Defending Networked Resources Against Floods of Unwelcome Requests

by

Michael Walfish

S.M., Electrical Engineering and Computer Science, M.I.T., 2004
A.B., Computer Science, Harvard University, 1998

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

Massachusetts Institute of Technology

February 2008

This version of the dissertation contains an addendum to the submitted version. The addendum is Appendix F. This version also makes a few other small changes.

Defending Networked Resources Against Floods of Unwelcome Requests

by

Michael Walfish

Submitted to the Department of Electrical Engineering and Computer Science on November 8, 2007, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

ABSTRACT

The Internet is afflicted by "unwelcome requests", defined broadly as spurious claims on scarce resources. For example, the CPU and other resources at a server are targets of denial-of-service (DoS) attacks. Another example is spam (i.e., unsolicited bulk email); here, the resource is human attention. Absent any defense, a very small number of attackers can claim a very large fraction of the scarce resources.

Traditional responses identify "bad" requests based on content (for example, spam filters analyze email text and embedded URLs). We argue that such approaches are inherently gameable because motivated attackers can make "bad" requests look "good". Instead, defenses should aim to allocate resources proportionally (so if 10% of the requesters are "bad", they should be limited to 10% of the scarce resources).

To meet this goal, we present the design, implementation, analysis, and experimental evaluation of two systems. The first, *speak-up*, defends servers against application-level denial-of-service by encouraging all clients to automatically send *more* traffic. The "good" clients can thereby compete equally with the "bad" ones. Experiments with an implementation of speak-up indicate that it allocates a server's resources in rough proportion to clients' upload bandwidths, which is the intended result.

The second system, DQE, controls spam with per-sender email quotas. Under DQE, senders attach stamps to emails. Receivers communicate with a well-known, untrusted *enforcer* to verify that stamps are fresh and to cancel stamps to prevent reuse. The enforcer is distributed over multiple hosts and is designed to tolerate arbitrary faults in these hosts, resist various attacks, and handle hundreds of billions of messages daily (two or three million stamp checks per second). Our experimental results suggest that our implementation can meet these goals with only a few thousand PCs. The enforcer occupies a novel design point: a set of hosts implement a simple storage abstraction but avoid neighbor maintenance, replica maintenance, and mutual trust.

One connection between these systems is that DQE needs a DoS defense—and can use speak-up. We reflect on this connection, on why we apply speak-up to DoS and DQE to spam, and, more generally, on what problems call for which solutions.

Dissertation Supervisor: Hari Balakrishnan
Title: Professor

To Jack and Ruth Radin

Contents

Figures

Previously Published Material

Chapter 3 significantly revises a previous publication [169]: M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense, *Proc. ACM SIGCOMM*, Sept. 2006. http://doi.acm.org/10.1145/1159913.1159948 © 2006 ACM.

Chapter 4 significantly revises a previous publication [168]: M. Walfish, J.D. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Distributed quota enforcement for spam control, *Proc. USENIX NSDI*, May 2006.

Appendix E and pieces of Chapter 4 appear in M. Walfish, J.D. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Supplement to "Distributed Quota Enforcement for Spam Control". Technical report, MIT CSAIL, MIT-CSAIL-TR-2006-33, May 2006. http://hdl.handle.net/1721.1/32542

Figure 3.15 (page 71) is reproduced from the following paper [138]: V. Sekar, N. Duffield, O. Spatscheck, J. van der Merwe, and H. Zhang. LADS: Large-scale automated DDoS detection system. *Proc. USENIX Technical Conference*, June 2006. The author gratefully acknowledges the permission of Vyas Sekar and his co-authors and of the USENIX association to reproduce this figure.

Acknowledgments

For months, I have been referring to this document as "my diss".[1] Fortunately, the custom of acknowledgments lets me end the dissing with a list of props. May I list properly.

The list begins with Hari Balakrishnan, my adviser, and Scott Shenker, my step-adviser in their research union. For me, Hari has been the consummate "justified optimist": he has made the difficult seem, and become, manageable. Scott's mastery of the Offhand Suggestion That Solves the Quandary at Hand has been clutch. The two of them have given me a tremendous amount, and here I can thank them only for a subset. First, their advice. It has ranged usefully over nudges, pokes, pushes, and, when needed, shoves. Without it, I would not have done most of what I did in graduate school. Second, their support, both teaching me How It is Done and socializing me to our research community. Third, their humor. Last and most relevant to this dissertation, our joint work. I am very much in their intellectual debt.

I am likewise massively indebted to co-author David Karger, whose contributions and style of thinking imbue this dissertation. I do not understand how someone can think so fast yet be so relaxed and patient, but I am grateful for this combination.

Co-author Mythili Vutukuru is the reason that our SIGCOMM paper, and likely this dissertation, was submitted on time. Under extreme pressure (which never shook her) she, among other things, created a durable, powerful infrastructure that continued to pay dividends as I prepared this dissertation. Co-author J.D. Zamfirescu delivered analogously for our NSDI paper, and his contributions and influence dwell in Chapter 4 and Appendix E.

Frans Kaashoek's friendly skepticism, detailed comments, and high standards improved not only this dissertation, for which he was a committee member, but also my past written and spoken work, and my general research taste. I am grateful for these things.

[1] I did not invent this moniker.

Committee member Mark Handley's careful reading, gentle but pointed comments, and cheery mix of enthusiasm and skepticism contributed much not only to this work but also to my morale. At various times and in various ways, Mark has been very supportive, and here I thank him for those things and for his trans-Atlantic readership and travel to my defense.

* * *

This dissertation builds on several papers [167–169]. Those papers were much improved by comments from, and conversations with, the following people: Ben Adida, Dave Andersen, Micah Brodsky, Russ Cox, Jon Crowcroft, Frank Dabek, Nick Feamster, Michel Goraczko, Jaeyeon Jung, Brad Karp, Dina Katabi, Sachin Katti, Eddie Kohler, Christian Kreibich, Maxwell Krohn, Karthik Lakshminarayanan, Vern Paxson, Adrian Perrig, Max Poletto, Sean Rhea, Rodrigo Rodrigues, Srini Seshan, Emil Sit, Sara Su, Arvind Thiagarajan, Mythili Vutukuru, Shabsi Walfish, Andrew Warfield, Keith Winstein, J.D. Zamfirescu, the HotNets 2005 attendees, and the anonymous reviewers for HotNets, SIGCOMM, and NSDI.

Also, I thank Jay Lepreau, Mike Hibler, Leigh Stoller, and the rest of Emulab for indispensable infrastructure and ace user support. And I thank the NDSEG fellowship program and the NSF for generous financial support.

* * *

I am fortunate to have been at MIT CSAIL for the past five years. There, I have profited from the suggestions, high standards, and expertise of many.

Robert Morris sets an inspirational example.

Maxwell Krohn and Russ Cox have entertained, with good nature, a preposterous volume of questions. In so doing, they have extracted me from more than a few tight situations and contributed much to my education. They are generous souls. I thank them for good advice, good feedback, good taste, good software, good conversations, and good fun.

Nick Feamster and Dave Andersen have served as great examples of Grad School (And Its Aftermath) Done Right. They have also given heaps of crucial advice, comments on drafts and talks, and sample code.

I am indebted to my officemates, present and past. Mythili Vutukuru is an inspiring collaborator. The curiosity and high standards of Jakob Eriksson, Bret Hull, Jaeyeon Jung, Arvind Thiagarajan and, recently, Ramki Gummadi gave rise to great feedback and great conversations. Over the years, Emil Sit has given valuable pointers and talk feedback.

Jeremy Stribling is a hilarious man. And totally hardcore.

I have learned much and received helpful feedback from many others at MIT, including Dan Abadi, Magdalena Balazinska, Micah Brodsky, Vladimir Bychkovsky, Austin Clements, Dorothy Curtis, Frank Dabek, Bryan Ford, Michel Goraczko, Anjali Gupta, Kyle Jamieson, Srikanth Kandula, Dina Katabi, Sachin Katti, Jon Kelner, Nate Kushman, Chris Lesniewski-Laas, Barbara Liskov, Sam Madden, Petar Maymounkov, David Mazières, Martin Rinard, Rodrigo Rodrigues, Stan Rost, Sara Su, Russ Tedrake, Alex Vandiver, Keith Winstein, John Wroclawski, and Alex Yip.

I thank Eddie Kohler for much typographical guidance, for `otftotfm`, and for the inspiring example of his dissertation.

Vivek Goyal and John Guttag gave particularly good suggestions on practice job talks, which are reflected in some of the "phrasing" of this dissertation. Both have also given me key advice at key times.

Butler Lampson's suggestions caused Appendix F.

I thank Sheila Marian for cheer.

* * *

There are a number of people who influenced me to enter graduate school in the first place and who equipped me with the needed skills.

Brad Karp has been encouraging my interest in Computer Science and research since 1995, when I was an undergraduate. I would not have been aware of this path—let alone taken it—had I not known him and his infectious passion for the field. He made many key suggestions over the years, including pointing me to Digital Fountain, and has spent entirely too much time over the last six years giving me clutch advice, support, and help.

Going back eight years, the Ph.D.s at Digital Fountain—particularly Vivek Goyal, Armin Haken, Diane Hernek, Mike Luby, Adrian Perrig, and Amin Shokrollahi—motivated me by example, whether they knew it or not. I am especially grateful for Diane's encouragement and Mike's support.

Going back even further, Mr. Arrigo taught me the joy of math and related disciplines, and it was he who encouraged me to take a CS course as an undergraduate, back when I knew nothing of the field. And Mrs. Leerburger, more than anyone, taught me to write.

And going back further, further still, I thank my parents, the Walfishes, and their parents, the Walfishes and Radins—for nature and nurture.

1

Introduction

Spam, defined as unsolicited bulk email, had been a background annoyance since the famous "DEC email" of 1978, the first known spam [156]. In the late 1990s, however, the volume of spam increased sharply, flooding inboxes and making email unusable for many recipients [37, 155]. In response, email providers and individual recipients deployed spam filters, which kept messages with certain words (e.g., "Viagra") out of inboxes. Spammers must have thought that their problem was one simply of presentation, not underlying message, for they began to traffic in euphemism: filter writers now had to block messages containing "encoded" words (e.g., "V!@gr@"). But spammers changed tactics again, and today their advertisements appear inside excerpts from sources like *The New York Times*, *Hamlet*, and "Seinfeld"—which are difficult for filters to recognize as spam—or inside images and audio files that are difficult for a computer to interpret. And there are anecdotal reports that spammers can respond within hours to changes in popular spam filters [94].

In this environment, it is hard to get filters right. Yet, people still filter. They have to: spam is roughly 75% of all email sent [106, 107, 150]. The result of filtering in this regime—a regime in which most email is spam, yet much spam looks legitimate to a computer—is that *legitimate email is sometimes kept from the recipient's inbox.* Anecdotal evidence suggests that the rate of such "false positives" is 1% [30, 116], with some estimating their economic damage at hundreds of millions of dollars annually [31, 48]. While we have no way to verify these numbers, we can vouch for the personal inconvenience caused by false positives. *Email is no longer reliable.*

* * *

Spam's increasing sophistication parallels developments in *denial-of-service* (DoS), a phenomenon that has many incarnations and is defined at a high

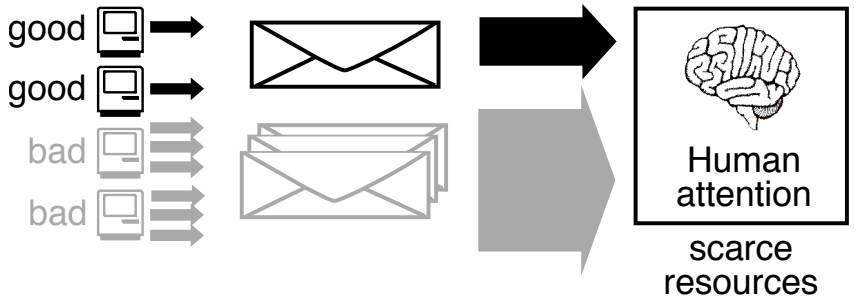
level as consuming a scarce resource to prevent legitimate clients of the resource from doing so. (We discuss some of the motives for conducting DoS attacks in the next chapter.) One of the original DoS attacks was to flood network links with ICMP [119] traffic. However, such attacks are easily filtered by placing a module in front of the flooded link. Next, many attackers turned to TCP SYN floods, which send a server spurious requests to establish TCP sessions. However, these floods can be heavily mitigated with TCP SYN cookies [21], which push the burden of session setup to clients, thereby preventing a small number of clients from overwhelming a server. In response, attackers have moved to *distributed* denial-of-service (known as *DDoS*), in which a large set of machines carries out the types of attacks above.

Lately, a particularly noxious form of DDoS has emerged, namely *application-level attacks* [73], in which the attackers mimic legitimate client behavior by sending proper-*looking* requests. Examples include HTTP requests for large files [129, 136], making queries of search engines [38], and issuing computationally expensive requests (e.g., database queries or transactions) [82]. For the savvy attacker, the appeal of this attack over a link flood or TCP SYN flood is two-fold. First, far less bandwidth is required: the victim's computational resources—disks, CPUs, memory, application server licenses, etc.—can often be depleted by proper-looking requests long before its access link is saturated. Second, because the attack traffic is "in-band", it is harder to identify and thus more potent.

* * *

So where does such unwelcome traffic—spam, DDoS, and other unsavory traffic, like viruses and worms—come from? The answer is: in many cases, from a flourishing Internet underworld. The eco-system of this underworld is well-developed, and we outline its structure in Chapter 2. For now, we just highlight a few aspects. First, the inhabitants of this underworld—organized criminals seeking profit and misguided middle-schoolers seeking bragging rights—are motivated and adaptable. Second, they send much of the unwelcome traffic from a low-cost attack platform—a collection of other people's computers that they have compromised and now control remotely. Third, the traffic is often *disguised* not only in terms of *content*, as with spam and DDoS that is difficult to detect (as described above), but also in terms of *location*: because the commandeered computers can be all over the world, it is hard to filter the traffic by looking at the network or geographic origin of the traffic.

(a)



(b)

FIGURE 1.1—Two examples of the abstract problem. In the problem, the two populations' requests are indistinguishable.

The state-of-affairs that has so far been described is vexing, and it highlights a fundamental, hard question. That question motivates this dissertation, and it is best phrased by casting the above situation in abstract terms.

1.1 THE PROBLEM IN ABSTRACT TERMS

The situation that we are concerned with has the following characteristics:

Scarce resources; bad clients may issue far more requests than good ones. There is a population of good and bad clients that make requests for some scarce resource. The situation becomes problematic when the bad clients individually issue far more requests than the good clients. For "resources" and "requests", we adopt broad notions. For example, requests could be HTTP requests, with the scarce resources being the CPU, RAM, and disk of a Web server, as in Figure 1.1(a). But requests could also be email messages, in which case the requested resource is human attention, as in Figure 1.1(b).

Open resources. By *open*, we mean that network protocols and social mores permit any client to make a request of the resource. For example, spam claims human attention by abusing the fact that anyone can, given a recipient's address, send email to that recipient. The resources of search engines (e.g., Google) and travel distribution sites (e.g., Orbitz) are another example: these sites devote significant computational resources—seconds of CPU time or more—to any client that sends them a request. (Request latencies to Google are of course fractions of a second, but it is highly likely that each request is concurrently handled by tens of CPUs or more.)

Can't differentiate good and bad reliably. First, we mean that good and bad *requests* are not differentiable. By "differentiate", we mean with computational means. (For example, a human can certainly detect spam, but the human's attention is the very thing that we are trying to protect.) Second, we mean that good and bad *clients* may not be differentiable: each bad client could adopt multiple identities (e.g., by pretending to have 200 IP addresses), meaning that an abusively heavy consumer of a resource might not be identifiable as such.

Spam has the three characteristics just listed. Other examples with these characteristics are HTTP floods of travel distribution sites, as discussed above; floods of DNS (Domain Name System) requests to sites that expose a database via a DNS front-end (e.g., the blacklist Spamhaus [158]); and floods of remote procedure calls (RPCs) (e.g., for exhausting the resources of a service like OpenDHT [130], in which clients are invited to consume storage anonymously, and make requests by RPC).

In these situations, classifying requests based on their contents amounts to applying a heuristic. Yet such heuristics are inherently gameable[1] and, when they err, may cause active harm by blocking valid requests (as in the case of spam filters blocking legitimate email, described earlier). Trying to identify and limit heavy users explicitly doesn't work either: a bad client making 100 times as many requests as a good client can thwart such a defense by adopting 100 separate identities.

Thus, we have the following question: *How can one defend against attacks on open, scarce resources, in which the bad clients make many more requests than the good ones, yet good and bad requests look alike?*

This is the question that this dissertation is trying to answer.

[1] I thank Maxwell Krohn for the words "heuristic" and "gameable" in this context.

1.2 PHILOSOPHY

Because, in our problem statement, telling apart "good" and "bad" requests and clients is difficult or impossible, our defenses will not even try to make this distinction. Indeed, in contrast to the prevailing "detect and block" ethos of current defenses, the defenses in this dissertation have an egalitarian ethos in that they treat all clients the same.

Specifically, our goal is to *allocate resources proportionally* to all clients. If, for example, 10% of the requesters of some scarce resource are "bad", then those clients should be limited to 10% of the resources (though the defense does not "know" that these clients are "bad"). To further specify the goal, the allocation to clients ought to reflect their *actual numbers* (as opposed to their *virtual identities*). Thus, any defense should be robust against a client manufacturing multiple identities to try to increase its share.

But what if 90% of the requesting clients are bad? In this case, meeting our goal still accomplishes *something*, namely limiting the bad clients to 90% of the resources. However, this "accomplishment" is likely insufficient: unless the resources are appropriately over-provisioned, the 10% "slice" that the good clients can claim will not meet their demand. While this fact is unfortunate, observe that if the bad clients look exactly like the good ones but vastly outnumber them, then *no* defense works. (In this case, the only recourse is a proportional allocation together with heavy over-provisioning.)

One might object that our philosophy "treats symptoms", rather than removing the underlying problem. However, as argued in Chapter 2, eliminating the root of the problem—compromised computers and underground demand for their services—is a separate, long-term effort. Meanwhile, a response to the symptoms is needed today.

1.3 CONTENTS OF THE DISSERTATION

Much of this dissertation focuses on the design, implementation, analysis, and evaluation of two systems that seek the proportional allocation goal stated above.

Speak-up. First, *speak-up*, the subject of Chapter 3, defends against application-level DDoS (as defined above). With speak-up, a server so victimized *encourages* its clients: it causes them, resources permitting, to send *higher* volumes of traffic. (The resulting extra traffic is automatic; the human owner of the client does not act.) The key insight of this defense is

as follows: we suppose that bad clients are already using most of their upload bandwidth so cannot react to the encouragement, whereas good clients have spare upload bandwidth so can send drastically higher volumes of traffic. As a result, the traffic into the server inflates, but the good clients are much better represented in the traffic mix than before and thus capture a much larger fraction of the server's resources than before.

Another way to explain speak-up is to say that the server, when attacked, *charges clients bandwidth for access*. Because the mechanism asks all clients for payment and does not distinguish among them, speak-up upholds the philosophy above. (In practice, speak-up can achieve only a *roughly* proportional allocation because it allocates service to clients in proportion to their bandwidths, which are not all equal. As we will argue in Chapter 3, a fully proportional allocation does not appear possible, given the threat.)

Of course, speak-up's use of bandwidth as a computational *currency* (i.e., a resource that the server asks clients to expend to get service) raises questions. However, we aim to show throughout Chapter 3 that speak-up is appropriate and viable under certain conditions. We also compare speak-up to other defenses and, in particular, compare bandwidth to CPU and memory cycles, which are the computational currencies that have previously been proposed [2, 10, 11, 45, 46, 80, 98, 170].

DQE. The second system in this dissertation, the subject of Chapter 4, is *Distributed Quota Enforcement (DQE)*, which controls spam by limiting email volumes directly. Under DQE, each sender gets a quota of stamps and attaches a stamp to each email. Receivers communicate with a well-known *quota enforcer* to verify that the stamp on the email is fresh and to cancel the stamp to prevent reuse. The receiving host delivers only messages with fresh stamps to the human user; messages with used stamps are assumed to be spam. The intent is that the quotas would be set such that, unlike today, no one could send more than a tiny fraction of all email. Because spammers need huge volumes to be profitable, such quota levels would probably drive them out of business. However, the system does its job even if they remain solvent.

This approach upholds the philosophy above: under DQE, an email—a request for human attention—is valid if the sender has not exhausted its quota; the contents of the email do not matter. Moreover, the quota allocation process does not try to determine which senders are spammers.

The focus of our work on DQE is the enforcer. Its design and implementation must meet several technical challenges: the enforcer must scale

to current and future email volumes (around 80 billion emails are sent per day [77, 123], and we target 200 billion emails, which is roughly two million messages per second), requiring distribution over many machines, perhaps across several organizations; it must tolerate faults in its underlying machines without letting much spam through; it must resist attacks; it must tolerate mutual distrust among its constituent hosts; and it should use as few machines as possible (to simplify management and hardware costs).

In addition to the technical requirements above, DQE raises some policy, economic, and pragmatic questions, notably: Which entity (or entities) will allocate quotas? How will that entity set quotas? Does DQE admit a plausible path to deployment and then widespread adoption? Although these issues are challenging, we are optimistic that they have viable solutions (as we discuss in Chapter 4). Moreover, even if they cannot be solved, the technical components of DQE are still useful building blocks for other systems.

Connections. This dissertation also considers the connections between these two systems. For one thing, we will see that DQE's enforcer can use a variant of speak-up to defend itself! (See §4.4.5.) More broadly, Chapter 5 situates speak-up and DQE in a spectrum of possible solutions to the abstract problem described in §1.1. We discuss what types of problems call for which solutions, why we apply speak-up to DDoS and DQE to spam (rather than vice-versa), and why DQE can defend itself with speak-up (but not vice-versa).

Interactions. Of course, if speak-up and DQE are successful, adversaries may shift to different tactics and attacks. In §4.7 and Chapter 6, we consider the interaction between these defenses and attackers' strategies.

1.4 CONTRIBUTIONS & RESULTS

We divide our contributions and results into three categories:

Articulation and philosophy. The contributions in this category are a definition of the abstract problem, including a recognition that spam and DDoS can be viewed as two incarnations of the same abstract problem; our philosophy of defense to this attack, namely avoiding heuristics and instead setting proportional allocation as the goal; and a comparison of several abstract solutions in the same framework.

Bandwidth as a currency. Speak-up's first contribution is to introduce bandwidth as a computational currency. One advantage of charging in this resource is that it is most likely adversaries' actual constraint.

A second contribution is to give four methods of charging in this currency. All of the methods incorporate a front-end to the server that does the following when the server is over-subscribed: (a) rate-limits requests to the server; (b) encourages clients to send more traffic; and (c) allocates the server in proportion to the bandwidth spent by clients. The method that we implement and evaluate is a *virtual auction*: the front-end causes each client to automatically send a congestion-controlled stream of dummy bytes on a separate payment channel. When the server is ready to process a request, the front-end admits the client that has sent the most bytes.

The collection of mechanisms for charging bandwidth is interesting for several reasons. First, the mechanisms are conceptually and practically simple. Second, they resist gaming by adversaries. Third, they find the price of service (in bits) automatically; the front-end and clients do not need to guess the correct price or communicate about it. Last, the mechanisms apply to other currencies (and the existing literature on computational currencies has not proposed mechanisms that have all of the properties of speak-up's mechanisms).

Beyond these methods, another contribution is to embody the idea in a working system. We implemented the front-end for Web servers. When the protected Web server is overloaded, the front-end gives JavaScript to unmodified Web clients that makes them send large HTTP POSTs. These POSTs are the "bandwidth payment". Our main experimental finding is that this implementation meets our goal of allocating the protected server's resources in rough proportion to clients' upload bandwidths.

Large-scale quota enforcement. DQE is in the family of email postage systems (which we review in §4.10). This dissertation's contribution to that literature is solving the many technical problems of large-scale, distributed quota enforcement (listed above in §1.3) with a working system, namely the enforcer. We show that the enforcer is practical: our experimental results suggest that our implementation can handle 200 billion messages daily (a multiple of the world's email volume) with a few thousand dedicated PCs.

Though we discuss the enforcer mostly in the context of spam control, it is useful as a building block in other contexts, both for enforcing quotas on resources other than human attention and for other applications.

Interesting aspects of the enforcer are as follows. The enforcer stores

billions of key-value pairs (canceled stamps) over a set of mutually untrusting nodes. It relies on just one trust assumption, common in distributed systems: that the constituent hosts are determined by a trusted entity. It tolerates Byzantine and crash faults in its nodes. It achieves this fault-tolerance despite storing only one copy (roughly) of each canceled stamp, and it gives tight guarantees on the average number of reuses per stamp. Each node uses an optimized internal key-value map that balances storage and speed, and nodes shed load to avoid "distributed livelock" (a problem that we conjecture exists in many other distributed systems when they are overloaded).

