is a $0, i$-node with $i > 0$, 2^{k+1} if it is a 0,0 node, and zero otherwise; we define the potential of a tree to be the sum of the potentials of its nodes. Insertion of a new node increases the potential by at most one. No insertion step, whether bottom-up or top-down, can increase the potential. If the root has rank k and it is promoted, the potential decreases by 2^{k+1}.

Theorem 2.5.2. *In a relaxed red-black tree built by m insertions, each with either bottom-up or top-down rebalancing, intermixed with arbitrary deletions, the rank of the root is at most $\lg(m+1) - 1$, and the height of the root is at most $2\lg(m+1)$.*

Proof. The only increase in potential is caused by insertions of new nodes and totals at most $m - 1$. (An insertion into an empty tree does not increase its potential.) If the root has rank k, there must have been a promotion of the root for each rank between 0 and $k - 1$ inclusive, decreasing the potential by $2^{k+1} - 2$. Thus $2^{k+1} \leq m + 1$, which implies $k \leq \lg(m + 1) - 1$. The rank rule implies that the rank of the grandparent of a node is greater than the rank of the node, which implies that the height of the root is at most $2\lg(m + 1)$. □

By truncating the exponential potential function, we can show that the number of rebalancing steps of rank k is exponentially small in k.

Theorem 2.5.3. *In a relaxed red-black tree built from an empty tree by m insertions, each with either bottom-up or top-down rebalancing, intermixed with arbitrary deletions, the number of rebalancing steps of rank k is at most $m/2^k$.*

Proof. The theorem is immediate for $k = 0$. Fix $k > 0$. Redefine the exponential potential to be zero for all nodes of rank k or greater. Inserting a new node still increases the potential by at most one, and no insertion rebalancing step can increase it, but a non-terminal bottom-up step of rank k, which must be a promotion, decreases it by 2^k, as does a top-down rebalancing step of rank k. Since each bottom-up step of rank k is preceded by a non-terminal bottom-up step of rank $k - 1$, the theorem follows. □

We conclude this section with a few comments about ravl trees versus relaxed red-black trees. Rebalancing does fewer promotions in the latter than in the former, at least locally. One the other hand, the height bound is smaller by a constant factor for ravl trees with bottom-up rebalancing than for relaxed red-black trees, and by increasing the look-ahead we can also make it smaller for ravl trees with top-down rebalancing. In a ravl tree the height of a node is at most its rank, but in a relaxed red-black tree the height of a node is at most twice its rank plus one. Thus to compare Theorems 2.3.3 and 2.4.3 with 2.5.3, we need to compare ϕ (the base in Theorem 2.3.3) or $\sqrt{2}$ (the base in Theorem 2.4.3) with $\sqrt{2}$ (the square root of the base in Theorem 2.5.3). If rebalancing is bottom-up, or if rebalancing is top-down and the look-ahead is at least 3, the comparison favors ravl trees. Determining which variant of which of these data structures is best under what actual circumstances is a subject for experimental investigation.

2.6 Rebuilding the Tree

As the ratio of the number of deletions to the number of insertions approaches one, the height of a ravl tree or a relaxed red-black tree can become $\omega(\log n)$, although it remains $\mathrm{O}(\log m)$. For many applications this is not a concern, but for those in which it is, we can keep the height $\mathrm{O}(\log n)$ by periodically rebuilding the tree. How to do the rebuilding, and how often, are interesting questions that deserve careful study. Here we offer a simple rebuilding method and some thoughts on how often to rebuild. We discuss the rebuilding of ravl trees; rebuilding of relaxed red-black trees is analogous.

To rebuild the tree, we initialize a new tree to empty. Then we traverse the old tree in symmetric order, deleting each visited node and inserting it into the new tree. Traversing the old tree takes $\mathrm{O}(n)$ time. To facilitate building the new tree, we store the nodes on its right spine in a stack, bottommost node on top. Each insertion into the new tree takes $\mathrm{O}(1)$ amortized time, and rebuilding the entire tree takes $\mathrm{O}(n)$ time. The new tree has height at most $\lg n + 1$: every child is a 1- or 2-child, and the 2-children have parents on the right spine. The new tree also has potential $\mathrm{O}(n)$ for any of the potential functions we have considered.

To decide when to rebuild the tree, we keep track of n and of the rank r of the root. If we are using bottom-up rebalancing, we rebuild the tree whenever $r > \log_\phi n + c$, where c is a small positive constant. Then the rebuilding time is $\mathrm{O}(1/(\phi^c - 1))$ per deletion. The larger we choose c, the smaller the overhead for rebuilding, but the larger the height can become as a function of n. If we allow c to grow as a function of n, we can make the rebuilding time $\mathrm{o}(1)$ per deletion while still maintaining a height bound of $\log_\phi n$ plus a lower-order term. If we are using top-down rebalancing, we rebuild the tree whenever $r > 2\lg n + c$.

We can also make the rebuilding incremental. For example, we can start the rebuilding when the height bound is violated and move two nodes from the old to the new tree after each insertion or deletion. During rebuilding, we do each insertion in the old or new tree as appropriate: such an operation is in the new tree if the key of the new item is at most that of the last item moved, in the old tree otherwise. We store the left spine of the old tree and the right spine of the new tree in stacks, so that the next node to be deleted from the old tree, and its insertion location in the new tree, can be found in $\mathrm{O}(1)$ time. We must update these stacks during insertions and deletions, but this takes $\mathrm{O}(1)$ amortized time per insertion or deletion. If n is the number of items in the old tree when rebuilding starts, the number in the new tree will be between $n/2$ and $2n$ when rebuilding stops.

Whether the tree is rebuilt incrementally or all at once, the tree height is always at most $\log_\phi n + \mathrm{O}(1)$ with bottom-up rebalancing or $2\lg n + \mathrm{O}(1)$ with top-down rebalancing, and Theorems 2.3.1 and 2.3.3, or 2.4.1 and 2.4.3 hold, respectively.

2.7 Good and bad alternatives

In this section we explore the effect of alternative insertion and deletion methods on ravl trees and relaxed red-black trees. Our conclusion, based not on a proof but on consideration of several alternative methods, is that any method for which bounds like ours hold must rebalance on insertion and must store $\Omega(\log\log m)$ bits of balance information. In particular, in a rank-based scheme, insertions and deletions cannot increase node ranks except by rebalancing steps. We now attempt to justify this conclusion.

An alternative way to do deletions in search trees is to do them lazily, via the "tombstone" method: to delete an item, remove it from its node but leave its key, so that search is still possible. If a new item with the same key is later inserted, store it in the node containing its key. The tombstone method avoids the need to swap items between nodes during deletion. In analyzing this method, one can ignore deletions and consider trees built only by insertions. If the tombstone method is applied to AVL trees with bottom-up rebalancing on insertion, Theorem 2.3.2 is immediate, and Theorem 2.3.3 follows from Theorem 6.1 in our paper on wavl trees [80]. Alternatively, one can rebalance top-down on insertion, in which case the results of Section 2.4 hold. If the tombstone method is applied to red-black trees, Theorem 2.5.2 is immediate since it holds for red-black trees, and Theorem 2.5.3 holds as well.

The drawback of the tombstone method is that if the number of deletions approaches the number of insertions, the space required by the tree can become superlinear in the number of items. One can keep the space usage linear by deleting each empty leaf and replacing each empty node with a missing child by its non-missing child [26]. Then every empty node has two non-empty children, and the number of nodes is at most $2n - 1$. This gives a variant of the ravl tree or relaxed red-black tree in which deletion is done without swapping, by deleting the item but not its key if it is in a node with two non-null children, or otherwise deleting the node containing the item and replacing this node by its non-empty child if it has one. This method can produce leaves of arbitrarily large rank as well as arbitrarily large rank differences, just as in the original versions of ravl and relaxed red-black trees.

An even more relaxed way to do rebalancing is to avoid rebalancing during both insertions and deletions but maintain a separate thread (or threads, in a multi-threaded implementation) that does rebalancing. This idea has been studied by several authors (e.g.,, [81, 95, 128, 129]), who call their data structures "relaxed balanced trees" of various kinds. These papers derive bounds on the total number of rebalancing steps that must be done to restore the balance of the tree, as a function of the number of updates (insertions and deletions). The problem with this approach is that a sequence of n insertions of items in increasing order will produce a linear tree, in which searches will be extremely expensive until the balancing process has a chance to do its work. Our results, on the other hand, offer a way to completely avoid rebalancing on deletion while still maintaining, *at all times*, a logarithmic bound on search time. A promising idea is to combine rebalancing on insertion with a rebalancing thread that gradually repairs the incremental damage done by deletions. Exploring the efficiency of such a method is an interesting direction for future research.

The tombstone method without node deletions produces a data structure in which all nodes have constant rank difference, and hence each node needs to store $\mathrm{O}(1)$ bits of balance information, but the number of nodes is $\Theta(m)$, not $\Theta(n)$. This leaves the question of whether there is a method that avoids rebalancing on deletion while using only n nodes and $\mathrm{O}(1)$ balance bits per node. We show that several approaches to this question fail, including the approach used by the database provider in the episode mentioned in the introduction.

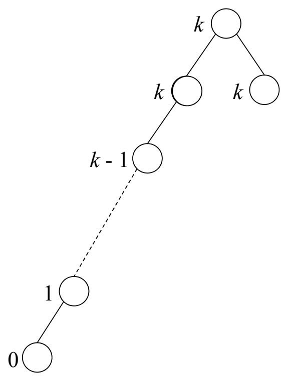


Figure 2.4: Counterexample for an alternative insertion method in relaxed red-black trees that uses one balance bit in every node. Node ranks are shown to the left.

Suppose we store rank differences instead of ranks in the nodes, and we want to keep the rank differences bounded (to 1 or 2 in a ravl tree, 0 or 1 in a relaxed red-black tree). How do we do insertions? The most naïve idea is to avoid computing ranks while walking along the search path, merely inserting each new node with a fixed rank difference. If this difference is positive, we immediately run into the problem that a sequence of insertions of items in sorted order will produce a linear tree, making search times linear. The alternative is to give new nodes a rank difference of zero. But by mixing deletions with insertions, we can still build a linear tree. In the case of a relaxed red-black tree, giving a new node a rank difference of zero is equivalent to coloring it red, which is what the insertion algorithm for standard red-black trees does. Insert three items in sorted order into an initially empty relaxed red-black tree. The third insertion creates a 0-child of a 0-child, causing a rotation and resulting in a tree whose root has two 0-children. Now repeat the following sequence of updates indefinitely: insert an item bigger than all previous ones, delete the two biggest items, and insert two items each bigger than all previous ones. Each sequence of an insertion, two deletions, and two insertions adds a 1-child to the left spine of the tree and produces a root with two red children; the tree consists of the left spine of the root and the right child of the root. (See Figure 2.4.) Thus an intermixed sequence of insertions and deletions can build a tree of linear depth. This is the method that was used by the database

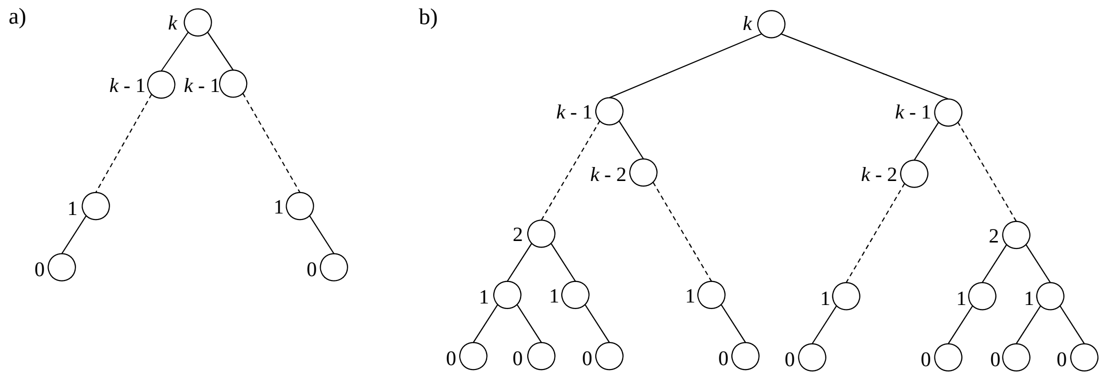


Figure 2.5: Counterexamples for two alternative methods of insertion and deletion in ravl trees that use one balance bit per non-root node. Node ranks are shown to the left.

provider. It is easy to construct a similar counterexample for the variant of ravl trees in which each new leaf has a rank difference of zero.

It is not surprising that making every new node a 0-child should create problems, since doing so can increase our rank-based potential functions by an arbitrarily large amount, destroying our analyses. We can avoid this by maintaining the rank of the tree root. Then we can compute the rank of each node visited during a search by summing rank differences. In a ravl tree, if we give a new node a rank equal to the maximum of zero and two less than the rank of its parent (thereby giving it a rank difference of 0, 1, or 2), then adding such a node increases the potential by at most one, and all our analyses still hold. The equivalent method in a relaxed red-black tree is to give a new node a rank equal to the maximum of zero and one less than the rank of its parent, thereby giving it a rank difference of 0 or 1. Interestingly, the method of giving a new node in a ravl tree a rank equal to the maximum of zero and one less than the rank of its parent (thereby giving it a rank difference of 0 or 1) fails, as the following counterexample shows. (See Figure 2.5.) For arbitrary $k \geq 1$, in $\mathrm{O}(k^2)$ insertions and deletions build a tree T_k of height k consisting of a root of rank k and a left and right spine, with each child having a rank difference of 1 and the two leaves having rank zero, as follows. For $k = 1$, do one insertion into an empty tree followed by one insertion of an item smaller than the one in the root and one insertion of an item larger than the one in the root. For $k > 1$, start with T_{k-1}. Do $2(k-1)$ insertions to give every non-leaf a second child of rank difference one. Then insert an item smaller than all those in the tree followed by an insertion of an item larger than all those in the tree. The first of these will increase the rank of the root; the second will make the root a 1,1-node. Finally, delete all the leaves that now have rank difference two. The result is T_k. Thus $\mathrm{O}(n)$ updates suffice to build a tree of height $\Theta(\sqrt{n})$.

The idea in the previous paragraph allows us to maintain $\mathrm{O}(1)$ rank differences during insertions, but what about during deletions? Consider ravl trees. Suppose that we leave insertion unmodified (so that all new nodes get a rank of zero), but we modify deletion so that when a node with one child is deleted, its child (which replaces it) gets a rank difference of two. This delays the problem illustrated by the counterexample in the previous paragraph but does not avoid it, as the following counterexample shows. (See Figure 2.5.) For arbitrary $k \geq 1$, in $\mathrm{O}(k^3)$ insertions and deletions build a tree T'_k of height k consisting of a root of rank k in which every child is a 1-child, every leaf has rank zero, every node on the left and right spines has two children, and the other non-leaves have one child. For $k = 1$, build $T'_1 = T_1$. For $k > 1$, start with T'_{k-1}. Insert an item less than all items in the tree and an item greater than all items in the tree. This increases the length of both spines by one, promotes all the items on the old spines including the root, and leaves the non-spine children of nodes on the spine with rank difference 2. To each of the leaves of rank zero and rank difference two, add a child via an insertion. This promotes the parent and results in a path of two nodes, each of rank difference one. For each of the remaining 2-children, proceed as follows. Delete the 2-child, replacing it by its only child, which increases in rank by one and becomes a 2-child. The leaf at the bottom of the path descending from this node now has rank one. Do two insertions to give this bottom node two 1-children. Delete the new 2-child, increasing the rank of the two new leaves to one. Choose one of these new leaves and do two insertions to give it two 1-children. Continue in this way until every node on the path down from the spine has two 1-children, and the topmost node on the path is a 2-child. Do one more insertion to add a child to one of the rank-0 leaves at the bottom of the path. This promotes every node along the path, creating a path of 1-children all the way to the spine. Now delete all the 2-children of nodes on this path. Repeating this construction for every 2-child of a node on the left or right spine produces T'_k. The number of updates to build T'_k from T'_{k-1} is $\mathrm{O}(k^2)$, so the total number of updates to build T'_k from an empty tree is $\mathrm{O}(k^3)$. Thus $\mathrm{O}(n)$ updates suffice to build a tree of height $\Theta(n^{1/3})$. Changing the insertions so that they add nodes of non-negative rank but rank difference 0, 1, or 2 does not affect this counterexample, since all insertions add nodes of rank 0. The counterexample also works if deletions use the tombstone method, since all deletions are of leaves or nodes with one child. A similar counterexample exists for relaxed red-black trees in which deletions are modified to keep all rank differences 0 or 1.

The counterexample in the previous paragraph suggests (but does not prove) that keeping the height logarithmically bounded using $\mathrm{O}(1)$ balance bits per node requires a deletion method that does not increase any ranks. We can obtain such a method by making sure that only leaves are deleted. This gives a valid method, but it seems to require arbitrarily long sequences of item swaps during deletions. There are three cases of deletion. To delete an item e, if e is in a leaf, merely delete the leaf. If e is in a node with a right child, swap e with the first item in the right subtree of the node containing e, and repeat this step until e is in a leaf; then delete the leaf. If e is in a node with a left child but no right child, proceed symmetrically: swap e with the last item in the left subtree of the node containing e, and repeat this step until e is in a leaf; then delete the leaf. The time for a deletion is $\mathrm{O}(h + 1)$. If we use this deletion method and modify insertions so that a leaf added by a deletion has a rank equal to the maximum of zero and the rank of its parent minus two, we obtain a variant of ravl trees in which every node has rank difference 1 or 2 and the bounds we have derived hold. To represent ranks we need to store the rank of the root plus one bit per node. The same idea applies to relaxed red-black trees. We have no better bound on the number of swaps per deletion than $\mathrm{O}(\log m)$. Whether this can be reduced to $\mathrm{O}(1)$ amortized per deletion via, for example, a periodic rebuilding scheme is a question we leave for further study.

2.8 To Rebalance on Deletion or Not?

Let us compare ravl trees and relaxed red-black trees to standard kinds of balanced trees. Deletion is much simpler in the former than in the latter. The price we pay for this simplicity is that the height bound is logarithmic in the number of insertions rather than the number of nodes, and each node needs to store $\Theta(\log \log m)$ bits of balance information rather than $\mathrm{O}(1)$. Rebalancing in ravl trees and relaxed red-black trees affects node exponentially infrequently in their height. The same is true in some kinds of standard balanced trees, including wavl trees and red-black trees [80, 156], but the bounds are not as good, because deletion rebalancing interferes with insertion rebalancing. Using periodic rebuilding as discussed in Section 2.6, we can reduce the height bound in ravl and relaxed red-black trees to logarithmic in the number of nodes. Using the multi-swapping deletion method discussed in Section 2.7, we can reduce the amount of balance information per node to $\mathrm{O}(1)$ in ravl trees and relaxed red-black trees, as long as we store the rank of the root as well. Storing the ranks explicitly, however, is only a small space penalty, and it reduces the context needed in rebalancing steps, so it seems a small price to pay. Indeed, at least some authors [13**?**] have advocated storing ranks in standard balanced trees, since it simplifies rebalancing. Doing this eliminates the space advantage of standard balanced trees. Our tentative conclusion is that our theoretical results favor ravl trees and relaxed red-black trees over their standard cousins.

We have also done some preliminary experiments in which we compared ravl trees and relaxed red-black trees (without periodic rebuilding) to standard red-black trees and wavl trees, on typical input sequences. Our results show that ravl trees and relaxed red-black trees perform significantly fewer rotations and balance information updates than the other

Test	Red-black trees				Wavl trees			
	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen
1. Random	26.44	116.07	10.47	15.63	29.55	133.74	10.39	15.09
2. Queue	50.32	285.13	11.38	22.50	50.33	184.53	11.20	14.00
3. Working set	41.71	185.35	10.51	16.18	43.69	159.69	10.45	15.35
4. Static Zipf	25.24	112.86	10.41	15.46	28.27	130.93	10.34	15.05
5. Dynamic Zipf	23.18	103.47	10.48	15.66	26.04	125.99	10.40	15.16

Test	Relaxed red-black trees				Ravl trees			
	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen	# rots $\times 10^6$	# bals $\times 10^6$	avg plen	max plen
1. Random	11.45	38.91	11.30	18.90	14.32	80.61	11.11	16.75
2. Queue	33.56	67.10	11.94	23.50	33.55	134.22	11.38	14.00
3. Working set	22.38	50.25	11.61	19.36	28.00	119.92	11.20	16.64
4. Static Zipf	10.78	38.47	12.73	24.69	13.48	78.03	11.12	17.68
5. Dynamic Zipf	10.24	37.83	11.36	18.86	12.66	74.28	11.11	16.84

Table 2.1: Performance comparison of red-black, wavl, relaxed red-black, and ravl trees on typical input sequences (rots = rotations, bals = balance information updates, avg plen = average path length, max plen = maximum path length).

trees, at the cost of slightly greater average and maximum path lengths. All balanced tree implementations were written in C; all reported quantities are machine-independent.

We generated five tree operation sequences, each performing a total of 2^{26} operations on a tree of size $n = 2^{13}$. To isolate the effect of rebalancing, only insertions and deletions were performed; the expected cost of interspersed accesses can be inferred from the average and maximum path lengths of the tree after each operation. Table 2.1 summarizes our results; the average and maximum path lengths reported are the average values over all operations. The first, fourth, and fifth operation sequences perform insertions and deletions on randomly selected items, chosen uniformly at random in the first sequence and according to a Zipf distribution [**?** **?**] with rank exponent $\alpha = 0.9346$ in the fourth and fifth sequences. (This value of α is based on a classic measurement study of the number of unique visitors seen by America Online on December 1, 1997 [**?**].) The fifth sequence simulates a dynamic Zipf distribution by randomly selecting an item and promoting it to the most popular rank after each operation (this simulates the "flash crowd" or "slashdot" effect often seen in websites). The second operation sequence simulates a queue by inserting the items in order and repeatedly deleting the smallest item in the tree and inserting an item larger than all other items in the tree. The third operation sequence randomly selects an item and inserts or deletes the $\lg n$ items centered around this item in symmetric order.

The results in Table 2.1 show that ravl trees performed significantly fewer rotations and balance information updates—over 42% and 35% fewer, respectively, on average—than red-black trees and wavl trees on the tested sequences. Similarly, relaxed red-black trees performed over 51% and 68% fewer rotations and balance information updates, respectively, on average. The price for this improvement is a slight increase in the average and maximum path length of the resulting trees: under 5.6% and 4.3% greater, respectively, for ravl trees, and under 11.3% and 33.4% greater, respectively, for relaxed red-black trees, on average. Wavl trees performed more rotations and balance information updates than red-black trees, but maintained better average and maximum path lengths. The same comparison holds for ravl trees and relaxed red-black trees.

We plan to conduct more thorough experiments on these and other balanced tree implementations, such as left-leaning red-black trees [23, 151]. In particular, we are investigating the performance of the trees on worst-case sequences, for which periodic rebuilding in ravl trees may be required to provide competitive performance.

2.9 Remarks

We have shown that one can obtain logarithmic worst-case search time in binary search trees that are rebalanced only after insertions, not after deletions. The resulting data structures are simpler than standard balanced search trees, and are preferred by database providers. Our results seem to require either that $\Theta(\log \log m)$ balance bits be stored per node, or that deletion be modified to delete only leaves, which seems to require cascaded swapping of items. Whether this can be proved or disproved is an open question. Also open is the best way to do incremental rebuilding to overcome the cumulative effect of deletion without rebalancing. On the experimental side, it would be valuable to do a systematic evaluation of the practical performance of ravl trees and relaxed red-black trees as compared to red-black trees, wavl trees, and other standard kinds of trees. Ravl trees and relaxed red-black trees combine simplicity with efficiency and may well be very useful in practice.

Chapter 3

Prophecy: A Scalable Solution for Independent Byzantine Failures

Replication techniques are now the norm in Internet services, in order to achieve both reliability and scalability. For example, this is the reason the database provider in Chapter 2 replicated its database system. However, leveraging active agreement to *mask* failures, whether to handle fail-stop behavior [**?**] using protocols like Paxos [105, 130], or arbitrary Byzantine failures [107] using a BFT protocol [31], is not yet widely used. There is some movement in this direction from industry—such as Google's Chubby [28] and Yahoo!'s Zookeeper [179] coordination services, and Google's globally-distributed Spanner database, all based on Paxos [105, 130]—but these services only mask benign faults, and most are used to manage infrastructure, not customer-facing services.

And yet non-fail-stop failures in customer-facing services continue to occur, much to the chagrin and concern of service providers. We listed several examples of real-world Byzantine failures in the introduction. Some of these failures were independent in nature [11, 60, 167], and thus using a BFT protocol could have prevented them. (Recall that BFT protocols assume a fixed bound on the number of simultaneous failures, which is a valid assumption when failures are independent.) Other failures [71, 182], including the outage that befell the database provider, were correlated in nature, affecting many (or all) servers simultaneously. For these failures, simply using a BFT protocol would not have helped. We would also need a way to make the failures independent, such as by running distinct implementations of the service at different replicas, a form of N-version programming [17]. In fact, such an approach could have been employed by the database provider to prevent its outage. Databases and file systems are ideal candidates for opportunistic N-version programming [32], because they expose a common interface (e.g., ODBC [68] and NFS [127], respectively) and are available in a variety of off-the-shelf implementations. In fact, several BFT systems [32, 67, 170] use off-the-shelf implementations in this way, with little or no modification.

While such a BFT database system could have prevented the outage, it would come at a cost to scalability. Namely, the system would not support load balancing read requests to individual replicas, because an individual replica could return an incorrect (possibly arbitrary) result. In general, while extensive prior work on improving performance results of BFT protocols [1, 10, 31, 32, 43, 73, 83, 100, 101, 170, 176, 178, 180] has reduced their latency to that of unreplicated reads to individual servers [37, 73, 100, 176], the throughput of these systems falls far short. This is simple math: a minimum of four replicas [31] (or sometimes even six [1]) are required to tolerate one faulty replica, and at least three must execute each request. More generally, $3f + 1$ replicas are needed to tolerate f faulty replicas, and $2f+1$ must execute each request. For datacenters in the (tens of) thousands of servers, requiring four times as many servers for the same throughput may be a non-starter. Even services that already replicate their data, such as the Google File System [69] and the database provider in Chapter 2, would see their throughput drop significantly when using BFT agreement.

But if the replication cost of BFT is provably necessary [25], something has to give. One might view our work as a thought experiment that explores the potential benefit of placing a small amount of trusted software or hardware in front of a replicated service.

After all, wide-area client access to an Internet service is typically mediated by some middlebox, which is then at least trusted to provide access to the service. Further, a small and simple trusted component may be less vulnerable to problems such as misconfigurations or Heisenbugs. And by treating the back-end service as an abstract entity that exposes a limited interface, this simple device may be able to interact with both complex and varied services. Our implementation of such a device has less than 3000 lines of code.

Barring such a solution, most system designers opt either for cheaper techniques (to avoid the costs of state machine replication) or more flexible techniques (to ensure service availability under heavy failures or partitions). The design philosophies of Amazon's Dynamo [46], GFS [69], and other systems [52, 59, 169] embrace this perspective, providing only eventually-consistent storage. On the other hand, the tension between these competing goals persists, with some systems in industry re-introducing stronger consistency properties. Examples include timeline consistency in Yahoo!'s PNUTS [41], per-user cache invalidation on Facebook [53], and linearizability in Google's Spanner database [42]. Nevertheless, we are unaware of any major use of *agreement* at the front-tier of customer-facing services. In this chapter, we challenge the assumption that the tradeoff between strong consistency and cost in these services is fundamental.

This chapter presents Prophecy, a system that lowers the performance overhead of fault-tolerant agreement for customer-facing Internet services, at the cost of slightly weakening its consistency guarantees. At Prophecy's core is a trusted *sketcher* component that mediates client access to a service replica group. The sketcher maintains a compact history table of observed request/response pairs; this history allows it to perform fast, load-balanced reads when state transitions do not occur (that is, when the current response is identical to that seen in the past) and slow, replicated reads otherwise (when agreement is required). The sketcher is a flexible abstraction that can *interface with any replica group*, provided it exposes a limited set of defined functionality. This chapter, however, largely discusses Prophecy's use with BFT replica groups.

Property	BFT	D-Prophecy	Prophecy
Trusted components	No	No	Yes
Modified clients	Yes	Yes	No
Session length	Long	Long	Short, long
Load-balanced reads	No	Yes	Yes
Consistency	Linearized	Delay-once	Delay-once

Table 3.1: Comparison of a traditional BFT system, D-Prophecy, and Prophecy.

Results and outline

When used with BFT replica groups that guarantee linearizability [85], Prophecy significantly increases throughput through its use of fast, load-balanced reads. However, it relaxes the consistency properties to what we term *delay-once* semantics. We introduce the notion of delay-once consistency and define it formally. Intuitively, it implies that faulty nodes can at worst return slightly stale (not arbitrary) data.

We also derive a distributed variant of Prophecy, called D-Prophecy, that similarly improves the throughput of traditional fault-tolerant systems. D-Prophecy achieves the same delay-once consistency but *without any trusted components*.

We have implemented both variants of Prophecy and applied them to BFT replica groups. We evaluate their performance on realistic workloads, not just null workloads as typically done in the literature. Prophecy adds negligible latency compared to standard load balancing, while providing an almost linear-fold increase in throughput. We also show how to scale out Prophecy to support large replica groups or many replica groups. In the latter case, we assume that each replica group satisfies the requirement that fewer than $1/3$ of its nodes are faulty. Depending on the scenario, this assumption may not be practical: we show how to remove it in Chapter 4.

Since Prophecy optimizes read requests, it is most effective in read-mostly workloads where state transitions are rare. We conduct a measurement study of the Alexa top-25 websites and show that over 90% of requests are for mostly static data. We also characterize the dynamism in the data.

Table 3 summarizes the different properties of a traditional BFT system, D-Prophecy, and Prophecy. The remainder of this chapter is organized as follows. In §3.1 we motivate the design of D-Prophecy and Prophecy, and we describe this design in §3.2. In §3.3 we define delay-once consistency and analyze Prophecy's implementation of this consistency model. In §3.4 we discuss extensions to the basic system model that consider scale and complex component topologies. We detail our prototype implementation in §3.5 and describe our system evaluation in §3.6. In §3.7 we present our measurement study. We review related work in §5 and conclude in §5.6.

3.1 Motivating Prophecy's Design

One might rightfully ask whether Prophecy makes unfair claims, given that it achieves performance and scalability gains at the cost of additional trust assumptions compared to traditional fault-tolerant systems. This section motivates our design through the lens of BFT systems, in two steps. First, we improve the performance of BFT systems on realistic workloads by introducing a cache at each replica server. By optimizing the use of this cache, we derive a distributed variant of Prophecy that does not rely on any trusted components. Then, we apply this design to customer-facing Internet services, and show that the constraints of these services are best met by a shared, trusted cache that proxies client access to the service replica group. The resulting system is Prophecy.

In our discussion, we differentiate between *write requests*, or those that modify service state, and *read requests*, or those that simply access state.

3.1.1 Traditional BFT Services and Real Workloads

A common pitfall of BFT systems is that they are evaluated on null workloads. Not only are these workloads unrealistic, but they also misrepresent the performance overheads of the system. Our evaluation in §3.6 shows that the cost of executing a non-null read request in the PBFT system [31] dominates the cost of agreeing on the ordering of the request, even when the request is served entirely from main memory. Thus the PBFT read optimization, which optimistically avoids agreement on read requests, offers little or no benefit for most realistic workloads. Improving the performance of read requests requires optimizing the *execution* of the reads themselves.

Unlike write requests, which modify service state and hence must be executed at each replica server, read requests can benefit from causality tracking. For example, if there are no causally-dependent writes between two identical reads, a replica server could simply cache the response of the first read and avoid the second read altogether.[1] However, this requires (1) knowledge of the causal dependencies of all write requests, and (2) a response cache of all prior reads at each replica server. The first requirement is unrealistic for many applications: a single write may modify the service state in complex ways. Even if we address this problem by invalidating the entire response cache upon receiving any write, the space needed by such a cache could be prohibitive: a cache of Facebook's $60+$ billion images on April 30, 2009 [126], assuming a scant 1% working-set size, would occupy approximately 15TB of memory. Thus, the second requirement is also unrealistic.

Instead of caching each response r, the replica servers can store a compact, collision-resistant sketch $s(r)$ to enable *cache validation*. That is, when a client issues a read request for r, only one replica server executes the read and replies with r, while the remaining replica servers reply with $s(r)$ from their caches. The client accepts r only if the replica group agrees on $s(r)$ and if $s(r)$ validates r. Thus, even if the replica that returns r is faulty, it cannot make the client accept arbitrary data; in the worst case, it causes the client to accept a stale version of r. Therefore we only need to ask one replica to execute the read, effectively implementing what we call a *fast read*. Fast reads drastically improve the throughput of read requests and can be load-balanced across the replica group to avoid repeated stale results. The replica servers maintain a fresh cache by updating it during

[1]Other causality-based optimizations, such as client-side speculation [176] or server-side concurrent execution [101] are also possible, but are complementary to any cache-based optimizations.

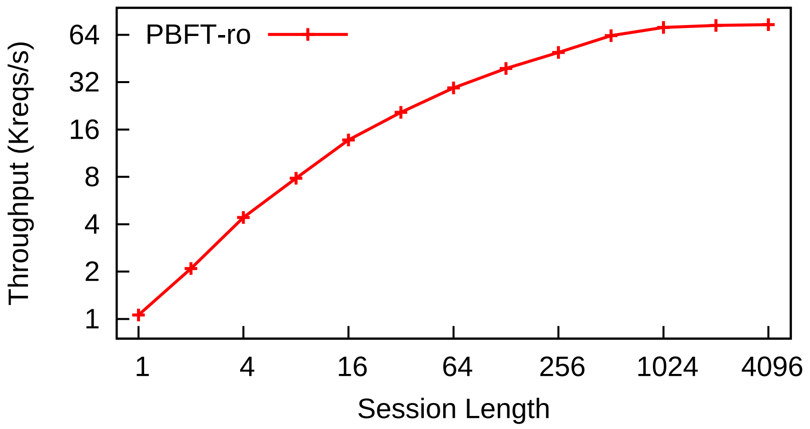


Figure 3.1: PBFT's throughput in the thousands of requests per second for null requests in sessions of varying length. Note that both axes are log scale.

regular (replicated) reads, which are issued when fast reads fail. Using a compact cache reduces the memory footprint of the Facebook image working set to less than 27GB.

We call the resulting system Distributed Prophecy, or D-Prophecy, and call the consistency semantics it provides *delay-once consistency*.

3.1.2 BFT Internet Services

An oft-overlooked issue with BFT systems, including D-Prophecy, is that they are *implicitly* designed for services with long-running sessions between clients and replica servers (or at least always presented and evaluated as such). Clients establish symmetric session keys with each replica server, although the overhead of doing so is not typically included when calculating system performance. Figure 3.1 shows the throughput of the PBFT implementation as a function of session length, with all relevant optimizations enabled including the read optimization (indicated by 'ro'). As sessions get shorter, throughput is drastically reduced because replicas need to decrypt and verify clients' new session keys. For PBFT sessions consisting of 128 read requests, throughput is half of its maximum, and for sessions consisting of 8 read requests, throughput is one-tenth of its maximum.

The assumption of long-lived sessions breaks down for Internet services, however, which are mostly characterized by *short-lived sessions* and *unmodified clients*. These properties make it impractical for clients to establish per-session keys with each replica. Moreover, depending on clients to perform protocol-specific tasks leads to poor backwards compatibility for legacy clients of Internet services (e.g., web browsers), where cryptographic support is not easily available [4]. Instead, we might turn to using an entity knowledgeable of the BFT protocol to proxy client requests to a service replica group. And since Internet services already rely on the correct operation of local middleboxes (at least with respect to service availability), we extend this reliance by converting the middlebox into a trusted proxy. The trusted proxy interfaces multiple short-lived sessions between clients and itself with a single long-lived session between itself and the replica group, acting as a client in the traditional BFT sense.

When using proxied client access to a D-Prophecy group, there is no need to maintain redundant caches at each replica server: a shared cache at the trusted proxy suffices, and it preserves delay-once consistency. A fast read now mimics the performance of an unreplicated read, as the proxy only asks one replica server for r and validates the response with its (local) copy of $s(r)$. Since the cache is compact, the proxy remains a small and simple trusted component, amenable to verification. We call this system Prophecy, and present its design in §3.2.

3.1.3 Applications

The delay-once semantics of Prophecy imply that faulty nodes can at worst return stale (not arbitrary) data. This semantics is sufficient for a variety of applications. For example, Prophecy would be able to protect against the Facebook and Flickr mishaps mentioned in the introduction, because it would not allow arbitrary data to reach the client. Applications that serve inherently static (write-once) data are also good candidates, because here a “stale” response is as fresh as the latest response. In §3.7 we demonstrate the propensity for static data in today’s most popular websites.

Social networks and “Web 2.0” applications are good candidates for delay-once consistency because they typically do not require all writes to be immediately visible. Consider the following example from Yahoo!’s PNUTS system [41]. A user wants to upload spring-break photos to an online photo-sharing site, but does not want his mother to see them. So, he first removes her from the permitted access list of his database record and then adds the spring-break photos to this record. A consistency model that allows these updates to appear in different orders at different replicas, such as eventual consistency [54], is insufficient: it violates the user’s intention of hiding the photos from his mother. Delay-once consistency only allows stale data to be returned, not data out-of-order: if the photos are visible, then the access control update must have already taken place. Further, once the user has “refreshed” his own page and sees the photos, he is guaranteed that his friends will also see them.

For applications where writes are critical, such as a bank account, delay-once consistency is appropriate because it ensures that writes follow the protocol of the replica group. Although reads may return stale results, they can only do so in a limited way, as we discuss in §3.3. On the other hand, there are some applications for which delay-once consistency is not beneficial, such as those that critically depend on reading the latest data (e.g., a rail signaling service), or those that return non-deterministic content (e.g., a CAPTCHA generator).

3.2 System Design

We first define a sketcher abstraction that lies at the heart of Prophecy. For a more traditional setting, we use this sketcher to design a distributed variant of Prophecy, or D-Prophecy. We then present the design of Prophecy.

3.2.1 The Sketcher

Prophecy and D-Prophecy use a sketcher to improve the performance of read requests to an existing replica group. A *sketcher* maintains a history table of compact, collision-resistant sketches of requests and responses processed by a replica group. Each entry in the history table is of the form $(s(q), s(r))$, where q is a request, r is the response to q, and s is the sketching function used for compactness (s typically makes use of a secure hash function like SHA-1). The sketcher looks up or updates entries in the history table using a standard get/set interface, keyed by $s(q)$. In Prophecy, only read requests and responses are stored in the history table.

The specific use of the sketcher and its interaction with the replica group differs between Prophecy and D-Prophecy. However, both systems require the replica group to support the following request interface:

- $\texttt{RESP} \leftarrow \textit{fast}(\texttt{REQ}\ q)$
- $(\texttt{RESP}\ \ r,\ \texttt{SEQ\_NO}\ \ \sigma) \leftarrow \textit{replicated}(\texttt{REQ}\ q)$

We expect the $fast$ interface to be new for most replica groups. The $replicated$ interface should already exist, but may need to be extended to return sequence numbers. No modifications are made to the replica group beyond what is necessary to support the interfaces, in either system.

3.2.2 D-Prophecy

Figure 3.2 shows the system model of D-Prophecy. Except for the sketcher, all other entities are standard components of a replicated service: clients send requests to (and receive responses from) a service implemented by N replica servers, according to some replication protocol like PBFT. Each replica server is augmented with a sketcher that maintains a history table for read requests. The history table is read by the $fast$ interface and updated by the $replicated$ interface, as follows.

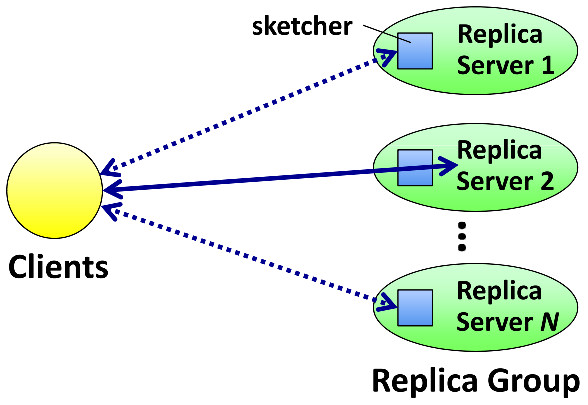


Figure 3.2: Executing a fast read in D-Prophecy. Only one replica server executes the read (bold line); the others return the response sketch in the history table (dashed lines).

A client issues a fast read q by sending it to all replica servers and choosing one of them to execute q and return r. The policy for selecting a replica server is unspecified, but a uniformly random policy has especially useful properties (see §3.3.2). The other replicas use their sketcher to lookup the entry for $s(q)$ and return the corresponding response sketch $s(r)$, or null if the entry does not exist. If the client receives a quorum of non-null response sketches that match the sketch of the actual response, it accepts the response. The quorum size depends on the replication protocol; we give an example below. Otherwise, we say a *transition* has occurred and the client reissues the request as a replicated read. A replicated read is executed according to the protocol of the replica group, with one additional step: all replica servers use their sketcher to update the entry for $s(q)$ with the new value of $s(r)$, before sending a response to the client.

Readers familiar with the PBFT protocol will notice that fast reads in D-Prophecy look very similar to PBFT optimized reads. However, there is a crucial difference: PBFT requires every replica server to execute the read, while D-Prophecy requires only one such execution, performing in-memory lookups of $s(r)$ at the rest. For non-null workloads, this represents a significant performance improvement, as shown in §3.6. On the flip side, each replica server requires additional memory to store its history table, though in practice this overhead is small. The quorum size required for fast reads is identical to the quorum

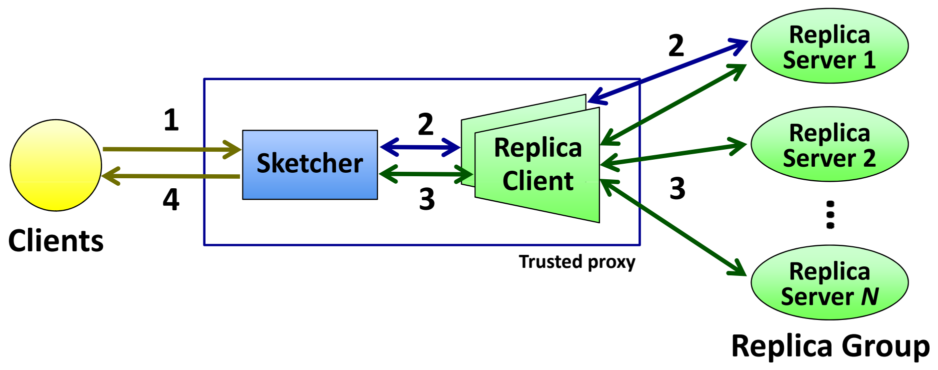


Figure 3.3: Prophecy mediating access to a replica group.

size required for optimized reads: $2f + 1$ responses suffices with some caveats (see §5.1.3 of [29]), and $3f + 1$ always suffices, for a group with $N = 3f + 1$ replica servers.

The architecture of D-Prophecy resembles that of a traditional BFT system: clients establish session keys with the replica servers and participate fully in the replication protocol. As we observed in §3.1.2, this makes D-Prophecy unsuitable for Internet services, with their environment of short-lived sessions and unmodified clients. This motivates the design of Prophecy, discussed next.

3.2.3 Prophecy

Figure 3.3 shows the simplest realization of Prophecy's system model. (We consider extensions to the basic model in §3.4.) There are four types of entities: clients, sketchers, replica clients, and replica servers. Unmodified clients' requests to a service are handled by the sketcher; together with the replica clients, this serves as the trusted proxy described in §3.1.2. The replica clients interact with the service, implemented by a group of N replica servers, according to some replication protocol.

The sketcher issues each request through a replica client; the next subsection details the handling of requests. Functionally, the sketcher in Prophecy plays the same role as the per-replica-server sketchers in D-Prophecy. Architecturally, however, its role is quite different. In Prophecy, a fast read is sent only to the single replica server that executes it, and neither the $fast$ nor $replicated$ interface accesses the history table directly. Thus, the replica group is treated as a black box. Since the sketcher is external to the replica group, writes processed by the group may no longer be visible or discernible to the sketcher; i.e., there may exist an *external write channel*. Since only replica clients interact directly with the replica servers, each replica client can maintain a single, long-lived session with each replica server. Wide-area clients are shielded from any churn in the replica group and are unaware of the replication protocol: the only responses they see are those that have already been accepted by the sketcher.

The type of session used between clients and the sketcher is left open by our design, as it may vary from service to service. For example, services that only allow read or simple write operations (e.g., HTTP GETs and POSTs) may use unauthenticated sessions. A service like Facebook may use authentication only during user login, and use unauthenticated cookie-based sessions after that. Finally, services that store private or protected data, such as an online banking system, may secure sessions at the application level (e.g., using HTTPS). Prophecy's architecture makes it easy to cope with the overhead of client-sketcher authentication, because one can simply add more sketchers if this overhead grows too high (see §3.4). To achieve the same scale-out effect, traditional BFT systems like PBFT and D-Prophecy would need to add entire replica groups.

Handling a Request

The sketcher stores two additional fields with each entry $(s(q), s(r))$ in the history table: the sequence number σ associated with r, and a 2-bit value b indicating whether $s(q)$ is *whitelisted* (always issued as a fast read), *blacklisted* (always issued as a replicated request), or neither (the default). The sketch $s(r)$ is empty for whitelisted or blacklisted requests. Algorithm 1 describes the processing of a request and is illustrated in Figure 3.3 (numbers on the right correspond to the numbered steps in the figure).

Algorithm 1 Processing a request at the sketcher.

```
Receive request $q$ from client                                        (1)
if $q$ is a read request then
    $(s(q), s(r), \sigma, b) \leftarrow$ Lookup $s(q)$ in history table
    if $(s(r) \neq$ null$)$ and $(b \neq$ blacklisted$)$ then
        $r' \leftarrow fast(q)$                                        (2)
        if $(s(r') = s(r))$ or $(b =$ whitelisted$)$ then
            return $r'$ to client                                      (4)
        end if
    end if
    $(r', \sigma') \leftarrow replicated(q)$                           (3)
    if $(s(r) =$ null$)$ or $(\sigma' > \sigma)$ then
        Update history table with $(s(q), s(r'), \sigma', b)$
    end if
else
    $(r', \sigma') \leftarrow replicated(q)$                           (3)
end if
return $r'$ to client                                                  (4)
```

Prophecy requires a sequence number to be returned by $replicated$, as it seeks to issue concurrent requests to the replica group using multiple replica clients. Concurrency allows reads to execute in parallel to improve throughput. Unfortunately, a sketcher that issues requests concurrently has no way of discerning the correct order of replicated reads by itself, i.e., the order they were processed by the replica group. Thus, it relies on the sequence number returned by $replicated$ to ensure that entries in the history table always reflect the latest system state.

The sketcher requires some application-specific knowledge of the format of q and r. This information is used to determine if q is a read or write request, and to discard extraneous or non-deterministic information from q or r while computing $s(q)$ or $s(r)$. For example, in our prototype implementation of Prophecy, an HTTP request is parsed by an HTTP protocol handler to extract the URL and HTTP method of the request; the same handler removes the date/time information from HTTP headers of the response. In practice, the required application-specific knowledge is minimal and limited to parsing protocol headers; the payload of the request or response (e.g., the HTTP body) is treated opaquely by the sketcher.

Whitelisting and blacklisting add flexibility to the handling of requests, but may require additional application-specific knowledge. One use of blacklisting that does not require such knowledge is to dynamically blacklist requests that exhibit a high frequency of transitions (e.g., dynamic content). This allows the sketcher to avoid issuing fast reads that are very likely to fail. (We do not currently implement this optimization.)

3.2.4 Performance

In our analysis and evaluation, the sketcher is able to accommodate all read requests in its history table without evicting any entries. If needed, a replacement policy such as LRU may be used, but this is unlikely: our current implementation can store up to 22 million unique entries using less than 1GB of memory.

The performance savings of a sketcher come from the ability to execute fast, load-balanced reads whose responses match the entries of the history table. Thus, Prophecy and D-Prophecy are most effective in read-mostly workloads. We can estimate the savings by looking at the cost, in terms of per-replica processing time, of executing a read in these systems. Let t be the probability that a state transition occurs in a given workload. Let C_R be the cost of a replicated read and C_r the cost of a fast read (excluding any sketcher processing in the case of D-Prophecy), and let C_{hist} be the cost of computing a sketch and performing a lookup/update in a history table. Below, we calculate the expected cost of a read in Prophecy and D-Prophecy when used with a BFT replica group that uses PBFT's read optimization. For comparison, we include the cost of the unmodified BFT group; here, t' is the probability that a PBFT optimized read fails.

$$\begin{aligned}
&\text{Prophecy:} && [C_r + 2C_{hist}] + [t(NC_R + C_{hist})] \\
&\text{D-Prophecy:} && [C_r + (N-1)C_{hist}] + [t(NC_R + NC_{hist})] \\
&\text{BFT:} && [NC_r] + [t'NC_R]
\end{aligned}$$

The addends on the left and right of each equation show the cost of a fast read and a replicated read, respectively. The equations do not include optimizations that benefit all systems equally, such as separating agreement from execution [180]. Prophecy performs two lookups in the history table during a fast read (one before and one after executing the read), and one update to the history table during a replicated read. D-Prophecy performs a history table lookup at all but one replica server during a fast read, and an update to the history table of each replica server during a replicated read. These equations show that Prophecy operates at maximum throughput when there are no transitions, because only one replica server processes each request, as compared to over $2/3$ of the replica servers in the BFT system (assuming, idealistically, that only a necessary quorum of replica servers execute the optimized read, and the remaining replicas ignore it). Since $C_{hist} \ll C_r$ for non-null workloads—the former involves an in-memory table lookup, the latter an actual read—this is a factor of over $(2/3)N$ improvement. D-Prophecy's savings are similar for the same reason. Although t' may be significantly less than t in practice—given that PBFT optimized reads may still succeed even when a state transition occurs—our evaluation in §3.6 reveals that the benefit of PBFT optimized reads over replicated reads is small for real workloads. Finally, while Prophecy's throughput advantage degrades as t increases, we demonstrate in §3.7 that t is indeed low for popular web services.

3.3 Consistency Properties

Despite their relatively simple designs, the consistency properties of Prophecy and D-Prophecy are only slightly weaker than those of the replica group. In this section, we formalize the notion of *delay-once consistency* introduced in §3.1. Delay-once consistency is a derived consistency model; here, we derive it from linearizability [85], the consistency model of most BFT protocols, and obtain *delay-once linearizability*. Then, we show how Prophecy implements delay-once linearizability.

3.3.1 Delay-once Linearizability

A history of requests and responses executed by a service is linearizable if it is equivalent to a sequential history [103] that respects the irreflexive partial order on requests imposed by their real-time execution [85]. Request X precedes request Y in this order, written $X \prec Y$, if the response of X is received before Y is sent. Suppose one client sends requests (R^a, W^b, R^c) to the service and another client sends requests (W^d, R^e, R^f, W^g), with partial order $\{R^a \prec R^e, W^g \prec R^c\}$. Then a valid linearized history could look like the following:

$$\langle R^a_0, W^d_1, W^b_2, R^e_2, R^f_2, W^g_3, R^c_3 \rangle.$$

The R's and W's represent read and write requests, and subscripts represent the service state reflected in the response to each request (following [76]). In contrast to this history, the following is a valid delay-once linearizable history, though it is not linearizable:

$$\langle R^a_0, W^d_1, W^b_2, R^e_0, R^f_2, W^g_3, R^c_2 \rangle.$$

Requests R^e and R^c have stale responses because they do not reflect the state update caused by sequentially precedent writes (note that the staleness of R^e's response is discernible to the issuing client, whereas the staleness of R^c's response is not). At a high level, a delay-once history looks like a linearized history with reads that reflect the state of prior reads, but not necessarily prior writes. The manner in which reads can be stale is not arbitrary, however. Specifically, a history H is *delay-once linearizable* if the subsequence of write requests in H, denoted by $H|_W$, satisfies linearizability, and if read requests satisfy the following property:

> **Delay-once property**. For each read request R_x in H, let R_y and W_z be the read and write request of maximal order in H such that $R_y \prec R_x$ and $W_z \prec R_x$. Then either $x = y$ or $x = z$.

Delay-once linearizability implies both monotonic read and monotonic write consistency, but not read-after-write consistency. If $\prec_H$ is the partial order of the history H, delay-once linearizability respects $\prec_{H|_W}$ but not $\prec_H$, due to the possible presence of stale reads.

The delay-once property ensures two things: first, reads never reflect state older than that of the latest read (they are only *delayed* to *one* stale state), and second, reads that are updated reflect the latest state immediately. Thus, a system that implements delay-once consistency is *responsive*. To verify if a read in a delay-once consistent history H is stale, one can check the following:

> **Staleness indicator**. Given a read request R_x in H, let W_y be the write request of maximal order in H such that $W_y \prec R_x$. R_x is stale if and only if $x < y$.