Stepping back from these techniques, we observe that the enforcer occupies a novel point in the design space for distributed systems, a point notable for simplicity. The enforcer implements a storage abstraction, yet its constituent nodes need neither keep track of other nodes, nor perform replica maintenance, nor use heavyweight cryptography, nor trust one another. The enforcer gets away with this simplicity because we tailored its design to the semantics of quota enforcement, specifically, that the effect of lost data is only that adversaries' effective quotas increase.

1.5 CONFRONTING CONTROVERSY

The author has given over twenty audio-visual presentations that covered the key ideas in this dissertation. These presentations have had at least five incarnations, have adopted different perspectives, and have been delivered to a range of audiences, from those familiar with the related work to general computer science audiences. The reception was the same every time:

"What??"

We therefore conclude that there is an element of controversy in these contents. For this reason, Appendices A and B answer common questions. Readers with immediate questions are encouraged to turn to these appendices now. And, while the main text tries to answer many of these questions, consulting these appendices while reading that text may still be useful.

Finally, for explicit critiques of speak-up, DQE, and the work as a whole, please see, respectively, §3.9, §4.11, and Chapter 6.

2

Background

In this chapter, we describe the eco-system that launches spam, DDoS, and other attacks; give a sense for the scale of the problem; and then highlight several important aspects. The abstract problem in §1.1 has been distilled from this context and from the specifics of the attacks themselves (described in later chapters).

2.1 AN INTERNET UNDERWORLD & ITS ECO-SYSTEM

The eco-system, in rough contours, works as follows. (For more detail, see [16, 75, 76, 99, 163] and citations therein.) First, there are people who specialize in finding *exploits*, i.e., bugs in operating systems and applications that allow a remote person or program to take control of a machine.

Next, the exploit-finders are compensated for their discoveries by *virus and worm authors*, whose wares use exploits to spread from machine to machine. Like real diseases, these "software diseases", commonly called *malware*, can spread either automatically (each machine compromised by an exploit searches for, and then compromises, other machines that are vulnerable to the same exploit) or as a result of ill-advised human behavior (visiting sketchy Web sites can lead to machine compromise if the Web browser on the machine is vulnerable to an exploit). Of course, malware can also spread without exploiting software flaws. For example, software programs, especially those of uncertain provenance, may include bundled malware; a side effect of installing such software is to install the malware. As another example, some viruses are installed when the virus emails copies of itself (to spam lists or to a human's address book) and recipients follow the email's instructions to install the attached program, which is the virus itself.

The malware authors are in turn compensated for their efforts by *bot herders*, who deploy the malware, hoping that its infectiousness will compromise a large population of machines (anywhere from hundreds to millions), leaving those machines under the control of the bot herder. Such a commandeered machine is known as a *bot* or *zombie*, and a collection of them is a bot network, or a *botnet*.

Bot herders profit from renting out their botnets.

Finally, those who rent the botnet make their profit from various activities that we now list. Spam is profitable because a small percentage of recipients do respond [57]. Launching DDoS attacks can be profitable if the attack is used to suppress a competing online business [38, 136] or to conduct extortion [111, 129, 160]. One can also use a botnet to host *phishing* Web sites (in phishing, people receive email, purportedly from their bank, that encourages them to disclose financial information at a Web site controlled by the attacker). A final activity is to log the key strokes of the human owner of the compromised machine, thereby gaining access to bank accounts or other "assets".

Of course, as only a single strand in the food chain, the account just given is an approximation. Nevertheless, it should give the reader enough context for the remainder of this chapter.

2.2 NUMBERS OF BOTS & BOTNETS

We first consider how many bots there might be worldwide and then how they are organized into botnets.

Estimates of the total number of bots are varied. According to Network World [112], "Symantec says it found 6 million infected bots in the second half of 2006. Currently, about 3.5 million bots are used to send spam daily, says ... a well-known botnet hunter". The author has personally heard informal estimates of tens of millions of bots and, based on these conversations, believes that 20 million bots is a high estimate of the worldwide total. Moreover, as Rajab et al. observe, the total number of bots that are online at any given moment is likely much smaller than the total number of infected computers [125].

In terms of how the bots are divided, there are likely a few big ones that are hundreds of thousands, or millions, strong [24, 40, 73, 75, 104, 153]. However, most botnets are far smaller. We consider average botnet size more completely at the end of the next chapter (§3.10.2); for now, we

simply relate one of the points made there, which is that 10,000 members is a large botnet.

Thus, while the *aggregate* computing power of bots represents several percentage points of the roughly one billion computers worldwide [34], no one bot herder can command anywhere near those resources. Moreover, bot herders frequently compete with each other to acquire bots; it is unlikely that they would merge their armies.

Nevertheless, bots are vexing and threatening. We now summarize some of the difficulties that they pose.

2.3 KEY CHARACTERISTICS OF THE PROBLEM

The "bot problem"—meaning the existence of bots themselves and also the attacks that they launch—is difficult for reasons that include the following:

- **Adversaries are motivated, adaptable, and skilled.** As mentioned in Chapter 1, the inhabitants of this eco-system are a combination of financially-motivated professionals and status-conscious teens (their notion of status is strange). Both groups have an incentive to be "good at being bad".

- **Bots are unlikely to go extinct.** Bots propagate as a result of two things that are notoriously difficult to control completely: bugs and human behavior. For this reason, we argue that while the number of bots may decrease in the future, the phenomenon will be with us for a long time to come. The problems caused by bots (spam, DDoS, etc.) must therefore be treated in isolation (even if they are just symptoms); we cannot wait for the "bot problem" to be eliminated.

- **Attack platform is cheap and powerful.** Botnets function as a large collection of inexpensive computing resources: bot rental rates, cents per bot week per host [73, 161], are orders of magnitude cheaper than hardware purchases. Moreover, bots allow an attacker to hide his actual geographic and network whereabouts since it is the bots, not the attacker's personal computers, that send the objectionable traffic.

- **Adversarial traffic can be disguised as normal traffic.** This disguise exists on two levels. First, as illustrated in the beginning of Chapter 1, adversaries disguise the content of their traffic. Second, because a bot army may be widely distributed, its bots often look, from a network

perspective, like a set of ordinary clients. It may therefore be difficult or impossible for the victim (the target of a DDoS attack, an email server receiving a lot of spam, etc.) or even the victim's Internet Service Provider (ISP) to filter the attack traffic by looking at the network or geographic origin of the traffic.

— **No robust notion of host identity in today's Internet.** We mean two things here. First, via *address hijacking*, attackers can pretend to have multiple IP addresses. Address hijacking is more than a host simply fudging the source IP address of its packets—a host can actually be *reachable* at the adopted addresses. We describe the details of this attack in Appendix C; it can be carried out either by bots or by computers that the attacker actually owns. Second, while address hijacking is of course anti-social, there is socially acceptable Internet behavior with a similar effect, namely deploying NATs (Network Address Translators) and proxies. Whereas address hijacking allows one host to adopt several identities, NATs and proxies cause multiple hosts to share a single identity (thousands of hosts behind a proxy or NAT may share a handful of IP addresses). As a result of all of these phenomena, the recipient of a packet in today's Internet may be unable to attribute that packet to a particular client.

We will further specify the threats relevant to the next two chapters in those chapters (see §3.3 and §4.1). The descriptions presented in this chapter are common to both threats.

3

Speak-up

This chapter is about our defense to application-level DDoS. This attack, defined at the beginning of Chapter 1, is a specific instance of the abstract problem described in §1.1. Recall that in this attack, computer criminals send well-formed requests to a victimized server, the intent being to exhaust a resource like CPU, memory, or disk bandwidth that can be depleted by a request rate far below what is needed to saturate an access link.

Current DDoS defenses try to *slow down the bad clients*. Though we stand in solidarity with these defenses in the goal of limiting the service that attackers get, our approach is different. It is to encourage *all clients to speak up*, rather than sit idly by while attackers drown them out.

To justify this approach, and to illustrate how it reflects the philosophy in §1.2, we now discuss three categories of defense. The first approach that one might consider is to **over-provision massively**: in theory, one could purchase enough computational resources to serve both good clients and attackers. However, anecdotal evidence [111, 151] suggests that while sites provision additional *link* capacity during attacks [25, 120], even the largest Web sites try to conserve *computation* by detecting bots and denying them access, using the methods in the second category.

We call this category—approaches that try to distinguish between good and bad clients—**detect-and-block**. Examples are profiling by IP address [9, 29, 103] (a box in front of the server or the server itself admits requests according to a learned demand profile); profiling based on application-level behavior [128, 147] (the server denies access to clients whose request profiles appear deviant); and CAPTCHA-based defenses [61, 82, 109, 151, 166], which preferentially admit humans. These techniques are powerful because they seek to block or explicitly limit unauthorized users, but their discriminations can err, as discussed in Chapter 1. Indeed, detection-based approaches become increasingly brittle as attackers' mimicry of legitimate

clients becomes increasingly convincing (see §3.9.2 for elaboration of this point).

It is for this reason that our philosophy in §1.2 calls for proportional allocation, i.e., giving every client an equal share of the server without trying to tell apart good and bad. Unfortunately, in today's Internet, attaining even this goal is impossible. As discussed in Chapter 2, address hijacking (in which one client appears to be many) and proxies (in which multiple clients appear to be one) prevent the server from knowing how many clients it has or whether a given set of requests originated at one client or many.

As a result, we settle for a *roughly* fair allocation. At a high level, our approach is as follows. The server makes clients reveal how much of some resource they have; examples of suitable resources are CPU cycles, memory cycles, bandwidth, and money. Then, based on this revelation, the server should arrange things so that if a given client has a fraction f of the clientele's total resources, that client can claim up to a fraction f of the server. We call such an allocation a *resource-proportional allocation*. This allocation cannot be "fooled" by the Internet's blurry notion of client identity. Specifically, if multiple clients "pool" their resources, claiming to be one client, or if one client splits its resources over multiple virtual clients, the allocation is unchanged.

Our approach is kin to previous work in which clients must spend some resource to get service [2, 10, 11, 41, 45, 46, 53, 80, 98, 114, 148, 170, 172]. We call these proposals **resource-based** defenses. Ours falls into this third category. However, the other proposals in this category neither articulate, nor explicitly aim for, the goal of a resource-proportional allocation.[1]

The preceding raises the question: which client resource should the server use? This chapter investigates *bandwidth*, by which we mean *available upstream bandwidth to the server*. Specifically, when the server is attacked, it encourages all clients to consume their bandwidth (as a way of revealing it); this behavior is what we promised to justify above.

A natural question is, "Why charge clients bandwidth? Why not charge them CPU cycles?" In §3.9.1, we give an extended comparison and show that bandwidth has advantages (as well as disadvantages!). For now, we simply state that many of this chapter's contributions apply to both currencies. Those contributions, mentioned in §1.4, include (1) articulating the goal of a resource-proportional allocation; (2) giving a family of mechanisms to

[1] An exception is a recent paper by Parno et al. [114], which was published after our work [169]; see §3.9.1.

charge in the currency that are simple, that find the correct "price", that do not require clients or the server to know or guess the price, and that resist gaming by adversaries; and (3) a viable system that incorporates one of these mechanisms and works with unmodified Web clients.

This chapter presents these contributions in the context of *speak-up*, a system that defends against application-level DDoS by charging clients bandwidth for access. We believe that our work in [167, 169] (on which this chapter is based) was the first to investigate this idea, though [70, 142] share the same high-level motivation (see §3.9.1).

* * *

The chapter proceeds in two stages. The first stage is a quick overview. It consists of the general threat (this section), the high-level solution (§3.1), and responses to five common questions (§3.2). The second stage follows a particular argument. Here is the argument's outline:

— We give a detailed description of the threat and of two conditions for addressing the threat (§3.3). The first of these conditions is inherent in any defense to this threat.

— We then describe a design goal that, if met, would mitigate the threat—and fully defend against it, under the first condition (§3.4.1).

— We next give a set of designs that, under the second condition, meet the goal (§3.4.2–§3.4.5).

— We describe our implementation and our evaluation of that implementation; our main finding is that the implementation roughly meets the goal (§3.7–§3.8).

— At this point, having shown that speak-up "meets its spec", we consider whether it is an appropriate choice: we compare speak-up to alternatives, critique it, and reflect on the plausibility of the threat itself (§3.9–§3.10).

With respect to this plausibility, one might well wonder how often application-level attacks happen today and whether they are in fact difficult to filter. We answer this question in §3.10: according to anecdote, current application-level attacks happen today, but they are primitive. However, in evaluating the need for speak-up, we believe that the right questions are not about how often the threat *has* happened but rather about whether the

threat *could* happen. (And it can.) Simply put, prudence requires proactivity. We need to consider weaknesses before they are exploited.

At the end of the chapter (§3.11), we depart from this specific threat, in two ways. First, we observe that one may, in practice, be able to combine speak-up with other defenses. Second, we mention other attacks, besides application-level denial-of-service, that could call for speak-up.

3.1 HIGH LEVEL EXPLANATION

Speak-up is motivated by a simple observation about bad clients: they send requests to victimized servers at much higher rates than legitimate clients do. (This observation has also been made by many others, including the authors of profiling and detection methods. Indeed, if bad clients *weren't* sending at higher rates, then, as long as their numbers didn't dominate the number of good clients, modest over-provisioning of the server would address the attack.)

At the same time, *some* limiting factor must prevent bad clients from sending even more requests. We posit that in many cases this limiting factor is bandwidth. The specific constraint could be a physical limit (e.g., access link capacity) or a threshold above which the attacker fears detection by profiling tools at the server or by the human owner of the "botted" host. For now, we assume that bad clients exhaust all of their available bandwidth on spurious requests. In contrast, good clients, which spend substantial time quiescent, are likely using a only small portion of their available bandwidth. The key idea of speak-up is to exploit this difference, as we now explain with a simple illustration.

Illustration. Imagine a simple request-response server, where each request is cheap for clients to issue, is expensive to serve, and consumes the same quantity of server resources. Real-world examples include Web servers receiving single-packet requests, DNS (Domain Name System) front-ends such as those used by content distribution networks or infrastructures like CoDoNS [127], and AFS (Andrew File System) servers. Suppose that the server has the capacity to handle c requests per second and that the aggregate demand from good clients is g requests per second, $g < c$. Assume that when the server is overloaded it randomly drops excess requests. If the attackers consume all of their aggregate upload bandwidth, B (which for now we express in requests per second) in attacking the server, and if $g + B > c$, then the good clients receive only a fraction $\frac{g}{g+B}$ of the server's

FIGURE 3.1—Illustration of speak-up. The figure depicts an attacked server, $B + g > c$. In (a), the server is not defended. In (b), the good clients send a much higher volume of traffic, thereby capturing much more of the server than before. The good clients' traffic is black, as is the portion of the server that they capture.

resources. Assuming $B \gg g$ (if $B \approx g$, then over-provisioning by moderately increasing c would ensure $g + B < c$, thereby handling the attack), the bulk of the server goes to the attacking clients. This situation is depicted in Figure 3.1(a).

In this situation, current defenses would try to slow down the bad clients. But what if, instead, we arranged things so that when the server is under attack *good clients send requests at the same rates as bad clients*? Of course, the server does not know which clients are good, but the bad clients have already "maxed out" their bandwidth (as assumed above). So if the server encouraged *all* clients to use up their bandwidth, it could speed up the good ones without telling apart good and bad. Doing so would certainly inflate the traffic into the server during an attack. But it would also cause the good clients to be much better represented in the mix of traffic, giving them much more of the server's attention and the attackers much less. If the good clients have total bandwidth G, they would now capture a fraction $\frac{G}{G+B}$ of the server's resources, as depicted in Figure 3.1(b). Since $G \gg g$, this fraction is much larger than before.

In §3.4, we make the preceding under-specified illustration practical. Before doing so, we answer several nagging questions (§3.2) and then characterize the threat that calls for speak-up (§3.3).

3.2 FIVE QUESTIONS

How much aggregate bandwidth does the legitimate clientele need for speak-up to be effective? Speak-up helps good clients, no matter how much bandwidth they have. Speak-up either ensures that the good clients get all the service they need or increases the service they get (compared to an attack

without speak-up) by the ratio of their available bandwidth to their current usage, which we expect to be very high. Moreover, as with many security measures, speak-up "raises the bar" for attackers: to inflict the same level of service-denial on a speak-up defended site, a much larger botnet—perhaps several orders of magnitude larger—is required. Similarly, the amount of over-provisioning needed at a site defended by speak-up is much less than what a non-defended site would need.

Thanks for the sales pitch, but what we meant was: how much aggregate bandwidth does the legitimate clientele need for speak-up to leave them unharmed by an attack? The answer depends on the server's spare capacity (i.e., 1−utilization) when not under attack. Speak-up's goal is to allocate resources in proportion to the bandwidths of requesting clients. If this goal is met, then for a server with spare capacity 50%, the legitimate clients can retain full service if they have the same aggregate bandwidth as the attacking clients (see §3.4.1). For a server with spare capacity 90%, the legitimate clientele needs only 1/9th of the aggregate bandwidth of the attacking clients. In §3.10.2, we elaborate on this point and discuss it in the context of today's botnet sizes.

Then couldn't small Web sites, even if defended by speak-up, still be harmed? Yes. There have been reports of large botnets [24, 40, 73, 75, 104, 153]. If attacked by such a botnet, a speak-up-defended site would need a large clientele or vast over-provisioning to withstand attack fully. However, most botnets are much smaller, as we discuss in §3.10.2. Moreover, as stated in §1.2, *every* defense has this "problem": no defense can work against a huge population of bad clients, if the good and bad clients are indistinguishable.

Because bandwidth is in part a communal resource, doesn't the encouragement to send more traffic damage the network? We first observe that speak-up inflates traffic only to servers currently under attack—a very small fraction of all servers—so the increase in total traffic will be minimal. Moreover, the "core" appears to be heavily over-provisioned (see, e.g., [54]), so it could absorb such an increase. (Of course, this over-provisioning could change over time, for example with fiber in homes.) Finally, speak-up's additional traffic is congestion-controlled and will share fairly with other traffic. We address this question more fully in §3.5.

Couldn't speak-up "edge out" other network activity at the user's access link, thereby introducing an opportunity cost? Yes. When triggered, speak-up *may* be a heavy network consumer. Some users will not mind this fact. Others will, and they can avoid speak-up's opportunity cost by leaving the attacked service (see §3.9.1 for further discussion of this point).

3.3 THREAT MODEL & APPLICABILITY CONDITIONS

The preceding section gave a general picture of speak-up's applicability. We now give a more precise description. We begin with the threat model, which is a concrete version of the abstract problem statement in §1.1, and then state the conditions that are required for speak-up to be most effective.

Speak-up aims to protect a *server*, defined as any network-accessible service with scarce computational resources (disks, CPUs, RAM, application licenses, file descriptors, etc.), from an *attacker*, defined as an entity (human or organization) that is trying to deplete those resources with legitimate-looking requests (database queries, HTTP requests, etc.) As mentioned in Chapter 1, such an assault is called an application-level attack [73]. The clientele of the server is neither pre-defined (otherwise the server can install filters to permit traffic only from known clients) nor exclusively human (ruling out proof-of-humanity tests [61, 82, 109, 113, 151, 166]).

Each attacker sends traffic from many compromised hosts, and this traffic obeys all protocols, so the server has no easy way to tell from a single request that it was issued with ill intent. Moreover, as mentioned in Chapter 2, the Internet has no robust notion of host identity. Since a determined attacker can repeatedly request service from a site while pretending to have different IP addresses, we assume that an abusively heavy client of a site will not always be identifiable as such.

Most services handle requests of varying difficulty (e.g., database queries with very different completion times). While servers may not be able to determine the difficulty of a request *a priori*, our threat model presumes that the attacker *can* send difficult requests intentionally.

We are not considering link attacks. We assume that the server's access links are not flooded; see condition C2 below.

The canonical example of a service that is threatened by the attack just described is a Web server for which requests are computationally intensive, perhaps because they involve back-end database transactions or searches (e.g., sites with search engines, travel sites, and automatic update services

for desktop software). Often, the clientele of these sites is partially or all non-human. Beyond these server applications, other vulnerable services include the capability allocators in network architectures such as TVA [178] and SIFF [177].[2]

There are many types of Internet services, with varying defensive requirements; speak-up is not appropriate for all of them. For speak-up to fully defend against the threat modeled above, the following two conditions must hold:

C1 **Adequate client bandwidth.** To be unharmed during an attack, the good clients must have in total roughly the same order of magnitude (or more) bandwidth than the attacking clients. This condition is fundamental to any defense to the threat above, in which good and bad clients are indistinguishable: as discussed in §1.2, if the bad population vastly outnumbers the good population, then *no* defense works.

C2 **Adequate link bandwidth.** The protected service needs enough link bandwidth to handle the incoming request stream (and this stream will be inflated by speak-up). This condition is one of the main costs of speak-up, relative to other defenses. However, we do not believe that it is insurmountable. First, observe that *most Web sites use far less inbound bandwidth than outbound bandwidth* (most Web requests are small, yet generate big replies).[3] Thus, the inbound request stream to a server could inflate by many multiples before the inbound bandwidth equals the outbound bandwidth. Second, if that headroom is not enough, then servers can satisfy the condition in various ways. Options include a permanent high-bandwidth access link, co-location at a data center, or temporarily acquiring more bandwidth using services like [25, 120]. A further option, which we expect to be the most common deployment scenario, is ISPs—which of course have significant bandwidth—offering speak-up as a service (just as they do with other DDoS defenses

[2] Such systems are intended to defend against DoS attacks. Briefly, they work as follows: to gain access to a protected server, clients request tokens, or capabilities, from an allocator. The allocator meters its replies (for example, to match the server's capacity). Then, routers pass traffic only from clients with valid capabilities, thereby protecting the server from overload. In such systems, the capability allocator itself is vulnerable to attack. See §3.9.3 for more detail.

[3] As one datum, consider Wikimedia, the host of `http://www.wikipedia.org`. According to [174], for the 12 months ending in August, 2007, the organization's outbound bandwidth consumption was six times its inbound. And for the month of August, 2007, Wikimedia's outbound consumption was eight times its inbound.

today), perhaps amortizing the expense over many defended sites, as suggested in [3].

Later in this chapter (§3.10), we reflect on the extent to which this threat is a practical concern and on whether the conditions are reasonable in practice. We also evaluate how speak-up performs when condition C2 isn't met (§3.8.8).

3.4 DESIGN

In this section, we aim to make practical the high-level illustration in §3.1. We use the notation from that section, and we assume that all requests cause equal server work.

We begin with requirements (§3.4.1), then develop two specific ways to realize these requirements (§3.4.2, §3.4.3). We then consider the connections between these approaches as we reflect more generally on the space of design possibilities (§3.4.5). We also consider various attacks (§3.4.4). We revisit our assumptions in §3.5 and describe how speak-up handles heterogeneous requests in §3.6.

3.4.1 Design Goal and Required Mechanisms

Design Goal. As explained at the beginning of this chapter, speak-up's principal goal is to allocate resources to competing clients in proportion to their bandwidths:

> Consider a server that can handle c requests per second. If the good clients make g requests per second in aggregate and have aggregate bandwidth of G requests per second to the server, and if the bad clients have aggregate bandwidth of B requests per second, then the server should process good requests at a rate of $\min(g, \frac{G}{G+B}c)$ requests per second.

If this goal is met, then modest over-provisioning of the server (relative to the legitimate demand) can satisfy the good clients. For if it is met, then satisfying them requires only $\frac{G}{G+B}c \geq g$ (i.e., the piece that the good clients *can* get must exceed their demand). This expression translates to the *idealized server provisioning requirement*:

$$c \geq g\left(1 + \frac{B}{G}\right) \stackrel{\text{def}}{=} c_{id},$$

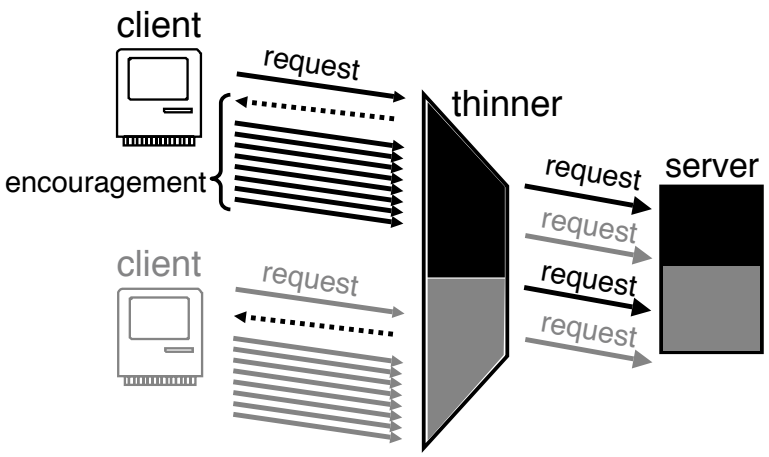


FIGURE 3.2—The components of speak-up. The thinner rate-limits requests to the server. The encouragement signal is depicted by a dashed line from thinner to client (the particular encouragement mechanism is unspecified). In this example, there are two clients, and they send equal amounts of traffic. Thus, the thinner's proportional allocation mechanism (unspecified in this figure) admits requests so that each client claims half of the server.

which says that the server must be able to handle the "good" demand (g) and diminished demand from the bad clients ($B\frac{g}{G}$). For example, if $B = G$ (a special case of condition C1 in §3.3), then the required over-provisioning is a factor of two ($c \geq 2g$). In practice, speak-up cannot exactly achieve this ideal because limited cheating is possible. We analyze this effect in §3.4.4.