The staleness property explains why object-based systems like web services fare particularly well with delay-once consistency. In these systems, state updates to one object are isolated from other objects, so staleness can only occur between writes and reads to the same object.

The above derivation of delay-once consistency is based on linearizability, but derivations from other consistency models are possible. For example, a weaker condition called read-after-write consistency also yields meaningful delay-once semantics.

3.3.2 Prophecy's Consistency Semantics

We now show that Prophecy implements delay-once linearizability when used with a replica group that guarantees linearizability, such as a PBFT replica group. A similar (but simpler) argument shows that D-Prophecy achieves delay-once linearizability, omitted here due to space constraints.

Prophecy inherits the system and network model of the replica group. When used with a PBFT replica group, we assume an asynchronous network between the sketcher and the replica group that may fail to deliver messages, may delay them, duplicate them, or deliver them out-of-order. Replica clients issue requests to the replica group one at a time; requests are retransmitted until they are received. We do not make any assumptions about the organization of the service's state; for example, the service may be a monolithic replicated state machine [104, 150] or a collection of numerous, isolated objects [85]. The sketcher may process requests concurrently. We model this concurrency by allowing the sketcher to issue requests to multiple replica clients simultaneously; the order in which these requests return from replica clients is arbitrary. Updates to service state may not be discernible or visible to the sketcher—i.e., there may exist an external write channel—as discussed in §3.2.3. We show that Prophecy achieves delay-once linearizability despite concurrent requests and external writers.

Our analysis of Prophecy's consistency requires a non-standard approach because it is the sketcher, not the replica servers, that enforces this consistency, and because fast reads are executed by individual replicas. In particular, we introduce the notion of an *accepted history*. Let H_i for $1 \leq i \leq N$ be the history of all write requests executed by replica server i and all fast read requests executed by i that were accepted by the sketcher. Let R_s be the history of all replicated read requests accepted by the sketcher. An accepted history A_i is the union of H_i and R_s, for each replica server i. The position in A_i of each replicated read in R_s is well defined because all reads are accepted at a single location (the sketcher) and all replicated requests are totally ordered by linearizability. We claim that the accepted history A_i is delay-once linearizable.

To see this, observe that replicated requests satisfy linearizability because they follow the protocol of the replica group. The sketcher ensures that replicated reads update the history table according to this order by using the sequence numbers returned by the *replicated* interface. Further, the sketcher only accepts a fast read if it reflects the state of the latest replicated read. Since A_i contains all replicated reads accepted by the sketcher (not just those accepted by i), and since accepted fast reads never reflect new state, it follows that all fast reads in A_i must satisfy the delay-once property. While A_i may not contain all write requests accepted by the replica group (e.g., if i is missing an update), this only affects i's ability to participate in replicated reads, and does not violate delay-once linearizability. Thus, we conclude that A_i is delay-once linearizable.

Limiting staleness via load balancing Stale responses are returned by faulty replica servers or correct replica servers that are out-of-date. We can easily verify if an accepted history contains stale responses by checking the staleness indicator defined in §3.3. To limit the number of stale responses, the *fast* interface dispatches fast reads from all clients uniformly at random over the replica servers.[2] Let g be the fraction of faulty or out-of-date replica servers currently in the replica group. If g is a constant, then g^k, the probability that k consecutive fast reads are sent to these servers, is exponentially decreasing. For BFT protocols, $g < 2/3$ assuming a worst-case scenario where the maximum number of correct nodes are out-of-date. For a replica group of size 4, the probability that $k > 6$ is less than 1.6%.

[2]We assume for simplicity that the random selection is secure, though in practice faulty replica servers may hamper this process. The latter is an interesting problem, but outside the scope of this work.

3.4 Scale and Complex Architectures

This section describes extensions to the basic Prophecy model in order to integrate fault tolerance into larger-scale and more complex environments.

Scaling through multiple sketchers In the basic system model of Prophecy (Figure 3.3), the sketcher is a single bottleneck and point-of-failure. We address this limitation by using multiple sketchers to build a sketching core, as follows. First, we horizontally partition the global history table, based on $s(q)$'s, into non-overlapping regions, e.g., using consistent hashing [91]. We assign each region to a distinct sketcher, which we refer to as *response sketchers*. The partitioning preserves delay-once semantics because only a single sketcher stores the entry for each $s(q)$. Second, we build a two-level sketching system as shown in Figure 3.4, where the first tier of *request sketchers* demultiplex client requests. That is, given a request q, any of a small number of request sketchers computes $s(q)$ and forwards q to the appropriate response sketcher. Using a one-hop distributed hash table (DHT) [91**?**] to manage the partitioning works well, given the network's small, highly-connected nature. The response sketchers (the members of this DHT) issue requests to the replica group(s) and sketch the responses, ultimately returning them to the clients. (Importantly, the replica servers in Figure 3.4 need not be part of a single replica group, but may instead be organized into multiple groups.) The larger number of response sketchers reflects the asymmetric bandwidth requirements of network protocols like HTTP. We evaluate the scaling benefits of multiple response sketchers in §3.6.7.

Handling sketcher failures The sketching core handles failure and recovery of sketchers seamlessly, because it can rely on the join and leave protocol of the underlying DHT. Since request sketchers direct client requests, they maintain the partitioning of the DHT. To preserve delay-once semantics, this partitioning must be kept consistent [28, 179] to avoid sending requests from the same region of the history table to multiple response sketchers.

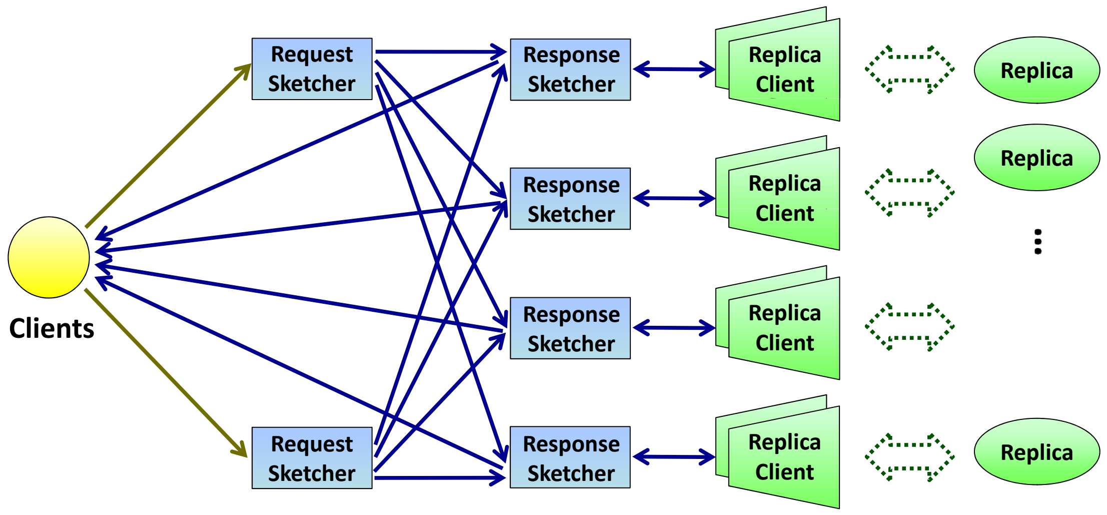


Figure 3.4: Scaling out Prophecy using multiple sketchers.

Prophecy's support for blacklisting simplifies this task, however. In particular, whenever a region of the history table is being relinquished or acquired between response sketchers, we can allow more than one response sketcher to serve requests from the same region provided the entire region is blacklisted (forcing all requests to be replicated). Once the partitioning has stabilized, the new owner of the region can unset the blacklist bit. As a result, membership dynamics can be handled smoothly and simply, at the cost of transient inefficiency but not inconsistency.

Mediating loosely-coupled groups A sketching core can be shared by the multiple, loosely-coupled components that typically comprise a real service. Alternatively, components that operate in parallel can use Prophecy via dedicated sketchers. Components that operate in series, such as multi-tier web services, can use Prophecy prior to each tier. However, applying agreement protocols in series introduces nontrivial consistency issues. We leave this problem to future work.

3.5 Implementation

Our implementation of Prophecy and D-Prophecy is based on PBFT [31]. We used the PBFT codebase given its stable and complete implementation, as well as newer results [10] showing its competitiveness with Zyzzyva and other recent protocols (much more so than was originally indicated [100]). We implemented and compared three proxied systems (Prophecy, proxied PBFT without optimized reads, and proxied PBFT with optimized reads), as well as three non-proxied ("direct") systems (D-Prophecy, PBFT without optimized reads, and PBFT with optimized reads). In our evaluation, we will compare proxied systems only with other proxied systems, and similarly for direct systems, as the architectures and assumptions of the two models are fundamentally different. The proxied systems do not authenticate communication between clients and the sketcher, though they easily can be modified to do so with equivalent overheads.

We implemented a user-space Prophecy sketcher in about 2,000 lines of C++ code using the Tamer asynchronous I/O library [99]. The sketcher forks a process for each core in the machine (8 in our test cluster), and the processes share a single history table via shared memory. The sketcher interacts with PBFT replica clients through the PBFT library. The pool of replica clients available to handle requests is managed as a queue. The sketching function uses a SHA-1 hash [125] over parts of the HTTP header (for requests) and the entire response body (for responses). The proxied PBFT variants share the same code base as the sketcher, but do not perform sketching, issue fast reads, or create or use the history table.

We modified the PBFT library in three ways: to add support for fast reads (about 20 lines of code), to return the sequence numbers (about 20 LOC), and to add support for D-Prophecy (about 100 LOC). Additional modifications enabled the same process to use multiple PBFT clients concurrently (500 LOC), and modified the simple server distributed with PBFT to simulate a webserver and allow "null" writes (500 LOC), as null operations actually have 8-byte payloads in PBFT. We also wrote a PBFT client in about 1000 lines of

C++/Tamer that can be used as a client in direct systems and as a replica client in proxied systems.

3.6 Evaluation

This section quantifies the performance benefits and costs of Prophecy and D-Prophecy, by characterizing their latency and throughput relative to PBFT under various workloads. We explore how the system's throughput characteristics change when we modify a few key variables: the processing time of the request, the size of the response, and the client's session length. Finally, we examine how Prophecy scales with the replica group size.

3.6.1 Experimental Setup

All of our experiments were run in a 25-machine cluster. Each machine has eight 2.3GHz cores and 8GB of memory, and all are connected to a 1Gbps switch.

The proxied systems are labeled Prophecy, pr-PBFT (proxied PBFT), and pr-PBFT-ro (proxied PBFT with the read optimization). The direct systems are labeled D-Prophecy, PBFT, and PBFT-ro (PBFT with the read optimization). Multicast and batching are not used in our experiments, as they do not impact performance when using read optimizations; all other PBFT optimizations are employed. Unless otherwise specified, all experiments used four replica servers, a single sketcher/proxy machine for the proxied systems, and a single client machine. The proxied experiments used 40 replica clients across eight processes at the sketcher/proxy, and had 100 clients establish persistent HTTP connections with the sketcher/proxy. The direct experiments used 40 clients across eight processes. These numbers were sufficient to fully saturate each system without degrading performance. All experiments use infinite-length sessions between communicating entities (except for the one evaluating the effect of session length). Throughput experiments were run for 30-second intervals and throughput was averaged over each second.

System	median	1st	99th
pr-PBFT	433	379	706
pr-PBFT-ro	296	255	544
Prophecy	256	216	286
Prophecy-100	617	553	768
PBFT	286	272	309
PBFT-ro	144	135	168
D-Prophecy	144	129	197
D-Prophecy-100	429	412	574

Table 3.2: Latency in μs for serial null reads.

In some experiments, we report numbers for Prophecy-X or D-Prophecy-X, which signifies that the systems experienced state transitions $X\%$ of the time.

3.6.2 Null Workload

Latency Table 3.2 shows the median and 99th percentile latencies for 100,000 serial null requests sent by a single client. All systems displayed low latencies under $1ms$, although the proxied systems have higher latencies as each request must traverse an extra hop. Prophecy, pr-PBFT-ro, D-Prophecy, and PBFT-ro all avoid the agreement phase during request processing and thus have notably lower latency than their counterparts. Prophecy-100 and D-Prophecy-100 represent a worst-case scenario where every fast read fails and is reissued as a replicated read.

Throughput Figure 3.5 shows the aggregate throughput of the proxied systems for executing null requests. We achieve the desired transition ratio by failing that fraction of fast reads at the sketcher.

Since replica servers can execute null requests cheaply, the sketcher/proxy becomes the system bottleneck in these experiments. Nevertheless, Prophecy achieves 69% higher throughput than pr-PBFT-ro due to its load-balanced fast reads, which require fewer packets to be processed by replica servers. As the transition ratio increases, however, Prophecy's

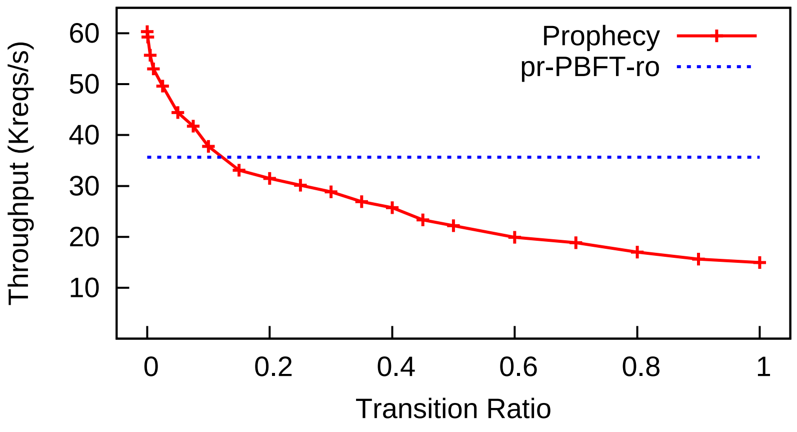


Figure 3.5: Throughput of null reads for proxied systems (Prophecy, pr-PBFT, and pr-PBFT-ro).

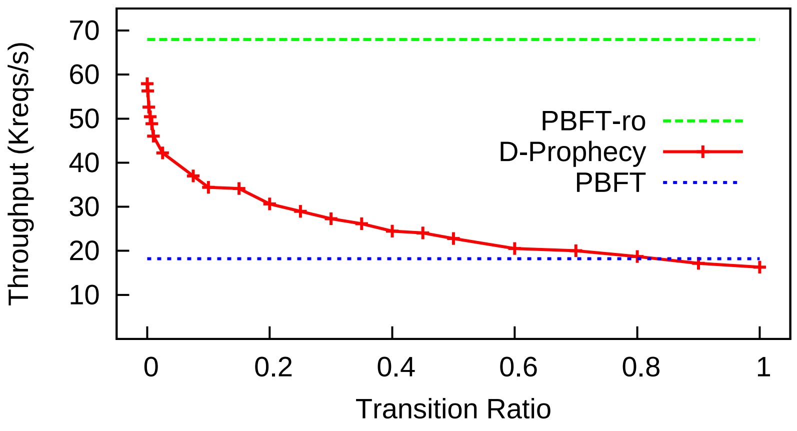


Figure 3.6: Throughput of null reads for direct systems (D-Prophecy, PBFT, and PBFT-ro).

advantage decreases because fewer fast reads match the history table. For example, when transitions occur 15% of the time—a representative ratio from our measurement study in §3.7—Prophecy's throughput is 7% lower than pr-PBFT-ro's.

Figure 3.6 depicts the aggregate throughput of the direct systems. In this experiment, 40 clients across two machines concurrently execute null requests. D-Prophecy's throughput is 15% lower than PBFT-ro's when there are no transitions, and 50% lower when there are 15% transitions. D-Prophecy derives no performance advantage from its fast reads because the optimized reads of PBFT take no processing time, while D-Prophecy pays the overhead for sketching and history table operations.

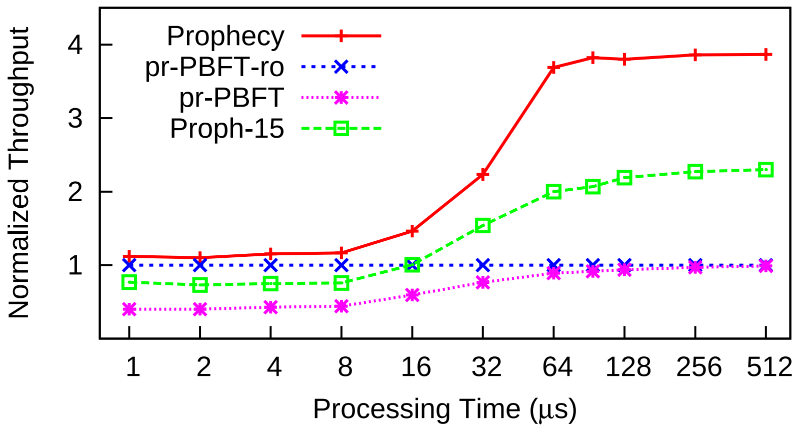


Figure 3.7: Throughput of proxied systems as processing time increases, normalized against pr-PBFT-ro.

3.6.3 Server Processing Time

The previous subsection shows that when requests take almost no time to process, Prophecy improves throughput only by decreasing the number of packets at each replica server, while D-Prophecy fails to achieve better throughput. However, when the replicas perform real work, such as the computation or disk I/O associated with serving a webpage, Prophecy's improvement is more dramatic.

Figures 3.7 and 3.8 demonstrate how varying processing time affects the throughput of proxied systems (normalized against pr-PBFT-ro) and direct systems (normalized against PBFT-ro), respectively. As the processing time increases—implemented using a busy-wait loop—the cost of executing requests begins to dominate the cost of agreeing on their order. This decreases the effectiveness of PBFT's read optimization, as evidenced by the increase in pr-PBFT's throughput relative to pr-PBFT-ro, and similarly between PBFT and PBFT-ro. At the same time, the higher execution costs dramatically increase the effectiveness of load balancing in Prophecy and D-Prophecy. Their throughput approaches 3.9 times the baseline, which is only 2.5% less than the theoretical maximum.

The effectiveness of load-balancing is more pronounced in Prophecy than in D-Prophecy for two main reasons. First, Prophecy's fast reads involve only one replica

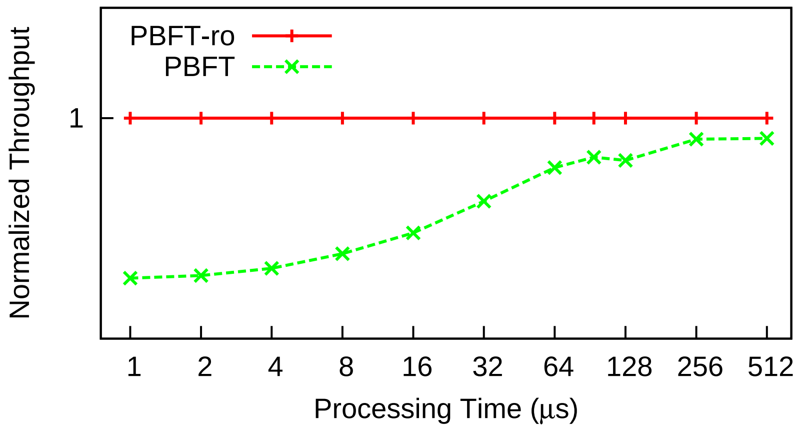


Figure 3.8: Throughput of direct systems as processing time increases, normalized against PBFT-ro.

server, while D-Prophecy's fast reads involve all replicas, even though only a single replica actually executes the request. Second, Prophecy performs sketching and history table operations at the sketcher, whereas D-Prophecy implements such functionality on the replica servers, stealing cycles from normal processing.

3.6.4 Integration with Apache Webserver

We applied Prophecy to a replica group in which each server runs the Apache webserver [14], appropriately modified to return deterministic results. Upon receiving a request, a PBFT server dispatches the request body to Apache via a persistent TCP connection over localhost.

Figure 3.9 shows the aggregate throughput of the proxied systems for serving a 1-byte webpage. When there are no transitions, Prophecy's throughput is 372% that of pr-PBFT-ro. At the representative ratio of 15%, Prophecy's throughput is 205% that of pr-PBFT-ro. The processing time of Apache is enough to dominate all other factors, causing Prophecy's use of fast reads to significantly boost its throughput.

Figure 3.10 shows the throughput of direct systems. With no transitions, D-Prophecy's throughput is 265% that of PBFT-ro, and 141% when there are 15% transitions.

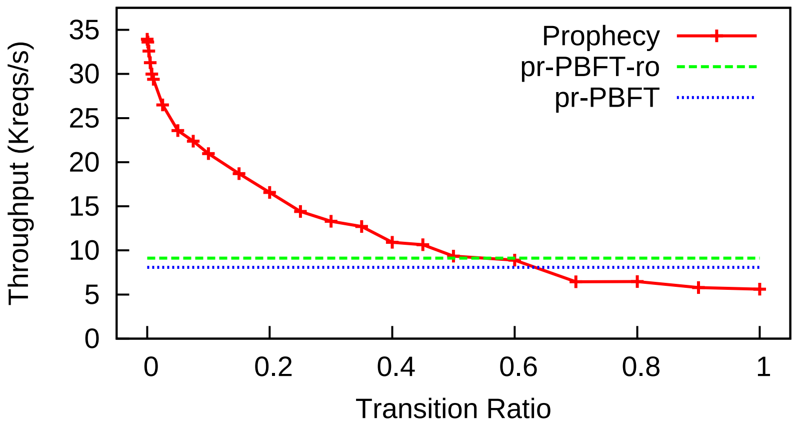


Figure 3.9: Throughput of reads of a 1-byte webpage to Apache webservers for proxied systems.

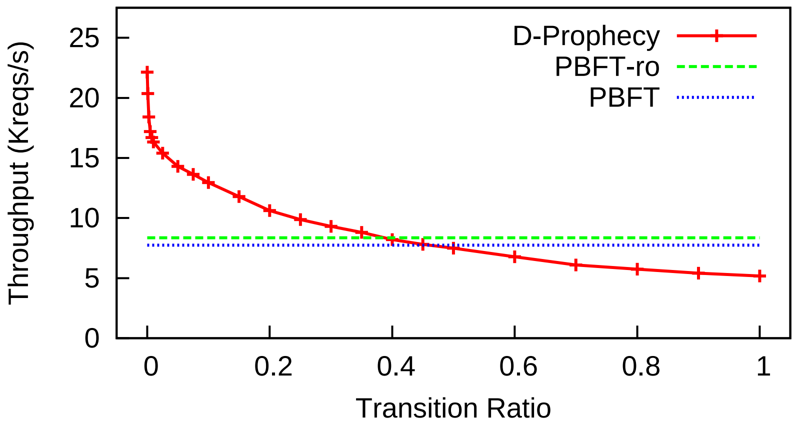


Figure 3.10: Throughput of concurrent reads of a 1-byte webpage to Apache webservers for direct systems.

In these experiments, the local HTTP requests to Apache took an average of $94\mu s$. For the remainder of this section, we use a simulated processing time of $94\mu s$ within replica servers when answering requests.

3.6.5 Response Size

Next, we evaluate the proxied systems' performance when serving webpages of increasing size, as shown by Figure 3.11. As the response size increases, fewer replica clients were needed to maximize throughput. At the same time, Prophecy's throughput advantage

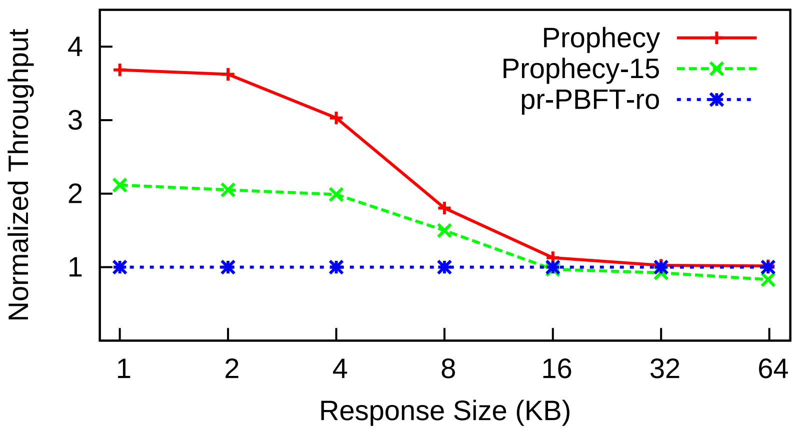


Figure 3.11: Throughput of proxied systems as response size increases, normalized against pr-PBFT-ro.

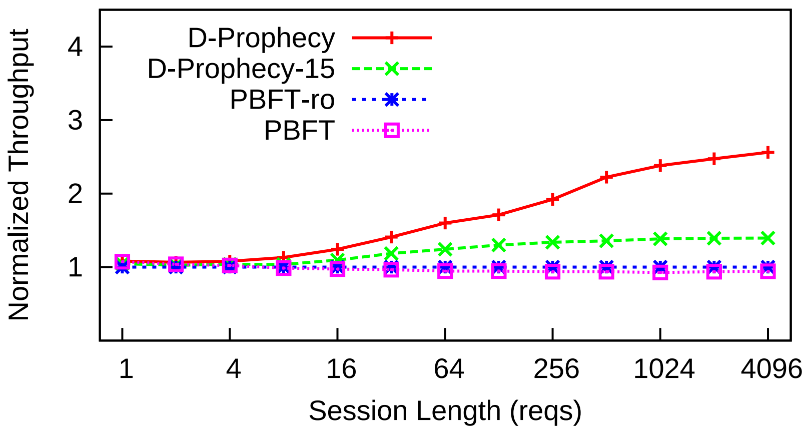


Figure 3.12: Throughput of direct systems as session length increases, normalized against PBFT-ro.

decreases as the response size increases, as the sketcher/proxy becomes the bottleneck in each scenario. Increasing the replica servers' processing time shifts this drop in Prophecy's throughput to the right, as it increases the range of response sizes for which processing time is the dominating cost. Note that we only evaluate the systems up to 64KB responses, because PBFT communicates via UDP, which has a maximum packet size of 64KB.

3.6.6 Session Length

Our experiments with direct systems so far did not account for the cost of establishing authenticated sessions between clients and replica servers. To establish a new session, the client must generate a symmetric key that it encrypts with each replica server's public key, and each replica server must perform a public-key decryption. Given the cost of such operations, the performance of short-lived sessions can be dominated by the overhead of session establishment, as we discussed in §3.1.2.

Figure 3.12 demonstrates the effect of varying session length on the direct systems, in which each request per session returns a 1-byte webpage. We find that the throughput of PBFT and PBFT-ro are indistinguishable for short sessions, but as session length increases, the cost of session establishment is amortized over a larger number of requests, and PBFT-ro gains a slight throughput advantage. Similarly, D-Prophecy achieves its full throughput advantage only when sessions are very long.