Required Mechanisms. Any practical realization of speak-up needs three mechanisms. The first is a way to limit requests to the server to c per second. However, rate-limiting alone will not change the server's allocation to good and bad clients. Since the design goal is that this allocation reflect available bandwidth, speak-up also needs a mechanism to reveal that bandwidth: speak-up must perform *encouragement,* which we define as *causing a client to send more traffic—potentially much more—for a single request than it would if the server were not under attack*. Third, given the incoming bandwidths, speak-up needs a *proportional allocation* mechanism to admit clients at rates proportional to their delivered bandwidths.

Under speak-up, these mechanisms are implemented by a front-end to the server, called the *thinner*. As depicted in Figure 3.2, the thinner implements encouragement and controls which requests the server sees. Encouragement and proportional allocation can each take several forms, as we will see in the two variations of speak-up below (§3.4.2, §3.4.3). And we will see

in §3.4.5 that the choices of encouragement mechanism and proportional allocation mechanism are orthogonal.

Before presenting these specifics, we observe that today when a server is overloaded and fails to respond to a request, a client typically times out and retries—thereby generating more traffic than if the server were unloaded. However, the bandwidth increase is small (since today's timeouts are long). In contrast, encouragement (which is initiated by an agent of the *server*) will cause good clients to send significantly more traffic—while still obeying congestion control.

3.4.2 Aggressive Retries and Random Drops

In the version of speak-up that we now describe, the thinner implements proportional allocation by dropping requests at random to reduce the rate to c. To implement encouragement, the thinner, for each request that it drops, immediately asks the client to retry. This synchronous *please-retry* signal causes the *good* clients—the bad ones are already "maxed out"—to retry at far higher rates than they would under silent dropping. (Silent dropping happens in many applications and in effect says, "please try again later", whereas the thinner says, "please try again *now*".)

With the scheme as presented thus far, a good client sends only one packet per round-trip time (RTT) while a bad client can keep many requests outstanding, thereby manufacturing an advantage. To avoid this problem, we modify the scheme as follows: without waiting for explicit please-retry signals, the clients send repeated retries in a congestion-controlled stream. Here, the feedback used by the congestion control protocol functions as implicit please-retry signals. This modification allows all clients to pipeline their requests and keep their pipe to the thinner full.

One might ask, "To solve the same problem, why not enforce one outstanding retry per client?" or, "Why not dispense with retries, queue clients' requests, and serve the oldest?" The answer is that clients are not identifiable: with address hijacking, discussed in Chapter 2, one client may claim to be several, and with NATs and proxies, several clients (which may individually have plenty of bandwidth) may appear to be one. Thus, the thinner can enforce neither one outstanding retry per "client" nor any other quota scheme that needs to *identify* clients. Ironically, *taxing* clients is easier than identifying them: the continuous stream of bytes that clients are asked to send ensures that each is charged individually.

Indeed, speak-up is a currency-based scheme (as we said earlier), and the price for access is the average number of retries, r, that a client must

send. Observe that the thinner does not communicate r to clients: good clients keep resending until they get through (or give up). Also, r automatically changes with the attack size, as can be seen from the expressions for r, derived below.

This approach fulfills the design goal in §3.4.1, as we now show. The thinner admits incoming requests with some probability p to make the total load reaching the server be c. There are two cases. Either the good clients cannot afford the price, in which case they exhaust all of their bandwidth and do not get service at rate g, or they can afford the price, in which case they send retries until getting through. In both cases, the price, r, is $1/p$. In the first case, a load of $B + G$ enters the thinner, so $p = \frac{c}{B+G}$, $r = \frac{B+G}{c}$, and the good clients can pay for $G/r = \frac{G}{G+B}c$ requests per second. In the second case, the good clients get service at rate g, as required, and $r = B/(c - g)$ (as we show immediately below). Note that in both cases r changes with the attack size, B.

To see that $r = B/(c-g)$ in the second case, observe that the "bad load" that actually reaches the server reduces from B, the attackers' full budget, to Bp. Thus, the thinner's dropping, combined with the fact that good clients retry their "good load" of g until getting through, results in the equation $g + Bp = c$, which implies $r = 1/p = B/(c - g)$.

3.4.3 Explicit Payment Channel and Virtual Auction

We now describe another encouragement mechanism and another proportional allocation mechanism; we use these mechanisms in our implementation and evaluation. They are depicted in Figure 3.3. For encouragement, the thinner does the following. When the server is oversubscribed, the thinner asks a requesting client to open a separate *payment channel*. The client then sends a congestion-controlled stream of bits on this channel. (Conceptually, the client is padding dummy bytes to its request.) We call a client that is sending bits a *contending* client; the thinner tracks how many bits each contending client sends.

The proportional allocation mechanism is as follows. Assume that the server notifies the thinner when it is ready for a new request. When the thinner receives such a notification, it holds a *virtual auction*: it admits to the server the contending client that has sent the most bits, and it terminates the corresponding payment channel.

As with the version in §3.4.2, the price here emerges naturally. Here, it is expressed in bits per request. The "going rate" for access is the winning bid from the most recent auction. We now consider the average price. Here,

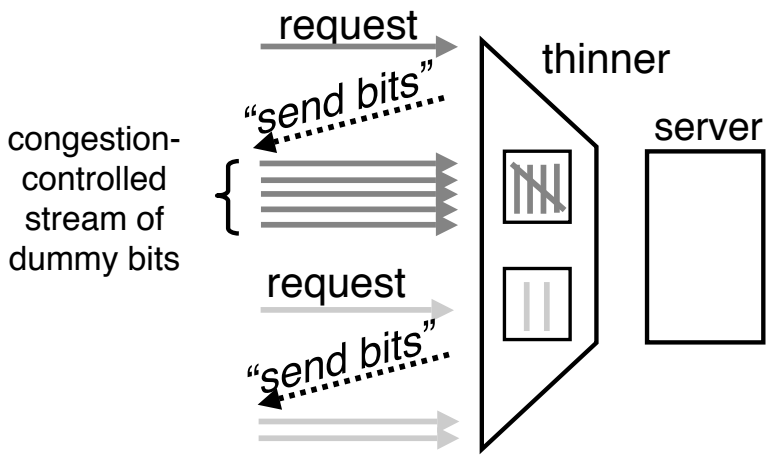


FIGURE 3.3—Speak-up with an explicit payment channel. For each request that arrives when the server is busy, the thinner asks the requesting client to send dummy bits. Imagine that an auction is about to happen. The dark gray request will win the auction because it has five units of payment associated with it, compared to only two units for the light gray request.

we express B and G in bits (not requests) per second and assume that the good and bad clients are "spending everything", so $B + G$ bits per second enter the thinner. Since auctions happen every $1/c$ seconds on average, the average price is $\frac{B+G}{c}$ bits per request.

However, we cannot claim, as in §3.4.2, that good clients get $\frac{G}{G+B}c$ requests served per second: the auction might allow "gaming" in which adversaries consistently pay a lower-than-average price, forcing good clients to pay a higher-than-average price. In the next section, we show that the auction can be gamed but not too badly, so all clients do in fact see prices that are close to the average.

3.4.4 Cheating and the Virtual Auction

In considering the robustness of the virtual auction mechanism, we begin with a theorem and then describe how practice may be both worse and better than this theory. The theorem is based on one simplifying assumption: that requests are served with perfect regularity (i.e., every $1/c$ seconds).

Theorem 3.1 In a system with regular service intervals, any client that continuously transmits an α fraction of the average bandwidth received by the thinner gets at least an $\alpha/2$ fraction of the service, regardless of how the bad clients time or divide up their bandwidth.

Proof: Consider a client, X, that transmits an α fraction of the average bandwidth. The intuition is that to keep X from winning auctions, the other clients must deliver substantial payment.

Because our claims are purely about proportions, we choose units to keep the discussion simple. We call the amount of bandwidth that X delivers between every pair of auctions a *dollar*. Suppose that X must wait t auctions before winning k auctions. Let t_1 be the number of auctions that occur until (and including) X's first win, t_2 the number that occur after that until and including X's second win, and so on. Thus, $\sum_{i=1}^{k} t_i = t$. Since X does not win until auction number t_1, X is defeated in the previous auctions. In the first auction, X has delivered 1 dollar, so at least 1 dollar is spent to defeat it; in the next auction 2 dollars are needed to defeat it, and so on until the $(t_1 - 1)^{st}$ auction when $t_1 - 1$ dollars are spent to defeat it. So $1 + 2 + \cdots + (t_1 - 1) = t_1(t_1 - 1)/2$ dollars are spent to defeat X before it wins. More generally, the total dollars spent by other clients over the t auctions is at least

$$\sum_{i=1}^{k} \frac{t_i^2 - t_i}{2} = \sum_{i=1}^{k} \frac{t_i^2}{2} - \frac{t}{2}.$$

This sum is minimized, subject to $\sum t_i = t$, when all the t_i are equal, namely $t_i = t/k$. We conclude that the total spent by the other clients is at least

$$\sum_{i=1}^{k} \frac{t^2}{2k^2} - \frac{t}{2} = \frac{t^2}{2k} - \frac{t}{2}.$$

Adding the t dollars spent by X, the total number of dollars spent is at least

$$\frac{t^2}{2k} + \frac{t}{2}.$$

Thus, recalling that α is what we called the *fraction* of the total spent by X, we get

$$\alpha \leq \frac{2}{(t/k + 1)}.$$

It follows that

$$\frac{k}{t} \geq \frac{\alpha}{2 - \alpha} \geq \frac{\alpha}{2},$$

i.e., X receives at least an $\alpha/2$ fraction of the service. □

Observe that this analysis holds for each good client separately. It follows that if the good clients deliver *in aggregate* an α fraction of the bandwidth, then *in aggregate* they will receive an $\alpha/2$ fraction of the service. Note that this claim remains true regardless of the service rate c, which need not be known to carry out the auction.

Theory versus practice. We now consider ways in which the above theorem is both weaker and stronger than what we expect to see in practice. We begin with weaknesses. First, consider the unreasonable assumption that requests are served with perfect regularity. To relax this assumption, the theorem can be extended as follows: for service times that fluctuate within a bounded range $[(1-\delta)/c\,,(1+\delta)/c\,]$, X receives at least a $(1-2\delta)\alpha/2$ fraction of the service. However, even this looser restriction may be unrealistic in practice. And pathological service timings violate the theorem. For example, if many request fulfillments are bunched in a tiny interval during which X has not yet paid much, bad clients can cheaply outbid it during this interval, *if* they know that the pathology is happening and are able to time their bids. But doing so requires implausibly deep information.

Second, the theorem assumes that a good client "pays bits" at a constant rate given by its bandwidth. However, the payment channel in our implementation runs over TCP, and TCP's slow start means that a good client's rate must grow. Moreover, because we implement the payment channel as a *series* of large HTTP POSTs (see §3.7), there is a quiescent period between POSTs (equal to one RTT between client and thinner) as well as TCP's slow start for each POST. Nevertheless, we can extend the analysis to capture this behavior and again derive a lower bound for the fraction of service that a given good client receives. The result is that if the good client has a small fraction of the total bandwidth (causing it to spend a lot of time paying), and if the HTTP POST is big compared to the bandwidth-delay product, then the client's fraction of service is not noticeably affected (because the quiescent periods are negligible relative to the time spent paying at full rate).

We now consider the strength of the theorem: it makes no assumptions at all about adversarial behavior. We believe that in practice adversaries will attack the auction by opening many concurrent TCP connections to avoid quiescent periods, but the theorem handles every other case too. The adversary can open few or many TCP connections, disregard TCP semantics, or send continuously or in bursts. The only parameter in the theorem is the total number of bits sent in a given interval by other clients.

The theorem does cede the adversary an extra factor of two "advantage" in bandwidth (the good client sees only $\alpha/2$ service for α bandwidth). This advantage arises because the proof lets the adversary control exactly when its bits arrive—sending fewer when the good client's bid is small and more as the bid grows. This ability is powerful indeed—most likely stronger than real adversaries have. Nevertheless, even with this highly pessimistic assumption about adversarial abilities, speak-up can still do its job: the re-

quired provisioning has only increased by a factor of two over the ideal from §3.4.1, and this provisioning is still far less than what would be required to absorb the attack without speak-up.

To see that the required provisioning increases by a factor of two, observe that the theorem says that good clients can get service of up to $c\alpha/2 = \frac{Gc}{2(G+B)}$ requests per second. Yet good clients need service of g requests per second. Thus, the required provisioning, which we denote c_{req}, must satisfy $\frac{Gc_{req}}{2(G+B)} \geq g$. This inequality yields $c_{req} \geq 2c_{id}$.

In §3.8.4, we quantify the adversarial advantage in our experiments by determining how the factors mentioned in this section—quiescent periods for good clients, bad clients opening concurrent connections—affect the required provisioning above the ideal.

3.4.5 Design Space

We now reflect on the possible designs for speak-up and then discuss how we chose which one to implement and evaluate.

Axes

We have so far presented two designs: "aggressive retries and random drops" (§3.4.2) and "payment channel and virtual auction" (§3.4.3). These designs are drawn from a larger space, in which there are two orthogonal axes that correspond to the required mechanisms from §3.4.1:

A1 **Encouragement method**:	A2 **Allocation mechanism**:
— Aggressive retries	— Random drops
— Payment channel	— Virtual auction

Thus, it is possible to imagine two other designs. We discuss them now.

"Retries and Virtual Auction"

In this design, clients send repeated retries in-band on a congestion-controlled stream. The thinner conducts a periodic auction, selecting as the winner the request with the most retries (rather than the most bits) up to that point. We can apply Theorem 3.1 to this design, as follows. The theorem describes clients' bandwidths in terms of a made-up unit (the "dollar"), so we need only take this unit to be retries between auctions, rather than bits between auctions.

"Payment Channel and Random Drops"

In this design,[4] clients pay bits out of band. As in the "virtual auction" designs, the thinner divides time into service intervals (i.e., time lengths of $1/c$ seconds), making an admission decision at the end of each interval. In this design, however, the intervals are independent. For a given interval, the thinner records how many bits clients have sent in that interval. At the end of an interval, the thinner chooses randomly. Specifically, a request that has sent a fraction f of the total bits *in that interval* is admitted by the thinner with probability f. To show that this design achieves our goal, we use the following theorem; like Theorem 3.1, it relies on the assumption that requests are served with perfect regularity.

Theorem 3.2 Under this design, any client that continuously delivers a fraction α of the average bandwidth received by the thinner gets a fraction α of service, in expectation, regardless of how the other clients time or divide up their bandwidth.

Proof: As in Theorem 3.1, consider a single client, X, and again assume that X delivers a *dollar* between service intervals. We will examine what happens over t time intervals. Over this period, X delivers t dollars of bandwidth. We are given that all of the clients together deliver t/α dollars over this period, so the other clients deliver $t/\alpha - t$ dollars.

Now, consider each of the i intervals. In each interval, the service that X expects is the same as the probability that it is admitted, which is $1/(1+b_i)$, where b_i is the bandwidth delivered by the other clients in interval i. By linearity of expectation, the total expected service received by X is

$$\sum_{i=1}^{t} \frac{1}{1+b_i} \quad \text{subject to} \quad \sum_{i=1}^{t} b_i = \frac{t}{\alpha} - t.$$

The minimum—which is the worst case for X—happens when the b_i are equal to each other, i.e., $b_i = 1/\alpha - 1$. In that case, the expected service received by X is

$$\sum_{i=1}^{t} \frac{1}{1+1/\alpha-1} = \alpha t,$$

so X can expect at least an α fraction of the total service. □

[4] A good question by Jeff Erickson caused us to think of this approach.

An advantage of this approach is that the thinner need not keep track of how many bits have been sent on behalf of each request, as we now explain. (Of course, depending on the scenario, the thinner may still need per-request state, such as congestion control state or other connection state.) We can regard the thinner's tasks as (a) receiving a stream of bits in each interval (each packet brings a set of bits on behalf of a request); (b) at the end of the interval, choosing a bit uniformly at random; and (c) admitting the request on whose behalf the "winning bit" arrived. These tasks correspond to admitting a request with probability proportional to the number of bits that were sent on behalf of that request in the given interval.

To implement these tasks, the thinner can use *reservoir sampling* [85] with a reservoir of one bit (and its associated request). Reservoir sampling takes as input a stream of unknown size and flips an appropriately weighted coin to make a "keep-or-drop" decision for each item in the stream (a "keep" decision evicts a previously kept item). The weights on the coins ensure that, when the stream ends, the algorithm will have chosen a uniformly random sample of size k (in our case, $k = 1$). A further refinement avoids making a decision for each item (or bit, in our context): once the algorithm keeps an item, it chooses a random value representing the *next* item to admit; it can then discard all intermediate items with a clear conscience [165].

Comparing the Possibilities

Before we actually compare the alternatives, observe that one of the designs is under-specified: in the description of "aggressive retries and random drops" in §3.4.2, we did not say how to set p, the drop probability. However, the design and theorem above suggest one way to do so: the thinner divides up time and selects one client to "win" each interval (rather than trying to apply a drop probability to each retry independently such that the total rate of admitted requests is c). With this method of setting p, every design in this space shares the same high-level structure: clients pipeline bits or requests, and the thinner selects a client once every $1/c$ seconds. The designs are thus directly comparable.

The differences among the designs are as follows. Axis A1 is primarily an implementation distinction, and which choice is appropriate depends on the protected application and on how speak-up fits into the communication protocol between clients and servers.

Axis A2 is more substantive. Here, we have a classic trade-off between random and deterministic algorithms. The "virtual auction" is gameable in a limited way, but clients' waiting times are bounded: once a client has paid

enough, it is served. "Random drops" is the opposite: it is not at all gameable, but our claims about it apply only to long-term averages. At short time scales, there will be significant variation in the server's allocation and thus in waiting times. In particular, a typical coupon-collector analysis shows that if there are n equivalent clients that continually make requests, some of the clients will have to wait an expected $O(n \log n)$ intervals to get service. (Regard each interval as picking a client uniformly at random.) Another difference between the two options is that one can implement "random drops" with less state at the thinner (by using reservoir sampling, as described above). Which choice on axis 2 is appropriate depends on one's goals and taste.

Rationale for our choices. For our prototype (§3.7), we chose the payment channel over in-band retries for reasons related to how JavaScript drives Web browsers. We chose the virtual auction over random drops because we wanted to avoid variance.

Other designs. One might wonder whether the design space is larger than these four possibilities, which share a similar structure. Indeed, we used to be enamored of a different structure, namely the version of "random drops" described at the beginning of §3.4.2. The charm of that version was that its thinner was stateless. However, we ultimately rejected that approach because, as described in §3.4.2, clients need to keep their pipes to the thinner full (otherwise, recall, bad clients could manufacture a bandwidth advantage). This requirement implies that the thinner must maintain congestion control state for each client, ending the dream of a stateless thinner.

3.5 REVISITING ASSUMPTIONS

We have so far made a number of assumptions. Below we address four of them in turn: that aside from end-hosts' access links, the Internet has infinite capacity; that no bottleneck link is shared (which is a special case of the first assumption, but we address them separately); that the thinner has infinite capacity; and that bad clients consume all of their upload bandwidth when they attack. In the next section, we relax the assumption of equal server requests.

3.5.1 Speak-up's Effect on the Network

No flow between a good client and a thinner individually exhibits anti-social behavior. In our implementation (see §3.7), each payment channel comprises a series of HTTP POSTs and thus inherits TCP's congestion control. For UDP applications, the payment channel could use the congestion manager [14] or DCCP [87]. (Bad clients can refuse to control congestion, but this behavior is a link attack, which speak-up does not defend against; see §3.3.) However, individually courteous flows do not automatically excuse the larger rudeness of increased traffic levels, and we must ask whether the network can handle this increase.

To answer this question, we give two sketchy arguments suggesting that speak-up would not increase total traffic much, and then consider the effect of such increases. First, speak-up inflates *upload* bandwidth, and, despite the popularity of peer-to-peer file-sharing, most bytes still flow in the *download* direction [54]. Thus, inflating upload traffic even to the level of download traffic would cause an inflation factor of at most two. Second, only a very small fraction of servers is under attack at any given time. Thus, even if speak-up did increase the traffic to each attacked site by an order of magnitude, the increase in overall Internet traffic would still be small.

Whatever the overall traffic increase, it is unlikely to be problematic for the Internet "core": both anecdotes from network operators and measurements [54] suggest that these links operate at low utilization. And, while the core cannot handle *every* client transmitting maximally (as argued in [164]), we expect that the fraction of clients doing so at any time will be small—again, because few sites will be attacked at any time. Speak-up will, however, create contention at bottleneck links (as will any heavy user of the network), an effect that we explore experimentally in §3.8.7.

3.5.2 Shared Links

We now consider what happens when clients that share a bottleneck link are simultaneously encouraged by the thinner. For simplicity, assume two clients behind bottleneck link l; the discussion generalizes to more clients. If the clients are both good, their individual flows roughly share l, so they get roughly the same piece of the server. Each may be disadvantaged compared to clients that are not similarly bottlenecked, but neither is disadvantaged relative to the other.

If, however, one of the clients is bad, then the good client has a problem: the bad client can open n parallel TCP connections (§3.4.4), claim roughly

an $n/(n+1)$ fraction of l's bandwidth, and get a much larger piece of the server. While this outcome is unfortunate for the good client, observe, first, that the *server* is still protected (the bad client can "spend" at most l). Second, while the thinner's encouragement might instigate the bad client, the fact is that when a good and bad client share a bottleneck link—speak-up or no—the good client loses: the bad client can always deny service to the good client. We experimentally investigate such sharing in §3.8.6.

3.5.3 Provisioning the Thinner

For speak-up to work, the thinner must be uncongested: a congested thinner could not "get the word out" to encourage clients. Thus, the thinner needs enough bandwidth to absorb a full DDoS attack and more (which is condition C2 in §3.3). It also needs enough processing capacity to handle the dummy bits. (Meeting this requirement is far easier than provisioning the *server* to handle the full attack because the thinner does not do much per-request processing.) We now argue that meeting these requirements is plausible.

One study [138] of observed DoS attacks found that the 95th percentile of attack size was in the low hundreds of Mbits/s (see Figure 3.15 in §3.10.2), which agrees with other anecdotes (e.g., [162]). The traffic from speak-up would presumably be multiples larger since the good clients would also send at high rates. However, even with several Gbits/s of traffic in an attack, the thinner's requirements are not insurmountable.

First, providers readily offer links, even temporarily (e.g., [25, 120]), that accommodate these speeds. Such bandwidth is expensive, but co-located servers could share a thinner, or else the ISP could provide the thinner as a service (see condition C2 in §3.3). Second, we consider processing capacity. Our unoptimized software thinner running on commodity hardware can handle 1.5 Gbits/s of traffic and tens or even hundreds of thousands of concurrent clients; see §3.8.1. A production solution would presumably do much better.

3.5.4 Attackers' Constraints

The assumption that bad clients are today "maxing out" their *upload* bandwidth was made for ease of exposition. The required assumption is only that *bad clients consistently make requests at higher rates than legitimate clients*. Specifically, if bad clients are limited by their *download* bandwidth, or they are not maxed out at all today, speak-up is still useful: it *makes* upload band-

width into a constraint by forcing everyone to spend this resource. Since bad clients—even those that aren't maxed out—are more active than good ones, the imposition of this upload bandwidth constraint affects the bad clients more, again changing the mix of the server that goes to the good clients. Our goals and analysis in §3.4 still hold: they are in terms of the bandwidth *available* to both populations, not the bandwidth that they actually *use* today.

3.6 HETEROGENEOUS REQUESTS

We now generalize the design to handle the more realistic case in which the requests are unequal. There are two possibilities: either the thinner can tell the difficulty of a request in advance, or it cannot. In the first case, the design is straightforward: the thinner simply scales a given bandwidth payment by the difficulty of the associated request (causing clients to pay more for harder requests).

In the remainder of this section, we address the second case, making the worst-case assumption that although the thinner does not know the difficulty of requests in advance, attackers do, as given by the threat model in §3.3. If the thinner treated all requests equally (charging, in effect, the average price for any request), an attacker could get a disproportionate share of the server by sending only the hardest requests.

In describing the generalization to the design, we make two assumptions:

— As in the homogeneous case, the server processes only one request at a time. Thus, the "hardness" of a computation is measured by how long it takes to complete. Relaxing this assumption to account for more complicated servers is not difficult, as long as the server implements processor sharing among concurrent requests, but we don't delve into those details here.

— The server exports an interface that allows the thinner to SUSPEND, RESUME, and ABORT requests. (Many transaction managers and application servers support such an interface.)

At a high level, the solution is for the thinner to break time into quanta, to view a request as comprising equal-sized *chunks* that each consume a quantum of the server's attention, and to hold a virtual auction for each quantum. Thus, if a client's request is made of x chunks, the client must win

x auctions for its request to be fully served. The thinner need not know x in advance for any request.

In more detail: rather than terminate the payment channel once the client's request is admitted (as in §3.4.3), the thinner extracts an *on-going* payment until the request completes. Given these on-going payments, the thinner implements the following procedure every τ seconds (τ is the quantum length):

1. Let v be the currently-active request. Let u be the contending request that has paid the most.
2. If u has paid more than v, then SUSPEND v, admit (or RESUME) u, and set u's payment to zero.
3. If v has paid more than u, then let v continue executing but set v's payment to zero (since v has not yet paid for the *next* quantum).
4. Time-out and ABORT any request that has been SUSPENDed for some period (e.g., 30 seconds).

This scheme requires some cooperation from the server. First, the server should not SUSPEND requests that hold critical locks; doing so could cause deadlock. Second, SUSPEND, RESUME, and ABORT should have low overhead.

In general, the approach described in this section could apply to other defenses as well (though, to our knowledge, no one has proposed it). For example, a profiler could allocate quanta based on clients' historical demand rather than on how many bits clients have paid.

3.7 IMPLEMENTATION

We implemented a prototype thinner in C++ as an OKWS [90] Web service using the SFS toolkit [101]. It runs on Linux 2.6. Any JavaScript-capable Web browser can use our system; we have successfully tested our implementation with Firefox, Internet Explorer, Safari, and a custom client that we use in our experiments.

The thinner is designed to be easy to deploy. It is a Web front-end that is intended to run on a separate machine "in front" of the protected Web server (which we will call just the *server* in the remainder of this section). Moreover, there are not many configuration parameters. They are:

— The *capacity* of the protected server, expressed in requests per second.

— A *list of URLs and regular expressions* that correspond to "hard requests". Each URL and regular expression is associated with a *difficulty level*. This difficulty level is relative to the capacity. For example, if the server's capacity is 100 requests per second, and if the thinner is configured such that a given URL has difficulty 2, then the thinner assumes that for the server to handle that request takes an average of .02 seconds.

— The *name or address* of the server.

— A custom *"please wait" screen* that humans will see while the server is working and while their browser is paying bits. Existing computationally intensive Web sites already use such screens.

When the thinner gets a request, it first checks whether that request is on the list of hard requests. If not, it passes the request directly to the server and feeds the response back to the client on behalf of the server.

On the other hand, if the Web client has requested a "hard" URL, the thinner immediately replies with the "please wait" screen. If there are no other connections to the thinner (i.e., if the server is not oversubscribed), then the thinner submits the client's request to the protected server. After the server processes the request and replies to the thinner, the thinner returns to the client (1) JavaScript that wipes the "please wait" screen and (2) the contents of the server's reply.

If, however, other clients are communicating with the thinner (i.e., if the server is oversubscribed), the thinner adds JavaScript after the "please wait" HTML. As depicted in Figure 3.4, this JavaScript causes the client's browser to dynamically construct, and then submit, a one-megabyte HTTP POST containing random bytes. (One megabyte reflects some browsers' limits on POSTs.) This POST is the client's bandwidth payment (§3.4.3). If, while sending these dummy bytes, the client wins an auction (we say below when auctions happen), the thinner terminates the POST and submits the client's request to the server, as above. And, as above, the server then replies to the thinner, the thinner wipes the "please wait" screen, etc.

If the client completes the POST without winning an auction, then the thinner returns JavaScript that causes the browser to send another POST, and the process described in the previous paragraph repeats. The thinner correlates the client's payments with its request via a "request id" field in all HTTP requests. This field could be forged by a client, but such forgery is not

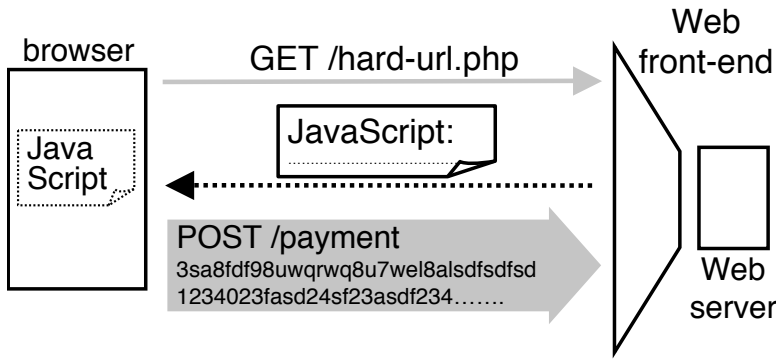


FIGURE 3.4—Implementation of the payment channel. When the server is busy, the thinner, implemented as a Web front-end, sends JavaScript to clients that causes them to send large HTTP POSTs. The thinner ends a POST if and when the client wins the auction.

cause for concern: it amounts to contributing bandwidth to another client or splitting bandwidth over two virtual clients, both of which are behaviors that the design *assumes* of clients (see page 26, §3.4.1, and §3.4.4).

Auctions. The thinner holds auctions (and demands bandwidth payment) whenever it has more than one connection open (this state corresponds to over-subscription of the server). The server does not tell the thinner whether it is free. Rather, the thinner uses the configuration parameters (specifically, the server's capacity and the difficulty of requests) to meter requests to the server in the obvious way: if we assume for a moment that all requests are of unit difficulty, then the thinner holds an auction every $1/c$ seconds. Backing off of this assumption, if a request of difficulty level d has just been admitted to the server, then the thinner will hold the next auction d/c seconds later. To handle difficult requests fairly, the thinner scales clients' payments by the difficulty level, and the auction winner is based on the scaled payments.

* * *

One can configure the thinner to support hundreds of thousands of concurrent connections by setting the maximum number of connection descriptors appropriately. (The thinner evicts old clients as these descriptors deplete.) With modern versions of Linux, the limit on concurrent clients is not per-connection descriptors but rather the RAM consumed by each open connection.

Our thinner implementation allocates a protected server in rough proportion to clients' bandwidths.	§3.8.2, §3.8.5
In our experiments, the server needs to provision only 37% beyond the bandwidth-proportional ideal to serve 99.98% of the good requests.	§3.8.3, §3.8.4
Our unoptimized thinner implementation can sink 1.5 Gbits/s of uploaded "payment traffic".	§3.8.1
On a bottleneck link, speak-up traffic can crowd out other speak-up traffic and non-speak-up traffic.	§3.8.6, §3.8.7
When the thinner has less bandwidth than required (i.e., when condition C2 from §3.3 is not met), speak-up does not achieve a bandwidth-proportional allocation but still yields a better allocation than having no defense.	§3.8.8

TABLE 3.1—Summary of main evaluation results.

3.8 EXPERIMENTAL EVALUATION

To investigate the effectiveness and performance of speak-up, we conducted experiments with our prototype thinner. Our primary question is how the thinner allocates an attacked server to good clients. To answer this question, we begin in §3.8.2 by varying the bandwidth of good (G) and bad (B) clients, and measuring how the server is allocated with and without speak-up. We also measure this allocation with server capacities above and below the ideal in §3.4.1. In §3.8.3, we measure speak-up's latency and byte cost. In §3.8.4, we ask how much bad clients can "cheat" speak-up to get more than a bandwidth-proportional share of the server. §3.8.5 shows how speak-up performs when clients have differing bandwidths and latencies to the thinner. We also explore scenarios in which speak-up traffic shares a bottleneck link with other speak-up traffic (§3.8.6) and with non-speak-up traffic (§3.8.7). Finally, we measure how speak-up performs when the thinner's bandwidth is under-provisioned (§3.8.8); that is, we measure the effect of not meeting condition C2 in §3.3. Table 3.1 summarizes our results.

3.8.1 Setup and Method

All of the experiments described here ran on the Emulab testbed [47]. The clients run a custom Python Web client and connect to the prototype thin-

ner in various emulated topologies. The thinner runs on Emulab's "PC 3000", which has a 3 GHz Xeon processor and 2 GBytes of RAM; the clients are allowed to run on any of Emulab's hardware classes.

The protected server is an Apache Web server that runs on the same host as the thinner. The thinner is configured so that "hard" requests are those to a particular URL, U, that corresponds to a simple PHP script. That PHP script responds to HTTP GET requests for U by first sleeping for $1/c$ seconds and then returning a simple text file. c is the server capacity that we are modeling and varies depending on the experiment. The thinner sends the server requests for U no more often than once every $1/c$ seconds. If a request arrives while the server is still "processing" (really, sleeping on behalf of) a previous one, the thinner replies with JavaScript that makes the client issue a one megabyte HTTP POST—the payment bytes (see §3.7).

All experiments run for 600 seconds. Each client runs on a separate Emulab host and generates *requests* for U. All requests are identical. Each client's requests are driven by a Poisson process of rate λ requests/s. However, a client never allows more than a configurable number w (the window) of outstanding requests. If the stochastic process "fires" when more than w requests are outstanding, the client puts the new request in a backlog queue, which drains when the client receives a response to an earlier request. If a request is in this queue for more than 10 seconds, it times out, and the client logs a service denial.

We use the behavior just described to model both good and bad clients. A bad client, by definition, tries to capture more than its fair share. We model this intent as follows: in our experiments, bad clients send requests faster than good clients, and bad clients send requests concurrently. Specifically, we choose $\lambda = 40$, $w = 20$ for bad clients and $\lambda = 2$, $w = 1$ for good clients.

Our choices of B and G are determined by the number of clients that we are able to run in the testbed and by a rough model of today's client access links. Specifically, in most of our experiments, there are 50 clients, each with 2 Mbits/s of access bandwidth. Thus, $B + G$ usually equals 100 Mbits/s. This scale is smaller than most attacks. Nevertheless, we believe that the results generalize because we focus on how the prototype's behavior differs from the theory in §3.4. By understanding this difference, one can make predictions about speak-up's performance in larger attacks.

Because the experimental scale does not tax the thinner, we separately measured its capacity and found that it can handle loads comparable to recent attacks. At 90% CPU utilization on the hardware described above

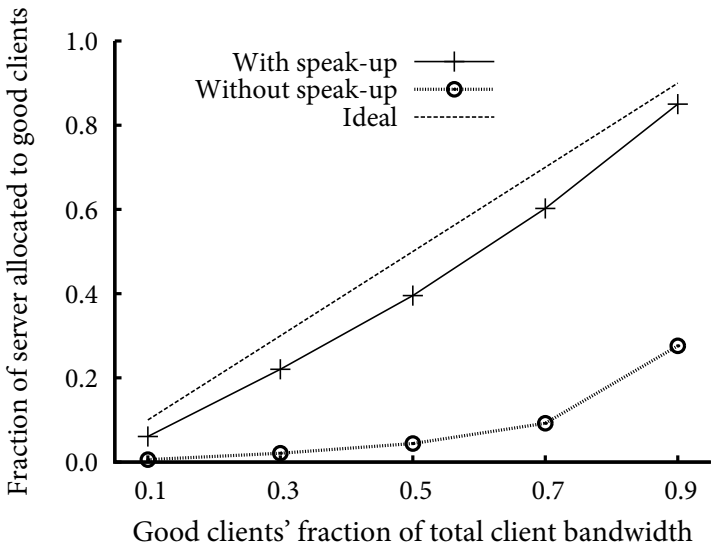


FIGURE 3.5—Server allocation when $c = 100$ requests/s as a function of $\frac{G}{G+B}$. The measured results for speak-up are close to the ideal line. Without speak-up, bad clients sending at $\lambda = 40$ requests/s and $w = 20$ capture much more of the server.

with multiple gigabit Ethernet interfaces, in a 600-second experiment with a time series of 5-second intervals, the thinner sinks payment bytes at 1451 Mbits/s (with standard deviation of 38 Mbits/s) for 1500-byte packets and at 379 Mbits/s (with standard deviation of 24 Mbits/s) for 120-byte packets. Many recent attacks are roughly this size; see §3.5.3 and §3.10.2. The capacity also depends on how many concurrent clients the thinner supports; the limit here is only the RAM for each connection (see §3.7).

3.8.2 Validating the Thinner's Allocation

When the rate of incoming requests exceeds the server's capacity, speak-up's goal is to allocate the server's resources to a group of clients in proportion to their aggregate bandwidth. In this section, we evaluate to what degree our implementation meets this goal.

In our first experiment, 50 clients connect to the thinner over a 100 Mbits/s LAN. Each client has 2 Mbits/s of bandwidth. We vary f, the fraction of "good" clients (the rest are "bad"). In this homogeneous setting, $\frac{G}{G+B}$ (i.e., the fraction of "good client bandwidth") equals f, and the server's capacity is $c = 100$ requests/s.

Figure 3.5 shows the fraction of the server allocated to the good clients as a function of f. Without speak-up, the bad clients capture a larger fraction of the server than the good clients because they make more requests

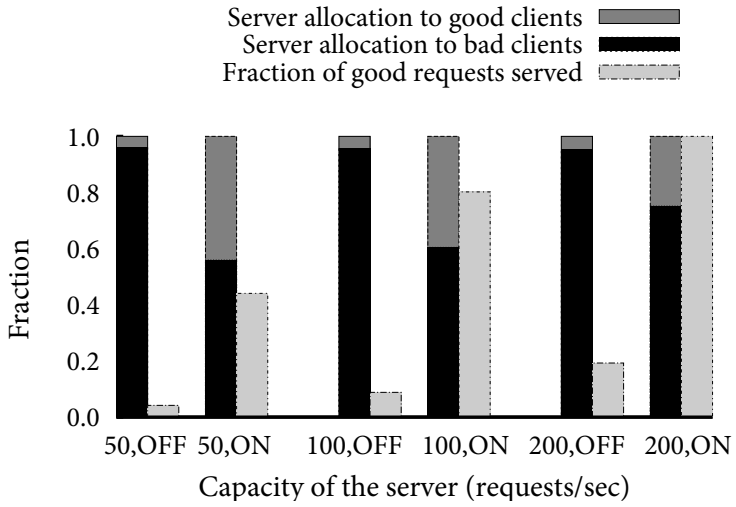


FIGURE 3.6—Server allocation to good and bad clients, and the fraction of good requests that are served, without ("OFF") and with ("ON") speak-up. c varies, and $G = B = 50$ Mbits/s. For $c = 50, 100$, the allocation is roughly proportional to the aggregate bandwidths, and for $c = 200$, all good requests are served.

and the server, when overloaded, randomly drops requests. With speak-up, however, the good clients can "pay" more for each of their requests—because they make fewer—and can thus capture a fraction of the server roughly in proportion to their bandwidth. The small difference between the measured and ideal values is a result of the good clients not using as much of their bandwidth as the bad clients. We discussed this adversarial advantage in §3.4.4 and further quantify it in §3.8.3 and §3.8.4.

In the next experiment, we investigate different "provisioning regimes". We fix G and B, and measure the server's allocation when its capacity, c, is less than, equal to, and greater than c_{id}. Recall from §3.4.1 that c_{id} is the minimum value of c at which all good clients get service, if speak-up is deployed and if speak-up allocates the server exactly in proportion to client bandwidth. We set $G = B$ by configuring 50 clients, 25 good and 25 bad, each with a bandwidth of 2 Mbits/s to the thinner over a LAN. In this scenario, $c_{id} = 100$ requests/s (from §3.4.1, $c_{id} = g(1 + \frac{B}{G}) = 2g = 2 \cdot 25 \cdot \lambda = 100$), and we experiment with $c = 50, 100, 200$ requests/s.

Figure 3.6 shows the results. The good clients get a larger fraction of the server with speak-up than without. Moreover, for $c = 50, 100$, the allocation under speak-up is roughly proportional to the aggregate bandwidths, and for $c = 200$, all good requests are served. (The bad clients get a larger

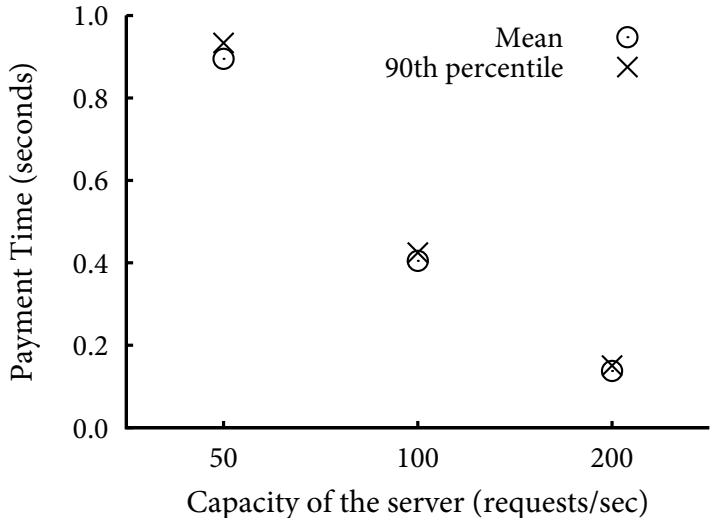


FIGURE 3.7—Mean time to upload dummy bytes for good requests that receive service. c varies, and $G = B = 50$ Mbits/s. When the server is not overloaded ($c = 200$), speak-up introduces little latency.

share of the server for $c = 200$ because they capture the excess capacity after the good requests have been served.) Again, one can see that the allocation under speak-up does not exactly match the ideal: from Figure 3.6, when speak-up is enabled and $c = c_{id} = 100$, the good demand is not fully satisfied.

3.8.3 Latency and Byte Cost

We now explore speak-up's byte cost and a pessimistic estimate of its latency cost for the same set of experiments (c varies, 50 clients, $G = B = 50$ Mbits/s).

For the pessimistic latency cost, we measure the length of time that clients spend uploading dummy bytes, as seen at the client. Figure 3.7 shows the averages and 90th percentiles of these measurements for the served good requests. The reasons that this measurement is a pessimistic reflection of speak-up's true latency cost are as follows. First, for the good clients, speak-up decreases average latency (because speak-up serves more good requests). Second, even calling this measurement the *per-request* latency cost is pessimistic because that view unrealistically implies that, *without* speak-up, the "lucky" requests (the ones that receive service) never have to wait. Third, any other resource-based defense would also introduce some latency.

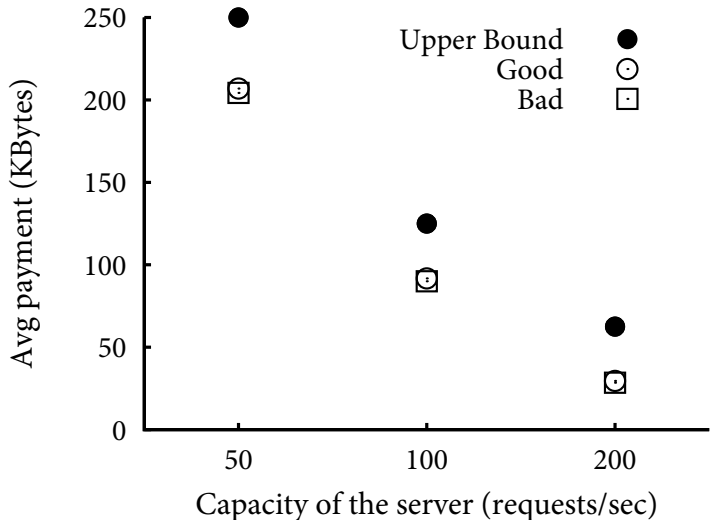


FIGURE 3.8—Average number of bytes sent on the payment channel—the "price"—for served requests. c varies, and $G = B = 50$ Mbits/s. When the server is overloaded ($c = 50, 100$), the price is close to the upper bound, $(G+B)/c$; see the text for why they are not equal.

For the byte cost, we measure the number of bytes uploaded for served requests—the "price"—as recorded by the thinner. Figure 3.8 shows the average of this measurement for good and bad clients and also plots the theoretical average price, $(G + B)/c$, from §3.4.3, which is labeled "Upper Bound".

We make two observations about this data. The first is that when the server is under-provisioned, good clients pay slightly more for service than bad ones. The reason is as follows. *All* contending clients tend to overpay: the client that will win the next auction continues to pay until the auction happens rather than stopping after it has paid enough to win. And since good clients pay at a faster rate *per request*, they tend to overshoot the "true" price (the second-highest bid) more than the bad clients do. Note, however, that the overpayment by any client is bounded by $\frac{1}{c}\cdot$ (bandwidth of a client) because a client can overpay for at most the time between two auctions.

The second observation is that the actual price is lower than the theoretical one. The reason is that clients do not consume all of their bandwidth. We now explain why they do not, considering the different values of c in turn.

For $c = 50$, each good client spends an average of 1.46 Mbits/s (determined by tallying the total bits spent by good clients over the experiment).

This average is less than the 2 Mbits/s access link because of a quiescent period between when a good client issues a request and when the thinner replies, asking for payment. This period is 0.22 seconds on average, owing mostly to a long backlog at the thinner of requests and payment bytes (but a little to round-trip latency). When not in a quiescent period, a good client consumes most of its access link, delivering 1.85 Mbits/s on average, inferred by dividing the average good client payment (Figure 3.8) by the average time spent paying (Figure 3.7). *Bad* clients, in contrast, keep multiple requests outstanding so do not have "down time". For $c = 50$, they spend an average of 1.84 Mbits/s.

The $c = 100$ case is similar to $c = 50$.

We now consider $c = 200$. Recall that the upper bound on price of $(G+B)/c$ is met only if all clients actually pay all of their bandwidth. However, if the server is over-provisioned and all of the good clients' requests are served, as happens for $c = 200$, then the good clients do not pay the maximum that they are able. For each request, a good client pays enough to get service, and then goes away, until the next request. This behavior causes a lower "going rate" for access than is given by the upper bound.

3.8.4 Empirical Adversarial Advantage

As just discussed, bad clients deliver more bytes than good clients in our experiments. As a result of this disparity, the server does not achieve the ideal of a bandwidth-proportional allocation. This effect was visible in §3.8.2.

To better understand this adversarial advantage, we ask, What is the minimum value of c at which all of the good demand is satisfied? To answer this question, we experimented with the same configuration as above ($G = B = 50$ Mbits/s; 50 clients) but for more values of c. We found that at $c = 137$, 99.98% of the good demand is satisfed and that at $c = 140$, all but one of the good clients' 30,157 requests is served. $c = 137$ is 37% more provisioning than c_{id}, the capacity needed under exact proportional allocation. We conclude that a bad client can cheat the proportional allocation mechanism but only to a limited extent—at least under our model of bad behavior.

We now revisit that model. First, we chose $w = 20$ arbitrarily. It might be true that with a smaller or larger value for w, the bad clients could capture more of the server. Second, bad clients do not "optimize". As one example, in the $c = 50$ experiment, the average time between when the thinner returns JavaScript to a bad client and when the thinner actually gets bits

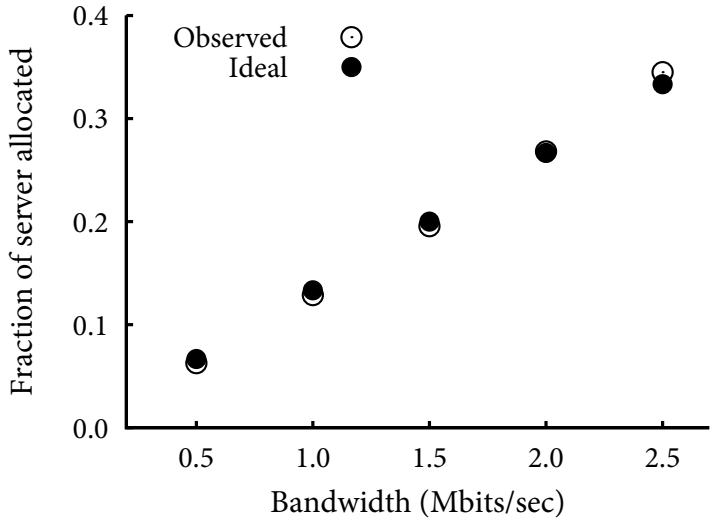


FIGURE 3.9—Heterogeneous client bandwidth experiments with 50 LAN clients, all good. The fraction of the server ($c = 10$ requests/s) allocated to the ten clients in category i, with bandwidth $0.5 \cdot i$ Mbits/s, is close to the ideal proportional allocation.

from that client is roughly two full seconds. During those two seconds, the bad client is effectively paying for $w - 1$ (or fewer requests) rather than w requests, so perhaps bad clients are not realizing their full adversarial advantage. Indeed, one could imagine a bad client setting w adaptively or concentrating bandwidth on particular requests. Nevertheless, the analysis in §3.4.4 shows that bad clients cannot do much better than the naïve behavior that we model.

3.8.5 Heterogeneous Network Conditions

We now investigate the server's allocation for different client bandwidths and RTTs. We begin with bandwidth. We assign 50 clients to 5 categories. The 10 clients in category i ($1 \leq i \leq 5$) have bandwidth $0.5 \cdot i$ Mbits/s and are connected to the thinner over a LAN. All clients are good. The server has capacity $c = 10$ requests/s. Figure 3.9 shows that the resulting server allocation to each category is close to the bandwidth-proportional ideal.