We do not evaluate the effect of session lengths in the proxied systems, because they currently do not authenticate communication with the clients. Authentication can easily be incorporated into these systems, however, at a similar cost to Prophecy and pr-PBFT. That said, proxied systems can better scale up the maximum rate of session establishment than direct systems, as we observed in §3.2.3: each additional proxy provides a linear rate increase, while direct systems require an entire new replica group for a similar linear increase.

3.6.7 Scaling Out

Finally, we characterize the scaling behavior of Prophecy and proxied PBFT systems. By increasing the size of their replica groups, PBFT systems gain resilience to a greater number of Byzantine faults (e.g., from one fault per 4 replicas, to four faults per 13 replicas). However, their throughput does not increase, as each replica server must still execute every request. On the other hand, Prophecy's throughput can benefit from larger groups, as it can

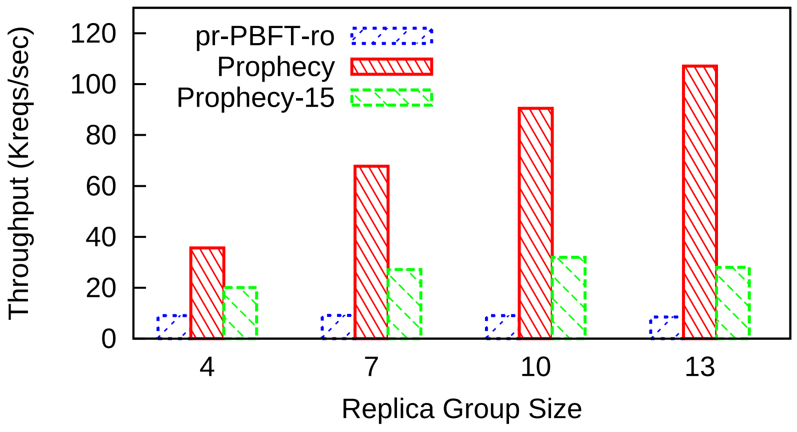


Figure 3.13: Throughput of Prophecy and pr-PBFT-ro with varying replica group sizes.

load balance fast reads over more replica servers. As the sketcher can become a bottleneck in the system at higher read rates, we used two sketchers for a 7-replica group and three sketchers for a 10- and 13-replica group.

Figure 3.13 shows the throughput of proxied systems for increasing group sizes. Prophecy's throughput is 395%, 739%, 1000%, and 1264% that of pr-PBFT-ro, for group sizes of 4, 7, 10, and 13 replicas, respectively. Prophecy does not achieve such a significant throughput improvement when experiencing transitions, however. We see that a 15% transition ratio prevents Prophecy from handling more than $32,000$ req/s, which it achieves with a replica group of size 10. Thus, under moderate transition rates, further increasing the replica group size will only increase fault tolerance, not throughput.

3.7 Measurement Study of Alexa Sites

The performance savings of Prophecy are most pronounced in read-mostly workloads, such as those involving DNS: of the 40K names queried by the ConfiDNS system [135], 95.6% of them returned the same set of IP addresses every time over the course of one day. In web services, it is less clear that transitions are rare, given the pervasiveness of so-called "dynamic content".

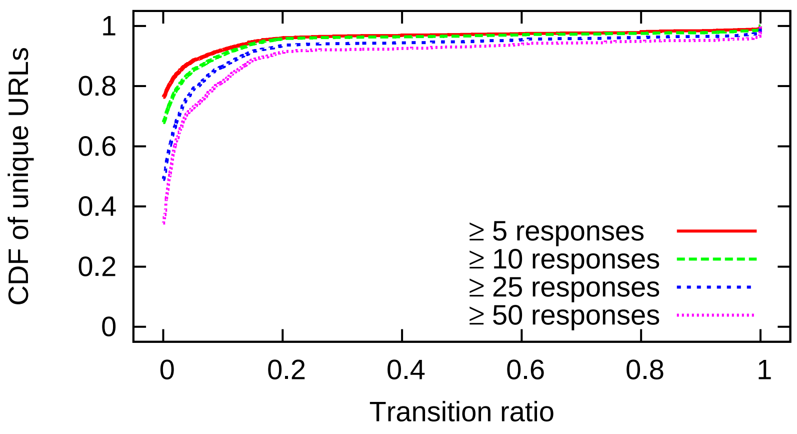


Figure 3.14: A CDF of requests over transition ratios.

To investigate this dynamism, we collected data from the Alexa top 25 websites by scripting a Firefox browser to reload the main page of each site every 20 seconds for 24 hours on Dec. 29, 2008. Among the top sites were `www.youtube.com`, `www.facebook.com`, `www.skyrock.com`, `www.yahoo.co.jp`, and `www.ebay.com`.[3] The browser loads and executes all embedded objects and scripts, including embedded links, JavaScript, and Flash, with caching disabled. We captured all network traffic using the tcpflow utility [51], and then ran our HTTP parser and SHA-1-based sketching algorithm to build a compact history of requests and responses, similar to the real sketcher.

Our measurement results show that transitions are rare in most of the downloaded data. We demonstrate a clear divide between very static and very dynamic data, and use Rabin fingerprinting [138] to characterize the dynamic data. Finally, we isolate the results of individual geographic "sites" using a CIDR prefix database.

3.7.1 Frequency of Transitions

For each unique URL requested during the experiment, we measured the ratio of state transitions over repeated requests. Figure 3.14 shows a CDF of unique URLs at different

[3]While one might argue that BFT agreement is overkill for many of the sites in our study, our examples in the introduction show that Heisenbugs and one-off misconfigurations can lead to embarrassing, high-profile events. Prophecy protects against these mishaps without the performance penalty normally associated with BFT agreement.

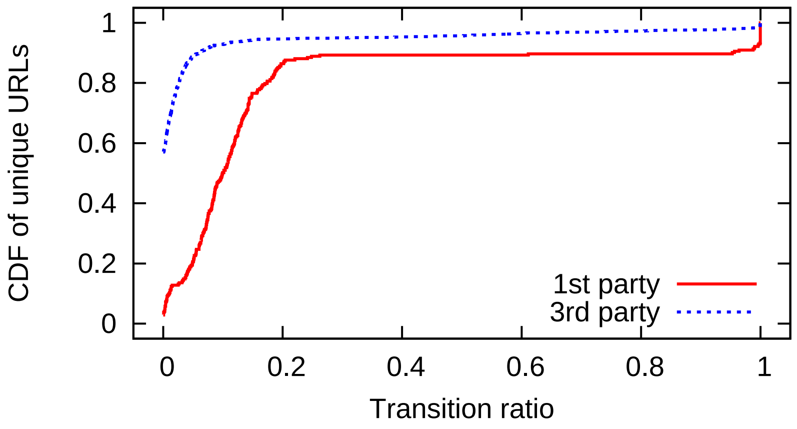


Figure 3.15: A CDF over transition ratios of first-party vs. third-party URLs.

transition ratios. We separately plotted those URLs based on the number of requests sent to each one, given that embedded links generate a variable number of requests to some sites. (Where not specified, the minimum number of requests used is 25.) We see that roughly 50% of all data accessed is purely static, and about 90% of all requests have fewer than 15% state transitions. These numbers confirmed our belief that most dynamic websites are actually dynamic compositions of very static content. The same graph scaled by the average response size of each request yields very similar curves (omitted), suggesting that Figure 3.14 also reflects the total response throughput at each transition ratio.

Figure 3.15 is the same plot as Figure 3.14 but divided into first-party URLs, or those targeted at an Alexa top website, and third-party URLs, or those targeted at other sites (given that first-party sites can embed links to other domains for image hosting, analytics, advertising, etc.). The graph shows that third-party content is much more static than first-party content, and thus third-party content providers like CDNs and advertisers could benefit substantially from Prophecy.

The results in this section are conservative for two reasons. First, they reflect a workload of only three requests per minute per site, when in reality there may be tens or hundreds of thousands of requests per minute. Second, many URLs—though not enough to cause space problems in a real history table—saw only a few requests, but returned identical responses, suggesting that our HTTP parser was conservative in parsing them as unique URLs. An important characteristic of all of the graphs in this section is the relatively flat line across the middle: this suggests that most data is either very static or very dynamic.

3.7.2 Characterizing Dynamic Data

Dynamic data degrades the performance of Prophecy because it causes failed fast reads to be resent as replicated reads. Often, however, the amount of dynamism is small and may even be avoidable. To investigate this, we characterized the dynamism in our data by using Rabin fingerprinting to efficiently compare responses on either side of a transition. We divided each response into chunks of size 1K in expectation [120], or a minimum of 20 chunks for small requests.

Our measurements indicate that 50% of all transitions differ in at least 30% of their chunks, and about 13% differ in all of their chunks. Interestingly, the *edit distance* of these transitions was much smaller: we determined that 43% of all transitions differ by a single contiguous insertion, deletion, or replacement of chunks, while preserving at least half or no more than doubling the number of original chunks. By studying transitions with low edit distance, we can identify sources of dynamism that may be refactorable. For example, a preliminary analysis of around 4,000 of these transitions (selected randomly) revealed that over half of them were caused by load-balancing directives (e.g., a number appended to an image server name) and random identifiers (e.g., client IDs) placed in embedded links or parameters to JavaScript functions. In fact, most of the top-level pages we downloaded, including seemingly static pages like `www.google.com`, were highly dynamic for this exact reason. A more in-depth analysis is slated for future work.

3.7.3 Site-Based Analysis

A "site" represents a physical datacenter or cluster of machines in the same geographic location. A single site may host large services or multiple services. Having demonstrated

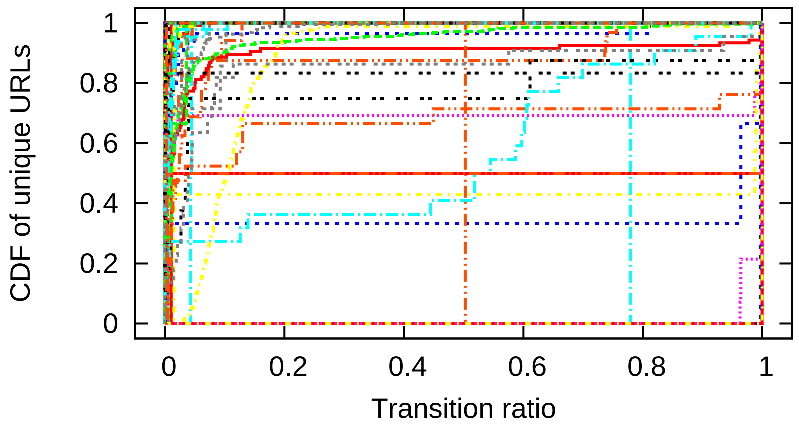


Figure 3.16: A CDF of URLs over transition ratios for all sites for which CIDR data was available.

Prophecy's ability to scale out in such environments, we now study the potential benefit of deploying Prophecy at the sites in our collected data. To organize our data into geographic sites, we used forward and reverse DNS lookups on each requested URL and matched the resulting IP addresses against a CIDR prefix database. (This database, derived from data supplied by Quova [137], included over 2 million distinct prefixes, and is thus significantly finer-grained than those provided by RouteViews [146].) Requests that mapped to the same CIDR prefix were considered to be part of the same site. Figure 3.16 shows an overlay of the transition plots of each site. From the figure, a few sites serve very static data or very dynamic data only, but most sites serve a mix of very static and very dynamic data. All but one site (`view.atdmt.com`) show a clear divide between very static and very dynamic data.

3.8 Related Work

A large body of work has focused on providing strong consistency and availability in distributed systems. In the fail-stop model, state machine replication typically used primary copies and view change algorithms to improve performance and recover from failures [105, 130]. Quorum systems focused on tradeoffs between overlapping read and write sets [70, 84]. These protocols have been extended to malicious settings, both for Byzantine fault-tolerant replicated state machines [31, 32, 107], Byzantine quorum systems [1, 114], or some hybrid of both [43]. Modern approaches have optimized performance via various techniques, including by separating agreement from execution [180], using optimistic server-side speculation on correct operation [100], reducing replication costs by optimizing failure-free operation [178], and allowing concurrent execution of independent operations [101]. Prophecy's history table is motivated by the same assumption as this last approach—namely, that many operations/objects are independent and hence often remain static over time.

Given the perceived cost of achieving strong consistency and a particular desire to provide "always-on" write availability, even in the face of partitions, a number of systems opted for cheaper techniques. Several BFT replicated state machine protocols were designed with weaker consistency semantics, such as BFT2F [111], which weakens linearizability to fork* consistency, and Zeno [159], which weakens linearizability to eventual consistency. Several filesystems were designed in a similar vein, such as SUNDR [110] and systems designed for disconnected [82, 97] or partially-connected operation [134]. BASE [136] explored eventual consistency with high scalability and partition tolerance; the foil to database ACID properties. More recently, highly-scalable storage systems being built out within datacenters have also opted for cheaper consistency techniques, including the Google File System [69], Yahoo!'s PNUTS [41], Amazon's Dynamo [46], Facebook's Cassandra [52], eBay's storage techniques [169], or the popular approach of using Memcached [59] with a backend relational database. These systems take this approach partly because they view stronger consistency properties as infeasible given their performance (throughput) costs; Prophecy argues that this tradeoff is not necessary for read-mostly workloads.

Recently, several works have explored the use of trusted primitives to cope with Byzantine behavior. A2M [34] prevents faulty nodes from lying inconsistently by using a trusted append-only memory primitive; TrInc [108] uses a simple trusted counter primitive to achieve the same goal; and CheapBFT uses the same primitive as TrInc while reducing the number of active replicas during failure-free operation. These systems require only $2f + 1$ replicas to tolerate f faults, and thus they can be used with Prophecy instead of PBFT to reduce replication costs. Chun et al. [33] introduced a lightweight BFT protocol for multi-core single-machine environments that runs a trusted coordinator on one core, similar in philosophy to Prophecy's approach of extending the trusted computing base to include the sketcher.

Prophecy is unique in its application to customer-facing Internet services and its ability to load-balance read requests across a replica group while retaining good consistency semantics. Perhaps closest to Prophecy's semantics is the PNUTS system [41], which supports a load-balanced read primitive that satisfies timeline consistency (all copies of a record share a common timeline and only move forward on that timeline). Delay-once linearizability is strictly stronger than timeline consistency, however, because it does not allow a client to see a copy of a record that is more stale than a copy the client has already seen (whereas timeline consistency does).

There has been some work on using history as a consistency or security metric for particular applications. Aiyer et al. [6, 7] develop k-quorum systems that bound the staleness of a read request to one of the last k written values. Using Prophecy with a k-quorum system may be synergistic: Prophecy's load-balanced reads are less costly than quorum reads, and k-quorum systems can protect against an adversarial scheduler that attempts to hamper Prophecy's load balancing. The Farsite file system [5] uses historical sketches to validate read requests, but requires a lease-based invalidation protocol to keep sketches strongly consistent. The system modifies clients extensively and requires knowledge of causal dependencies (if these constraints are ignored, then D-Prophecy can easily be modified to achieve the same consistency as Farsite). Pretty Good BGP [93] whitelists BGP advertisements whose new route to a prefix includes its previous originating AS, while other routes require manual inspection. ConfiDNS [135] uses both agreement and history to make DNS resolution more robust. It requires results to be static for a number of days and agreed upon by some number of recursive DNS resolvers. Perspectives [175] combines history and agreement in a similar way to verify the self-signed certificates of SSH or SSL hosts on first contact. Prophecy can be viewed as a framework that leverages history and agreement in a general manner.

3.9 Remarks

Prophecy leverages history to improve the throughput of Internet services by expanding the trusted middlebox between clients and a service replica group, while providing a consistency model that is very promising for many applications. D-Prophecy achieves the same benefits for more traditional fault-tolerant services. Our prototype implementations of Prophecy and D-Prophecy easily integrate with PBFT replica groups and are demonstrably useful in scale-out topologies. Performance results show that Prophecy achieves 372% of the throughput of even a read-optimized PBFT group with 4 replicas, and scales linearly as the number of replicas increases. Our evaluation demonstrates the need to consider a variety of workloads, not just null workloads as typically done in the literature. Finally, our measurement study of the Internet's most popular websites demonstrates that a read-mostly workload is applicable to web service scenarios.

Chapter 4

Commensal Cuckoo: Scaling BFT by Composing Many Groups

The Prophecy system in the previous chapter scales the performance of BFT groups on read-mostly workloads. However, as the number of writes in the workload increases, Prophecy's performance benefit diminishes, and beyond a 50-50 mixture of reads and writes, Prophecy offers no benefit at all. The problem is that every write request, and every load-balanced read that fails and must be reissued as a replicated read, is executed by over $2/3$ of the nodes. Moreover, each node maintains a full copy of the system state. Thus to build BFT systems at a large scale, we must partition the system into smaller groups that operate on disjoint (or at least minimally-overlapping) partitions of the system state and functions. Client requests are then routed to the appropriate group.

One approach, as we observed in Section 3.4, is to run multiple instances of Prophecy in parallel, each with its own BFT group. In fact, many systems combine BFT groups in this way, as we will later see. But this assumes that fault occurrences are perfectly and statically distributed across the groups, i.e., every group has fewer than $1/3$ faults by assumption. Such an assumption may not be practical in some scenarios. For example, in an open peer-to-peer system like the Vuze DHT [**?**] (a million-node BitTorrent tracker), any machine connected to the Internet can participate, and nodes may leave and join the system repeatedly.

More concretely, the challenge in a partitioned system subject to a Byzantine adversary is to ensure that every group maintains less than a fixed *local* fraction of faulty nodes (e.g., $1/3$), despite some fixed (smaller) *global* fraction of faulty nodes that are controlled by the adversary. A group fails when its fraction of faulty nodes exceeds the local threshold. The system fails if even a single group fails, since such a group may behave arbitrarily: for example, it may delete its portion of the system state and try to corrupt other groups. We assume that the adversary can coordinate the faulty nodes in an arbitrary, adaptive manner. In particular, it may initiate a *join-leave attack* [45, 49, 162], wherein it repeatedly rejoins certain faulty nodes to the system using fresh identities [49] with the goal of compromising one or more groups. Such attacks pose a significant threat [45, 49, 162] and have been launched successfully on real DHT systems [177]. We call this the *secure group partitioning* problem.

Several systems [44, 96, 102, 141, 144, 173] and some proposals [143, 145] have used multiple BFT groups for scalability, but these solutions rely on a central configuration service to manage system-wide membership, or they maintain this information at every node. Other systems offer better decentralization [5, 30, 88, 89, 109, 118, 121, 160], e.g., using a group for each directory of a file system [5], but they assume that faulty nodes are distributed randomly or even perfectly across groups. Thus, to our knowledge, all systems are vulnerable to join-leave attacks. For example, Rodrigues and Liskov [142, 144] build a DHT by mapping nodes and data to a virtual $[0, 1)$ space using consistent hashing [91], and form BFT groups out of contiguous sets of four nodes, each group tolerating one fault. Even without a join-leave attack, such a perfect distribution of faults cannot be achieved even if faults occur uniformly randomly: a standard balls-in-bins argument shows that some group will have $\omega(1)$ faulty nodes with high probability.

Some theoretical constructions of fault-tolerant DHTs assume that nodes fail randomly (independent of their location) [86, 122], but these assumptions break down in the presence of adaptive join-leave attacks. Recent years have seen constructions that can provably withstand join-leave attacks [18, 19, 20, 55, 149]. Of these, the most promising scheme for DHTs that does not keep the system in a hyperactive state—e.g., by forcing nodes to rejoin the system periodically [18]—is the scheme of Awerbuch and Scheideler [19]. They propose a simple, event-based scheme called the *cuckoo rule*: when a node wishes to join the system, place it at a random location $x \in [0, 1)$ and move, or *cuckoo*, all nodes in a constant-sized interval surrounding x to new random locations in $(0, 1]$. Using this rule, they prove that groups of size $\mathrm{O}(\log n)$ remain correct for any polynomial number of rounds in n, where n is the number of correct nodes in the system. However, as we will show, the constants in their scheme are prohibitively large, so either groups must be very large (hundreds of nodes) or the global fraction of faults must be trivially low for the system to survive a reasonable number of rounds.

Results and outline

In this chapter, we propose a scheme called *commensal cuckoo* that significantly improves the performance of the (parasitic) cuckoo rule. We demonstrate that the cuckoo rule fails largely due to "bad luck": bad events that occur with non-negligible probability, like consecutive malicious joins to the same group. Thus, our approach is to partially derandomize the cuckoo rule, which we do in two ways. First, we ensure that the number of nodes cuckood during a join deterministically matches the expected amount. Second, we allow groups to *vet* the join process, that is, reject join attempts if they have not received sufficiently many new nodes since the last join. Join vetting has surprisingly deeper benefits: it naturally addresses a known [19, 21] vulnerability in the cuckoo rule and suggests the possibility of allowing faulty nodes to *choose* their join location. Using commensal cuckoo, we are able to maintain smaller groups of 64 nodes (as opposed to hundreds), while tolerating a global fraction of faulty nodes between 32x–41x larger than that of the cuckoo rule.

We define the secure group partitioning problem in §4.1 and examine the cuckoo rule in §4.2, using simulations to understand why it fails. We introduce our improved scheme, the commensal cuckoo rule, in §4.3. Commensal cuckoo is just one (critical) piece of a larger set of mechanisms needed to solve the secure group partitioning problem. We review these complementary problems in §4.4, as well as our proposed solutions. We conclude in §4.5.

4.1 Problem definition

Let $N = n + \epsilon n$ be the size of the system, such that n nodes are correct and ϵn nodes are faulty. The global fraction of faulty nodes is thus $\epsilon/(1+\epsilon)$. Initially, the n correct nodes are mapped to random locations in $[0,1]$. Next, the adversary joins the ϵn faulty nodes one by one. Finally, the adversary begins executing rejoin operations (a leave followed by a join) on whichever faulty nodes it wants, even basing its decision on the entire system state. A *round* consists of a single rejoin operation. Our goal is to devise an efficient join rule such that, with high probability (i.e., at least $1 - 1/n$) and for any polynomial number of rounds, the system can be partitioned into intervals $I \in [0,1)$ that satisfy the following conditions [19]:

- *Balance condition: I* contains $\Theta(|I| \cdot n)$ nodes.
- *Correctness condition: I* has less than $1/3$ faulty nodes.

The nodes in such an interval comprise a *group*. In line with our discussion above, we assume that groups are disjoint to maximize parallelism, and each group runs a BFT protocol to perform tasks such as generate pseudorandom numbers and agree on membership changes. In principle, the constant $1/3$ can be replaced with any constant less than or equal to $1/2$, a flexibility we exploit later.

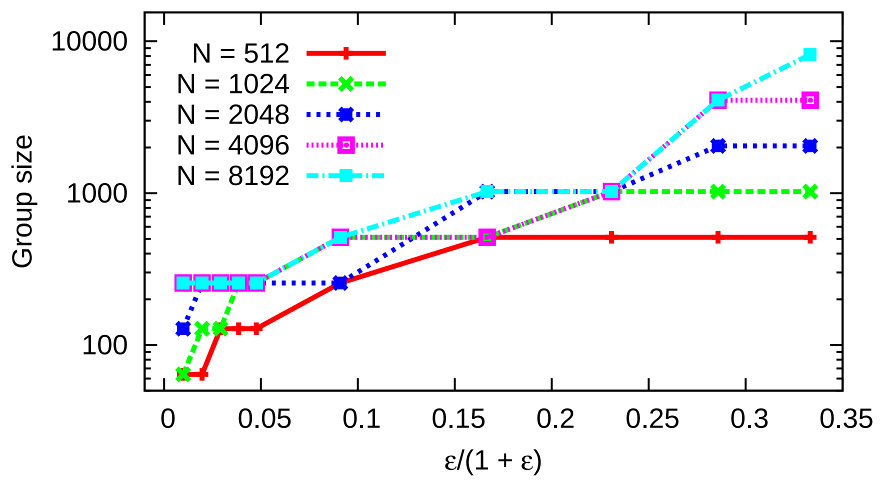


Figure 4.1: (Cuckoo rule) Minimum group size (in powers of 2) needed to tolerate different ϵ for 100,000 rounds, where $\epsilon/(1+\epsilon)$ is the global fraction of faulty nodes. Groups must be large (i.e., 100s to 1000s of nodes) to guarantee correctness.

Our discussion below abstracts away several lower-level mechanisms required to implement a secure group partitioning algorithm, such as a mechanism for constructing and routing verifiable messages between groups. Since the algorithms we present can be understood without these mechanisms, we postpone their discussion to §4.4.

4.2 Cuckoo Rule

Awerbuch and Scheideler [19] propose the following simple join rule. For a fixed $k > 0$, define a *k-region* to be a region in $[0, 1)$ of size k/n that starts at an integer multiple of k/n. For technical (divisibility) reasons, k-regions are rounded from above to the closest value $1/2^r$ where r is an integer.

Cuckoo rule. *When a new node wants to join the system, place it at a random* $x \in [0, 1)$ *and move (cuckoo) all nodes in the unique* k*-region containing* x *to random locations in* $[0, 1)$.

We call the new node's join a *primary join* and the subsequent joins of the cuckood nodes *secondary joins*. Awerbuch and Scheideler prove that in steady state, provided $\epsilon < 1/2 - 1/k$, all regions of size $O(\log n)/n$ have $O(\log n)$ nodes (i.e., they are balanced)

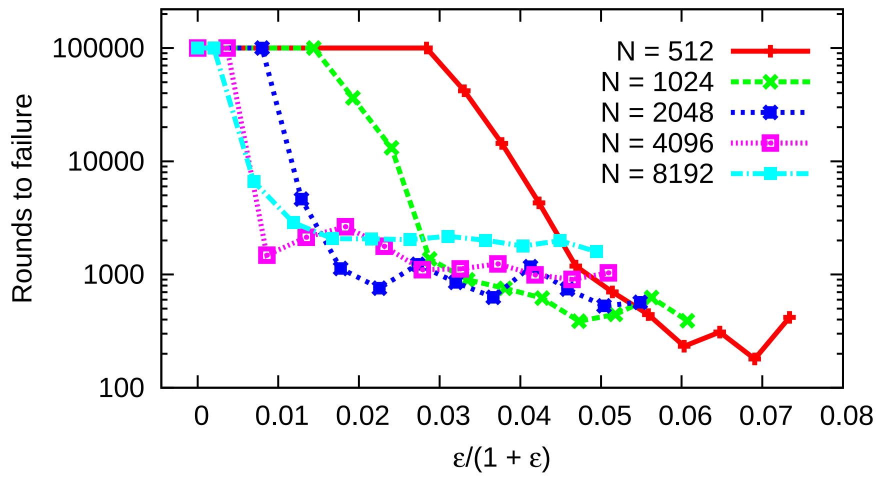


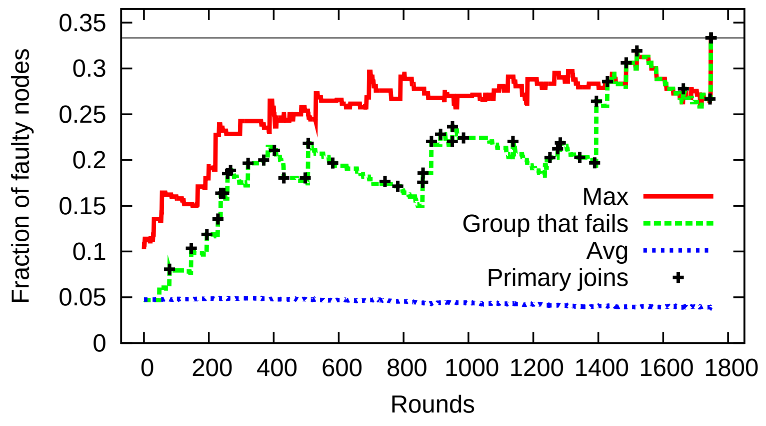
Figure 4.2: (Cuckoo rule) Number of rounds the system maintained correctness with an average group size of 64 nodes, for varied global fractions of faulty nodes. Simulation was halted after 100,000 rounds. The global faulty fraction is trivially low and system longevity drops sharply, for all N.

of which less than $1/3$ are faulty (i.e., they are correct), with high probability, for any polynomial number of rounds. Thus these intervals are the groups. Their analysis further implies that the best adversarial strategy is to target a single group and repeatedly rejoin faulty nodes that lie outside the group.

In practice, the random location of the primary join is generated by the group initially contacted by the new node, and the random locations of the secondary joins are generated by the group that owns the primary join location (where the new node ultimately joins).

Cuckoo rule analysis. We first observe that the optimal strategy of the adversary is actually different from that claimed in [19]—namely, target a single group, and have nodes not in that group rejoin—once constant factors are taken into account. At the beginning of each round, the adversary should sort all groups by increasing fraction of faulty nodes, and should have a faulty node belonging to the group with the lowest fraction attempt to rejoin the system. This Markovian strategy always maintains the largest and most promising number of targeted groups.

Using this modified adversarial strategy, we simulated the cuckoo rule to investigate the different constant factors involved. These factors arise from the use of Chernoff bounds in



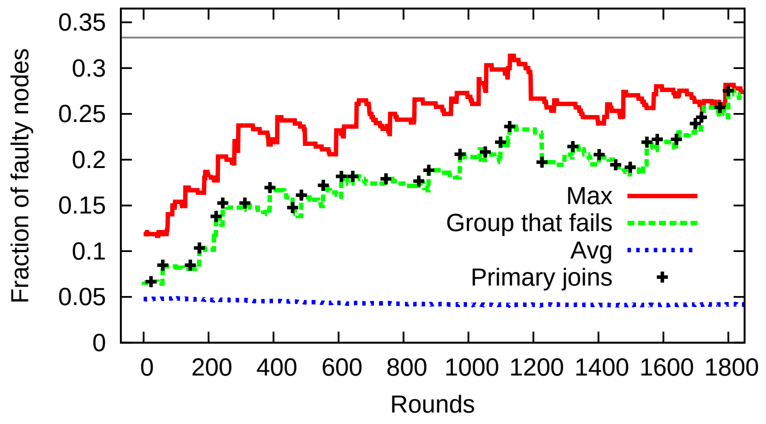


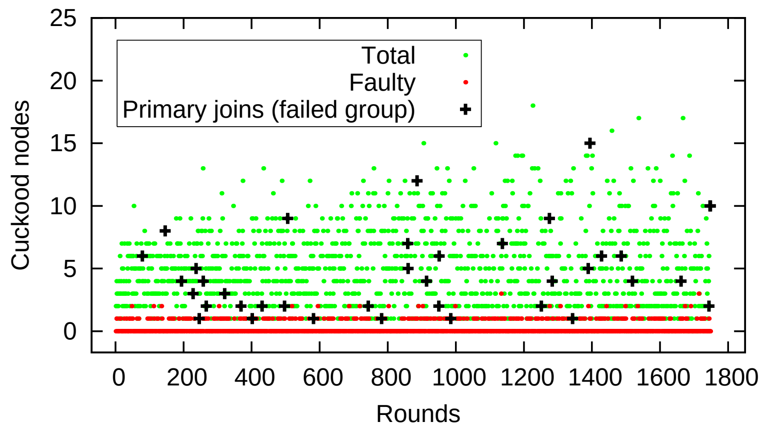
Figure 4.3: Evolution of a group that fails for the cuckoo rule (top) vs. commensal cuckoo rule (bottom). Failure occurs when the group's faulty fraction exceeds $1/3$, shown as a horizontal line. Primary joins (the black crosses) are often to blame for the group's ultimate failure, but their effect is more gradual for the commensal cuckoo rule, which does not fail until round 3158. For both rules, the maximum faulty fraction per group is much higher than the average, and the initial maximum fraction group was not the final one that failed.

the analysis, as well as union bounds over all groups and all rounds for which the balance and correctness conditions must hold. In our experiments, we scaled k to reflect the total number of nodes N instead of n, so that the expected number of total nodes cuckood from a k-region is k. For simplicity, we refer to this scaled quantity as k itself.

Figure 4.1 shows the minimum (average) group size required for the system to remain correct for 100,000 rounds in three consecutive trials, optimizing over k, for different values of N and ϵ (recall that $\epsilon/(1+\epsilon)$ is the global fraction of faulty nodes). We increased the group size in powers of 2 to avoid divisibility issues. As the figure shows, this size is in the hundreds of nodes for any reasonable global faulty fraction. Even when $\epsilon/(1+\epsilon) = 0.01$, the minimum group size is 256 for $N \geq 2048$. This situation is degenerate, actually, because the *total* number of faulty nodes in the system is itself less than $1/3$ of the average group size. This means that even if all faulty nodes were collocated in the same group, the group would still be correct (in expectation), unless it was much smaller than the average group size. The fact that the system still fails suggests that the cuckoo rule causes groups to become highly imbalanced. In fact, the optimal value of k in these cases was always less than 2, indicating that the system preferred to cuckoo very few nodes.

To determine what it would take to support groups of size 64—perhaps a performance upper-bound for any replicated system—we ran the simulation with this constraint and optimized over ϵ and k. Figure 4.2 shows the results, where each simulation is an average of three trials running for a maximum of 100,000 rounds, at which point the system was deemed non-faulty. For $N \geq 1024$, $\epsilon/(1+\epsilon) < 0.015$ to achieve a non-faulty result, dropping to 0.002 when $N = 8192$. Note that some degradation in this threshold is to be expected as N increases, because the fixed average group size becomes smaller relative to the global number of faulty nodes ϵn. However, all of the thresholds (for different N) still fall into the degenerate realm described above. They are also sharp, as the number of rounds to failure drops dramatically when ϵ is increased (note the log scale). Our goal is to increase these thresholds.

In order to achieve this increase, we gain a deeper understanding of what goes wrong by examining the evolution of a group that eventually fails. Figure 4.3 (left) plots the fraction of faulty nodes in such a group over time, for a system with $N = 8192, \epsilon = 0.05, k = 4$, and an average group size of 64 as before. We immediately see two problems. First, each primary join (indicated by a black cross) causes the faulty fraction of the group to jump. Although the fraction reduces slightly due to churn in the system caused by joins in other groups, it does not fully recover to its original level. Second, some primary joins occur very close together—the probability of such bad events is non-negligible, and can be easily calculated. These joins have the worst impact. In contrast, during the period from round



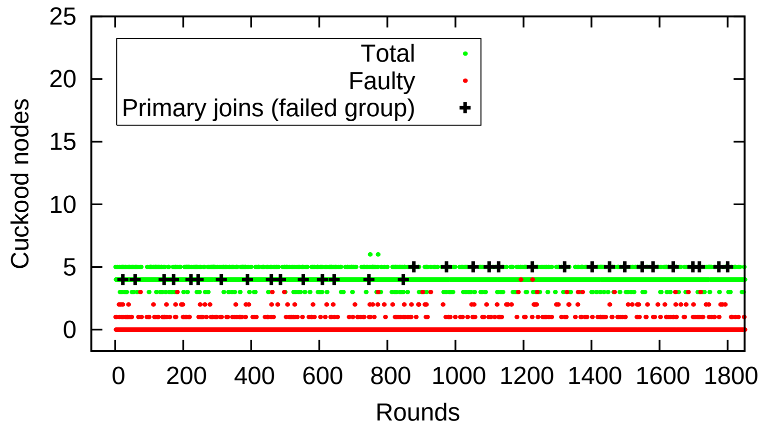


Figure 4.4: Number of cuckood nodes during joins for the cuckoo rule (top) vs. commensal cuckoo rule (bottom). The expected value is 4. The cuckoo rule yields higher variance in the number of evicted nodes, likely due to increasing non-uniformity of nodes across the keyspace, as k-regions are evicted *en masse*.

507 to 858, there are relatively few primary joins, giving the group enough time to reduce its faulty fraction.

To investigate the first problem, Figure 4.4 (left) shows the number of nodes cuckood in each round. Although this number starts out concentrated around its expected value of $k = 4$, it spreads out over time, ranging from 0 to as high as 18. This indicates that k-regions are getting increasingly "clumpy", likely due to the fact that primary joins empty out an entire k-region. To investigate the second problem, Figure 4.5 plots a CDF of the number of secondary joins in between successive primary joins to the failed group. This number is also expected to be $k = 4$, since an interval of size i is joined every $1/i$ rounds in expectation, during which time k/i other nodes are expected to be cuckood, of which

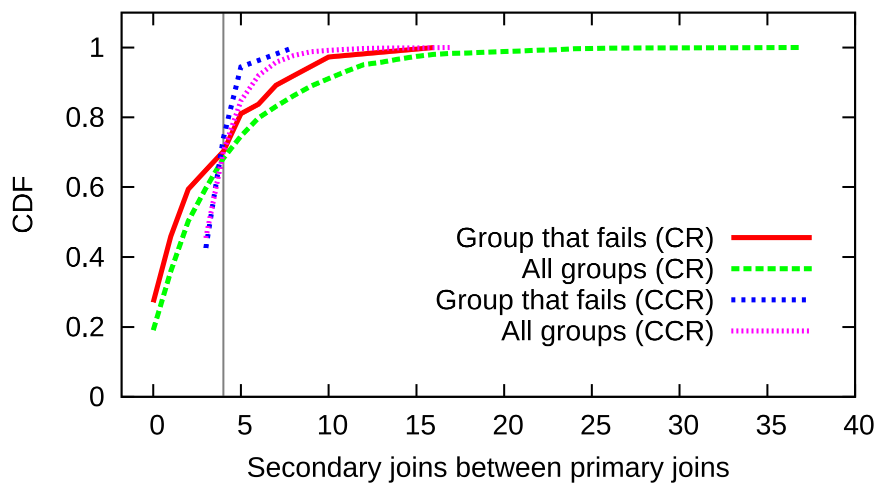


Figure 4.5: Number of secondary joins between successive joins for the cuckoo rule (CR) vs. commensal cuckoo rule (CCR). Commensal cuckoo leads to results more tightly concentrated around the expected value of 4, shown as a vertical line.

$(i/1)(k/i) = k$ are expected to land in the interval. However, the actual number shows enough variance to be problematic, with a considerable fraction at 0, and values as high as 16 for the failed group and 37 over all groups.

4.3 Commensal Cuckoo Rule

The two problems discovered in §4.2 can be remedied in two natural ways. First, to make cuckoos more consistently sized and evade the "clumpiness" problem, we cuckoo k nodes chosen randomly from the group being joined, instead of all nodes in the k-region surrounding the join location. However, using k for every cuckoo actually makes things worse, because a group that is light (i.e., has less than the average group size g), will continue to become lighter, eventually allowing the adversary to compromise it. Thus, to ensure that lighter groups cuckoo less and heavier groups cuckoo more, we scale k by the group's current size relative to g. Most prior work (e.g., [18, 55]) calculates g based on an estimate of n and a target group size $c \log n$ for some *a priori* fixed $c > 0$. However, g can also be chosen at the onset of the system (as in most of our experiments), not as a function of

n, to reflect the scalability of the higher-level application running in each group. Such an approach is practical if g is large enough for the maximum n ever expected to be seen.

Second, to address the problem of inconsistently spaced primary joins, we allow a group to *vet* join attempts by refusing them if it has not received a sufficient number of secondary joins since the last primary join. This allows the group to replenish the nodes it lost during the last cuckoo. Together, these techniques derandomize crucial aspects of the cuckoo rule, yielding our new rule:

Commensal cuckoo rule. *When a new node wants to join the system, pick a random* $x \in [0, 1)$. *If the group containing* x *has not received at least* $k - 1$ *secondary joins since its last primary join, start over with a new random* $x \in [0, 1)$. *Otherwise, place the node at* x *and move (cuckoo)* kg'/g *random nodes in the group to random locations in* $[0, 1)$, *where* g' *is the group's current size and* g *is the average group size.*

Interestingly, these two modifications are synergistic. By ensuring the expected (weighted) number of nodes are cuckood during each primary join, commensal cuckooing ensures a sufficient number of secondary joins, which allows a group to wait for enough of them before accepting another primary join. At the same time, by vetting repeated joins attempts, the adversary is forced to join distinct groups, which roughly speaking ensures that all groups are joined, resulting in $\sum kg'/g = (k/g) \sum g' = Nk/g$ total secondary joins. Thus, if groups wait for slightly less than k secondary joins between primary joins, a joining node need only try $\mathrm{O}(1)$ times before finding a group that accepts it. Cuckooing a weighted number of nodes but waiting for a fixed number of secondary joins has another benefit: it pushes the group's size towards the average size g.

We omit a detailed analysis of the commensal cuckoo rule due to space constraints.

4.3.1 Comparative evaluation

Figures 4.4 (right) and 4.5 show the number of cuckood nodes and the number of secondary joins between successive primary joins, respectively, using the commensal cuckoo rule. Compared to the results of the cuckoo rule, these numbers are tightly concentrated around their expected values of 4. The combined effect is seen in Figure 4.3 (right), where the group that ultimately fails does so in a more gradual and consistent manner than with the cuckoo rule. More importantly, commensal cuckoo is able to benefit from larger values of k: using $k = 6$ instead of 4 in Figure 4.3 results in no group failures for over 100,000 rounds.

Table 4.1 shows the largest value of $\epsilon/(1 + \epsilon)$ achieved by each scheme, optimizing over k, that allows a system with an average group size of 64 to remain correct for at least 100,000 rounds in three consecutive trials. Commensal cuckoo tolerates a global faulty fraction over 32x larger than that of the cuckoo rule while maintaining less than $1/3$ faulty nodes in each group. Since commensal cuckoo also improves with increasing k, we enforced an upper bound of $k = 12$; this was sufficient to scale performance with increasing N. The cuckoo rule does not improve by increasing k when the average group size is small, as we discussed in §4.2. In general, commensal cuckoo evicts more nodes per join than the cuckoo rule, but this eviction is needed to redistribute faulty nodes in the system.

Interestingly, commensal cuckoo's technique of join vetting has deeper benefits than those outlined above. As presented, the cuckoo rule in §4.2 has a known vulnerability [19, 21]: when joining a faulty node, the adversary may repeatedly cause the random number generation for the primary join location to fail—causing no cuckoos to occur—until it receives a join location of its liking. Awerbuch and Scheideler address this problem [21] by spawning artificial cuckoos whenever a join attempt fails. Join vetting, on the other hand, naturally evades this vulnerability, because regardless of the adversary's behavior, a group will not accept a primary join unless sufficient secondary joins have occurred.

The very property that saved us above, however, opens the door to a new type of liveness attack: if too few secondary joins occur in every group, the system might deadlock because it might be the case that no group accepts a join. The adversary can implement such an attack, for example, by having faulty nodes ignore secondary join operations (and leave

	<1/3 faulty per group			<1/2 faulty per group		
N	CR	CCR	Gain	CR	CCR	Gain
512	0.0284	0.0739	2.60x	0.0534	0.1854	3.47x
1024	0.0144	0.0757	5.25x	0.0293	0.1759	6.00x
2048	0.0080	0.0695	8.70x	0.0144	0.1803	12.5x
4096	0.0036	0.0693	19.0x	0.0080	0.1647	20.6x
8192	0.0020	0.0651	32.4x	0.0040	0.1660	41.4x

Table 4.1: Maximum $\epsilon/(1+\epsilon)$ of the cuckoo rule (CR) vs. commensal cuckoo rule (CCR), in order for groups of average size 64 to remain correct for at least 100,000 rounds.

the system instead). However, we can protect against this attack in much the same way Awerbuch and Scheideler protected against failed join attempts above. Namely, the group that issues a secondary join of a node can inform the group containing the secondary join location, so the latter group can increment its count of secondary joins even if the node does not arrive within a given time frame. (We assume correct nodes arrive within this time frame.) If the node attempts to join the group after this point, it will have to do so as a primary join. Section §4.4 discusses techniques for dealing with groups that become too light, e.g., as a result of these "no-shows".

The power of join vetting suggests the following potential modification to the commensal cuckoo rule: *allow faulty nodes to attempt a primary join at* any *location*, not just a random one. We conjecture that commensal cuckoo with this modification remains secure.

4.3.2 Higher fault thresholds

Table 4.1 also lists the largest $\epsilon/(1+\epsilon)$ achieved when less than $1/2$ of the nodes in a group are allowed to be faulty. These results are significantly better: commensal cuckoo tolerates a global faulty fraction over 41x larger than that of the cuckoo rule. An upper bound of $k = 8$ was sufficient to scale this performance with increasing N. Several techniques exist for improving the resiliency of BFT protocols to $1/2$, such as using a broadcast channel [139] which can be implemented using trusted primitives [34, 108], or separating the thresholds of consistency and availability [106]. We discuss the first approach in detail in Chapter 5. The latter approach allows a group to remain correct as long as fewer than $1/2$ of its nodes are faulty, even though it may become unavailable before that. This means that a correct group may become unresponsive, but we can mitigate this occurrence by leveraging the other groups in the system, e.g., as backups or as sources of replacement nodes.

4.4 Towards a complete solution

Commensal cuckoo relies on several lower-level mechanisms that we have assumed in previous sections. We briefly discuss some of those mechanisms here.

Secure routing. In order to send messages between the groups involved in a join operation, as well as to process client requests that may arrive at any group, we need a mechanism for routing messages. The cuckoo rule [19] uses a routing scheme based on de Bruijn graphs that requires $\mathrm{O}(\log n)$ hops and $\mathrm{O}(\log^3 n)$ total messages. Recent work [147, 181] reduces the latter overhead to $\mathrm{O}(\log n)$ messages in expectation. One might even combine these ideas with $\mathrm{O}(1)$-hop DHT constructions [75] to reduce both the number of hops and messages, at the cost of increased per-group state.

Group authentication. A group must be able to verify a message it receives even if it has no knowledge of the sending group's membership. The cuckoo rule relies on all-to-all connectivity between adjacent groups along a routing path to authenticate messages. However, cryptographic techniques can significantly improve the performance and flexibility of authentication in the DHT [181]. In particular, threshold signatures [158] and distributed key generation [94] can be used to assign a public/private key pair to each group that remains constant despite changes in the group's membership. Specifically, the g group members use a (t, g)-threshold signature scheme to cooperatively sign a message with their private key shares, despite up to t faulty nodes. When the group's membership changes, a new set of key shares corresponding to the same public/private key pair is generated and distributed. The new shares reveal no additional information to faulty nodes than their old shares, even in combination.

Bootstrapping and heavy churn. Our analysis of the cuckoo and commensal cuckoo rules assumes the system is in steady state. However, protocols to bootstrap the system and handle heavy churn are also needed. Prior fault-tolerant DHTs describe such protocols [18, 55], but they use groups that overlap and are based on the current (estimated) value of n. Thus, all groups necessarily change if n changes by a sufficiently large amount. Our scheme supports an alternative approach that chooses a target group size g at the onset of the system, based on the scalability of the intra-group protocol, as discussed in §4.3. This enables a bootstrapping protocol that creates an initial group spanning the entire $[0, 1)$ interval and uses split and merge operations to divide a region or merge two regions, respectively, if they get too heavy or too light. Such a protocol is simpler when n is small, and it localizes the effect of increasing n. For the bootstrapping protocol to work, we must assume that the number of faulty nodes is at most ϵn for all values of n.

Other attacks. Our fault model currently allows the adversary to control the behavior of only faulty nodes. A different type of attack is one of denial-of-service (DoS), in which the adversary forces a correct node to leave the system, e.g., by overwhelming it with spurious traffic. Awerbuch and Scheideler propose an extension [20] to the cuckoo rule that withstands such an attack; commensal cuckoo is compatible with this extension, and we believe some of the ideas presented in this chapter can be applied to the extended rule as well.

The adversary may also launch a DoS attack on the data layer of the DHT, for example by crafting a series of insert or lookup requests that target a particular region or node. Awerbuch and Scheideler handle such attacks by *proactively* replicating data items across groups [19]. A *reactive* approach may also be practical and more efficient. This is because, unlike with join-leave attacks, it is possible to *detect* when a data-layer attack occurs by measuring the current request load. The system could use this information to adaptively replicate data items on-the-fly.

4.5 Remarks

Commensal cuckoo is a practical scheme for partitioning a large-scale system into many small groups that remain correct despite adversarial join-leave attacks. By carefully managing when it is acceptable for new nodes to join groups, as well as balancing which existing nodes are evicted by such joins, commensal cuckoo can support significantly smaller group sizes and a higher fraction of faulty nodes than the state-of-the-art, the cuckoo rule. Commensal cuckoo relies on several important mechanisms to solve the secure group partitioning problem. In our future work, we plan to design protocols for these mechanisms along the lines of §4.4.

Chapter 5

Partial broadcast: Scaling BFT by Tolerating More Faults

Both the Prophecy system (Chapter 3) and the commensal cuckoo algorithm (Chapter 4) experience significant benefits if the fault resilience of a BFT group can be increased. In both cases, the cost of replication is reduced, which improves the performance of replicated requests in each group. In the case of commensal cuckoo, it also dramatically increases the number of global faults that can be tolerated across all groups (Table 4.1). This chapter explores how adding broadcast channels to the communication model can increase the resilience of a BFT group.

It is instructive to take a step back and ask where the requirement of fewer than $1/3$ faulty nodes comes from. All BFT protocols have, at their core, a solution to the *Byzantine agreement* problem: a set of n processors, each with an initial value and any f of whom may be arbitrarily faulty, must reach agreement on a value proposed by one of the correct processors.[1] Lamport, Shostak, and Pease showed in 1982 that in the standard communication model of a complete synchronous network of pairwise authenticated channels,

[1] This variant of the problem is called *consensus*. The variant where only a single processor has an initial value is called *reliable broadcast*. Consensus implies reliable broadcast, but the reverse is only true if faulty processors are in the minority, i.e., $n > 2f$.

Byzantine agreement is possible if and only if $n > 3f$ [107]. Subsequent work showed that the $n > 3f$ bound is remarkably robust to changes in the underlying communication and computation model [25, 50, 92].

A closer inspection of these results reveals that the fundamental reason for requiring $n > 3f$ processors is that faulty processors can *equivocate*, i.e.,, say different things to different processors. For instance, in a synchronous system with 3 processors, a single faulty processor can consistently send different messages to the two correct processors and make them agree to different values [107]. In asynchronous systems, the adversary's ability to delay messages (in addition to its ability to equivocate) can confound the correct processors even if cryptography is used [50]. Thus, several researchers have considered using stronger communication primitives such as broadcast channels. Broadcast channels mitigate equivocation by ensuring that a message appears identically at all recipients on the channel. In the synchronous model, Rabin and Ben-Or [139] introduced a global broadcast channel and achieved Byzantine agreement (in fact, any multiparty protocol) if and only if $n > 2f$. Fitzi and Maurer [57] added $\Theta(n^3)$ *partial broadcast* channels among every set of three processors to achieve Byzantine agreement if and only if $n > 2f$. Ravikant et al. [140] reduced the number of 3-processor channels required for $n = 2f + h$, where h is an integer in $[1..f]$, assuming sufficient connectivity in the underlying network. However, they get asymptotically tight results only for the same case $h = 1$ as Fitzi and Maurer do. Finally, Considine et al. [40] generalized partial broadcast to sets of $b > 3$ processors, achieving reliable broadcast (but not consensus) when $n > f(b + 1)/(b - 1)$.

The above overview of previous work illustrates that the gap between $n > 3f$ and $n > 2f$ represents a fundamental *price of equivocation*. In this chapter, we give a complete and tight characterization of this price, by studying the relationship between the fraction of processor 3-tuples that are prevented from equivocating, modeled as 3-processor partial broadcast channels, and the fault resilience required to solve Byzantine agreement. Specifically, we study the use, application, and algorithmic implications of 3-processor broadcast channels to Byzantine agreement. We view these channels as 3-hyperedges in the processor graph. Whereas prior work has almost exclusively focused on resilience $n > 2f$, we show that the range $n \geq 2f + h$, where $h \in [1..f]$ allows for significantly more efficient constructions by requiring asymptotically fewer than $\binom{n}{3} = \Theta(n^3)$ 3-hyperedges, assuming standard requirements on the connectivity of the underlying graph [113, 140]. We derive asymptotically tight bounds for all $h \in [1..f]$. Interestingly, this problem turns out to be a natural generalization of a well-studied *hypergraph coloring problem*. Although prior theoretical work has been limited to the synchronous model, we show how to apply our results to various other models as well, such as the partially synchronous model used by the BFT groups in the previous chapters [31].

The motivation for studying 3-hyperedges is two-fold. First, a 3-hyperedge (x, y, z) represents the fundamental unit of equivocation: a faulty processor x says different things to correct processors y and z. Second, while it is possible (and likely more efficient) to create a single or larger broadcast channel with x and the processors it has 3-hyperedges with, the expressiveness of 3-hyperedges may be useful in practice. For example, if hyperedges (x, y, z) and (x, y, w) exist but not (x, z, w), then a protocol may require x to use partial broadcast when sending a message to y and z or to y and w, but not when sending a message to z and w.