We now consider RTT, hypothesizing that the RTT between a good client and the thinner will affect the allocation, for two reasons. First, at low prices, a client will have sent the full price—that is, the requisite number of bytes to win the virtual auction—before TCP has "ramped up" to fill the client's pipe. In these cases, clients with longer RTTs will take longer to pay. Second, and more importantly, each request has at least one associated

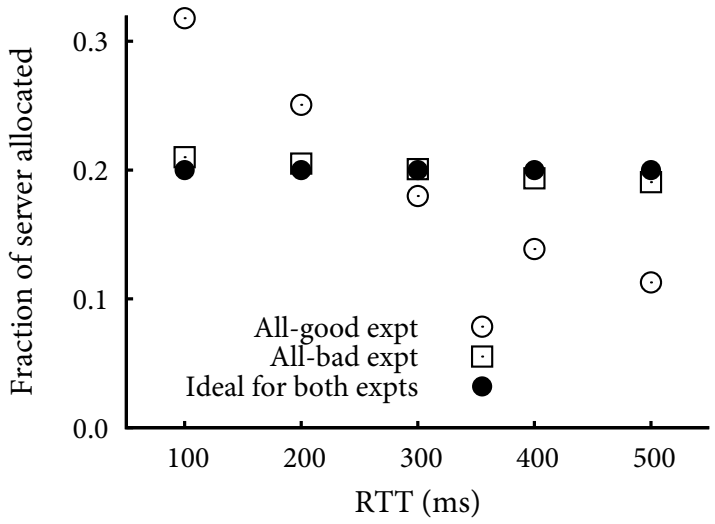


FIGURE 3.10— Two sets of heterogeneous client RTT experiments with 50 LAN clients, all good or all bad. The fraction of the server ($c = 10$ requests/s) captured by the 10 clients in category i, with RTT $100 \cdot i$ ms, varies for good clients. In contrast, bad clients' RTTs don't matter because they open multiple connections.

quiescent period (see §3.8.1 and §3.8.3), the length of which depends on RTT. In contrast, bad clients have multiple requests outstanding so do not have "down time" and will not be much affected by their RTT to the thinner.

To test this hypothesis, we assign 50 clients to 5 categories. The 10 clients in category i ($1 \leq i \leq 5$) have RTT $= 100 \cdot i$ ms to the thinner, giving a wide range of RTTs. All clients have bandwidth 2 Mbits/s, and $c = 10$ requests/s. We experiment with two cases: all clients good and all bad. Figure 3.10 confirms our hypothesis: good clients with longer RTTs get a smaller share of the server while for bad clients, RTT matters little. This result may seem unfortunate, but the effect is limited: for example, in this experiment, no good client gets more than double or less than half the ideal.

3.8.6 Good and Bad Clients Sharing a Bottleneck

When good clients share a bottleneck link with bad ones, good requests can be "crowded out" by bad ones before reaching the thinner (see §3.5.2). We quantify this observation with an experiment that uses the following topology, depicted in Figure 3.11: 30 clients, each with a bandwidth of 2 Mbits/s, connect to the thinner through a common link, l. The capacity of l is 20 Mbits/s. l is a bottleneck because the clients behind l can generate 60 Mbits/s. Also, 5 good and 5 bad clients, each with a bandwidth of 2 Mbits/s,

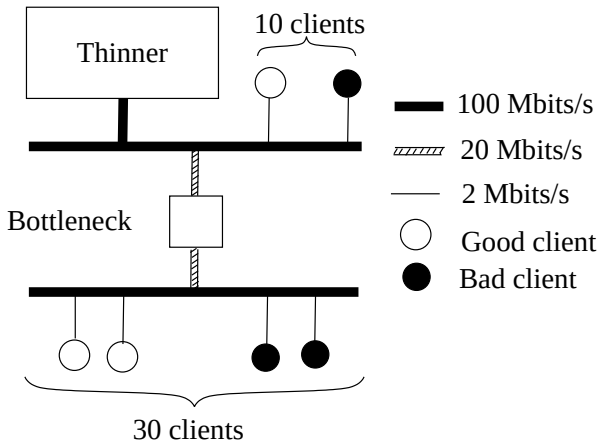


FIGURE 3.11—Network topology used to measure the impact of good and bad clients sharing a bottleneck link (§3.8.6).

connect to the thinner directly through a LAN. The server's capacity is $c = 30$ requests/s. We vary the number of good and bad clients behind l, measuring three cases: first, 5 good and 25 bad clients; second, 15 good and 15 bad clients; and third, 25 good and 5 bad clients.

Based on the topology, the clients behind l should capture half of the server's capacity. In fact, they capture slightly less than half: in the first case, they capture 43.8%; in the second, 47.7%; and in the third, 47.1%.

Next, we measure the allocation of the server to the good and bad clients behind l. We also measure, of the good requests that originate behind l, what fraction receive service. Figure 3.12 depicts these measurements and compares them to the bandwidth-proportional ideals. The ideal for the first measurement is given simply by the fraction of good and bad clients behind l. The ideal for the second measurement, f_{id}, is 0.25, and it is calculated as follows. Let G_l be the ideal bandwidth available to the good clients behind l. $G_l = 2\frac{20}{60}n$, where n is the number of good clients behind l. The fraction $\frac{20}{60}$ reflects the fact that in the ideal case, the bottleneck restricts every client equally. Further, let $g_l = n\lambda$ be the rate at which the good clients behind l issue requests. Of the good requests that originate behind l, the ideal fraction that would be served, f_{id}, is the bandwidth-proportional server piece for the good clients behind l divided by those clients' demand:

$$f_{id} = \frac{\frac{G_l}{G+B}c}{g_l} = \frac{\frac{G_l}{40}30}{g_l} = \frac{2\frac{20}{60}n30}{40n\lambda} = \frac{2\frac{20}{60}n30}{80n} = 0.25.$$

The figure shows that the good clients behind l are heavily penalized. The

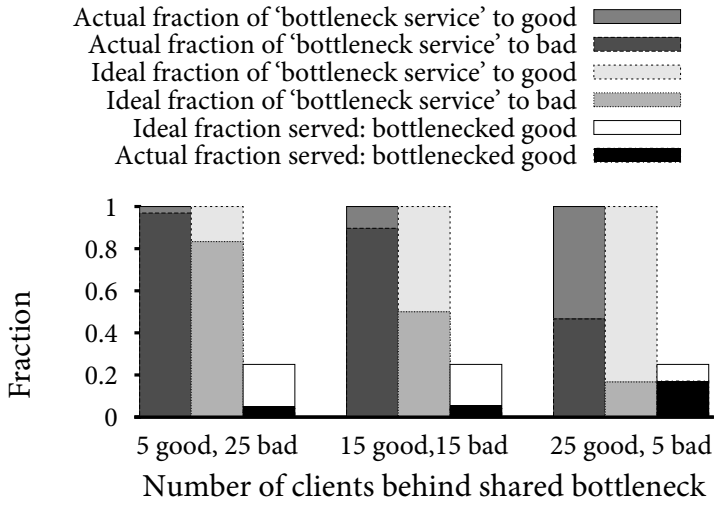


FIGURE 3.12—Server allocation when good and bad clients share a bottleneck link, *l*. "Bottleneck service" refers to the portion of the server captured by all of the clients behind *l*. The actual breakdown of this portion (left bar) is worse for the good clients than the bandwidth-proportional allocation (middle bar) because bad clients "hog" *l*. The right bar further quantifies this effect.

reason is that each bad client keeps multiple connections outstanding so captures much more of the bottleneck link, *l*, than each good client. This effect was hypothesized in §3.5.2.

3.8.7 Impact of Speak-up on Other Traffic

We now consider how speak-up affects other traffic, specifically what happens when a TCP endpoint, *H*, shares a bottleneck link, *m*, with clients that are uploading dummy bits. The case when *H* is a TCP sender is straightforward: *m* will be shared among *H*'s transfer and the speak-up uploads. When *H* is a TCP receiver, the extra traffic from speak-up affects *H* in two ways. First, ACKs from *H* will be lost (and delayed) more often than without speak-up. Second, for request-response protocols (e.g., HTTP), *H*'s request can be delayed. Here, we investigate these effects on HTTP downloads.

We experiment with the following setup: 10 good speak-up clients share a bottleneck link, *m*, with *H*, a host that runs the HTTP client `wget`. *m* has a bandwidth of 1 Mbit/s and one-way delay 100 ms. Each of the 11 clients has a bandwidth of 2 Mbits/s. On the other side of *m* are the thinner (fronting a server with $c = 2$ requests/s) and a separate Web server, *S*. In each experiment, *H* downloads a file from *S* 100 times.

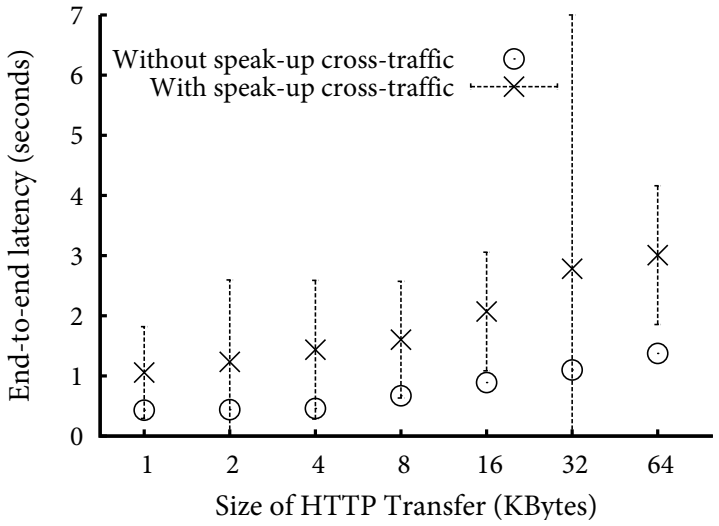


FIGURE 3.13—Effect on an HTTP client of sharing a bottleneck link with speak-up clients. Graph shows means of end-to-end HTTP download latencies with and without cross-traffic from speak-up, for various HTTP transfer sizes (which are shown on a log scale). Graph also shows standard deviations for the former measurements (the standard deviations for the latter are less than 1.5% of the means). Cross-traffic from speak-up has a significant effect on end-to-end HTTP download latency.

Figure 3.13 shows the mean download latency for various file sizes, with and without the speak-up traffic. The figure also shows standard deviations for the former set of measurements. For the latter set, the standard deviations are less than 1.5% of the means.

There is significant "collateral damage" to "innocently bystanding" Web transfers here: download times inflate between 2 and 3.2× for the various transfer sizes. However, this experiment is quite pessimistic: the RTTs are large, the bottleneck bandwidth is highly restrictive (roughly 20× smaller than the demand), and the server capacity is low. While speak-up is clearly the exacerbating factor in this experiment, it will not have this effect on every link.

3.8.8 Under-provisioned Thinner

We now explore how speak-up performs when condition C2 from §3.3 is not met. That is, we measure what happens when the thinner's incoming bandwidth is less than the combined bandwidth of its current clients. In this case, we expect bad clients to claim a disproportionate share of the server. The reason is as follows. When the thinner's access link is overloaded, the

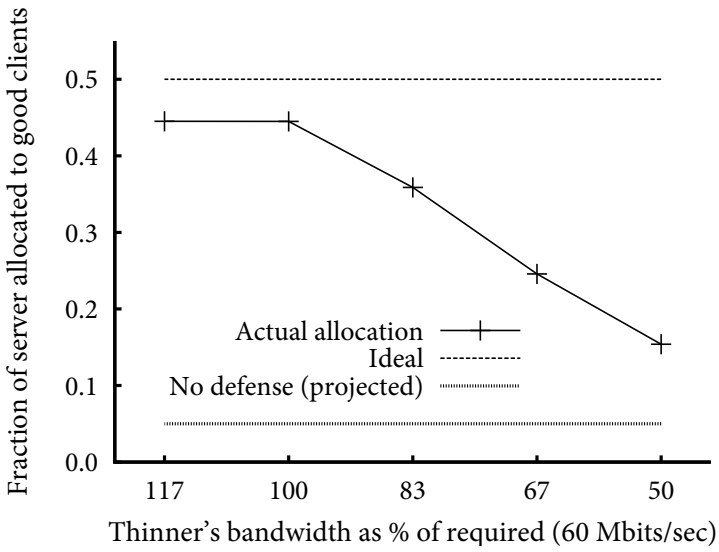


FIGURE 3.14—Server allocation as a function of the thinner's bandwidth provisioning. $G = B = 30$ Mbits/s, and $c = 30$ requests/s. Graph depicts the actual allocation that we measured under speak-up, as well as the ideal allocation and our prediction of the allocation that would result if no defense were deployed. As the thinner becomes more under-provisioned, bad clients capture an increasingly disproportionate fraction of the server. However, even in these cases, speak-up is far better than having no defense.

thinner will not "hear" all of the incoming requests. Thus, not all of the incoming requests will receive timely encouragement. Because good clients make fewer requests and keep fewer requests outstanding, a good client, relative to a bad client, has a higher probability of having no "encouraged" requests outstanding. Thus, on average, a good client is quiescent more often than a bad client, meaning that a good client pays fewer bits and hence gets less service.

To measure this effect, we perform an experiment with the following topology: 15 good and 15 bad clients, each with a bandwidth of 2 Mbits/s, connect to the thinner over a LAN. Observe that the thinner's required bandwidth provisioning is 60 Mbits/s. Our experiments vary the thinner's actual access bandwidth from 30 Mbits/s, representing a thinner that is under-provisioned by 50%, to 70 Mbits/s, representing a thinner that is amply provisioned. For each case, we measure the fraction of the server that goes to good and bad clients.

Figure 3.14 depicts the results, along with the ideal allocation and the allocation that we project would result if no defense were deployed. This latter allocation is the ratio of the two populations' request rates: $\lambda = 2$ for a good client and $\lambda = 40$ for a bad client (see §3.8.1).

As the thinner becomes increasingly under-provisioned, bad clients capture increasingly more of the server, which is the effect that we hypothesized. Nevertheless, even in this case, speak-up is far better than nothing.

3.9 SPEAK-UP COMPARED & CRITIQUED

Having shown that speak-up roughly "meets its spec", we now compare it to other defenses against application-level DDoS attacks. (For pointers to the broad literature on other denial-of-service attacks and defenses, in particular link attacks, see the survey by Mirkovic and Reiher [108] and the bibliographies in [82, 109, 178].) Some of the defenses that we discuss below have been proposed; others are hypothetical but represent natural alternatives. As the discussion proceeds, we will also critique speak-up. Some of these critiques are specific to speak-up; some apply to speak-up's general category, to which we turn now.

3.9.1 Resource-based Defenses

This category was pioneered by Dwork and Naor [46], who suggested, as a spam defense, having receivers ask senders for the solutions to computationally intensive puzzles, in effect charging CPU cycles to send email. (Back later proposed a similar idea [11].) Since then, others have done work in the same spirit, using *proof-of-work* (as such schemes are known) to defend against denial-of-service attacks [10, 41, 45, 53, 80, 114, 170, 172]. Others have proposed memory, rather than CPU, cycles for similar purposes [2], and still others use money as the scarce resource [98, 148].

One of the contributions of speak-up is to introduce *bandwidth* as the scarce resource, or currency. Another contribution is the explicit goal of a resource-proportional allocation. Many of the other proposals strive for a notion of fairness, and a few implicitly achieve a resource-proportional allocation or something close, but none state it as an objective (perhaps because it only makes sense to state this objective after establishing that no robust notion of host identity exists). An exception is a recent paper by Parno et al. [114], which was published after our work [169].

We do not know of another proposal to use bandwidth as a currency. However, Sherr, Gunter, and their co-authors [70, 142] describe a related solution to DoS attacks on servers' computational resources. In their solution, good clients send a fixed number of copies of their messages, and the server only processes a fixed fraction of the messages that it receives,

thereby diminishing adversaries' impact. Our work shares an ethos but has a very different realization. In that work, the drop probability and repeat count are hard-coded, and the approach does not apply to HTTP. Further, the authors do not consider congestion control, the implications of deployment in today's Internet, and the unequal requests case. Also, Gligor [61] observes that a hypothetical defense based on client retries and timeouts would require less overhead but still provide the same qualitative performance bounds as proof-of-work schemes. Because this general approach does not meet his more exacting performance requirements, he does not consider using bandwidth as a currency.

Bandwidth vs. CPU

We chose bandwidth as a currency because we were originally motivated by the key observation in §3.1, namely that good clients likely have more spare upload capacity. However, our resource-proportional goal could just as well be applied to CPU cycles, so we must ask, "Why bandwidth? Why not use CPU cycles as the computational currency?" To answer this question, we now compare these two alternatives. We do not find a clear winner. Bandwidth strikes us as the more "natural" choice, but it can introduce more collateral damage.

We begin with a hedge; we discuss contributions of speak-up that are expressed in the context of bandwidth but could apply to other currencies:

Less mechanism, more desiderata. The existing CPU-based proposals [10, 41, 53, 80, 114, 170, 172] incorporate far more mechanism than speak-up: they require some or all of client modification, protocol modification (to accommodate the puzzle challenge and response), or a network-wide puzzle distribution system together with a trusted authority. We conjecture that this extra mechanism is unnecessary, at least for protecting Web applications. Consider a hypothetical defense that works just like speak-up, except instead of clients' browsers coughing up dummy bits, their browsers cough up solutions to small, fixed-size CPU puzzles; in each interval, the thinner admits the client that has solved the most number of puzzles. (Similar ideas, with different implementations, are described in [114, 170].) This defense would not require the mechanisms mentioned above.

Moreover, this hypothetical defense would, like speak-up, have the following desirable properties: it would find the price correctly (by "price", we mean the going rate of access, expressed in CPU cycles or puzzle difficulty level); it would find the price automatically, with no explicit control or adjustment (in some proposals, either the server sends clients puzzles of a

particular difficulty level or clients must guess a difficulty level); it would resist gaming; and it would work with unmodified clients. No existing CPU-based proposal has all of these properties. Thus, even if the reader is not convinced by the advantages of bandwidth, speak-up's auction mechanism is useful in CPU-based schemes too.

Advantages of bandwidth. In a head-to-head comparison, bandwidth has two advantages compared to CPU:

1. Implementation-independent. A scheme based on bandwidth cannot be gamed by a faster client implementation.[5] In the hypothetical CPU-based defense just given, good clients would solve puzzles in JavaScript; bad ones might use a puzzle-solving engine written in a low-level language, thereby manufacturing an advantage.

2. Bandwidth is attackers' actual constraint. Today, absent any defense, the apparent limit on attackers is bandwidth, not CPU power (if a bot, or any host, issues requests at a high rate, its access link will be saturated long before its CPU is taxed). Charging the resource that is the actual limiting factor yields two benefits.

First, charging *any* price in that currency (including one that is below what is needed to achieve a resource-proportional allocation) will have *some* effect on the attackers. For example, assume that bad clients have maxed out their bandwidth and that the "correct" price is 50× as many bits as are in a normal request. In this situation, if the server requires all clients to spend, say, twice as many bits per request rather than 50× as many bits, then it will halve the effective request rate from the bad clients while very probably leaving the good clients unaffected. CPU cycles offer no such benefit: to affect the bad clients at all, the price in CPU cycles must be high. Of course, in our design of speak-up, servers do not state the price as a fixed multiple (doing so would not achieve a resource-proportional allocation), but they could do so in a variant of the scheme.

The second benefit to charging in the scarce resource is as follows. If the server charged in a resource that was *not* scarce, namely CPU cycles, attackers could use their bandwidth to mount some *other* attack (e.g., a link flooding attack or other malfeasance that requires bandwidth). Granted, bots today do not, to our knowledge, multiplex themselves over several attacks,[6] but in theory bots could do so. And further granted, the protected

[5] I thank Trevor Blackwell for this observation (August, 2006).

[6] On the other hand, it is common for a host to be infected by multiple bots [56]; in that case, one could imagine hosts being multiplexed over several attacks.

server likely doesn't care about preventing bots from wreaking havoc elsewhere. However, the server's *ISP* might care. More generally, *charging the resource that is scarce limits attackers' total abilities compared to charging in a different resource.*

Disadvantages of bandwidth. Against the advantages above, bandwidth has three disadvantages relative to CPU cycles; we believe that none of them is fatal:

1. CPU is a purely local resource. When a client spends bandwidth, it may adversely affect a client that isn't "speaking up". However, an analogous situation holds for *any* network application that is a heavy bandwidth consumer. For example, BitTorrent is one of the predominant applications on the Internet, is a heavy consumer of clients' upload bandwidth (far more than speak-up, which is only invoked when the server is under attack), and likely causes collateral damage. Yet, BitTorrent seems to have gained acceptance anyway, so we do not view this disadvantage as fatal for bandwidth.

A possibly more serious concern in this category is that when two "speaking up" clients share a bottleneck link, they also, as a result, "share" an allocation at the server. Meanwhile, if the clients were paying in CPU cycles, they would not have this problem.

2. Asymmetry of CPU puzzles. Solving a puzzle is slow; checking it is fast. Bandwidth does not have an analogous property: the front-end to the server must sink all of the bandwidth that clients are spending. However, we do not view this disadvantage as fatal; see the discussion of condition C2 in §3.3.

3. Variable bandwidth costs. In some countries, customers pay their ISPs "per-bit". For those customers, access to a server defended by speak-up (and under attack) would cost more than usual. One could address this disadvantage by changing the implementation of speak-up slightly so that it gives humans the opportunity to express whether they want to pay bandwidth (e.g., one could imagine the thinner exposing the "going rate" in bits and letting customers choose whether to continue).

Stepping back from the comparison between CPU and bandwidth, we wonder whether there is a design that combines the two currencies to get the advantages of both. We leave this question for future work.

Drawbacks of Resource-based Schemes

We now discuss critiques of resource-based schemes in general. First, as discussed in condition C1 in §3.3, any scheme that is trying to achieve a roughly proportional allocation only works if the good clients have enough currency (a point made by Laurie and Clayton in the context of proof-of-work for spam control [92]).

A second disadvantage that inheres in these schemes is that they are only roughly fair. In the context of speak-up, we call this disadvantage *bandwidth envy*. Before speak-up, all good clients competed equally for a small share of the server. Under speak-up, more good clients are "better off" (i.e., can claim a larger portion of the server). But since speak-up allocates the server's resources in proportion to clients' bandwidths, high-bandwidth good clients are "more better off", and this inequality might be problematic. However, observe that unfairness only occurs under attack. Thus, while we think that this inequality is unfortunate, it is not fatal. A possible solution is for ISPs with low-bandwidth customers to offer access to high-bandwidth proxies whose purpose is to "pay bandwidth" to the thinner.[7] These proxies would have to allocate *their* resources fairly—perhaps by implementing speak-up recursively.

A third critique of resource-based approaches in general, and speak-up in particular, is that they treat *flash crowds* (i.e., overload from good clients alone) as no different from an attack. This fact might appear unsettling. However, observe that, for speak-up at least, the critique does not apply to the canonical case of a flash crowd, in which a hyperlink from `slashdot.org` overwhelms a residential Web site's access link: speak-up would not have been deployed to defend a low-bandwidth site (see §3.3). For sites in our applicability regime, making good clients "bid" for access when *all* clients are good is certainly not ideal, but the issues here are the same as with speak-up in general.

A final critique of resource-based schemes is that they give attackers *some* service so might be weaker than the schemes that we discuss next that seek to *block* attackers. However, under those schemes, a smart bot can imitate a good client, succeed in fooling the detection discipline, and again get *some* service.

[7] Ben Adida suggested this idea.

3.9.2 Detect-and-Block Defenses

The most commonly deployed defense [111] is a combination of link overprovisioning [25, 120] and profiling, which is a detect-and-block approach offered by several vendors [9, 29, 103]. These latter products build a historical profile of the defended server's clientele and, when the server is attacked, block traffic violating the profile. Many other detect-and-block schemes have been proposed; we now mention a few. In application-level profiling [128, 147], the server gives preference to clients who appear to have "typical" behavior. Resource containers [15] perform rate-limiting to allocate the server's resources to clients fairly (more generally, one can use Fair Queuing [42] to rate-limit clients based on their IP addresses). Defenses based on CAPTCHAs [166] (e.g., [109, 151]) use reverse Turing tests to block bots. Killbots [82] combines CAPTCHAs and rate-limiting, defining a bot as a non-CAPTCHA answering host that sends too many requests to an overloaded server.

One critique of detect-and-block methods is that they can err. CAPTCHAs can be thwarted by "bad humans" (cheap labor hired to attack a site or induced [118] to solve the CAPTCHAs) or "good bots" (legitimate, non-human clientele or humans who do not answer CAPTCHAs). As mentioned in Chapter 2 and the beginning of this chapter, schemes that rate-limit clients by IP address can err because of address hijacking and proxies. Profiling apparently addresses some of these shortcomings today (e.g., many legitimate clients behind a proxy would cause the proxy's IP address to have a higher baseline rate in the server's profile). However, in principle such "behavior-based" techniques can also be "fooled": a set of savvy bots could, over time, "build up" their profile by appearing to be legitimate clients, at which point they could abuse their profile and attack.

3.9.3 Mechanisms for Blocking Traffic

There has been a lot of recent research focusing on *mechanisms* for blocking traffic destined to servers under attack; the *policies* for such blocking are often unspecified. For this reason, we believe that this class of proposals is orthogonal to, and can be combined with, speak-up and the other members of the taxonomy presented at the beginning of the chapter. We now give more detail.