Results and outline

Let H be a 3-uniform hypergraph on n vertices (representing processors), where each 3-hyperedge represents a partial broadcast channel. Our main result is an asymptotically tight characterization of the necessary and sufficient number of 3-hyperedges required to achieve Byzantine agreement despite f faulty processors, for all $n \geq 2f + h$, h a positive integer in $[1..f]$. h is thus the parameter that interpolates between the well-studied cases $n > 2f$ and $n > 3f$. As in prior work [140], we assume the underlying graph is at least

$(2f+1)$-connected. Let $T_n(h)$ denote the minimum m such that there exists an H with m 3-hyperedges that achieves Byzantine agreement. We show:

- $T_n(h(n)) = \Theta\left(\frac{n^3}{h(n)^2}\right)$

For comparison, the only other existing work that gives results on this trade-off is by Ravikant et al. [140], who obtain an upper bound of $T_n(h(n)) = \mathrm{O}((f - h(n) + 1)f^2) = \mathrm{O}((n - 3h(n) + 1)n^2)$, which is up to a factor of $\Theta(n^2)$ off the correct bound, depending on $h(n)$. They also give a near-tight bound for the case $n = 2f + 1$, but this result is asymptotically identical to the trivial solution of including all 3-hyperedges. Their constructions are elementary and use a clever and simple recursive structure, though the analysis is non-trivial. We improve on their results by using the power of expander graphs, building hypergraphs out of existing expander constructions in order to exploit their high connectivity. Although we can prove our bound on $T_n(h(n))$ using a simple probabilistic method argument, in this work we are concerned with *explicit constructions*. A strong motivation for this goal is given by the following result.

Theorem 5.0.1. *Given a 3-uniform hypergraph $H = (V, E)$ with $|V| = n$, it is* co-NP-*complete to decide, for any $n = 2f + h$, $h \in [1..f]$, whether Byzantine agreement is possible in H despite f faulty processors.*

The proof of this theorem, a reduction from balanced bipartite independent set, can be found in Section 5.5. Since it is intractable to detect the possibility of Byzantine agreement in general, it is possible that explicit construction is the only reliable means of exploiting the efficiency gains of sparse fault-tolerant hypergraphs.

Our final result gives an exact bound for $U_n(h(n))$, the minimum m such that any H with m hyperedges achieves Byzantine agreement:

- $U_n(h(n)) = \binom{n}{3} - \frac{n-h(n)}{2} \cdot h(n)^2 + 1$

Section 5 discusses the application of our results to upper and lower bounds for Byzantine agreement in various models. Section 5.1 formally defines the problem and proves an equivalence to a more natural combinatorial problem, which we use in the remainder of the chapter. Section 5.2 proves the lower bound on $T_n(h(n))$ using a graph projection and counting arguments. Section 5.3 describes explicit constructions of H that match the upper bound on $T_n(h(n))$; we rely on a "lifting" procedure that converts existing constructions of Ramanujan graphs into hypergraphs with expander-like properties. Section 5.4 proves the bound on $U_n(h(n))$ using multivariate minimization techniques. Section 5.6 concludes with some open problems.

Algorithms and applications

The results in this chapter naturally give rise to new upper and lower bounds for Byzantine agreement in various models (augmented with partial broadcast channels). Specifically, our results apply to any proof that relies on the following intersection property between quorums of processors (made precise in Section 5.1), which we call *f-tolerance*: Consider a quorum of size $n - f$; although $n - f$ is the number of correct processors, a quorum of this size may still contain up to f faulty processors. Given two such quorums, the correct processors of one quorum may disagree with those of the other because faulty processors common to both may equivocate, unless their intersection contains at least one correct processor: $2(n - f) - n > f \implies n > 3f$. The key insight is that we get the same guarantee by replacing the correct processor x with a 3-hyperedge (x, y, z) such that y and z are correct processors in distinct quorums. Even if x is faulty, hyperedge (x, y, z) prevents it from equivocating to y and z and making their quorums agree on inconsistent values.

Ravikant et al. [140] prove that Byzantine agreement is possible in the synchronous model if and only if a set of conditions which includes f-tolerance holds. The remaining conditions are the standard connectivity requirements of the underlying graph, which we also assume in our setting. Thus Theorem 1 in their paper implies lower and upper bounds for Byzantine agreement in our setting as well. However, the hypergraphs they construct are suboptimal: they provide a tight bound only for $n = 2f + 1$ [140, Sections 5.2 and 5.3] and loose upper bounds for any $n = 2f + h, h \in [1..f]$ [140, Section 5.1]. By replacing their construction with ours from Section 5.3, we reduce the number of 3-hyperedges and the message complexity of their protocol by up to a factor of $\Theta(n^2)$.

In the asynchronous model, we can adapt the lower bound of Bracha and Toueg [25] to show that f-tolerance is necessary for Byzantine agreement. The proof is essentially the same as Theorem 3 in their paper, except that instead of any two $(n - f)$-sized quorums, a pair of quorums that violates f-tolerance must be used. We can obtain upper bounds for any $n = 2f + h$ by modifying the protocols of Bracha and Toueg [25, Figure 2] or Bracha [24], for example. We do this by overlaying our hypergraph construction onto the processor graph and requiring x to use hyperedge (x, y, z), if it exists, when sending a message to both y and z. Although this reduces the efficiency of the protocol, the relevant correctness proofs ([25, Theorem 4] and [24, Sections 3.2 and 6]) are readily adapted because they rely on f-tolerance. Similarly, there has been tremendous recent interest in the systems community on designing efficient Byzantine agreement protocols in a partially synchronous model with cryptography (e.g.,, [10, 43, 73, 90, 100]). This model is subject to Bracha and Toueg's lower bound [25], and essentially all upper bounds are derivatives of Castro and Liskov's protocol [29], which relies on f-tolerance for correctness [29, Invariants A.1.4 and A.1.5]. Therefore we can improve the resiliency of these protocols as above. This reduces their replication costs, which is often cited as an obstacle to practical deployment [90, 157, 178, 180].

There are several ways to implement 3-hyperedges in practice. One way is to use multicast groups; another is to use a shared cyptographic key; another is to use trusted primitives like an append-only log [34] or a trusted counter [108]. Depending on the implementation, it may not always be possible to force a processor x to use a 3-hyperedge *a priori*, but it is always possible for y and z to generate a proof of misbehavior (POM) [8] against x *a posteriori* if x violates the protocol. Several systems [8, 77, 100] use POMs in this manner.

Finally, if each 3-hyperedge is allowed to fail with some probability [57], our hypergraph construction reduces the probability of failure because there are fewer 3-hyperedges to union bound over.

Other related work

If cryptography is used, consensus is possible in the synchronous model when $n > 2f$ [50, 107]. Whereas consensus cannot be solved in the asynchronous model [56], the $n > 3f$ bound applies in this model when using randomized algorithms that terminate almost surely [25] or with probability $1 - \epsilon$ for fixed $\epsilon > 0$ [92], even if cryptography is used [50]. The same bound also applies in partially synchronous models [50].

Partial broadcast was first considered by Franklin, Wright, and Yung [62, 63] in the context of secure point-to-point communication over an incomplete network. Stronger primitives based on trusted subsystems and cryptography have also been used, such as *weak sequenced broadcast* [2] to solve weak Byzantine agreement in the asynchronous model, and *append-only log* [34] and *trusted incrementer* [108] to solve Byzantine agreement in a partially synchronous model. These primitives achieve resilience $n > 2f$.

We mention two other lines of work that are related to ours. The first considers hybrid fault models that combine Byzantine and crash failures (e.g.,, [66, 106]), in which optimal bounds on resilience [66, 106] depend on the number of faults of each type. The second considers a non-threshold adversary characterized by an *adversary structure* (e.g.,, [12, 58]), or a monotone set of subsets of processors any one of which may be faulty. It is known [58] that Byzantine agreement is possible if and only if no three sets cover all processors.

5.1 Problem Definition and an Equivalence

We model a system of n processors as a 3-uniform, n-vertex hypergraph $H = (V, E)$ where each edge $(x, y, z) \in E$ represents a partial broadcast channel. For a fixed integer f, we analyze the conditions under which Byzantine agreement is possible in H, when up to f processors are faulty. As in prior work [140], we assume the underlying graph is at least $(2f + 1)$-connected (e.g.,, via a complete set of 2-hyperedges (edges) connecting the processors). As explained in Section 5, this problem is equivalent to ensuring that in the intersection of any two size-$(n - f)$ quorums S and T, there exists a node z that cannot equivocate between correct nodes $x \in S, y \in T$. We assume w.l.o.g. that $S \cap T$ contains only faulty nodes, because a correct node in $S \cap T$ would prevent equivocation. This reduces the possibility of Byzantine agreement to the following property of H.

Definition 1. *A 3-uniform hypergraph $H = (V, E)$ with $|V| = n$ vertices is* f-tolerant *if $\forall\, S, T, Z \subset V$ and $S \neq T$ satisfying the conditions below, $\exists\, x \in S \backslash Z, y \in T \backslash Z, z \in S \cap T$ for which $(x, y, z) \in E$:*

- $|Z| \leq f$,
- $|S|, |T| \geq n - f$,
- $S \cap T \subseteq Z \subset S \cup T$.

The property as defined is somewhat unwieldy, because the sets S, T, and Z can overlap in a variety of ways. To simplify our problem statement, we introduce a new property on disjoint sets and show its equivalence. Consider the sets $A = S \setminus Z$, $B = T \setminus Z$, and $C = S \cap T$; A, B, and C are disjoint because $S \cap T \subseteq Z$. A and B contain the correct nodes in quorums S and T, respectively, and C contains the faulty nodes in their intersection. (There may be other faulty nodes in the two quorums, limiting the size of A and B, but only in sum.) We will shortly redefine f-tolerance in terms of the following notion.

Definition 2. *A 3-uniform hypergraph $H = (V, E)$ with $|V| = n$ vertices is h-*disjoint *if for all disjoint $A, B, C \subset V$ satisfying the conditions below, $\exists\, x \in A, y \in B, z \in C$ for which $(x, y, z) \in E$:*

- $|A|, |B|, |C| \geq h,$
- $|A| + |B| + |C| \geq \frac{n+3h}{2}.$

Figure 5.1 illustrates the equivalence of f-tolerance and h-disjointness. Note that although A, B, and C are symmetric in the above definition, we will often distinguish them as in Figure 5.1 for ease of exposition. The following theorem makes this equivalence precise:

Theorem 5.1.1. *Let H be a 3-uniform hypergraph on n vertices, and integer $f \geq \frac{n}{3}$. Then H is f-tolerant if and only if H is $(n - 2f)$-disjoint.*

Note that $n > 3f$, i.e., $f < \frac{n}{3}$, is the trivial case of f-tolerance. For such f, *all* 3-uniform hypergraphs on n vertices are f-tolerant.

Proof. First we show that $(n - 2f)$-disjointness implies f-tolerance. Given an S, T, Z satisfying the conditions of Definition 1, we show an A, B, C satisfying the conditions of Definition 2 such that an edge crossing A, B, and C certifies that the same edge crosses $S \setminus Z$, $T \setminus Z$, and $S \cap Z$. Then if H is $(n - 2f)$-disjoint, an edge between A, B, C must be present, so H must also be f-tolerant. Specifically, define $A = S \setminus Z$, $B = T \setminus Z$, and $C = S \cap T$. Since $S \cap T \subseteq Z$, A, B, and C are disjoint. Since $|Z| \leq f$, we have $|A|, |B| \geq (n-f)-f = n-2f = h$. Also, $|C| = |S \cap T| \geq |S|+|T|-n \geq 2(n-f)-n = n - 2f = h$. Finally, observe that

$$\begin{aligned}
|A| + |B| + |C| &= |S \setminus Z| + |T \setminus Z| + |S \cap T| \\
&= |S| + |T| - |S \cap T| - Z + |S \cap T| && \text{because } S \cap T \subseteq Z \subset S \cup T \\
&= |S| + |T| - |Z| \\
&\geq 2(n - f) - f = \frac{3}{2}(n - 2f) + \frac{n}{2} = \frac{n + 3h}{2}.
\end{aligned}$$

Thus, $(n - 2f)$-disjointness implies f-tolerance. We now show the reverse direction. Given an A, B, C satisfying the conditions of Definition 2, we show an S, T, Z satisfying the conditions of Definition 1 with $h = n - 2f$, such that an edge crossing $S \setminus Z$, $T \setminus Z$, and $S \cap T$ certifies that the same edge crosses A, B, and C. Then if H is f-tolerant, all the necessary edges must be present, so H must also be $(n - 2f)$-disjoint.

We may assume w.l.o.g. that $|A|+|B|+|C| = \frac{n+3h}{2} = 2n-3f$. Indeed, if $|A|+|B|+|C|$ is larger, we may consider subsets whose sum of sizes *does* equal $2n-3f$; any edge crossing these subsets also crosses A, B, C. Let $X, Y \subseteq V$ be two arbitrary (possibly empty) sets such that $X, Y, A, B,$ and C are pairwise disjoint, and

$$|X| = n - f - |A| - |C| \qquad\qquad |Y| = n - f - |B| - |C|.$$

Note that $|X|$ is nonnegative, because

$$|A| + |C| = 2n - 3f - |B| \leq 2n - 3f - (n - 2f) = n - f.$$

Similarly, $|Y|$ is nonnegative. We now verify that the total size of the five disjoint sets is at most n, so that it is possible to pick such sets.

$$\begin{aligned}
|A| + |B| + |C| + |X| + |Y| &= 2(n - f) - |C| \\
&\leq 2(n - f) - (n - 2f) = n
\end{aligned}$$

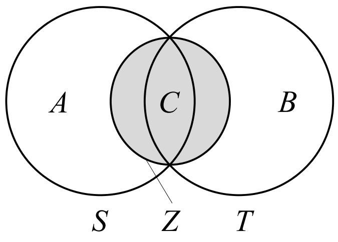


Figure 5.1: The relationship between sets S, T, Z of f-tolerance and sets A, B, C of h-disjointness. Shaded regions contain faulty nodes. The diagram above shows $C \subset Z$, but in general $C \subseteq Z$.

Now set $S = A \cup X \cup C \cup W$, $T = B \cup Y \cup C \cup W$, and $Z = C \cup X \cup Y$. Then

$$|S| = |A| + |X| + |C| = n - f.$$

Similarly, $|T| = n - f$. Finally,

$$\begin{aligned} |Z| = |C| + |X| + |Y| &= 2(n-f) - (|A| + |B| + |C|) \\ &\leq 2(n-f) - \frac{n + 3(n-2f)}{2} \\ &= 2n - 2f - 2n + 3f = f \end{aligned}$$

Finally, observe that $S \cap T = C \subseteq Z$ and $S \cup T = A \cup B \cup C \cup X \cup Y \supset Z$. Thus S, T, Z satisfy the conditions of f-tolerance, hence H contains an edge crossing $S \setminus Z, T \setminus Z, S \cap T$. These sets equal $A, B, C \cup W$, respectively.

Thus f-tolerance implies $(n - 2f)$-disjointness. $\square$

h-disjointness is equivalent to the notion of $(3, f)$-hyper-$(3f - n + 1)$-connectedness in [140]. However, we find our definition to be simpler and more clearly related to the hypergraph coloring literature, discussed below. The remainder of this chapter characterizes the hypergraphs that are h-disjoint by deriving tight bounds on the necessary and sufficient number of edges, $T_n(h)$ and $U_n(h)$ respectively. As we observed in Section 5, these results imply new upper and lower bounds for Byzantine agreement in different models. We start with $T_n(h)$:

Definition 3. *For positive integers n and h, $T_n(h)$ is the minimum m such that there exists an h-disjoint 3-uniform hypergraph with m edges.*

5.1.1 Related Combinatorial Problems

h-disjointness can be seen as a generalization of a rich body of work on *mixed hypergraph coloring* and *the upper chromatic number* (see Voloshin's book [172]). We present a single definition that essentially captures all of these concepts. A *k-heterochromatic coloring* of a hypergraph $H = (V, E)$ is a surjection $\chi : V \to [k]$ such that the restriction of χ to some $e \in E$ is injective. In other words, some edge has no repeated color. When H is k-uniform, this is equivalent to some edge being k-chromatic, as in h-disjointness.

A primary line of research in this area sought to analyze $f(n, k)$, the minimum number of edges among k-heterochromatically colorable, k-uniform, n-vertex hypergraphs [15, 16, 27, 47, 171], which was recently resolved up to lower order terms by Bujtás and Tuza [27]. The specific (earlier) result that is relevant to us is $f(n, 3) = \frac{n(n-2)}{3}$. h-disjointness has immediate connections to $f(n, 3)$, but introduces the additional concepts of *balance* and *partiality* in colorings, controlled by h. When $h = 1$, h-disjointness is equivalent to the condition that there is a trichromatic edge for all *small partial* colorings A, B, C, with $|A|, |B|, |C|$ non-empty, but total size only $|A|+|B|+|C| = (n+3)/2$. This condition is strictly stronger than requiring all complete colorings to have a trichromatic edge, because every complete coloring contains a small partial coloring. In contrast, when $h = f = n/3$, h-disjointness is equivalent to the condition that there is a trichromatic edge for all *balanced complete* colorings, with $|A| = |B| = |C| = n/3$. This is strictly weaker than restricting all complete colorings to have a trichromatic edge. For $1 < h < n/3$ the condition will be that all somewhat small, somewhat balanced colorings have a trichromatic edge.

Thus we may already state that $T_n(1) \geq f(n,3) = \frac{n(n-2)}{3}$ and $T_n(n/3) \leq f(n,3) = \frac{n(n-2)}{3}$. As we will see, we can in fact do much better and show a smooth transition $T_n(h) = \Theta\left(\frac{n^3}{h(n)^2}\right)$. In particular, this gives $T_n(1) = \Theta(n^3)$, and $T_n(n/3) = \Theta(n)$, both a factor of n from $f(n,3)$.

5.2 Lower Bounds

In this section, we will give asymptotically tight bounds on $T_n(h(n))$ for all non-decreasing functions $h(n)$. We will need the following notion of a hypergraph-projection.

Definition 4. *Let $H = (V, E)$ be a 3-uniform hypergraph, and $W \subseteq V$ a subset of its vertices. We define the* projection of H by W *as the* ***graph*** *$H_W = (V_W, E_W)$. The vertex set V_W is defined as $V \setminus W$. The edge set E_W has an edge (u, v) if and only if E has an edge (u, v, w), for some $w \in W$.*

This definition allows us to apply graph-theoretic theorems to hypergraphs, with potentially little loss. We extend the technique of [163], [15], [48], used to prove the aformentioned lower bound on k-heterochromatically colorable hypergraphs. They consider a hypergraph H for which every k-coloring contains a k-chromatic hyperedge, and proceed to lower bound the size of its edge set in two steps. First, they lower bound the number of edges in the projection of H by each $(k-2)$-vertex subset. Then, they upper bound the number of possible edge-projections of each hyperedge, giving a lower bound on the number of original hyperedges.

Our analysis is similar, with an added layer of complexity in lower bounding the number of edges in each projection. Because $h(n)$-disjointness implies that hyperedges cross somewhat balanced partitions of subsets of the vertices, we cannot assume that the projections are connected graphs. For large $h(n)$, we can only assume very weak conditions on the projections' connectivity. For small $h(n)$, we can assume conditions even stronger than connectedness. To address this, we prove a pair of results on the number of edges in a simple graph having appropriate connectivity conditions.

5.2.1 Linear $h(n)$

We first consider the regime in which $h(n)$ is linear in n. In fact, the bound we derive below holds for all legitimate $h(n)$, but it is asymptotically optimal only for linear $h(n)$. In the subsequent subsection, we will give a bound that holds for a smaller range of $h(n)$, but is asymptotically optimal for sublinear $h(n)$.

Theorem 5.2.1. *For any positive $h(n)$ that is bounded above everywhere by $\frac{n}{3}$,*

$$T_n(h(n)) \geq \frac{3}{4}n(1 - o(1)).$$

Proof. Let $H = (V, E)$ be a 3-uniform hypergraph on n vertices. Consider a coloring A, B, C of V, for which $|C| \leq n/3$. To satisfy $h(n)$-disjointness, H must contain a hyperedge that is trichromatic in A, B, C. In particular, for any bisection $(S, \bar{S})$ of $V \setminus C$, there is an edge in H_C crossing $(S, \bar{S})$. We will use the following lemma.

Lemma 5.2.2. *For **graph** $G = (V, E)$, $|V| = n$, if all bisections are crossed by at least one edge, then $|E| \geq n/2$.*

Proof. We will prove the contrapositive. Assume $|E| < n/2$. Let the connected components of G be $G_1, \ldots, G_t$. Observe that $t > n/2$, because $|E(G_i)| > \sum_{i=1}^{t} |V(G_i)|$, so

$$n/2 > |E| = \sum_{i=1}^{t} |E(G_i)| > \sum_{i=1}^{t} |V(G_i)| - 1 = n - t.$$

We will show how to allocate connected components to sets $S, \bar{S}$, so that each side has size exactly $n/2$. We treat each connected component as an integer representing its size, then simply apply the following claim.

Claim 5.2.3. *Let A be a multiset of integers such that $\sum_{a \in A} a = n$, and $t = |A| > n/2$. Then there exists $S \subseteq A$ such that $\sum_{a \in S} a = n/2$.*

This suffices to prove the lemma. □

Proof of Claim 5.2.3. We prove the claim by induction over n. Let the numbers in A be $a_1 \geq a_2 \geq \ldots \geq a_t$.

Base Case: If $n = 1$, the claim is vacuous, because there is no subset of A of size greater than 1. If $n = 2$, the only multiset of two integers that sums to 2 is $\{1, 1\}$, so let $S = \{1\}$.

Inductive Step: We construct a set of integers A', to which we will apply the claim inductively. Specifically, set $A' = A \cup \{a_1 - a_2\} \setminus \{a_1, a_2\}$. In other words, remove the two largest values, and replace them by their difference.

Let

$$n' = \sum_{a \in A'} = n - a_1 - a_2 + (a_1 - a_2) = n - 2a_2 \leq n - 2.$$

We have

$$|A'| = |A| - 1 > n/2 - 1 = (n-2)/2 \geq n'/2,$$

so we may apply the claim inductively. There thus exists a set $S' \subseteq A'$, with $|S'| = n'/2$ and $\sum_{a \in S'} a = n'/2$. Either S' or $\bar{S}' = A' \backslash S'$ contains the 'new' integer $a_1 - a_2$. Let Q refer to this set. Then consider the sets $Q \cup \{a_1\} \setminus \{a_1 - a_2\}$, and $(A' \setminus Q) \cup \{a_2\}$. The elements of the former set sum to $a_2 + \sum_{a \in Q} a$, and the latter set sums to $a_2 + \sum_{a \in A' \backslash Q} a$. But one of these sets has size $a_2 + n'/2$, and the other has size $a_2 + n'/2$. Since $n' = n - 2a_2$, one of the two sets has size $n/2$. Choose this set to be S. This proves the inductive hypothesis. □

To apply Lemma 5.2.2, observe that since $|V_C| \geq 2n/3$, we have $|E_C| \geq n/3$. Now we sum over all $|C| = n/3$.

$$\sum_{\substack{C \subset V \\ |C| = n/3}} |E_C| \geq \binom{n}{n/3} n/3. \tag{5.1}$$

In order to turn this into a bound on $|E|$, we need to upper bound the extent to which hyperedges in E are overcounted in (5.1). A hyperedge in E induces a (single) edge in E_C only if one of its vertices is in C, and two of them are not. For a given hyperedge in E, there are three possible vertices that could be in C. Conditioned on that vertex being in C, and the two other vertices being outside C, there are at most $\binom{n-3}{n/3-1}$ ways to choose the remaining vertices of C. Hence each edge in E contributes 1 to $|E_C|$ for at most $3 \cdot \binom{n-3}{n/3-1}$ distinct C.

Dividing out the maximum contribution of each edge gives our desired lowered bound:

$$\begin{aligned}
|E| &\geq \frac{\binom{n}{n/3} n/3}{3 \cdot \binom{n-3}{n/3-1}} \\
&= \frac{n!(n)(n/3-1)!(2n/3-2)!}{9(n-3)!(n/3)!(2n/3)!} \\
&= \frac{(n-1)(n-2)}{4n/3-2} \\
&= \frac{3}{4} n(1-o(1)).
\end{aligned}$$

This completes the proof of Theorem 5.2.1. □

5.2.2 Sublinear $h(n)$

Theorem 5.2.4. *For any function $h(n)$ that is bounded above everywhere by $\frac{n}{6}$,*

$$T_n(h(n)) \geq \Omega_n\left(\frac{n^3}{h(n)^2}\right).$$

Proof. First we will need a (weakened) generalization of Lemma 5.2.2.

Definition 5. *Let $G = (V, E)$ be a graph on n vertices. For integers a, b, we say that G is (a, b)*-crossing *if for all disjoint $X, Y \subseteq V$ such that $|X| = a$ and $|Y| = b$, there is an edge from X to Y. (That is, $\exists x \in X, y \in Y : (x, y) \in E$.)*

Lemma 5.2.5. *For positive $i \leq n/2$, every $(i, n/2)$-crossing graph on n vertices has at least $\frac{n^2}{2(n-i)} \left(\frac{n}{2i} - 1 \right)$ edges.*

Proof. Note that the bound is vacuous for $i = n/2$, so assume $i \leq n/2 - 1$. First observe that every subset of size i must have at least $n/2 - i + 1$ edges leaving it. This can be seen by contradiction: assume that some set X of size i has at most $n/2 - i$ edges leaving it, and hence at most $n/2 - i$ vertices in its neighborhood. Then take as Y the set of vertices in $V \setminus X$ with no edge to X. This set is of size at least $n - |X| - (n/2 - i) = n/2$. Since there is no edge from X to Y, the graph cannot be $(i, n/2)$-crossing.

We now show that the lemma in fact holds for any graph having the above boundary property. There are $\binom{n}{i}$ vertex sets of size i. We count at least $n/2 - i + 1$ edges out of each set. Each edge can only be counted for the sets that it leaves; there are $2\binom{n-2}{i-1}$ of these, because we must choose one of the two vertices, not choose the other one, and choose $i - 1$ other vertices. Hence, the total number of edges is at least

$$\begin{aligned}
\frac{\binom{n}{i}}{2\binom{n-2}{i-1}}(n/2 - i + 1) &= \frac{n!(i-1)!(n-i-1)!}{2i!(n-i)!(n-2)!}(n/2 - i + 1) \\
&= \frac{n(n-1)}{2i(n-i)}(n/2 - i + 1) \\
&= \frac{n}{2(n-i)} \left((n-1) \left(\frac{n}{2i} - 1 \right) + \frac{n-1}{i} \right) \\
&> \frac{n}{2(n-i)} \left((n-1) \left(\frac{n}{2i} - 1 \right) + \frac{n}{2i} - 1 \right) \\
&= \frac{n^2}{2(n-i)} \left(\frac{n}{2i} - 1 \right).
\end{aligned}$$

□

Now consider an $h(n)$-disjoint hypergraph $H = (V, E)$ on n vertices. For convenience, let $h = h(n)$. Since H is h-disjoint, there must exist a hyperedge for every A, B, C satisfying the conditions of Definition 2. In particular, consider a $C \subseteq V$ of size $|C| = h$. As in Section 5.2.1, we will show that the graph projection of H by C has many edges.

For $C \subseteq V$ having size $|C| = h$, let $G_C = (V_C, E_C)$ be the projection of H by C. We bound the size of each $|E_C|$. H is h-disjoint, so for any disjoint $A, B \subseteq V \setminus C$ such that $|A|, |B| \geq h$ and $|A| + |B| \geq \frac{n+3h}{2} - h = \frac{n+h}{2}$, there exists an edge $(x, y, z) \in E$ with $x \in A, y \in B, z \in C$. In other words, graph G_C has the following property: for $D \subseteq V_C$ with $|D| \geq \frac{n+h}{2}$, for all $A \subseteq D, B = D \setminus A$ with $|A|, |B| \geq h$, there is an edge in E_C from A to B.

When $h = 1$, the above says that each subset of V_C of size $\frac{n+1}{2}$ is connected. In generality, we would like to lower bound the number of edges in $|E_C|$, which we can do with Lemma 5.2.5. First let $n' = |V_C|$, so $n' = n - h$. Observe that G_C is $(h, n'/2)$-crossing, by choosing A as any set of size h, B as any disjoint set of size $\frac{n'}{2}$, and $D = A \cup B$, so that $|D| = |A| + |B| = h + \frac{n'}{2} = h + \frac{n-h}{2} = \frac{n+h}{2}$. Hence $|E_C| \geq \frac{n'^2}{2(n'-h)}\left(\frac{n'}{2h} - 1\right)$.

Now we must bound the extent to which each hyperedge is overcounted. A hyperedge can only be counted towards a given E_C if exactly one of its vertices is contained in C. There are then $\binom{n-3}{h-1}$ ways to choose the rest of the vertices. So each hyperedge contributes to $|E_C|$ for at most $3\binom{n-3}{h-1}$ values of C.

There are exactly $\binom{n}{h}$ sets C. Hence the total number of edges in H must be at least

$$|E| \geq \frac{\binom{n}{h}}{3\binom{n-3}{h-1}} \left(\frac{n'^2}{2(n'-h)}\right) \left(\frac{n'}{2h} - 1\right).$$

We have assumed that $h \leq n/6$, so $\frac{3}{2} \leq \frac{n}{4h}$ and hence

$$\frac{n'}{2h} - 1 = \frac{n-h}{2h} - 1 = \frac{n}{2h} - \frac{3}{2} \geq \frac{n}{4h}.$$

Similarly,

$$\frac{n'^2}{2(n'-h)} = \frac{(n-h)^2}{2(n-2h)} \geq \frac{(n-(n/6))^2}{2(n-2)} = \frac{25n^2}{72(n-2)}.$$

Then

$$\begin{aligned}
|E| &\geq \frac{\binom{n}{h}}{3\binom{n-3}{h-1}} \left(\frac{25n^3}{288h(n-2)}\right) \\
&= \frac{n!(h-1)!(n-2-h)!}{3h!(n-3)!(n-h)!} \left(\frac{25n^3}{288h(n-2)}\right) \\
&\geq \frac{25n^4}{864h^2(n-2)} = \Omega\left(\frac{n^3}{h^2}\right).
\end{aligned}$$

This completes the proof of Theorem 5.2.4. □

5.3 Upper Bounds

In this section, we give an asymptotically tight upper bound on $T_n(h(n))$ for almost all n, and all $1 \leq h(n) \leq n/3$. We do this by constructing near-Ramanujan expander graphs and converting them to "lifted" hypergraphs with expander-like properties. Our construction hence depends on the existence of sufficiently good expanders. These are probabilistically guaranteed to exist for all n; recall, however, that we are primarily interested in explicit constructions. Our result is fully constructive, with the exception that it relies on expander graphs that can be explicitly constructed for an infinite but incomplete set of values of n. As such, our result is only fully constructive for these n, which we do not consider a substantial weakness. To 'fill in the missing values' would require advances in explicit expander construction, which would immediately imply corresponding extensions of our algorithm.

Much of the difficulty of our analysis comes in explicitly bounding the degree. This is necessary to achieve an eigenvalue gap that can guarantee edges are well-distributed enough to induce a hyperedge across all "reasonable" colorings.

In what follows, an *algebraic* (n, d, λ)*-expander* will refer to an n-vertex, d-regular graph whose adjacency matrix has $\max(|\lambda_2|, |\lambda_n|) = \lambda$, where $\lambda_1 \geq \ldots \geq \lambda_n$ are the matrix eigenvalues.

Definition 6. *For graph $G = (V, E)$, we define a* lifted *3-uniform hypergraph $L(G) = (V, E')$ as follows. The edge set E' contains (x, y, z) if and only if at least two of the edges (x, y), (y, z), and (x, z) are present in G.*

In other words, for a given vertex x in G, we make hyperedges out of x and every pair of its neighbors. We claim that for an (n, d, λ)-expander G with the right parameters, hypergraph $H = L(G)$ is $h(n)$-disjoint and has a number of edges given by:

Theorem 5.3.1. $T_n(h(n)) \leq O\left(\frac{n^3}{h(n)^2}\right)$.

Proof. We construct a lifted 3-uniform hypergraph $H = L(G)$, where G is an (n, d, λ)-expander. Our goal is to determine the minimum λ such that H is $h(n)$-disjoint, as a function of d. Then using an expander G for which an upper bound on λ is known, we can derive a sufficient lower bound on d and hence the number of hyperedges in H.

To demonstrate $h(n)$-disjointness, we consider each partial 3-coloring A, B, C satisfying the conditions of Definition 2, and show that H contains a trichromatic edge for each such coloring. By the construction of H, it suffices to show that some set of vertices in C has edges in G to both A and B. Our main tool will be the Expander Mixing Lemma, which states that if $G = (V, E)$ is an (n, d, λ)-expander, then for any $S, T \subseteq V$, $\left||E(S,T)| - \frac{d|S|\cdot|T|}{n}\right| \leq \lambda\sqrt{|S| \cdot |T|}$. Additionally, we will need the following variant of the expander mixing lemma. We have not found this precise lemma in the literature, so we give a proof of it below.

Lemma 5.3.2 (Expander Vertex-Boundary Lemma)**.** *Let $G = (V, E)$ be an (n, d, λ)-expander. Then for any sets $S, T \subseteq V$,*

$$|S \cap N(T)| \geq |S| - \frac{2\lambda n}{d}\sqrt{\frac{|S|}{|T|}},$$

where $N(T)$ is the set of vertices having a neighbor in T.

Proof. The intuition is that if S is large, then the subset of S with edges to T cannot be small, because by the expander mixing lemma, a small set would have a small number of edges to T, but S must have a large number of edges to T. Let $S_T = S \cap N(T)$ and $s = |S|, t = |T|, s_T = |S_T|$. Then by definition $E(S,T) = E(S_T,T)$. By the expander mixing lemma, we have:

$$|E(S,T)| \geq \frac{d}{n}st - \lambda\sqrt{st} \quad \text{and} \quad |E(S_T,T)| \leq \frac{d}{n}s_Tt + \lambda\sqrt{s_Tt}.$$

Combining the inequalities and solving for s_T gives:

$$\begin{aligned} s_T &\geq s - \lambda\frac{n}{d\sqrt{t}}(\sqrt{s} + \sqrt{s_T}) \\ &\geq s - \frac{2\lambda n}{d}\sqrt{s/t}, \end{aligned}$$

where the last inequality follows because $s_T \leq s$. $\square$

Using these tools, we prove the following main lemma:

Lemma 5.3.3. *Let $A, B, C \subseteq V$ be colors of sizes $a \leq b \leq c$, respectively, of the vertices of $H = L(G)$. Define $\mathcal{F}(a,b,c) = \sqrt{\frac{cb}{a}}\left(\sqrt{a+b} - \sqrt{b}\right)$. If*

$$\lambda < \frac{d}{n}\mathcal{F}(a,b,c),$$

then H contains a trichromatic edge.

Before proving the lemma, we show how it implies the theorem. By picking a G with $\lambda < \frac{d}{n}\mathcal{F}(a,b,c)$, we ensure that H contains a trichromatic edge for all colorings A, B, C. But the conditions of h-disjointness do not require all colorings to have this property, and in particular we can show:

Claim 5.3.4. *For $a \leq b \leq c$ satisfying the conditions of Definition 2, $\mathcal{F}(a,b,c) \geq k\sqrt{h(n-h)})$ for a fixed constant $k > 0$.*

Proof. We wish to show that:

$$\sqrt{\frac{cb}{a}}\left(\sqrt{a+b}-\sqrt{b}\right) \geq k\sqrt{n(n-h)}$$

for some constant $k > 0$. To do this, we will show that $\sqrt{c} \geq k_1\sqrt{n-h}$ and $\sqrt{\frac{b}{a}}\left(\sqrt{a+b}-\sqrt{b}\right) \geq k_2\sqrt{h}$ for constants $k_1, k_2 > 0$, from which the theorem follows (with $k = k_1k_2$) because the LHS and RHS are non-negative in both inequalities. The conditions $a \leq b \leq c$ and $a+b+c = \frac{n+3h}{2}$ imply that $c \geq \frac{1}{3}\left(\frac{n+3h}{2}\right)$. Since both sides are positive, we have:

$$\sqrt{c} \geq \sqrt{\frac{1}{6}(n+3h)} > \sqrt{\frac{1}{6}(n-h)} = k_1\sqrt{n-h}$$

for $k_1 = \sqrt{\frac{1}{6}}$.

To lower bound the $\sqrt{\frac{b}{a}}\left(\sqrt{a+b}-\sqrt{b}\right)$ expression, we relax the $c \geq b$ and $a+b+c = \frac{n+3h}{2}$ constraints, and leave only the constraints $b \geq a$ and $a \geq h$. Now, since $a, b > 0$, we have for fixed a:

$$\begin{aligned}
\frac{d}{db}\left(\sqrt{\frac{b}{a}}\left(\sqrt{a+b}-\sqrt{b}\right)\right) &= \frac{d}{db}\left(\sqrt{\frac{b^2}{a}+b}-\frac{b}{\sqrt{a}}\right) \\
&= \frac{\frac{2b}{a}+1}{2\sqrt{\frac{b^2}{a}+b}} - \frac{1}{\sqrt{a}} \\
&= \frac{2+\frac{a}{b}}{2\sqrt{a}\sqrt{1+\frac{a}{b}}} - \frac{1}{\sqrt{a}} \\
&= \frac{2+\frac{a}{b}-2\sqrt{1+\frac{a}{b}}}{2\sqrt{a}\sqrt{1+\frac{a}{b}}} \\
&= \frac{\left(1-\sqrt{1+\frac{a}{b}}\right)^2}{2\sqrt{a}\sqrt{1+\frac{a}{b}}}
\end{aligned}$$

which is always positive, indicating that the function is monotonically increasing in b. Thus in order to minimize the function for fixed a, we choose b as small as possible, namely $b = a$. This gives:

$$\sqrt{\frac{b}{a}}\left(\sqrt{a+b}-\sqrt{b}\right) \geq \left(\sqrt{2}-1\right)\sqrt{a} \geq k_2\sqrt{h(n)}$$

for $k_2 = \sqrt{2}-1$. Thus the claim holds for $k = k_1 k_2 = \frac{\sqrt{2}-1}{\sqrt{6}}$. □

Thus it suffices to construct a G with $\lambda < \frac{dk}{n}\sqrt{h(n-h)}$. A Ramanujan graph has $\lambda \leq 2\sqrt{d-1} < 2\sqrt{d}$ and hence can be used if $2\sqrt{d} < \frac{dk}{n}\sqrt{h(n-h)}$. Rearranging gives $d > \frac{4n^2}{k^2 h(n-h)}$, which is satisfied if $d > \frac{6n}{k^2 h}$, since $h \leq n/3$. That is, if G is Ramanujan with $d = \Theta\left(\frac{n}{h}\right)$, then $H = L(G)$ is h-disjoint. Since H has a hyperedge for every pair of edges from a given vertex in G, H has maximum degree $O\left(\frac{n^2}{h^2}\right)$ and thus at most $O\left(\frac{n^3}{h^2}\right)$ hyperedges.

To prove the bound, it suffices to assert the existence of Ramanujan graphs for every n and d. In fact, a much stronger theorem holds: for every $\epsilon > 0$ and even $d \geq 4$, a random d-regular graph on n vertices satisfies $\lambda \leq 2\sqrt{d-1}+\epsilon$ with high probability [65]. Both ϵ and the requirement that d be even have an insubstantial effect on the final number of edges. This completes the proof of Theorem 5.3.1. □

We now give a proof of the main lemma. The idea is to first apply the expander vertex-boundary lemma to A and C, then the expander mixing lemma to B and the subset of C with neighbors in A. In so doing, we certify that A contains a vertex with edges to both B and C, in G. By the definition of a lifted hypergraph, this ensures that H contains a hyperedge crossing all three colors. Since we show this for arbitrary A,B,C satisfying the size bounds of Definition 2, this verifies the h-disjointness of H.

Proof of Lemma 5.3.3. Let $C_A = C \cap N(A)$ and $a = |A|, b = |B|, c = |C|, c_A = |C_A|$. By Lemma 5.3.2,

$$c_A \geq c - \frac{2\lambda n}{d}\sqrt{\frac{c}{a}}. \tag{5.2}$$

To prove the existence of a trichromatic edge in H, it suffices to show that $|E_G(C_A, B)| > 0$. By the expander mixing lemma,

$$|E_G(C_A, B)| \geq \frac{dbc_A}{n} - \lambda\sqrt{bc_A}.$$

Hence there exists an edge from C_A to B when $\frac{dbc_A}{n} > \lambda\sqrt{bc_A}$, which is equivalent to $c_A > \frac{\lambda^2 n^2}{bd^2}$, because all variables are non-negative. Substituting c_A with (5.2) and solving gives $\frac{n^2}{bd^2}\lambda^2 + \frac{2n}{d}\sqrt{\frac{c}{a}}\lambda - c < 0$. By the quadratic equation, this is equivalent to

$$\left(\lambda - \frac{\sqrt{c}bd}{n}\left(\sqrt{\frac{1}{b}+\frac{1}{a}} - \sqrt{\frac{1}{a}}\right)\right)\cdot$$
$$\left(\lambda + \frac{\sqrt{c}bd}{n}\left(\sqrt{\frac{1}{b}+\frac{1}{a}} + \sqrt{\frac{1}{a}}\right)\right) < 0.$$

Because λ is positive, the LHS is negative when the first term is negative. Thus we need:

$$\begin{aligned}\lambda &< \frac{\sqrt{c}bd}{n}\left(\sqrt{\frac{1}{b}+\frac{1}{a}} - \sqrt{\frac{1}{a}}\right)\\ &= \frac{\sqrt{c}bd}{n}\frac{\sqrt{a+b}-\sqrt{b}}{\sqrt{ab}}\\ &= \frac{d}{n}\sqrt{\frac{cb}{a}}\left(\sqrt{a+b}-\sqrt{b}\right).\end{aligned}$$

This concludes the proof of the lemma, and hence of Theorem 5.3.1. □

We also give a (slightly less general) explicit construction of such hypergraphs.

Theorem 5.3.5. *There is an algorithm that, for an infinite number of integers n, and any $h(n)$ bounded above by $n/3$, efficiently constructs an $h(n)$-disjoint hypergraph with $O\left(\frac{n^3}{h(n)^2}\right)$ hyperedges.*

In other words, by applying an explicit Ramanujan construction, we constructively achieve the result of Theorem 5.3.1, for an infinite number of values of n.

Proof. Extending the classic works of Lubotzky-Phillips-Sarnak [112], Margulis [115], and Morgenstern [119] on explicit constructions of Ramanujan graphs, Cioabă and Murty [35] give a construction that comes very close to the Ramanujan bound for nearly any graph size and degree.

Theorem 5.3.6 (Cioabă and Murty [35])**.** *Let $d \in \mathbb{Z}^+$ be such that $d-1$ is composite. For any positive ϵ, there exists an infinite sequence of graphs $\{G_i\}_{i=0}^{\infty}$ such that G_i is an $(n_i, d, (2+\epsilon)\sqrt{d-1})$-expander, and $n_i > n_{i-1}\ \forall\ i > 0$.*

Recall that Lemma 5.3.3 and Claim 5.3.4 together imply it suffices to construct an (n, d, λ)-expander G, where $\lambda < \frac{dk}{n}\sqrt{h(n)(n-h(n))}$ for a fixed k.

Pick $d = \frac{2(2+\epsilon)^2 n}{k^2 h(n)}$. Then [35] gives an algorithm to construct (n, d, λ)-expanders with $\lambda = \frac{(2+\epsilon)^2}{k}\sqrt{\frac{2n}{h(n)}}$. Then observe:

$$\begin{aligned}\frac{dk}{n}\sqrt{h(n)(n-h(n))} &= \frac{2(2+\epsilon)^2 nk}{k^2 h(n) n}\sqrt{h(n)(n-h(n))}\\ &\geq \frac{2(2+\epsilon)^2}{k}\sqrt{\frac{n(2/3)}{h(n)}}\\ &> \frac{(2+\epsilon)^2}{k}\sqrt{\frac{2n}{h(n)}} = \lambda,\end{aligned}$$

proving the theorem. □

5.4 A Sufficiency Condition for h-disjointness

In this section we consider a complementary question to that of the previous sections. Namely: how many hyperedges are necessary to ensure that *every* 3-uniform hypergraph of that size is $h(n)$-disjoint? Equivalently, what is the densest 3-uniform hypergraph that is *not* $h(n)$-disjoint? This question is relevant in practice, as it may be impossible in some systems to implement the set of 3-hyperedges exactly. The theorem below gives guarantees on reliability in such an oblivious setting.

Definition 7. *For integer $h \leq n/3$, the* sufficiency number *$U_n(h)$ is the minimum integer such that, for a 3-uniform hypergraph $H = (V, E)$ on n vertices, $|E| \geq U_n(h)$ implies that H is h-disjoint.*

Theorem 5.4.1. *For $h \leq n/3$,*

$$U_n(h) = \binom{n}{3} - \frac{n-h}{2} \cdot h^2 + 1.$$

In other words, $\frac{n-h}{2} \cdot h^2$ is the minimum number of edges one can remove from the complete 3-uniform n-vertex hypergraph, in order to ensure it is not h-disjoint.

Proof. Let $H = (V, E)$ be an n-vertex 3-uniform hypergraph that is not h-disjoint. By definition, there must be some partial coloring A, B, C of the vertices with $a = |A|, b = |B|, c = |C|$, such that there is no edge crossing A, B, C, and moreover $a, b, c \geq h$ and $a + b + c \geq \frac{n+3h}{2}$. For any 3-coloring, the complete 3-uniform hypergraph contains exactly abc crossing hyperedges. Hence for some a, b, c having the properties above, abc is the smallest number of edges that can be removed from the complete graph to make it not h-disjoint.

Claim 5.4.2. *For integers $a, b, c \geq h$ such that $a+b+c \geq \frac{n+3h}{2}$, abc is minimized by taking $a = h, b = h, c = \frac{n-h}{2}$.*

Proof. We will assume for the proof that $n \equiv h \pmod 2$. The second case is proved similarly.

First note that since $a+b+c \geq \frac{n+3h}{2}$, abc is minimized by taking $a+b+c = \frac{n+3h}{2}$. Decreasing the sum can always decrease the product. Hence, we may assume w.l.o.g. that $c = \frac{n+3h}{2} - a - b$, and minimize $g(a,b) = ab\left(\frac{n+3h}{2} - a - b\right)$. This gives the following optimization problem, for arbitrary n and h:

$$\begin{aligned} &\text{minimize} && g(a,b) \\ &\text{subject to} && a \geq h, \qquad b \geq h, \qquad a+b \leq \frac{n+h}{2} \end{aligned}$$

The constraints are linear and hence define halfspaces in the (a,b)-plane. These halfspaces define P, a polytope (triangle) in which the solution must lie. In particular, the optimal solution must either be a global minimum of $g(a,b)$ (and hence a root of the gradient); a minimum along one of the faces of the polytope; or a vertex of the polytope. We check each case in turn.

Proposition 5.4.3. *The following three facts about the constrained optima of $g(a,b)$ hold.*

- *The gradient of g has a single root inside P, and it is a global maximum.*
- *The minimum value of g along the faces of P is $\frac{(n+h)^2}{16}h$.*
- *The minimum value of g at a vertex of P is $\frac{h^2(n-h)}{2}$.*

Proof. We explore the three claims below.

Global minimum. The gradient of g has only a single root in the polytope.

$$g(a,b) = \left[\frac{b(n+3h)}{2} - 2ab - b^2, \frac{a(n+3h)}{2} - 2ab - a^2\right].$$

We may assume that $a \neq 0, b \neq 0$, since otherwise $[a,b]$ is not in the polytope. Hence $\frac{b(n+3h)}{2} - 2ab - b^2 = 0$ is equivalent to $a = \frac{n+3h}{4} - \frac{b}{2}$. Symmetrically, $b = \frac{n+3h}{4} - \frac{a}{2}$.

So $a = \frac{n+3h}{4} - \left(\frac{n+3h}{8} - \frac{a}{4}\right)$, i.e., $\frac{3}{4}a = \frac{n+3h}{8}$, i.e., $a = \frac{n+3h}{6}$. Symmetrically, $b = \frac{n+3h}{6}$. Therefore the only root of the gradient that may be in the polytope is $\left[\frac{n+3h}{6}, \frac{n+3h}{6}\right]$.

We now show that $\left[\frac{n+3h}{6}, \frac{n+3h}{6}\right]$ must be a maximum by examining the second derivatives of $g(a, b)$. The second derivatives are

- $g_{aa}(a, b) = -2b = -\frac{n+3h}{3}$
- $g_{bb}(a, b) = -2a = -\frac{n+3h}{3}$
- $g_{ab}(a, b) = \frac{n+3h}{2} - 2a - 2b = -\frac{n+3h}{6}$

The second derivative test says that if $g_{aa}(a, b)$ is negative, and $g_{aa}(a, b) \cdot g_{bb}(a, b) - g_{ab}(a, b)^2$ is positive, then $[a, b]$ is a local maximum. Indeed, $-\frac{n+3h}{3}$ is negative, and $\left(-\frac{n+3h}{3}\right)^2 - \left(-\frac{n+3h}{6}\right)^2$ is positive. Therefore the only root that may be within the polytope is a global maximum, so it cannot possibly be the minimum point of the polytope.

Faces. We consider the points along the faces of each constraint. In other words, we set the constraints to tight, then globally optimize the resulting function.

First make $a \geq h$ tight. $a = h$ means the new objective function is $hb\left(\frac{n+3h}{2} - h - b\right) = hb\left(\frac{n+h}{2} - b\right)$. We differentiate to find the optimum value of b for this function.

$$\frac{d}{db} hb\left(\frac{n+h}{2} - b\right) = \frac{h(n+h)}{2} - 2hb,$$

which is 0 only at $b = \frac{n+h}{4}$. Thus $g\left(h, \frac{n+h}{4}\right) = h\left(\frac{n+h}{4}\right)\left(\frac{n+h}{2} - \frac{n+h}{4}\right) = \frac{h(n+h)^2}{16}$ is a potential global minimum for g.

By symmetry, setting $b \geq h$ tight gives the same potential minimum.

Finally, set $a + b \leq \frac{n+h}{2}$ tight. Then the new objective function is $\left(\frac{n+h}{2} - b\right) b \left(\frac{n+3h}{2} - \left(\frac{n+h}{2} - b\right) - b\right) = \left(\frac{n+h}{2} - b\right) bh$. We differentiate to find the optimum value of b of this new function.

$$\frac{d}{db}\left(\frac{n+h}{2} - b\right) bh = \frac{h(n+h)}{2} - 2bh,$$

which is 0 only when $b = \frac{n+h}{4}$. Plugging this and $a = \frac{n+h}{4}$ (by symmetry) into g gives $g(a,b) = (\frac{n+h}{4})^2(\frac{n+3h}{2} - \frac{n+h}{2}) = \frac{(n+h)^2}{16}h$, the same value found for the other two constraints.

Vertices. We now consider the value of g at the three intersections of the three halfspaces. (Note that the faces only intersect in at most one point because they are unique and on two variables.)

Setting $a = h$ and $a+b = \frac{n+h}{2}$ gives $b = \frac{n-h}{2}$, hence $g(a,b) = h \cdot \frac{n-h}{2}\left(\frac{n+3h}{2} - h - \frac{n-h}{2}\right) = h \cdot \frac{n-h}{2} \cdot h$.

By symmetry, setting $b = h$ and $a + b = \frac{n+h}{2}$ also gives $g(a,b) = \frac{h^2(n-h)}{2}$.

Finally, setting $a = h$ and $b = h$ gives $g(a,b) = h^2\left(\frac{n+3h}{2} - 2h\right) = \frac{h^2(n-h)}{2}$, again. $\square$

We now observe how the proposition implies the theorem.

Comparison. There are only two possible minima in the polytope: $\frac{h(n+h)^2}{16}$ and $\frac{h^2(n-h)}{2}$. Observe that

$$\begin{aligned}\frac{(n+h)^2}{16} &= \frac{n^2+h^2+2nh}{16}\\ &= \frac{(n^2+9h^2-6nh)-8h^2+8nh}{16}\\ &= \frac{(n-3h)^2}{16} + \frac{h(n-h)}{2}\\ &\geq \frac{h(n-h)}{2}.\end{aligned}$$

Therefore $\frac{h(n+h)^2}{16} \geq \frac{h^2(n-h)}{2}$, so $\frac{h^2(n-h)}{2}$ is the minimum of the constrained $g(a,b)$. Recall that this was obtained by setting two faces to tight. In other words, set any two of a, b, c to h, and the other to $\frac{n-h}{2}$. $\square$

By the claim, removing $\frac{h^2(n-h)}{2}$ edges from the complete 3-uniform hypergraph ensures that a given valid coloring has no trichromatic edge. As a result, the hypergraph cannot be h-disjoint. Conversely, removing fewer edges cannot remove all the edges crossing any valid coloring. Hence $U_n(h) = \binom{n}{3} - \frac{n-h}{2} \cdot h^2 + 1$.

This completes the proof of Theorem 5.4.1. □

Theorem 5.4.4. *If we remove all edges incident on a given vertex, simultaneously from the complete n-vertex 3-uniform hypergraph, then we must remove $\Theta(n^2 h(n))$ edges to make it not $h(n)$-disjoint.*

Proof. As we know, some coloring of valid sizes a, b, c must have all edges crossing it removed. If we must remove all edges from a given vertex at once, then we clearly must remove all of the edges from all of the vertices in some color to all the other colors. Otherwise the remaining vertex in that color will still have edges to the other two colors.