Examples of proposed mechanisms include the recent literature on capabilities [7, 177, 178]; dFence [95], in which server operators can dynamically deploy middleboxes to filter problematic traffic; and an addressing

scheme in which hosts that are always clients cannot be addressed [74] (see the bibliographies of those papers for other examples). These proposals share a similar high-level structure: they describe systems to keep traffic from potential victims—under the assumption that the infrastructure or the protected host knows which packets are worth receiving. For example, with capabilities [7, 177, 178], servers are protected by *capability allocators* that act on their behalf. These allocators give requesting clients tokens, or capabilities, that routers understand. Clients then place the capabilities in the packets that they originate, and routers give such packets higher priority than packets without capabilities.

These techniques focus on *how* to block traffic, not on *which* traffic to block. For the latter function, the authors generally suggest detect-and-block techniques (e.g., CAPTCHAS), but they could easily use speak-up, as mentioned in §3.3. For example, when a server is over-subscribed, its capability allocator could conduct a bandwidth auction to decide which clients receive capabilities. Indeed, under the threat in which good and bad are indistinguishable, these proposals would *have* to use speak-up or another resource-based scheme! As an example, Parno et al. [114] advocate CPU puzzles for precisely this purpose, but their paper considers bandwidth as a candidate resource.

3.9.4 Summary

Because detect-and-block defenses can err, we favor resource-based defenses for the threat described in §3.3. We have argued that bandwidth is a natural (but not the only) choice for the resource and that speak-up's mechanisms may still be useful under other resource choices. Finally, we have addressed important critiques of resource-based approaches in general and speak-up in particular.

3.10 PLAUSIBILITY OF THE THREAT & CONDITIONS

Though we have argued that speak-up can be an appropriate defense, given the threat and conditions that we modeled in §3.3, we have so far not said to what extent the threat and conditions occur in practice. Below we address the following questions in turn: (1) How often is the threat manifest? (2) How often does condition C1 apply, i.e., how often are aggregate good and bad bandwidths roughly equal? (3) Is condition C2 reasonable, i.e., can we assume that servers have adequate link bandwidth? To answer

these questions, we synthesize the research and anecdotes of others, relying on secondary and tertiary sources.

Our discussion will be with reference to attackers' current abilities. If deployed, speak-up would certainly cause attackers to change their tactics; specifically, they would try to acquire more machines. We consider such dynamics, together with the next response from the academic and security communities, in §6.1. For now, we simply observe that attackers are already highly motivated to compromise machines; it follows that compromising *additional* machines will be costly for them.

3.10.1 The Threat

By "the threat", we mean requests that are (a) application-level (b) legitimate-looking and (c) of uncertain origin. We address these characteristics in turn. For (a), reports are mixed. A data set from Shadowserver that covers DDoS activity by ∼1500 botnets [140] controlled via Internet Relay Chat (IRC) does not indicate *any* application-level attacks, but Prolexic Technologies reportedly sees *mostly* this type [17]. However, in Prolexic's case, the requests are often ill-formed, so characteristic (b) does not hold. For (c), we know that some attackers hijack addresses for sending spam and that proxies are widespread (see Chapter 2). Also, bots are ever more sophisticated, and botnets are becoming smaller [17, 33, 35, 49, 78, 105, 125], presumably to fly under the "detection radar" of victims and the mitigation community. The combination of smarter but smaller bots will make the three characteristics above—which require smart bots but which conserve the adversary's resources—more likely.

Regardless, this discussion obscures two larger points. First, even if such attacks have not been observed in their pure form, it is not hard to carry them out: the *vulnerability* is real. Second, we believe that it is important to be proactive, that is, to identify weaknesses *before* they are exploited. Thus, even if the underground economy does not favor this attack today, we must ask—and try to answer—the question of how to defend against it.

3.10.2 Relative Sizes of Good and Bad Clientele

We first discuss the sizes of botnets and then discuss the implications for what types of sites speak-up can protect.

We begin by arguing that *most botnets today consist of fewer than 100,000 hosts, and even 10,000 hosts is a large botnet*. Although there have been reports of botnets of over 100,000 hosts [24, 40, 73, 75, 104, 153] (millions in

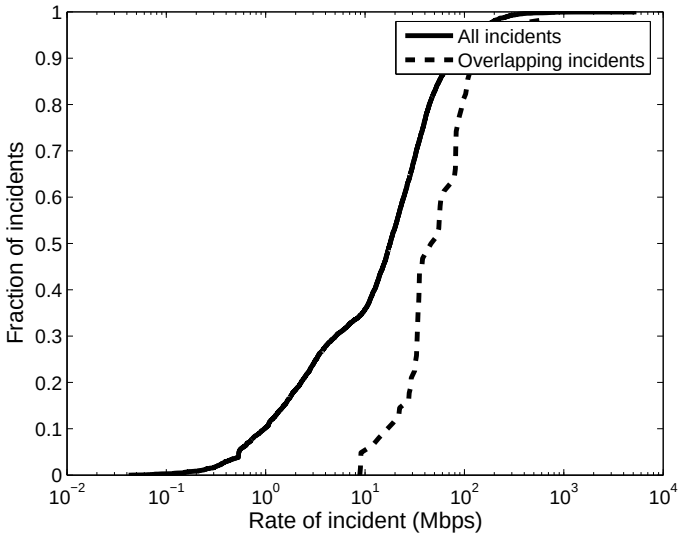


FIGURE 3.15—Rates of potential attack incidents observed by Sekar et al. [138]. This figure is reproduced, with permission, from Figure 16 of their paper.

the case of [153]), most botnets are far smaller. Freiling et al., in a study of 180 botnets in 2005, find that the largest botnets were up to 50,000 hosts, with some being just several hundred strong [56, §5]. Symantec reports that 2,000 to 10,000 hosts is a common range [104]. Shadowserver [139] tracks thousands of botnets and reports a total number of bots in the single millions, implying that the average botnet size is in the low thousands. Cooke et al. interviewed operators from Tier-1 and Tier-2 ISPs; their subjects indicated that whereas botnets used to have tens of thousands of nodes several years before, sizes of hundreds to thousands of nodes became the norm [35]. Rajab et al. find similar numbers—hundreds or thousands—for the "live populations" of various botnets [124, 125]. Indeed, they point out that some studies may report the "footprint"—the total number of hosts that are infected with the bot—which may account for the larger sizes. For speak-up, we are concerned only with the botnet's *current* firepower and hence with the "live population".

The observed sizes of link flooding attacks back up these rough estimates, as we now argue. First, consider the observations of Sekar et al. [138] about attacks in a major ISP; the relevant graph from their paper is reproduced in Figure 3.15, and the relevant line is the solid one. The graph shows that a 100 Mbits/s DoS attack is over the 90th percentile (specifically, it is at the 93rd percentile [137]). And 1 Gbit/s attacks are at the 99.95th per-

centile [137]. Their study, combined with anecdotes [73, 152, 162], suggests that although a small number of link-flooding attacks are over 1 Gbit/s, the vast majority are hundreds of Mbits/s or less. These sizes in turn suggest that the attacks are not being carried out by gigantic botnets. Our reasoning is as follows. One study found that the average bot has roughly 100 Kbits/s of bandwidth [143]. Thus, even if each bot uses only a tenth of its bandwidth during an attack, an attack at the 90th (resp., 99.95th) percentile could not have been generated by a botnet of larger than 10,000 (resp., 100,000) nodes. Our conclusion is that most attacks are launched from botnets that are under 10,000 hosts. (Of course, it is possible that the attackers controlled millions of hosts, each of which was sending traffic very slowly, but in that case, *no* defense works.)

Given these numbers, how big does the legitimate clientele have to be to fully withstand attack? As mentioned in §3.2, the answer depends on the server's utilization (or, what is the same thing, its degree of overprovisioning), as we now illustrate. Recall from §3.4.1 that, for the good clients to be unharmed in the ideal case, we must have $c \geq g(1+B/G)$. Letting $u = g/c$ be the usual utilization of the server, we get $G/B \geq u/(1-u)$. If the bandwidth of a population is proportional to the number of members, then the good clients must have $u/(1-u)$ as many members as the bots to fully withstand attack. Putting this result in context, if a service has spare capacity 90% (i.e., $u = 0.1$), speak-up can fully defend it (i.e., leave its good clients unharmed) against a 1,000-host (resp., 10,000-host) botnet if the good clients number $\sim$100 (resp., $\sim$1,000). If a service has spare capacity 50%, then the good clients must number $\sim$1,000 (resp., $\sim$10,000).

Many sites have clienteles of this order of magnitude: observe that these numbers refer to the good clients currently *interested* in the service, many of which may be quiescent. For example, `http://www.kayak.com`, a computationally-intensive travel distribution site, often claims to have at least tens of thousands of clients online. Many of these clients are humans pausing between queries, but, from speak-up's perspective, their machines count in the "current clientele". A more extreme example is Wikipedia, which, according to the Web analytics company Alexa [5], is one of the top 10 sites on the Web (as of August, 2007). According to Wikipedia's statistics [173], they get an average of 20,000 requests per second. Assuming (likely pessimistically) that a human reading Wikipedia makes a request once every ten seconds, the number of concurrent clients interested in the service is 200,000.

3.10.3 Costs for the Server

We discussed how to satisfy condition C2 in §3.3 and §3.5.3. And we showed in §3.8.8 that even if the condition isn't satisfied, speak-up still offers some benefit. Here, we just want to make two points about C2. First, any other scheme that were to seek a proportional allocation would also need condition *C1*. Thus, C2 represents the relative cost of speak-up to the server. Second, one can regard this cost as *paying bandwidth to save application-level resources*. We are not claiming that this trade-off is worthwhile for every server, only that speak-up creates such an option and that the option may appeal to some server owners.

3.11 REFLECTIONS

We first summarize speak-up's purpose and its costs and then discuss it in a broader context—how and when it combines with other defenses, and where else it may apply.

Summary. Our principal finding in this chapter has been that speak-up mostly meets the goal in §3.4.1—a roughly fair allocation, based on clients' bandwidths. Thus, speak-up upholds the philosophy in §1.2.

Speak-up certainly introduces costs, but they are not as onerous as they might appear. The first set of costs, to the victimized server, we discuss immediately above (condition C2). The second set of costs is to the network, because speak-up introduces traffic when servers are attacked. Yet, *every network application introduces traffic* (and some, like BitTorrent, introduce a whole lot more traffic). And, as with most other network applications, the traffic introduced by speak-up obeys congestion control. The third set of costs is to the end-user: there exist variable bandwidth costs (which we discussed in §3.9.1), and, also, speak-up may edge out other activity on the user's upload link. Such an opportunity cost is unfortunate, but opportunity costs apply to all resource-based defenses (e.g., if the server charged in CPU cycles, the user's tasks would compute slower).

Thus, while speak-up may not make sense for every site or every denial-of-service attack, we think that it is a reasonable choice for some "pairings". The benefit is that the implementation, and the overall idea, are ultimately quite simple. Moreover, the costs that we have been discussing have been for the "worst-case" threat; when that threat is only partially manifest, the costs of speak-up are lower, as we now describe.

Speak-up combined with other defenses. The threat described in §1.1 and §3.3 specifies that the server cannot identify its clients and that the bad clients can mimic the good ones; it is for this threat that speak-up is designed. Yet, this threat may not always hold in practice (as mentioned in §3.10), or it may hold only partially. In these cases, one can use other defenses, with speak-up being the "fallback" or "backstop".

We now give two examples of when other defenses would apply. First, in practice, a server may be able to recognize *some* of its legitimate clientele as such. For example, when a client makes a purchase at a Web server, the server can infer that the client is legitimate and issue a cookie to the client. Then, when the server is overloaded or attacked, it (a) prioritizes clients that present valid cookies but does *not* charge them bandwidth and (b) runs speak-up for the remainder of its clientele. This approach saves bandwidth, both for the known clients and for the server (because the known clients are not speaking up). This approach also gives low bandwidth clients a way to overcome their bandwidth disadvantage (discussed in §3.9.1)—become known to the server. The disadvantage of this approach is that the *unknown* legitimate clients will be competing for a smaller piece of the server so may be worse off, compared to a uniform application of speak-up.

A second example is that if some requests are ill-formed or violate the protocol, the server should drop them and run speak-up for the remaining clients. As with the approach above, this one may result in lower costs for the server—in this case, because fewer bad clients get service.

Other applications of speak-up. While we have discussed speak-up in the context of defending servers against application-level denial-of-service, it can apply more broadly. We now list three sample applications. First, one could use a variant of speak-up to guard against "Sybil attacks" [43] (i.e., attacks in which clients manufacture identities) in peer-to-peer networks: the protocol could require clients to send large and frequent heartbeat messages; clients would then be unable to "afford" many identities [83].

Second, in some contexts, speak-up could defend against link attacks: a link is fundamentally a resource and could be protected and allocated just like a server's computational resources. For instance, consider cases in which the "link" is a contained channel reserved for a particular class of traffic (as in the scenario of capability requests, covered by Parno et al. [114]). One could allocate such a channel roughly fairly—without needing a thinner—by requiring clients to send as much traffic as possible through the channel. Then, the channel would be over-subscribed, and a

client's chances of getting bits through the channel would be in proportion to the quantity of traffic that it sends.

Third, in DQE, the system covered in the next chapter, clients pay for email quotas. As we will see, DQE works with a range of currencies and policies; one option is for prospective senders to pay in bandwidth.

We now turn to DQE.

4

DQE

In this chapter, we view the attention of all of the world's email recipients as one aggregate resource, and we describe a system that regulates the consumption of this resource. The motivation for such a system is of course spam, which over-subscribes human attention and is an instance of the abstract problem in §1.1.

Consistent with the philosophy in §1.2, we look for solutions that have two broad characteristics. First, they should limit the number of messages sent, rather than try to divine their intent, as is done by spam filters. (Filtering is a *content-based* solution and thus inherently unreliable, as argued in Chapter 1.) Second, the limits on message volume should obey a rough notion of proportionality: no one sender should be able to send more than a tiny fraction of all email. Today, in contrast, a small number of spammers send more than three-quarters of all email [106, 107, 150]. If a system with both of these characteristics is deployed and adopted, then the world's inboxes will have only a small percentage of spam (unless a large percentage of email senders are spammers), and email that obeys the volume limits will be delivered reliably.

To satisfy these requirements, we turn to an approach using *quotas* or *bankable postage*. Several such schemes have been proposed before [1, 13, 89]. In general, these systems give every sender a *quota* of *stamps*. How this quota is determined varies among proposals; options include proof of CPU or memory cycles [1, 117], annual payment [13], having an email account with an ISP [89], having a driver's license [13], etc. The sending host or its email server attaches a stamp to each email message, and the receiving host or its email server *tests* the incoming stamp by asking a *quota enforcer* whether the enforcer has seen the stamp before. If not, the receiving host infers that the stamp is "fresh" and then *cancels* it by asking the enforcer to

store a record of the stamp. Only messages with fresh stamps are delivered by the receiving host to the human user's inbox; used stamps are presumed to indicate spam.

Later in the chapter, we explain in detail why we prefer this approach to alternatives and how this approach should be combined with other defenses (see §4.10.1). For now, observe that it upholds our requirements and philosophy: neither quota allocation nor enforcement uses content-based discrimination; the allocation of quotas is supposed to be such that no one sender is "allowed" to send outsized volumes; and quota enforcement ensures that only quota-obeying emails are seen by humans, thereby conserving human attention. However, for the approach to be viable, two things are required: (1) pragmatic policies for allocating quotas (really, allocating human attention) to achieve the rough proportionality goal; and (2) a technical mechanism for quota enforcement that can handle the volume of the world's email, without much cheating.

In this chapter, we focus on the second of these requirements, quota enforcement, though we briefly cover quota allocation (see §4.7). The reason for this imbalance of focus is that these two are different concerns: the former is a purely technical matter while the latter involves social, economic, and policy factors. In fact, our specific aim is to show that many technical hurdles in quota-based systems can be overcome.

To that end, this chapter describes the design and implementation of *DQE* (Distributed Quota Enforcement), a quota-based spam control system. DQE adopts and augments a proposal by Balakrishnan and Karger [13]. Their architecture meets desired properties not met by previous work, including separating allocation and enforcement, not trusting the enforcer, and preserving privacy (see §4.2.1). DQE's augmentation is the design, implementation, analysis, and experimental evaluation of an enforcer that meets a second set of challenges. These challenges include scaling to the volume of the world's email, tolerating faults, resisting attack, and achieving high throughput (see §4.2.2). Our experimental results suggest that, in addition to meeting these challenges, our implementation of the enforcer could handle 200 billion messages daily (a multiple of the world's email volume) with a few thousand dedicated PCs (see §4.6). Our work on the enforcer leads us to conclude that large-scale quota enforcement is practical and viable; that conclusion is this chapter's contribution to the spam control literature.

A second set of contributions in this chapter is independent of spam control. We believe that the enforcer is an interesting distributed system

in its own right. It is designed to store billions of key-value pairs (canceled stamps, in the spam context) over a set of mutually untrusting nodes (§4.4). It relies on just one trust assumption, common in distributed systems: that the constituent hosts are determined by a trusted entity. It tolerates Byzantine and crash faults in its nodes, but it does not need to be "Byzantine Fault Tolerant" [28], for it is allowed to give wrong answers sometimes. It achieves fault-tolerance by "replicating on demand" in response to such wrong answers (§4.4.1, §4.4.2). Each node uses, for its internal key-value map, a novel data structure that balances storage and speed (§4.4.3). Nodes prevent the enforcer's *aggregate* throughput from degrading under load—a phenomenon that we call "distributed livelock" and that we conjecture exists in many other distributed systems—by making only *local* decisions about which requests to drop (§4.4.4). And the enforcer is a candidate for protection by speak-up! (See §4.4.5.)

Apart from these techniques, what is most interesting to us about the enforcer is that it does a fairly large job with fairly little mechanism: it is designed to handle millions of requests per second and is fault-tolerant, yet the nodes do not need to maintain replicas, keep track of other nodes, route requests for each other, or trust each other. In fact, we believe that the enforcer is viable precisely *because* of its absence of mechanism. As we discuss in §4.11, the enforcer is likely to be a useful building block in other contexts.

* * *

This chapter's organization follows an argument that DQE can achieve our top-level goal of proportionally allocating human attention. The outline of the argument is as follows:

— We first consider quota enforcement (§4.2–§4.6). We wish to show that, given some allocation, DQE can enforce it. We do so by:

  * Defining technical goals that must be met for DQE to be viable (§4.2).
  * Describing the architecture of DQE, some of which is inherited from [13] (§4.3).
  * Detailing the enforcer's design, its implementation, and our evaluation of that implementation. We show that the enforcer can ensure, roughly, that any given allocation holds (i.e., the possible cheating is limited). We also show that our implementation

could handle the volume of the world's email with a few thousand machines. These two results demonstrate that DQE is viable (§4.4–§4.6).

— We then argue that pragmatic policies for allocation exist and that the total consumption of human attention is appropriately bounded (§4.7–§4.8).

— We next discuss possible paths to deployment and adoption (§4.9).

— Having argued that DQE is technically viable and that the non-technical aspects (allocation, adoption) are at least not insurmountable, we compare DQE to alternatives; we also state how DQE should be combined with other defenses (§4.10).

— Finally, we critique DQE and consider its applicability in other contexts (§4.11).

4.1 THE THREAT

We define *spammers* to be either authors or distributors of spam. Spammers may send spam either from their own computers or from botnets that they control. They can also use these botnets to attack DQE. Any spam-sending computer may temporarily hijack addresses (as discussed in Chapter 2 and observed in [126]). Also, we will regard spam as principally an economic activity (so it will be reasonable for us to consider a spammer's profit-per-message, in §4.7). For more detail about the problem of spam and the many solutions that have been proposed (some of which we cover in §4.10) see [37, 58, 64, 68, 157].

4.2 TECHNICAL REQUIREMENTS & CHALLENGES

We begin with architectural requirements and then discuss challenges that are specific to the enforcer. The former set of goals was articulated in [13] and satisfied by the architecture described in that paper, as we show in §4.3. The latter set of goals is unmet by previous proposals.

4.2.1 Protocol Requirements

Separate allocation and enforcement. Allocating quotas of stamps is a social, economic, and policy function; it requires great care; it does not re-

quire much computation; and it needs to happen at very coarse-grained timescales, such as once per year per sender. Enforcement is the exact opposite: it is a technical function; it can be performed "sloppily"; it requires significant computation; and it needs to happen once per email message. Given this contrast, Balakrishnan and Karger argue [13], and we concur, that allocation and enforcement should be performed separately—that is, at separate times, by separate entities, and in a way that allows any allocation policy to work with the enforcement mechanism.

No false positives. We want email to be reliable again. We assume reused stamps indicate spam. Thus, a fresh stamp must never appear to have been used before.

Untrusted enforcer. We do not know the likely economic model of the enforcer, whether *monolithic* (i.e., owned and operated by a single entity) or *federated* (i.e., many organizations with an interest in spam control donate resources to a distributed system). No matter what model is adopted, it would be wise to design the system so that clients place minimal trust in this infrastructure.

Privacy. To reduce (already daunting) deployment hurdles, we seek to preserve the current "semantics" of email. In particular, queries of the quota enforcer should not identify email senders (otherwise, the enforcer knows which senders are communicating with which receivers, violating email's privacy model), and a receiver should not be able to use a stamp to prove to a third party that a sender communicated with it.

4.2.2 Challenges for the Enforcer

Scalability. The enforcer must scale to current and future email volumes. Studies estimate that 80-90 billion emails will be sent daily this year [77, 123]. (We admit that we have no way to verify these claims.) We set an initial target of 200 billion daily messages (an average of about 2.3 million stamp checks per second) and strive to keep pace with future growth. To cope with these rates, the enforcer must be composed of many hosts.

Fault-tolerance. Given the required number of hosts, it is highly likely that some subset will experience crash faults (e.g., be down) or Byzantine faults (e.g., become subverted). The enforcer should be robust to these faults. In particular, it should guarantee no more than a small amount of stamp reuse, despite such failures.

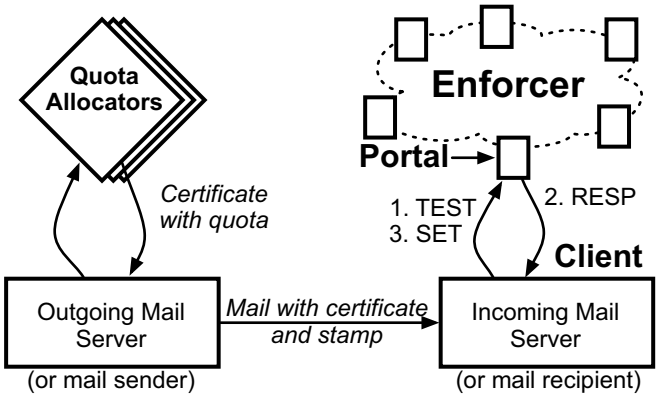


FIGURE 4.1—DQE architecture.

High throughput. To control management and hardware costs, we wish to minimize the required number of machines, which requires maximizing throughput.

Attack-resilience. Spammers will have a strong incentive to cripple the enforcer; it should thus resist denial-of-service (DoS) and resource exhaustion attacks.

Mutually untrusting nodes. In both federated and monolithic enforcer organizations, nodes could be compromised. In the federated case, even when the nodes are uncompromised, they may not trust each other. Thus, in either case, besides being untrust*ed* (by clients), nodes should also be untrust*ing* (of other nodes), even as they do storage operations for each other.

We now show how the above requirements are met, first discussing the general architecture in §4.3 and then, in §4.4, focusing on the detailed design of the enforcer.

4.3 DQE ARCHITECTURE

The architecture is depicted in Figure 4.1. We begin by discussing the format and allocation of stamps (§4.3.1), how stamps are checked and canceled (§4.3.2), and how that process satisfies the requirements in §4.2.1. These pieces—the high-level architecture, stamps, and the protocols in §4.3.2—are what DQE inherits from [13]. Indeed, most (but not all) of the ideas in §4.3.1 and §4.3.2 appeared in [13].

We also give an overview of the enforcer (§4.3.3) and survey vulnerabilities (§4.3.4). Although we will refer to "sender" and "receiver", we expect those will be, for ease of deployment, the sender's and receiver's respective email servers.

4.3.1 Stamp Allocation and Creation

The quota allocation policy is the purview of a few globally trusted quota allocators (QAs), each with distinct public/private key pair (QA_{pub}, QA_{priv}); the QA_{pub} are well known. A participant S constructs public/private key pair (S_{pub}, S_{priv}) and presents S_{pub} to a QA. The QA determines a quota for S and returns to S a signed certificate (the notation $\{A\}_B$ means that string A is signed with key B):

$$C_S = \{S_{pub}, \textit{expiration time}, \textit{quota}\}_{QA_{priv}}.$$

Anyone knowing QA_{pub} can verify, by inspecting C_S, that whoever owns S_{pub} has been allocated a quota. *expiration time* is when the certificate expires (in our implementation, certificates are valid for one year), and *quota* specifies the maximum number of stamps that S can use within a well-known epoch (in our implementation, each day is an epoch). Epochs free the enforcer from having to store canceled stamps for long time periods. Obtaining a certificate is the only interaction that participants have with a QA, and it happens on long time scales (e.g., yearly). As a result, allocation is separate from enforcement—one of the requirements in §4.2.1—and the QA can allocate quotas with great care.

Participants use the *quota* attribute of their certificates to create up to *quota* stamps in any epoch. A participant with a certificate may give its stamps to other email senders, which may be a practical way for an organization to acquire a large quota and then dole it out to individual users.

Each stamp has the form

$$\{C_S, \{i, t\}_{S_{priv}}\}.$$

Each i in $[1, \textit{quota}]$ is supposed to be used no more than once in the current epoch. t is a unique identifier of the current epoch. Because email can be delayed en route to a recipient, receivers accept stamps from the current epoch and the one just previous.

Quotas and stamps are reminiscent of micropayment systems [26, 60, 132]. In those systems, buyers get blocks of cash and mint payments from the blocks. We compare DQE and micropayments in §4.10.

1. S constructs STAMP $= \{C_S, \{i, t\}_{S_{priv}}\}$.

2. $S \rightarrow R : \{\text{STAMP}, msg\}$.

3. R checks that $i \leq quota$ (in C_S), that t is the current or previous epoch, that $\{i, t\}$ is signed with S_{priv} (S_{pub} is in C_S), and that C_S is signed with a quota allocator's key. If not, R rejects the message; the stamp is invalid. Otherwise, R computes POSTMARK = HASH(HASH(STAMP)).

4. $R \rightarrow$ enforcer : TEST(POSTMARK). Enforcer replies with x. If x is HASH(STAMP), R considers STAMP used. If x is "not found", R continues to step 5.

5. $R \rightarrow$ enforcer : SET(POSTMARK, HASH(STAMP)).

FIGURE 4.2—Stamp cancellation protocol followed by sender (S), receiver (R), and the enforcer. The protocol upholds the design goals in §4.2.1: it gives no false positives, preserves privacy, and does not trust the enforcer.

An alternative to senders creating their own stamps would be QAs distributing stamps to senders. We reject this approach because it would require a massive computational effort by the QAs.

4.3.2 Stamp Cancellation Protocol

This section describes the protocol followed by senders, receivers, and the enforcer. Figure 4.2 depicts the protocol.

For a given stamp attached to an email from sender S, the receiver R must check that the stamp is unused and must prevent reuse of the stamp in the current epoch. To this end, R checks that the value of i in the stamp is less than S's quota, that t identifies the current or just previous epoch, and that the signatures are valid. If the stamp passes these tests, R communicates with the enforcer using two UDP-based Remote Procedure Calls (RPCs): TEST and SET. R first calls TEST to check whether the enforcer has seen a fingerprint of the stamp; if the response is "not found", R then calls SET, presenting the fingerprint to be stored.[1] The fingerprint of the stamp

[1] One might wonder why receivers will SET after they have already received "service" from the enforcer in the form of a TEST reply. Our answer is that executing these requests is inexpensive, automatic, and damaging to spammers.

is HASH(STAMP), where HASH is a one-way hash function that is hard to invert.[2]

For this approach to work, *signatures must be deterministic*. If each message had many valid signatures, then senders could create many different values of STAMP for the same logical stamp. Each of the STAMP values would of course lead to a different value of HASH(STAMP), giving receivers no way to detect the multiple use. To get deterministic signatures that have provable cryptographic security, our implementation uses Full Domain Hash[3] with a large modulus [20, 36].[4]

Note that an adversary cannot cancel a victim's stamp before the victim has actually created it: the stamp contains a signature, so guessing HASH(STAMP) requires either finding a collision in HASH or forging a signature.

We now return to the requirements in §4.2.1. First, we discussed the separation of allocation and enforcement in §4.3.1. Second, false positives are impossible: because HASH is one-way, a reply of the fingerprint—HASH(STAMP)—in response to a TEST of the postmark—HASH(HASH(STAMP))—proves that the enforcer has seen the (postmark, fingerprint) pair. Thus, the enforcer cannot falsely cause an email with a novel stamp to be labeled spam. (The enforcer can, however, allow a reused stamp to be labeled novel; see §4.4.) Third, receivers do not trust the enforcer: they demand proof of reuse (i.e., the fingerprint). Finally, the protocol upholds current email privacy semantics: the enforcer sees hashes of stamps and not stamps themselves, so it cannot infer who sent the message corresponding to a given stamp. More details about this protocol's privacy properties are in [13].

[2] Our implementation uses SHA-1, which has recently been found to be weaker than previously thought [171]. We don't believe this weakness significantly affects our system because DQE stamps are valid for only two days, and, at least for the near future, any attack on SHA-1 is likely to require more computing resources than can be marshaled in this time. Moreover, DQE can easily move to another hash function.

[3] Public-key signature schemes often work as follows. They (a) hash the message to be signed, together with some randomness, thereby producing a non-deterministic digest and (b) apply the "sign" operation to this digest. In Full Domain Hash, one does not use randomness but, as with the other schemes, *does* take care that the hash function produces digests that are roughly uniformly distributed over the full domain of the signing function. Under a particular model of how hash functions work (the random oracle model [19]), breaking this approach is at least as hard (roughly) as breaking the underlying problem (e.g., RSA). The difference is that, with the randomness, breaking the signature scheme is a little "harder". To compensate for this effect, we choose a larger domain for the signing function.

[4] I am grateful to Shabsi Walfish for pointing me to these papers.

4.3.3 The Enforcer

The enforcer stores the postmarks and fingerprints of stamps canceled (i.e., SET) in the current and previous epochs. It comprises thousands of untrusted *storage nodes* (which we often call just "nodes"), with the list of approved nodes published by a trusted authority. The nodes might come either from a single organization that operates the enforcer for profit (perhaps paid by organizations with an interest in spam control) or else from multiple contributing organizations.

Clients, typically incoming email servers, interact with the enforcer by calling its interface, TEST and SET. These two RPCs are implemented by every storage node. For a given TEST or SET, the node receiving the client's request is called the *portal* for that request. Clients discover a nearby portal either via hard-coding or via DNS.

4.3.4 Remaining Vulnerabilities

As discussed in §4.3.2, attackers cannot forge stamps, cancel stamps that they have not seen, or induce false positives. DQE's remaining vulnerabilities are in two categories: unauthorized stamp *use* (i.e., theft) and stamp *re*-use. Since the purpose of the enforcer is to prevent reuse, we address the second category when describing the enforcer's design in §4.4. We discuss the first category in the remainder of this section.

Spammers can steal stamps from (1) the bots that they control; and (2) email servers along the path from the sending email client to the receiving email client. We now address these two cases. The gist of our argument is that the effect of such stealing is limited.

Stamp theft from "botted" computers. This theft can take three forms. First, assume that the human user of a "botted" host stores his stamps on the email server. Then, if the spammer intercepts the server's authentication of the user (e.g., by installing a key logger to discover a password), the spammer can compromise the email account and send stamped spam from there. However, the user's quota would be depleted, possibly alerting him to the compromise of his account. Moreover, out-of-band contact between the email provider and the customer could detect the theft, in analogy with credit card companies contacting customers to verify anomalous activity.

Next, assume that the end-user stores her stamps directly on her computer (which is "botted") and that her computer, rather than the email server, performs the stamp checks. The second form of theft is the bot simply stealing the user's stamps directly from the computer. The third form

is stealing stamps from *inbound* email before the DQE client software can cancel the stamps. In both of these cases, the human would again realize that something was wrong, either because she faced a depleted quota or found all of her legitimate email labeled spam. Moreover, we expect both of these forms of theft to be very rare because most people will not manage their stamps; doing so requires administration (to integrate stamps with the email client, etc.), which most users will prefer to delegate to the email server.

Note, also, that *even if* a spammer succeeds in stealing stamps from all of the hosts in his botnet—using any of the three attacks just mentioned—such theft is unlikely to increase spam much: a botnet with 100,000 hosts and a daily quota of 100 stamps per machine leads to 10 million extra spams, a small fraction of the tens of billions of daily spams today. We discuss this point in further depth when considering DQE's end-to-end effectiveness, in §4.8.

Stamp theft from email servers and relays. This theft can take three forms. First, the spammer can compromise a source email server, allowing the spammer to steal stamps directly. Second, the spammer can compromise a destination email server (in practice many email servers perform the source and destination roles), allowing him to steal stamps from inbound email. In both of these cases, the users of the compromised server would realize that something was wrong. Third, the spammer can compromise an intermediate relay between sender and receiver, allowing him to steal stamps from email passing through the relay. To thwart this attack, senders can encrypt emails (including the header that contains the stamp). We believe that these three forms of theft will not lead to much extra spam because (a) no single email server or relay is likely to carry a significant fraction of the world's email volume; and (b) email servers, being relatively hardened and supervised machines, are unlikely to be compromised en masse.

4.4 DETAILED DESIGN OF THE ENFORCER

The enforcer, depicted in Figure 4.3, is a high-throughput storage service that replicates immutable key-value pairs over a group of mutually untrusting, infrequently changing nodes. It tolerates Byzantine faults in these nodes. We assume a trusted *bunker*, an entity that communicates the system membership to the enforcer nodes. The bunker assigns random *identifiers*—whose purpose we describe below—to each node and infre-

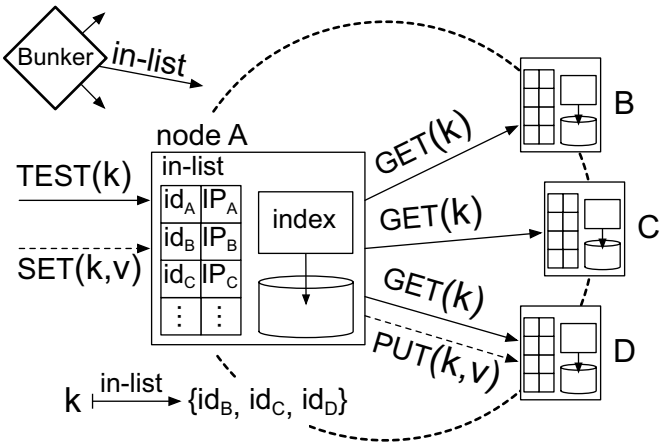


FIGURE 4.3—Enforcer design. A TEST induces multiple GETs; a SET induces one PUT. Here, *A* is the portal. The ids (id_A, id_B, etc.) are in a circular identifier space; their values are determined by the bunker.

quently (e.g., daily) distributes to each node an *in-list*, a digitally signed, authoritative list of the members' identifiers and IP addresses.[5]

Given the required size of the system—thousands of nodes (§4.6.5)—we believe that the bunker is a reasonable assumption. If a single organization operates the enforcer, the bunker can be simply the human who deploys the machines. If the enforcer is federated, a small number of neutral people can implement the bunker: managing a list of several thousand relatively reliable machines that are donated by various organizations is a "human scale" job. Moreover, because the enforcer is robust to failed nodes, adding machines to the in-list requires only light vetting, and removing crashed or compromised machines from the in-list can happen lazily (e.g., as the result of background auditing by the bunker or by other nodes). Of course, the bunker is a single point of vulnerability, but observe that *humans*, not computers, execute most of its functions. Nevertheless, to guard against a compromised bunker, nodes accept only limited daily changes to the in-list.

Clients' queries—e.g., TEST(HASH(HASH(stamp)))—are interpreted by the enforcer as queries on key-value pairs, i.e., as TEST(k) or SET(k, v), where $k = \text{HASH}(v)$. (Throughout, we use k and v to mean keys and values.)

Portals implement TEST and SET by invoking at other nodes a UDP-based RPC interface, internal to the enforcer, of GET(k) and PUT(k, v). To

[5] The bunker is a configuration server. We use the term bunker because it connotes an entity that can do its work without being connected. For example, if the bunker were attacked, it could disseminate the in-list by fax.

ensure that GET and PUT are invoked only by other nodes, the in-list can include nodes' public keys, which nodes can use to establish pairwise shared secrets for lightweight packet authentication (e.g., HMAC [88]).

The rest of this section describes the detailed design of the enforcer. We first specify TEST and SET (§4.4.1) and show that even with crash failures (i.e., down or unreachable nodes), the enforcer guarantees little stamp reuse (§4.4.2). We then show how nodes achieve high throughput with an efficient implementation of PUT and GET (§4.4.3) and a way to avoid degrading under load (§4.4.4). We then consider attacks *on* nodes (§4.4.5–§4.4.6) and attacks *by* nodes, and we argue that a Byzantine failure reduces to a crash failure in our context (§4.4.7). Our design decisions are driven by the challenges in §4.2.2, but the mapping between them is not clean: multiple challenges are relevant to each design decision, and vice versa.

4.4.1 TEST and SET

Each key k presented to a portal in TEST or SET has r *assigned* nodes that *could* store it; these nodes are a "random" subset (determined by k) of enforcer nodes. We say below how to determine r. To implement TEST(k), a portal invokes GET(k) at k's r assigned nodes in turn. The portal stops when either a node replies with a v such that $k = \text{HASH}(v)$, in which case the portal returns v to its client, or else when it has tried all r nodes without such a reply, in which case the portal returns "not found". To implement SET(k, v), the portal chooses *one* of the r assigned nodes uniformly at random and invokes PUT(k, v) there. Pseudo-code for TEST and SET is shown in Figure 4.4. The purpose of 1 PUT and r GETs—as opposed to the usual r PUTs and 1 GET—is to conserve storage.

A key's assigned nodes are determined by consistent hashing [84] in a circular identifier space using r hash functions.[6] The bunker-given identifier mentioned above is a random choice from this space. To achieve near-uniform per-node storage with high probability, each node actually has multiple identifiers [149] deterministically derived from its bunker-given one.

Churn

Churn generates no extra work for the system. To handle *intra-day* churn (i.e., nodes going down and coming up between daily distributions of the

[6] This use of consistent hashing [84] is reminiscent of DHTs [12], but the enforcer and DHTs have different structures and different goals; see §4.10.3.

```
procedure TEST(k)
 v ← GET(k)    // local check
 if v ≠ "not found" then return (v)
 // r assigned nodes determined by in-list
 nodes ← ASSIGNED_NODES(k)
 for each n ∈ nodes  do {
      v ← n.GET(k)    // invoke RPC
      // if RPC times out, continue
      if v ≠ "not found" and k == HASH(v) then return (v)
  }
 // all nodes returned "not found" or timed out
 return ("not found")

procedure SET(k, v)
 PUT(k, v)    // local store
 nodes ← ASSIGNED_NODES(k)
 n ← choose random n ∈ nodes
 n.PUT(k, v)    // invoke RPC
```

FIGURE 4.4—Pseudo-code for TEST and SET in terms of GET and PUT.

in-list), portals do not track which nodes are up; instead they apply to each PUT or GET request a timeout of several seconds with no retry, and interpret a timed-out GET as simply a "not found". (A few seconds of latency is not problematic for the portal's client—an incoming email server—because sender-receiver latency in email is often seconds and sometimes minutes.) Moreover, when a node fails, other nodes do not "take over" the failed node's data: the invariant "every (k, v) pair must always exist at r locations" is not needed for our application.

To handle *inter-day* churn (i.e., in-list changes), the assigned nodes for most (k, v) pairs must not change; otherwise, queries on previously SET stamps (e.g., "yesterday's" stamps) would fail. This requirement is satisfied because the bunker makes only minor in-list changes from day-to-day and because, from consistent hashing, these minor membership changes lead to proportionately minor changes in the assigned nodes [84].

4.4.2 Fault-Tolerance Analysis

We now show how to set r to prevent significant stamp reuse. As mentioned at the beginning of this chapter, we view everyone's attention as one aggregate resource. For this reason, we need only bound *total* stamp reuse; it is

acceptable if some stamps are reused more than others or some recipients get more spam than others.

We will assume that nodes, even subverted ones, do not abuse their *portal* role; we revisit this assumption in §4.4.7.

Our analysis depends on a parameter p, the fraction of the n total machines that fail during a two-day period (recall that an epoch is a day and that nodes store stamps' fingerprints for the current and previous epochs). We do not require that failures are independent, only that p is a reasonably small fraction (e.g., one-tenth or one-fifth). To avoid highly correlated failures (the most extreme of which is $p = 1$), we imagine each node choosing one of several different software implementations. We believe that it is reasonable to presume a few active implementations: an enforcer node's functions, being only a few thousand lines of code (see §4.5), are not hard to re-implement.

We don't distinguish the causes of failures—some machines may be subverted, while others may simply crash. To keep the analysis simple, we also do not characterize machines as reliable for some fraction of the time—we simply count in p any machine that fails to operate perfectly over the two-day period. Nodes that do operate perfectly (i.e., remain up and follow the protocol) during this period are called *good*. We believe that carefully chosen nodes can usually be *good* so that $p = 0.1$, for example, might be a reasonably conservative estimate. Nevertheless, observe that this model is very pessimistic: a node that is offline for a few minutes is no longer good, yet such an outage would scarcely increase total spam.

We first consider each stamp's expected reuse and then show that, with very near certainty, the actual total stamp reuse is close to the expected total—regardless of *which* subset of np nodes fails.

Bounding Expected Reuse Per Stamp

We now give some intuition for why a stamp's expected reuse is small. For a given stamp, reuse stops once the corresponding (k, v) pair is PUT on a good node (at that point, future TESTs will "find" (k, v)). If most enforcer nodes are good, this event usually happens quickly, limiting the total reuse per stamp. There is a possibility (with probability less than p^r) that none of the r assigned nodes is good. In this case, an adversary can reuse the stamp once at each of the n portals. (Infinite reuse is prevented by the "local PUT" in the first line of SET in Figure 4.4.) However, these "lucky" stamps do not worry us: recall that our goal is to keep small the *total* number of reuses

across all stamps. While such "lucky" stamps contribute to this total, they do so in a limited way.

We now make the preceding intuition more precise:

Theorem 4.1 Under the assumptions above, the expected number of uses of a stamp is less than $1/(1-p)^2 + p^r n$.

Proof: Observe that each apparently fresh use of a stamp induces a PUT to an assigned node (because of the receiver-enforcer protocol; see lines 4 and 5 in Figure 4.2). And, as mentioned above, once the stamp is PUT to a good assigned node, the adversary can no longer reuse that stamp successfully. Since PUTs are random, some will be to a node that has already received a PUT for the stamp (in which case the node is bad), while others are to "new" nodes. Each time a PUT happens on a new node, there is a $1-p$ chance that the node is good.

Now, consider a single stamp. We make a worst-case assumption that an adversary *tries* to reuse this stamp an infinite number of times.

Let I_i be an indicator random variable for the event that the stamp needs to be PUT to at least $i-1$ distinct nodes before hitting a good one, and let T_i be the number of PUTs, after $i-1$ distinct nodes have been tried, needed to get to the i^{th} distinct node. As a special case, let $T_{r+1} = n - \sum_{j=1}^{r} T_j$ to reflect the fact that if all r assigned nodes are bad, an adversary can reuse the stamp once at each portal. $\mathbb{E}[I_i] = \Pr[I_i = 1] \leq p^{i-1}$. (The inequality enters because p is the *fraction* of bad nodes, so a key's choice of "bad" nodes happens without replacement.) For $i \in \{1, \ldots, r\}, \mathbb{E}[T_i] = r/(r-i+1)$, since each PUT attempt for the stamp has a $(r-i+1)/r$ chance of selecting a new node. Finally, for $i \in \{1, \ldots, r+1\}$, the random variables I_i and T_i are independent. Then, assuming adversaries try to reuse each stamp *ad infinitum*, the expected number of PUTs (i.e., uses of the stamp) is:

$$
\begin{aligned}
&\mathbb{E}[I_1 T_1 + I_2 T_2 + \cdots + I_r T_r + I_{r+1} T_{r+1}] \\
&= \mathbb{E}[I_1]\mathbb{E}[T_1] + \mathbb{E}[I_2]\mathbb{E}[T_2] + \cdots + \mathbb{E}[I_r]\mathbb{E}[T_r] + \mathbb{E}[I_{r+1}]\mathbb{E}[T_{r+1}] \\
&\leq 1 + p\frac{r}{r-1} + p^2\frac{r}{r-2} + \cdots + p^{r-1}\frac{r}{1} + p^r\left(n - \sum_{j=1}^{r}\frac{r}{r-j+1}\right) \\
&= \sum_{i=0}^{r-1} p^i \frac{r}{r-i} + p^r\left(n - \sum_{j=1}^{r}\frac{r}{r-j+1}\right) \qquad (4.1)
\end{aligned}
$$

$$< \sum_{i=0}^{r-1} p^i \frac{r}{r-i} + p^r n. \tag{4.2}$$

Applying the inequality $r/(r-i) \leq i+1$ and taking the infinite sum, we can bound (4.2):

$$< \sum_{i=0}^{\infty} (i+1)p^i + p^r n = \frac{d}{dp} \sum_{i=0}^{\infty} p^i + p^r n,$$

giving the claimed upper bound of $1/(1-p)^2 + p^r n$. □

If we set $r = 1 + \log_{1/p} n$ and take $p = 0.1$, then, applying the theorem, a stamp's expected number of uses is less than $1/(1-p)^2 + p \approx 1 + 3p = 1.3$, close to the ideal of one use per stamp.

Bounding Total Reuse

Having considered a given stamp's expected reuse, we now aim to show that the actual total reuse stays close to the expected total reuse—*regardless of which subset of nodes fails*. This result means that an adversary can "choose" which *np* nodes fail, with little effect on the total stamp reuse. To establish this result, we first state a theorem and then reflect on what the theorem means in our context. We prove the theorem in Appendix D.

Theorem 4.2 Let K be the number of stamps that are active in a given day. If $K > (6n^2 + 300n)/\epsilon^2$, then, with probability at least $1 - e^{-100}$, there is no subset of size np whose failure leads to more than $(1+\epsilon)$ times the expected total use across all stamps. □

We anticipate $n \approx 1500$ (see §4.6.5). If we want $\epsilon = 0.05$ (i.e., 5% more use than is given by the expectation), then how large must K be for the theorem to hold? We need $K > (6 \cdot (1500)^2 + 300 \cdot 1500)/(0.05)^2 = 5.6$ billion stamps. We are in fact designing for K to be in the hundreds of billions, so the theorem holds for our scenario. Thus, we can be confident that there does not exist an unfortunate subset, that is, one whose failure leads to 5% more than the expected stamp reuse.

* * *

The above assumes that the network never loses RPCs (packet loss leads to extra stamp uses). If packets are lost often enough to substantially affect the enforcer's accuracy, then clients and portals can retry RPCs. Doing so

```
procedure GET(k)
 b ← INDEX.LOOKUP(k)
 if b == null then return ("not found")
 a ← DISK.READ(b)    // array a gets disk block b
 if k ∉ a then    // scan all keys in a
    return ("not found")  // index gave false location
 else return (v)    // v next to k in array a

procedure PUT(k, v)
 if HASH(v) ≠ k then return ("invalid")
 b ← INDEX.LOOKUP(k)
 if b == null then
    b ← DISK.WRITE(k, v)    // write is sequential
    // b is disk block where write happened
    INDEX.INSERT(k, b)
 else    // we think k is in block b
    a ← DISK.READ(b)    // array a gets disk block b
    if k ∉ a then    // false location: k not in block b
       b′ ← DISK.WRITE(k, v)
       INDEX.OVERFLOW.INSERT(k, b′)
```

FIGURE 4.5—Pseudo-code for GET and PUT. A node switches between batches of writes and reads; that asynchrony is not shown.

will lower the effective drop rate and make the false negatives from dropped packets a negligible contribution to total spam. Our implementation does not currently issue such retries.

4.4.3 Implementation of GET and PUT

In our early implementation, nodes stored their internal key-value maps in memory, which let them give fast "found" and "not found" answers to GETs. However, we realized that the total number of stamps that the enforcer must store makes RAM scarce. (For more detail, see Appendix F, which compares the total hardware cost of our current design to the total hardware cost of a design in which keys and values are stored fully in RAM.) Thus, nodes must store keys and values in a way that conserves RAM yet, as much as possible, allows high PUT and GET throughput. The rest of this section describes how they do so. In particular, their key-value stores have the following properties, which we justify below:

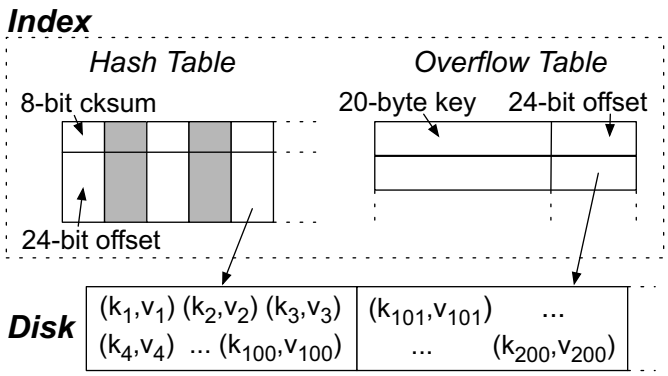


FIGURE 4.6—In-RAM index mapping from k to log block that holds (k, v).

1. PUTs are fast;
2. After a crash, nodes can recover most previously canceled stamps;
3. Each key-value pair costs 5.5 bytes rather than 40 bytes of RAM;
4. "Not found" answers to GETs are almost always fast;
5. "Found" answers to GETs require a disk random access.