The smallest a color A can be is size $h(n)$. So we must remove all of the edges incident on $h(n)$ vertices. There are $\binom{h(n)}{3}$ edges contained entirely within A, $\binom{h(n)}{2} \cdot \binom{(n+h(n))/2}{1}$ edges with two vertices in A, and $\binom{h(n)}{1} \cdot \binom{(n+h(n))/2}{2}$ edges with one vertex in A. Hence we must remove

$$\binom{h(n)}{3} + \binom{h(n)}{2} \cdot \frac{n + h(n)}{2} + h(n) \cdot \binom{(n + h(n))/2}{2} = \Theta(n^2 h(n))$$

edges in total. □

Note that for $h(n) = o(n)$, this is asymptotically larger than the minimum number of edges that can be removed from the complete 3-uniform hypergraph to make it not $h(n)$-disjoint.

5.5 Hardness of Deciding h-disjointness

In this section, we take a first step towards addressing algorithmic questions related to h-disjointness. In particular, given a 3-uniform hypergraph H, we would like to determine the minimum value $h_{opt}(n)$ such that H is $h_{opt}(n)$-disjoint, or barring this, an approximate value h that is as close to $h_{opt}(n)$ as possible. This question has practical value because it allows us to evaluate an existing hypergraph, or perhaps one constructed via the methods described in Section 5.3, for h-disjointness. Here we show that deciding whether H is h-disjoint is *co-NP*-complete.

Theorem 1 (restated). *Given a 3-uniform hypergraph $H = (V, E)$ with $|V| = n$, it is* co-NP*-complete to decide, for integers $h \leq n/3$, whether H is h-disjoint.*

Proof. The complement problem to h-disjointness is that of finding a disjoint $A, B, C \subseteq V$ satisfying the conditions of Definition 2 such that $\forall x \in A, y \in B, z \in C$, it holds that $(x, y, z) \notin E$. A certificate for this problem is the sets A, B, C, and it can be verified in $O(|A||B||C|)$ time by checking that all hyperedges (x, y, z) are not in E. Since complement h-disjointness is in *NP*, it follows that h-disjointness is in *co-NP*.

We show that complement h-disjointness is *NP*-hard by a reduction from balanced bipartite independent set (BBIS), which is *NP*-complete [9]. Given a balanced bipartite graph $G(X \cup Y; E)$ with $|X| = |Y| = n/2$ and a positive integer t, the decision BBIS problem is to find sets $A \subseteq X$, $B \subseteq Y$ with $|A| = |B| = t$ with no edges between A and B. Given an instance of the BBIS problem, we construct an instance of complement h-disjointness as follows. Create an empty 3-uniform hypergraph H with $n' = n + (n - t)$ vertices, where the first n vertices represent the vertices of G. On the $n - t$ vertices, create a complete 3-uniform hypergraph Z. Add a hyperedge (u, v, w) for each pair of vertices $u, v \in Z$ and every $w \in G$. Add a hyperedge (u, v, w) for each pair of vertices $u, v \in X$ and every $w \in Z$; do the same for every pair of vertices $u, v \in Y$. Finally, add a hyperedge (u, v, w) for each *edge* $(u, v) \in G$ and every $w \in Z$. The input to the complement h-disjointness problem is the hypergraph H and the positive integer $h = t$.

Given a solution (A, B) with $|A| = |B| = t$ to BBIS, we claim that A, B, Z is a solution to complement h-disjointness. Since $t \leq |X| = |Y| = n/2$, it follows that $|Z| = n - t \geq t$, so all $|A|, |B|, |Z| \geq t$. Also, $|A| + |B| + |Z| = t + t + (n - t) = \frac{n+(n-t)+3t}{2} = \frac{n'+3t}{2}$. Now, for a hyperedge to cross the sets A, B, Z, there must be some $u \in A, v \in B, w \in Z$ such that $(u, v, w) \in H$. By our construction, all but one type of hyperedge in H involve vertices in at most two of the sets $A \subseteq X$, $B \subseteq Y$, and Z. The exception are hyperedges (u, v, w)

where $(u, v) \in G$ and $w \in Z$. But since there are no edges crossing A, B, there is no hyperedge that crosses A, B, Z. Thus A, B, Z is a solution to complement h-disjointness.

In the reverse direction, suppose we have a solution A, B, C to complement h-disjointness. Since $|A| + |B| + |C| \geq \frac{n'+3t}{2} = n + t$, some vertices in Z must appear in the sets A, B, C. We claim that these vertices must appear in exactly one set. This is because if Z appears in all three sets, then there would exist a hyperedge crossing all three sets, since Z is a complete 3-uniform hypergraph. Similarly, Z cannot appear in exactly two sets, because then there would exist a hyperedge connecting two vertices of Z (one in each set) and a vertex in the third set (a vertex in G), also by our construction. Thus Z participates in exactly one set; assume w.l.o.g. that this set is C. We now claim that the vertices in X appear in at most two sets, and if they appear in two sets, one of those sets must be C. If X appears in both A and B, then there would exist a hyperedge connecting two vertices of X (one in each set) to a vertex of C, because C contains at least some vertices of Z by our argument above. Therefore, X appears in either A or B, but not both. The same argument shows that this is also true of Y. Combining these arguments with the fact that A, B, C are non-empty, it follows that the vertices of X and Y are split across A, B (though they may appear together in C). Since Z appears in C, A and B consist entirely of vertices in G. Finally, the same argument used in the forward direction above shows that there cannot exist an edge $(u, v) \in G$ between A and B, since then there would exist a hyperedge (u, v, w) to a vertex $w \in Z$ in C. Thus, since $|A|, |B| \geq t$, we can remove excess vertices so that $|A| = |B| = t$ and the resulting sets are a solution to BBIS. □

5.6 Remarks

This chapter studies the price of equivocation in distributed systems. Our tight bounds on the number of 3-processor partial broadcast channels required for Byzantine agreement describe the amount of equivocation a system can tolerate for a given level of redundancy.

Our results thus capture the *equivocation vs. redundancy trade-off*, an important metric in the cost-benefit analysis of a fault-tolerant system.

Several interesting theoretical questions remain. For example, given the hardness of deciding a system's resilience (h-disjointness) based on its partial broadcast channels, we are interested in approximation algorithms for this value. We would also like to understand the combinatorial properties of h-disjointness in greater depth. Can it be shown that h-disjoint hypergraphs *must* fundamentally be built on an underlying expander? How do our definitions and results scale to k-uniform hypergraphs, for $k > 3$?

On the practical side, it would be very interesting to find a realistic network that achieves h-disjointness naturally based on its broadcast (e.g.,, network hub) and point-to-point (e.g.,, network switch) connections, instead of constructing partial broadcast channels explicitly.

Chapter 6

Conclusions

Byzantine failures pose a significant challenge to building scalable Internet services because of their arbitrary nature. Standard techniques like load balancing requests to individual servers does not work, because a Byzantine-faulty server can return arbitrary results. And yet these failures continue to befall customer-facing services, much to the chagrin and concern of service providers.

This thesis describes scalable systems and algorithms for coping with Byzantine failures. There are two components to this approach: *fixing* the cause of the failure, and *masking* the effect of failures from customers. We described a real-world failure experienced by a database provider in the index data structure they were using, and fixed the cause of this failure by designing a new class of data structures called relaxed trees. Relaxed trees are simpler and more concurrent than standard search trees, while retaining most of their useful properties. Unfortunately, this particular failure could not have been masked from customers because it was correlated across all of the database servers, which were running identical code. We showed how to break this correlation by running distinct, off-the-shelf database implementations on each server, allowing us to bound the number of simultaneous failures that occur. Then, we run a Byzantine-fault-tolerant (BFT) protocol to mask the effect of the failures. Prophecy is a system that overcomes serious scalability limitations in existing BFT protocols, by using a cache validation technique to enable fast, load-balanced reads when results are historically consistent, while only slightly weakening the semantics of the BFT protocol. Prophecy achieves linear scale out on read-mostly workloads. To scale out further and to cope with write-heavy workloads, we devised an algorithm called commensal cuckoo that securely composes many small BFT groups, despite a dynamic adversary that can repeatedly leave and rejoin faulty nodes to the system. Such attacks are common in open peer-to-peer systems like BitTorrent. Compared to the state-of-the-art algorithm, commensal cuckoo tolerates 32x–41x more faulty nodes with groups as small as 64 nodes (as compared to hundreds). Finally, we improved the scalability of both Prophecy and commensal cuckoo by increasing the fault resilience of a BFT group from less than $1/3$ up to less than $1/2$. We did this by adding small (3-processor) partial broadcast channels that prevent faulty nodes from sending contradictory messages. We proved asymptotically tight bounds on the number of such channels needed for the complete range of fault resiliences, improving prior work by up to a quadratic factor.

Although our work focused on masking failures, this is not to say that fixing failures is less important. Masking failures prevents customers from seeing their effects, but if a failure (such as a software bug) does occur, then steps should be taken to remedy it. We did this for the database provider that experienced the outage. The other service providers listed in the introduction that suffered Byzantine failures also eventually fixed the causes of their failures.

As techniques for coping with Byzantine failures become cheaper, more service providers will employ them to avoid the severe costs of such a failure. This, in turn, will allow them to deliver richer, more reliable functionality to their customers. For example, a BFT BitTorrent system might be used to cache and distribute customer-facing web content, akin to a free alternative to Akamai. This thesis takes a step forward in that direction.

Bibliography

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *20th Symposium on Operating Systems Principles (SOSP)*, pages 59–74, 2005.

[2] I. Abraham, M. K. Aguilera, and D. Malkhi. Fast asynchronous consensus with optimal resilience. In *DISC*, pages 4–19, 2010.

[3] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.

[4] B. Adida. Helios: Web-based open-audit voting. In *USENIX Security*, July 2008.

[5] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[6] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. On the availability of non-strict quorum systems. In *DISC*, 2005.

[7] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. Byzantine and multi-writer K-quorums. In *DISC*, 2006.

[8] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *SOSP*, pages 45–58, 2005.

[9] Alon, Duke, Lefmann, Rodl, and Yuster. The algorithmic aspects of the regularity lemma. *J. Algorithms*, 16, 1994.

[10] L. Alvisi, A. Clement, M. Dahlin, M. Marchetti, and E. Wong. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[11] Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`, July 2008.

[12] S. Amitanand, I. Sanketh, K. Srinathant, V. Vinod, and C. P. Rangan. Distributed consensus in the presence of sectional faults. In *PODC*, pages 202–210, 2003.

[13] A. Andersson. Balanced search trees made simple. In *WADS*, volume 709, pages 60–71, 1993.

[14] `http://httpd.apache.org/`, 2009.

[15] J. L. Arocha, J. Bracho, and V. Neumann-Lara. On the minimum size of tight hypergraphs. *J. Graph Theory*, 16(4):319–326, 1992.

[16] J. L. Arocha and J. Tey. The size of minimum 3-trees. *J. Graph Theory*, 54(2):103–114, 2007.

[17] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *IEEE COMPSAC*, Nov. 1977.

[18] B. Awerbuch and C. Scheideler. Group spreading: A protocol for provably secure distributed name service. In *ICALP*, 2004.

[19] B. Awerbuch and C. Scheideler. Towards a scalable and robust DHT. In *SPAA*, 2006.

[20] B. Awerbuch and C. Scheideler. Towards scalable and robust overlay networks. In *IPTPS*, 2007.

[21] B. Awerbuch and C. Scheideler. Robust random number generation for peer-to-peer systems. *Theor. Comput. Sci.*, 410:453–466, 2009.

[22] R. Bayer. Binary B-trees for virtual memory. In *SIGFIDET*, pages 219–235, 1971.

[23] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[24] G. Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *PODC*, pages 154–162, 1984.

[25] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[26] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP*, pages 257–268, 2010.

[27] C. Bujtas and Z. Tuza. Smallest set-transversals of k-partitions. *Graph. Comb.*, 25:807–816, 2009.

[28] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, Nov. 2006.

[29] M. Castro. *Practical Byzantine Fault-Tolerance*. PhD thesis, Massachusetts Institute of Technology, 2000.

[30] M. Castro, P. Druschel, A. J. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI*, 2002.

[31] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[32] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Trans. Comp. Sys.*, 21(3):236–269, 2003.

[33] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine Byzantine-Fault Tolerance. In *USENIX ATC*, 2008.

[34] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *SOSP*, 2007.

[35] S. M. Cioaba. *Eigenvalues, expanders and gaps between primes*. PhD thesis, Queen's University, 2005.

[36] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP*, 2009.

[37] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. BFT: The time is now. In *LADIS*, 2008.

[38] CNET. Amazon cloud outage downs Netflix, Quora. `http://news.cnet.com/8301-1023_3-20089866-93/amazon-cloud-outage-downs-netflix-quora/`, Aug. 2011.

[39] CNET. Amazon cloud outage impacts Reddit, Airbnb, Flipboard. `http://news.cnet.com/8301-1023_3-57537499-93/amazon-cloud-outage-impacts-reddit-airbnb-flipboard/`, Oct. 2012.

[40] J. Considine, M. Fitzi, M. K. Franklin, L. A. Levin, U. M. Maurer, and D. Metcalf. Byzantine agreement given partial broadcast. *J. Cryptology*, 18(3):191–217, 2005.

[41] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, Aug. 2008.

[42] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[43] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, Nov. 2006.

[44] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-aware membership management for large-scale distributed systems. In *USENIX ATC*, 2009.

[45] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security*, 2003.

[46] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lak-shman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[47] K. Diao, G. Liu, D. Rautenbach, and P. Zhao. A note on the least number of edges of 3-uniform hypergraphs with upper chromatic number 2. *Discrete Math.*, 306(7):670–672, 2006.

[48] K. Diao, P. Zhao, and H. Zhou. About the upper chromatic number of a co-hypergraph. *Discrete Math.*, 220:67–73, 2000.

[49] J. R. Douceur. The Sybil attack. In *IPTPS*, 2002.

[50] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[51] J. Elson. tcpflow—A TCP Flow Recorder. `http://www.circlemud.org/~jelson/software/tcpflow/`, 2009.

[52] Facebook release cassandra: A structured storage system on a p2p network. `http://code.google.com/p/the-cassandra-project/`, 2008.

[53] Scaling out. `http://www.facebook.com/note.php?note_id=23844338919`, Aug. 2008.

[54] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1), 1999.

[55] A. Fiat, J. Saia, and M. Young. Making Chord robust to Byzantine attacks. In *ESA*, 2005.

[56] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[57] M. Fitzi and U. Maurer. From partial consistency to global broadcast. In *STOC*, pages 494–503, 2000.

[58] M. Fitzi and U. M. Maurer. Efficient Byzantine agreement secure against general adversaries. In *DISC*, pages 134–148, 1998.

[59] B. Fitzpatrick. Memcached: a distributed memory object caching system. `http://www.danga.com/memcached/`, 2009.

[60] Flickr phantom photos. `http://flickr.com/help/forum/33657/`, Feb. 2007.

[61] C. C. Foster. A study of AVL trees. Technical Report GER-12158, Goodyear Aerospace Corp., 1965.

[62] M. Franklin and M. Yung. Secure hypergraphs: Privacy from partial broadcast (extended abstract). In *STOC*, pages 36–44, 1995.

[63] M. K. Franklin and R. N. Wright. Secure communications in minimal connectivity models. In *Advances in Cryptology (EUROCRYPT)*, pages 346–360, 1998.

[64] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[65] J. Friedman. A proof of Alon's second eigenvalue conjecture. In *STOC*, pages 720–724, 2003.

[66] J. A. Garay and K. J. Perry. A continuum of failure models for distributed computing. In *6th International Workshop on Distributed Algorithms (WDAG)*, pages 153–165, 1992.

[67] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for Byzantine fault tolerant database replication. In *EuroSys*, pages 107–122, 2011.

[68] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.

[69] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[70] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, Dec. 1979.

[71] "This site may harm your computer" on every search result?!?! `http://googleblog.blogspot.com/2009/01/this-site-may-harm-your-computer-on.html`, Jan. 2009.

[72] J. Gray and A. Reuter, editors. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[73] R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic. The next 700 BFT protocols. In *EuroSys*, pages 363–376, 2010.

[74] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21, 1978.

[75] I. Gupta, K. P. Birman, P. Linga, A. J. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *IPTPS*, 2003.

[76] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC Systems Research Centre, 1985.

[77] A. Haeberlen, P. Kouznetsov, and P. Druschel. Peerreview: practical accountability for distributed systems. In *21st Symposium on Operating Systems Principles (SOSP)*, pages 175–188, 2007.

[78] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[79] B. Haeupler, S. Sen, and R. E. Tarjan. Rank-balanced trees. In *WADS*, pages 351–362, 2009.

[80] B. Haeupler, S. Sen, and R. E. Tarjan. Rank-balanced trees, 2012. In submission.

[81] S. Hanke, T. Ottmann, and E. Soisalon-Soininen. Relaxed balanced red-black trees. In *CIAC*, pages 193–204, 1997.

[82] J. Heidemann and G. Popek. File system development with stackable layers. *ACM Trans. Comp. Sys.*, 12(1), Feb. 1994.

[83] J. Hendricks, G. Ganger, and M. Reiter. Low-overhead Byzantine fault-tolerant storage. In *SOSP*, Oct. 2007.

[84] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comp. Sys.*, 4(1), Feb. 1986.

[85] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Sys.*, 12(3), 1990.

[86] K. Hildrum and J. Kubiatowicz. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *DISC*, 2003.

[87] A. Jaffe, T. Moscibroda, and S. Sen. On the price of equivocation in Byzantine agreement. In *PODC*, 2012.

[88] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *EuroSys*, 2006.

[89] A. Kapadia and N. Triandopoulos. Halo: High-assurance locate for distributed hash tables. In *NDSS*, 2008.

[90] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *European Conference on Computer Systems (EuroSys)*, pages 295–308, 2012.

[91] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, 1997.

[92] A. Karlin and A. C. Yao. Probabilistic lower bounds for Byzantine agreement and clock synchronization. Unpublished manuscript, 1984.

[93] J. Karlin, S. Forrest, and J. Rexford. Pretty Good BGP: Improving BGP by cautiously adopting routes. In *ICNP*, 2006.

[94] A. Kate and I. Goldberg. Distributed key generation for the internet. In *ICDCS*, 2009.

[95] J. L. W. Kessels. On-the-fly optimization of data structures. *Commun. ACM*, 26(11):895–901, 1983.

[96] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *HICSS*, 1998.

[97] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comp. Sys.*, 10(3), 1992.

[98] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[99] E. Kohler. Tamer. `http://read.cs.ucla.edu/tamer/`, 2009.

[100] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.

[101] R. Kotla and M. Dahlin. High-throughput Byzantine fault tolerance. In *DSN*, June 2004.

[102] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[103] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), 1979.

[104] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Sys.*, 6(2), 1984.

[105] L. Lamport. The part-time parliament. *ACM Trans. Comp. Sys.*, 16(2), 1998.

[106] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, pages 22–23, 2003.

[107] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[108] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: small trusted hardware for large distributed systems. In *NSDI*, 2009.

[109] H. C. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin. Flightpath: Obedience vs. choice in cooperative services. In *OSDI*, 2008.

[110] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[111] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

[112] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. *Combinatorica*, 8(3):261–277, 1988.

[113] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[114] D. Malkhi and M. K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[115] G. A. Margulis. Explicit group-theoretical constructions of combinatorial schemes and their application to the design of expanders and concentrators. *Probl. Inf. Transm.*, 24(1):39–46, 1988.

[116] K. Mehlhorn and A. Tsakalidis. An amortized analysis of insertions into AVL-trees. *SIAM Journal on Computing*, 15(1):22–33, 1986.

[117] J. Metzger. Managing simultaneous operations in large ordered indexes. Technical report, Technische Universität München, Institut für Informatik, TUM-Math, 1975.

[118] P. Mittal and N. Borisov. Shadowwalker: Peer-to-peer anonymous communication using redundant structured topologies. In *CSS*, 2009.

[119] M. Morgenstern. Existence and explicit constructions of $q + 1$ regular ramanujan graphs for every prime power q. *J. Comb. Theory Ser. B*, 62(1):44–62, 1994.

[120] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, Oct. 2001.

[121] A. Nambiar and M. Wright. Salsa: A structured approach to large-scale anonymity. In *CSS*, 2006.

[122] M. Naor and U. Wieder. Novel architectures for P2P applications: The continuous-discrete approach. *ACM Trans. Algorithms*, 3(3), 2007.

[123] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *NSDI*, 2006.

[124] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. on Comput.*, 2(1):33–43, 1973.

[125] *FIPS Publication 180-1: Secure Hash Standard*. Natl. Institute of Standards and Technology, Apr. 1995.

[126] F. E. Notes. Needle in a haystack: efficient storage of billions of photos. `http://www.facebook.com/note.php?note_id=76191543919`, Apr. 2009.

[127] B. Nowicki. NFS: Network File System Protocol specification. RFC 1094, Mar. 1989.

[128] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996.

[129] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *PODS*, pages 170–176, 1987.

[130] B. M. Oki and B. H. Liskov. Viewstamped replication: a general primary copy. In *PODC*, 1988.

[131] H. J. Olivié. A new class of balanced search trees: Half balanced binary search trees. *ITA*, 16(1):51–71, 1982.

[132] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual, FREENIX Track*, pages 183–191, 1999.

[133] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB, 2000.

[134] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[135] L. Poole and V. S. Pai. ConfiDNS: Leveraging scale and history to improve DNS security. In *WORLDS*, 2005.

[136] D. Pritchett. BASE: An ACID alternative. *ACM Queue*, 6(3), 2008.

[137] http://www.quova.com/, 2006.

[138] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981.

[139] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *STOC*, 1989.

[140] D. V. S. Ravikant, M. Venkitasubramaniam, V. Srikanth, K. Srinathan, and C. P. Rangan. On Byzantine agreement over (2, 3)-uniform hypergraphs. In *DISC*, pages 450–464, 2004.

[141] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Workshop on Theory and Practice in Distributed Systems*. 1995.

[142] R. Rodrigues. *Robust Services in Dynamic Systems*. PhD thesis, M.I.T., 2005.

[143] R. Rodrigues, P. Kouznetsov, and B. Bhattacharjee. Large-scale Byzantine fault tolerance: Safe but not always live. In *HotDep*, 2007.

[144] R. Rodrigues and B. Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Technical Report TR-2003-035, M.I.T., CSAIL, 2003.

[145] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *SIGOPS European workshop*, 2002.

[146] http://www.routeviews.org/, 2006.

[147] J. Saia and M. Young. Reducing communication costs in robust peer-to-peer networks. *Inf. Process. Lett.*, 106:152–158, 2008.

[148] B. Samadi. B-trees in a system with multiple users. *Inf. Proc. Lett.*, 5(4):107–112, 1976.

[149] C. Scheideler. How to spread adversarial nodes?: rotate! In *STOC*, 2005.

[150] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[151] R. Sedgewick. Left-leaning red-black trees. www.cs.princeton.edu/˜rs/talks/LLRB/LLRB.pdf, 2008.

[152] S. Sen and M. J. Freedman. Commensal cuckoo: Secure group partitioning for large-scale services. *SIGOPS Oper. Syst. Rev.*, 46(1):33–39, 2012.

[153] S. Sen, W. Lloyd, and M. J. Freedman. Prophecy: Using history for high-throughput fault tolerance. In *NSDI*, 2010.

[154] S. Sen and R. E. Tarjan. Deletion without rebalancing in multiway search trees. In *ISAAC*, 2009.

[155] S. Sen and R. E. Tarjan. Deletion without rebalancing in balanced binary trees. In *SODA*, 2010.

[156] S. Sen and R. E. Tarjan. Deletion without rebalancing in multiway search trees. *ACM Trans. Database Syst.*, 2013. To appear.

[157] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge: Reducing the costs of fast Byzantine replication in presence of unresponsive replicas. In *DSN*, 2010.

[158] V. Shoup. Practical threshold signatures. In *EUROCRYPT*, 2000.

[159] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent Byzantine-fault tolerance. In *NSDI*, 2009.

[160] A. Singh, T.-W. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *INFOCOM*, 2006.

[161] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[162] M. Srivatsa and L. Liu. Vulnerabilities and security threats in structured overlay networks: A quantitative analysis. In *ACSAC*, 2004.

[163] F. Sterboul. An extremal problem in hypergraph coloring. *J. Comb. Theory Ser. B*, 22(2):159 – 164, 1977.

[164] R. E. Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Inf. Proc. Lett.*, 16(5):253–257, 1983.

[165] R. E. Tarjan. Amortized computational complexity. *SIAM J. Algebraic and Disc. Methods*, 6:306–318, 1985.

[166] R. E. Tarjan. Efficient top-down updating of red-black trees. Technical Report TR-006-85, Department of Computer Science, Princeton University, 1985.

[167] TechCrunch. Facebook source code leaked. `http://www.techcrunch.com/2007/08/11/facebook-source-code-leaked/`, Aug. 2007.

[168] C. Technologies. The avoidable cost of downtime. `http://www.arcserve.com/us/~/media/files/supportingpieces/arcserve/avoidable-cost-of-downtime-summary.pdf`, Nov. 2010.

[169] F. Travostino and R. Shoup. eBay's scalability odyssey: Growing and evolving a large ecommerce site. In *LADIS*, Sept. 2008.

[170] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *SOSP*, 2007.

[171] V. I. Voloshin. On the upper chromatic number of a hypergraph. *Australas. J. Combin.*, 11:25–45, 1995.

[172] V. I. Voloshin. *Coloring Mixed Hypergraphs: Theory, Algorithms and Applications*, volume 17 of *Fields Institute Monographs*. AMS, 2002.

[173] P. Wang, I. Osipkov, N. Hopper, and Y. Kim. Myrmic: Provably secure and efficient DHT routing. Technical Report 2006/20, Univ. Minnesota, DTC, 2006.

[174] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: avoiding correlated failures in peer-to-peer systems. In *SRDS*, pages 362–369, 2002.

[175] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX ATC*, 2008.

[176] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, 2009.

[177] S. Wolchok, O. S. Hofmann, N. Heninger, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *NDSS*, 2010.

[178] T. Wood, R. Singh, A. Venkataramani, P. J. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *EuroSys*, pages 123–138, 2011.

[179] Yahoo! Hadoop Team. Zookeeper. `http://hadoop.apache.org/zookeeper/`, 2009.

[180] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, 2003.

[181] M. Young, A. Kate, I. Goldberg, and M. Karsten. Practical robust communication in dhts tolerating a Byzantine adversary. In *ICDCS*, 2010.

[182] ZDNet. AWS cloud accidentally deletes customer data. `http://www.zdnet.com/aws-cloud-accidentally-deletes-customer-data-3040093665/`, Aug. 2011.