As in previous systems [93, 122, 134], nodes write incoming data—key-value pairs here—to a disk log sequentially and keep an index that maps keys to locations in the log. In our system, the index lives in memory and maps keys to log *blocks*, each of which contains multiple key-value pairs. Our index can return *false locations*: it occasionally "claims" that a given key is on the disk even though the node has never stored the key.

When a node looks up a key k, the index returns either "not stored" or a block b. In the latter case, the node reads b from the on-disk log and scans the keys in b to see if k is indeed stored. Pseudo-code describing how GETs and PUTs interact with the index is shown in Figure 4.5.

We now describe the structure of the index, depicted in Figure 4.6. The index has two components. First is a modified open addressing hash table, the entries of which are divided into an 8-bit checksum and a 24-bit pointer to a block (of size, e.g., 4 Kbytes). A key k, like in standard open addressing as described by Knuth, "determines a 'probe sequence,' namely a sequence of table positions that are to be inspected whenever k is inserted or looked up" [86], with insertion happening in the first empty position. When insertion happens, the node stores an 8-bit *checksum* of k as well as a pointer to the block that holds k. (The checksum and probe sequence should be unpredictable to an adversary.) A false location happens when a lookup on key

k finds an entry for which the top 8 bits are *k*'s checksum while the bottom bits point to a block that does not hold *k*. This case is handled by the index's second component, an *overflow table* storing those (k, v) pairs for which *k* wrongly appears to be in the hash table. INDEX.LOOKUP(), in Figure 4.5, checks this table.

Analysis. Our analysis of the index focuses on the memory cost and the lookup speed.

We use the standard assumption that hash functions map each key to a random output, and in particular that the probe sequence for each key is an independent random sequence. We also assume that the checksum is an independent random value. Let $\alpha < 1$ be the load factor (i.e., ratio of non-empty entries to total entries) of the hash table. Let *N* be the number of keys that a node will store. We pessimistically assume that all *N* keys are already in the index.

We first calculate the probability that a key will be inserted in the overflow table. Consider a key, *k*, that the node is about to insert in the index. Each position in the probe sequence is empty with probability $1 - \alpha$. If the entry is not empty, then it has a matching checksum with probability $1/c$, where *c* is the number of distinct checksum values (256 in our case). Thus, a probe has one of three possible outcomes: empty (with probability $1 - \alpha$), matching checksum (with probability α/c) and non-matching checksum (with probability $\alpha(1 - 1/c)$). The node stops probing when one of the first two cases applies. The probability of the second case (matching checksum, which forces *k* into the overflow table), conditioned on the event that the node stopped probing, is equal to the probability of the second case divided by the probability of the first two cases, namely $\frac{\alpha/c}{1-\alpha+\alpha/c} = \frac{\alpha}{c+(1-c)\alpha}$. Thus, under our pessimistic assumption, *k* has probability $\frac{\alpha}{c+(1-c)\alpha}$ of winding up in the overflow table.

Then, if the node stores *N* keys, the expected number of keys in the overflow table is at most $\frac{N\alpha}{c+(1-c)\alpha}$. Each entry in the overflow table takes up 32 bytes (20 bytes for the key, 4 bytes for the disk offset, and 8 bytes for pointers to maintain the data structure; in our implementation, that data structure is a red-black tree). Since the hash table has size N/α and since each entry is exactly 4 bytes, the expected total size of the structure in bytes is

$$N\left(\frac{4}{\alpha} + \frac{32\alpha}{c + (1-c)\,\alpha}\right).$$

Taking $c = 256$ and $\alpha < 1$, the expression above is minimized at $\alpha^* = .85$. At this value of α, the expression equals $5.54N$. (We took c as a fixed quantity because the 8-bit checksum was easy to implement. We could have optimized further by minimizing the expression over values of c and α.)

We now calculate the expected number of lookups per key. Since each entry is empty with probability $1 - \alpha$, a node expects to inspect $1/(1 - \alpha)$ entries before finding the desired key or discovering it is absent. For $\alpha = 0.85$, $1/(1 - \alpha) = 6.66$.

* * *

We now return to the properties above. For property 1, PUTS are fast because the node, rather than interleaving reads and writes, does each in batches, yielding sequential disk writes. Property 2 holds because on booting, a node scans its log to rebuild the index. For property 3: as calculated above, the total expected size of the structure is $5.54N$ bytes, meaning that each of the N (k, v) pairs that the node stores costs 5.54 bytes. Property 4 holds because for *negative* GET(k) requests (i.e., k not found), nodes inspect an average of 6.66 entries in the probe sequence (as calculated above), and the rare false location incurs a disk random access. For *affirmative* GETS (i.e., reused stamps), the node visits an average of 6.66 entries to look up the block, b, that holds v; the node then does a disk random access to get b, as property 5 states.

These disk accesses are one of the enforcer's principal bottlenecks, as shown in §4.6.3. To ease this bottleneck, nodes cache recently retrieved (k, v) pairs in RAM.

Nodes use the block device interface rather than the file system. With the file system, the kernel would, on retrieving a (k, v) pair from disk, put in its buffer cache the entire disk block holding (k, v). However, most of that cached block would be a waste of space: nodes' disk reads exhibit no reference locality.

4.4.4 Avoiding "Distributed Livelock"

The enforcer must not degrade under high load. Such load could be from heavy legitimate use or from attackers' spurious requests (as in the next section). In fact, our implementation's capacity, measured by total correct TEST responses, did originally worsen under load. This section describes our change to avoid this behavior. See §4.6.6 for experimental evidence of the technique's effectiveness.

Observe that the packets causing nodes to do work are UDP RPC re-

quests or responses and that these packets separate into three classes. The classes are: (1) TEST or SET requests from clients; (2) GET or PUT requests from other enforcer nodes; and (3) GET or PUT responses. To achieve the enforcer's throughput goal, which is to maximize the number of successful PUTs and GETs, we have the individual nodes *prioritize these packet classes*. The highest priority class is (3), the lowest (1).

When nodes did not prioritize and instead served these classes round-robin, *overload*—defined as the CPU being unable to do the work induced by all arriving packets—caused two problems. First, each packet class experienced drops, so many GETs and PUTs were unsuccessful since either the request or the response was dropped. Second, the system admitted too many TESTs and SETs, i.e., it overcommitted to clients. The combination was *distributed* livelock: each node spent cycles on TESTs and SETs and meanwhile dropped GET and PUT requests and responses from *other* nodes.

Prioritizing the three classes, in contrast to round-robin, improves throughput and implements admission control: a node, in its role as portal, commits to handling a TEST or SET only if it has no other pending work in its role as node. We can view the work induced by a TEST or SET as a *distributed* pipeline; each stage is the arrival at any node of a packet related to the request. In this view, a PUT response, for example, indicates that the enforcer as a *whole* has done most of the work for the underlying SET request; dropping such a packet wastes work.

To implement the priorities, each of the three packet classes goes to its own UDP destination port and thus its own queue (socket) on the node. The node reads from the highest priority queue (socket) with data. If the node cannot keep up with a packet class, the associated socket buffer fills, and the kernel drops packets in that class.

An alternate way to avoid distributed livelock might be for a node to maintain a set of windows, one for every other node, of outstanding RPCs. With this approach, each node's incoming request queue would be bounded by the window size times the number of nodes, and hence no node would be overloaded. The reason that we rejected this approach is that we did not know how to set the size of the window.

The general approach described in this section—which applies the principle that, under load, one should drop from the beginning of a pipeline to maximize throughput—could be useful for other distributed systems. There is certainly much work addressing overload: see, e.g., SEDA [175, 176], LRP [44], and Defensive Programming [121] and their bibliographies; these proposals use fine-grained resource allocation to protect servers from

overload. Other work (see, e.g., Neptune [141] and its bibliography) focuses on clusters of equivalent servers, with the goal of proper allocation of requests to servers. All of this research concerns requests of single hosts, unlike the simple priority scheme described here, which concerns logical requests happening on several hosts.

Discussion. In reflecting on the technique just presented, we make two points. First, we have discussed *only* how to maintain throughput in the face of heavy load; we have said nothing about *which* requests and clients are served. In particular, one could imagine that, under attack, the enforcer keeps constant throughput but spends most of its resources on bad clients. The next section describes our defense to this situation.

Our second point is that the technique, as described, is a heuristic. It achieves a particular goal (maximizing the number of successful PUTs and GETs), and that goal serves our purpose, which is to prevent the number of correct TEST responses from decreasing under load. However, the real goal should be to *maximize* the correct TEST responses under load—giving such responses is the whole purpose of the enforcer. To achieve this other goal, a different priority scheme is likely needed. In particular, it is not at all clear that PUTs and GETs should have the same priority, that TESTs and SETs should have the same priority, or that all GET responses should have the same priority (e.g., perhaps responses from different "stages" of a given TEST request should be prioritized differently). However, there are difficult questions here because the priority scheme will in turn affect the ratio of TESTs and SETs that are presented, and, also, adversaries might try to game the scheme to thwart the enforcer. We are leaving to future work the question of what priority scheme is optimal.

4.4.5 Resource Exhaustion Attacks

Several years ago, a popular DNS-based block list (DNSBL) was forced offline [69], and a few months later another such service was attacked [159], suggesting that effective anti-spam services with open interfaces are targets for denial-of-service (DoS) attacks. If successful, DQE would be a major threat to spammers, so we must ensure that the enforcer resists attack. We do not focus on *packet floods*, in which attackers exhaust the enforcer's bandwidth with packets that are not well-formed requests. These attacks can be handled with various commercial (e.g., upstream firewalls) and academic (see [108] for a survey) solutions. We thus assume that enforcer nodes see only well-formed RPC requests.

The attack that we focus on in this section is a flood of spurious RPCs, which we call a *resource exhaustion attack*. The aim is to waste nodes' resources, specifically: disk random accesses on affirmative GETs, entries in the RAM index (which is exhausted long before the disk fills) for PUTs, and CPU cycles to process RPCs. This attack is difficult because one cannot differentiate "good" from "bad": requests are TEST(k) and SET(HASH(v), v) where k, v are any 20-byte values. Absent further mechanism, handling this attack requires the enforcer to be provisioned for the legitimate load plus as many TESTs and SETs as the attacker can send.

Resource exhaustion attacks appear difficult. Indeed, they have all of the vexing characteristics of the abstract problem in §1.1. Yet, *resource exhaustion attacks are a kind of application-level DDoS, so we can apply speak-up!*

We said in §3.3 that speak-up works best under conditions C1 and C2. We now revisit those conditions in the context of attacks on the enforcer. We begin with C2, which says that the enforcer should have ample bandwidth.

Attacks on the enforcer could be far larger than the attacks on individual servers that we discussed in Chapter 3. Those attacks are conducted by one or a small number of botnets. In contrast, the enforcer, if deployed, would be a threat to spammers' livelihood and could very well cause them to "join forces" and launch large aggregate attacks. If there are 20 million bots worldwide (see §2.2), and each has an average bandwidth of 100 Kbits/s (as mentioned in §3.10.2 based on a study [143]), then the adversaries could launch 2 Tbits/s of attack traffic. This amount is daunting, but the enforcer already needs to comprise roughly a thousand nodes (see §4.6.5). If each node has a bandwidth of 1 Gbit/s, then the enforcer would have an aggregate bandwidth of at least 1 Tbit/s, which is in striking distance of the largest possible attack. Observe that nodes need not have this bandwidth "year-round"; they could acquire it temporarily, when needed. So condition C2 seems within reach.

Condition C1, on the other hand, might go unmet: the good clients of the enforcer, the world's email servers, may not in aggregate have, or want to spend, terabits of bandwidth. But this unmet condition does not necessarily disqualify speak-up in this scenario. The reason is as follows. Recall that the purpose of condition C1 is to ensure that a server protected by speak-up need not over-provision much, relative to the good demand, to give all of its good clients service (see §3.4.1). However, in *this* scenario, the enforcer *plans* on most of its work being caused by adversarial behavior—because most email is spam. Moreover, as a result of this planning, we might be

able to dispense not only with C1 but also with a bandwidth-proportional allocation. We make these points more concrete below.

Defending the Enforcer with Options Inspired by Speak-up

We have claimed that speak-up can defend the enforcer. More accurately, the enforcer can employ any of several defenses that are inspired by speak-up. The choices are as follows, and we discuss them in turn:

1. Every node always charges a fixed bandwidth price for a TEST or a SET.
2. When overloaded, and only when overloaded, a node charges a fixed bandwidth price for a TEST or a SET.
3. When overloaded, and only when overloaded, a node conducts bandwidth auctions, i.e., it applies speak-up as described in Chapter 3.

Fixed bandwidth price. To explain why the first choice can defend against resource exhaustion attacks, and to see why both C1 and a bandwidth-proportional allocation are dispensable, let us make an assumption—which we revisit shortly—that attackers are *sending as much spam as they can*. Specifically, let us assume that they are limited by bandwidth. As in Chapter 3, this limit reflects either a constraint like access links or some threshold above which the attacker fears detection by the human owner of a compromised machine.

Observe that, independent of our assumption, the enforcer is indifferent between the attacker sending (1) a spurious TEST and (2) a single spam message, thereby inducing a legitimate TEST (and, rarely, a SET); the resources consumed by the enforcer are the same in both cases. Now, under the assumption above, we can neutralize resource exhaustion attacks by arranging for a TEST or a SET to require *the same amount of bandwidth as sending a spam*. Our reasoning is as follows. If attackers are "maxed out" and if sending a TEST and a spam costs the same bandwidth, then attackers cannot cause more TESTs and SETs than would be induced anyway by current email volumes—*for which the enforcer must already be provisioned*.

Of course, despite our assumption above, today's attackers are unlikely to be "maxed out". However, they have *some* bandwidth limit. If this limit and current spam volumes are the same order of magnitude, then the approach described here means that the enforcer's required provisioning, relative to what is already required to handle the world's spam volume, is a small constant factor.

If, however, a more pessimistic case materializes—the most extreme of which is our back-of-the-envelope calculation above of 2 Tbits/s of attack traffic—then the enforcer needs more provisioning or else needs to charge a price higher than the average number of bits in a spam. Concretely, if we assume that the enforcer is 1500 nodes and that each node has worst-case capacity 320·4 requests/s (see §4.6.5), then the aggregate capacity is ∼2 million requests/s, so the average price per request needs to be roughly (2 Tbits/s) / (2 million requests/s), which is approximately 1 Mbit, or 125 Kbytes, per request. If legitimate DQE clients cannot afford this price, then the enforcer needs more provisioning, which would lower the required price. For perspective, we now compare this fully pessimistic case to the optimistic baseline under this variant of speak-up. That baseline, as described above, is to set the price of a TEST equal to the average size of a spam. Many spams are on the order of 10 Kbytes. Thus, the optimistic price, roughly 10 Kbytes, is about 10× better than the fully pessimistic scenario, in terms of its effect on either the bandwidth cost to legitimate clients (125 Kbytes, assuming the same provisioning) or on the required enforcer over-provisioning.

In any case, all of this estimated over-provisioning is an upper bound: the most damaging spurious request is a TEST that causes a disk access by asking a node for an existing stamp fingerprint (§4.6.3), yet nodes cache key-value pairs (§4.4.3). If, for example, half of spurious TESTs generate cache hits, the required provisioning halves.

Fixed bandwidth price only under overload. The advantage of this approach is that when the enforcer is not under attack, good clients and the enforcer do not consume extra bandwidth. The disadvantage is that attackers can induce more TESTs and SETs, compared to the previous approach. The reason is as follows. Before a node is overloaded, TESTs and SETs at that node are "cheap". Thus, attackers can (a) make TEST and SET requests at each of the nodes, stopping just short of pushing them into overload and (b) spend their remaining bandwidth budget on sending spam (here, the attacker is "paying retail" for the TESTs and SETs induced by the spam). The result of this strategy is that attackers get some TESTs and SETs "at a discount", thereby allowing them to induce more than they could in the previous approach.

This disadvantage is limited because attackers can only get this bargain when consuming the *spare* capacity at each of the nodes. And the enforcer can compensate for this effect with increased provisioning.

Full-blown speak-up. The advantage of full-blown speak-up is that it may induce a much higher price, causing attackers to have even less impact. The disadvantage is that this higher price costs more bandwidth for the legitimate clients and the enforcer. Observe that charging less than this price still restricts attackers (as we argued for the previous two approaches) though gives them more than a bandwidth-proportional share. Yet, as mentioned earlier in this section, we probably do not need to restrict attackers to a bandwidth-proportional share because the enforcer is provisioned to give bad clients—and the requests that they induce—a disproportionate share anyway.

* * *

So far, we have not specified how enforcer nodes charge bandwidth. They have several options, including asking for long requests or demanding many copies of each request. In fact, the enforcer need not charge bandwidth: the discussion above could apply to a currency of CPU or memory cycles.

We have not yet addressed hotspots. Recall that we are not discussing link attacks in this section, so the type of hotspot that we now consider is a portal overloaded by spurious RPCs. If any particular portal is attacked, clients can use another one. Moreover, a bandwidth-limited attacker may not *want* to overload a portal because doing so amounts to wasting work that could be better spent issuing spurious TESTs and SETs to *other* portals. And, this adversarial strategy of not wasting work is precisely what the enforcer as a whole is already provisioned for, regardless of whether the spurious requests are concentrated at individual portals.

Finally, we note that even if the enforcer is knocked offline temporarily, the system can recover. During periods of enforcer outage, email servers can fall back on other spam defenses in the whitelisting family (see §4.10.1, page 126), perhaps queuing for later verification the emails that must go through the DQE checks.

4.4.6 Widespread, Simultaneous Stamp Reuse

One might imagine an attack in which the adversary simultaneously sends the same stamp to many recipients, in the hope of overloading the assigned nodes or otherwise thwarting the enforcer. However, we believe (but have not experimented to verify) that the enforcer will handle this attack properly. Consider a single portal that receives TEST(k), where k corresponds to the stamp in question. If the TEST arrives before any client has submitted SET(k, v), then the portal will return "not found". However, the window in

which portals are "ignorant" of the stamp is small: any client that receives "not found" will SET the (k, v) pair that corresponds to the stamp, at which point all future calls of TEST(k) will "find" (k, v).

This scenario does not overload the assigned nodes, for two reasons. First, the main bottleneck for an assigned node is disk random accesses (see §4.6.3), yet only one disk access per assigned node is required because the node will store the (k, v) pair in its RAM cache (§4.4.3). Second, portals cache in RAM the responses to TESTs and also perform "local PUTs" when clients call SET (§4.4.1). As a result, when a portal receives TEST(k) for a "popular" k, the portal will likely not need to contact the assigned node in the first place.

4.4.7 Adversarial Nodes

We now argue that for the protocol described in §4.4.1, a Byzantine failure reduces to a crash failure. Nodes do not route requests for each other. A node cannot lie in response to GET(k) because for a false v, HASH(v) would not be k (so a node cannot make a fresh stamp look reused). A node's only attack is to cause a stamp to be reused by ignoring PUT and GET requests, but doing so is indistinguishable from a crash failure. Thus, the analysis in §4.4.2, which applies to crash failures, captures the effect of adversarial nodes. Of course, depending on the deployment (federated or monolithic), one might have to assume a higher or lower p.

However, the analysis does not cover a node that abuses its portal role and endlessly gives its clients false negative answers, letting much spam through. Note, though, that if adversarial portals are rare, then a random choice is unlikely to find an adversarial one. Furthermore, if a client receives much spam with apparently fresh stamps, it may become suspicious and switch portals, or it can query multiple portals.

Another attack for an adversarial node is to execute spurious PUTs and GETs at other nodes, exhausting their resources. In defense, nodes maintain "put quotas" and "get quotas" for each other, which relies on the fact that the assignment of (k, v) pairs to nodes is balanced. Deciding how to set and apply these quotas is future work. The challenge is that a node will need to make *instantaneous* decisions yet will need to allocate fairly an *aggregate* resource—its capacity (total number of disk accesses and total RAM consumed) over the course of a day.

4.4.8 Limitations

The enforcer may be clustered, wide-area, or a combination of the two. Because our present concern is throughput, our implementation and evaluation are geared only to the fully clustered case. We briefly consider the wide-area case now. If the nodes are separated by low capacity links, distributed livelock avoidance (§4.4.4) is not needed, but congestion control is. Options include long-lived pairwise DCCP [87] connections or a scheme like STP in Dhash++ [39].

4.5 IMPLEMENTATION

We describe our implementation of the enforcer nodes and DQE client software; the latter runs at email senders and receivers, and handled the inbound and outbound email of several users for over six months.

4.5.1 Enforcer Node Software

The enforcer is a 5000-line event-driven C++ program that exposes its interfaces via XDR RPC over UDP. It uses libasync [101] and its asynchronous I/O daemon [93]. We modified libasync slightly to implement distributed livelock avoidance (§4.4.4). We have successfully tested the enforcer on Linux 2.6 and FreeBSD 5.3. We play the bunker role ourselves by configuring the enforcer nodes with an in-list that specifies random identifiers. We have not implemented per-portal quotas to defend against resource exhaustion by adversarial nodes (§4.4.7), a speak-up-related defense against resource exhaustion by adversarial clients (§4.4.5), HMAC for inter-portal authentication (§4.4), or rate-limiting of inbound PUTs (§4.6.3).

4.5.2 DQE Client Software

The DQE client software is two Python modules. The *sender* module is invoked by a `sendmail` hook; it creates a stamp (using a certificate signed by a virtual quota allocator) and inserts it in a new header in the departing message. The *receiver* module is invoked by `procmail`; it checks whether the email has a stamp and, if so, executes a TEST RPC over XDR to a portal. Depending on the results (no stamp, already canceled stamp, forged stamp, etc.), the module adds a header to the email for processing by filter rules. To reduce client-perceived latency, the module first delivers email to the recipient and then, for fresh stamps, asynchronously executes the SET.

The analysis (§4.4.2) accurately reflects how actual failures affect observed stamp reuse. Even with 20% of the nodes faulty, the average number of reuses is under 1.5.	§4.6.2
Microbenchmarks (§4.6.3) predict the enforcer's performance exactly. The bottleneck is disk accesses.	§4.6.4
The enforcer can handle 200 billion emails per day (a multiple of the current email volume) with a few thousand PCs. More specifically, the enforcer needs ∼5400 disks to handle this volume, provided that the peak:average ratio is 1.	§4.6.5
The scheme to avoid livelock (§4.4.4) meets its goal of preventing the rate of correct TEST responses from degrading under load.	§4.6.6

TABLE 4.1—Summary of evaluation results.

4.6 EVALUATION OF THE ENFORCER

In this section, we evaluate the enforcer experimentally. We first investigate how its observed fault-tolerance—in terms of the average number of stamp reuses as a function of the number of faulty machines—matches the analysis in §4.4.2. We next investigate the capacity of a single enforcer node, measure how this capacity scales with multiple nodes, and then estimate the number of dedicated enforcer nodes needed to handle 200 billion emails per day (our target volume; see §4.2.2). Finally, we evaluate the livelock avoidance scheme from §4.4.4. Table 4.1 summarizes our results.

All of our experiments use the Emulab testbed [47]. In these experiments, between one and 64 enforcer nodes are connected to a single LAN, modeling a clustered network service with a high-speed access link.

4.6.1 Environment

Each enforcer node runs on a separate Emulab host. To simulate clients and to test the enforcer under load, we run up to 25 instances of an open-loop tester, *U* (again, one per Emulab host). All hosts run Linux FC4 (2.6 kernel) and are Emulab's "PC 3000s", which have 3 GHz Xeon processors, 2 GBytes of RAM, 100 Mbits/s Ethernet interfaces, and 10,000 RPM SCSI disks.

Each *U* follows a Poisson process to generate TESTs and selects the portal for each TEST uniformly at random. This process models various email servers sending TESTs to various enforcer nodes. (As argued in [115], Pois-

son processes appropriately model a collection of many random, unrelated session arrivals in the Internet.) The proportion of *reused* TESTs (stamps[7] previously SET by U) to *fresh* TESTs (stamps never SET by U) is configurable. These two TEST types model an email server receiving a spam or non-spam message, respectively. In response to a "not found" reply—which happens either if the stamp is fresh or if the enforcer lost the reused stamp—U issues a SET to the portal that it chose for the TEST.

Our reported experiments run for 12 or 30 minutes. Separately, we ran a 12-hour test to verify that the performance of the enforcer does not degrade over time.

4.6.2 Fault-Tolerance

We investigate whether failures in the implemented system reflect the fault-tolerance analysis. Recall that this analysis (in §4.4.2) upper bounds the expected number of stamp uses in terms of the fraction of *bad* nodes, p. The analysis also shows that the worst-case stamp reuse is very close to the expectation. For this reason, we focus on the expectation in this section.

Recall that a node is considered *bad* if it is ever down while a given stamp is relevant (two days). Below, we model "bad" with crash faults, only. We do not model Byzantine faults explicitly because, as mentioned in §4.4.7, a Byzantine fault has the same effect as a crash fault—causing a stamp to be reused.

We run two experiments in which we vary the number of bad nodes. These experiments measure how often the enforcer—because some of its nodes have crashed—fails to "find" stamps it has already "heard" about.

In the first experiment, called *crashed*, the bad nodes are never up. In the second, called *churning*, the bad nodes repeat a 90-second cycle of 45 seconds of down time followed by 45 seconds of up time. Both experiments run for 30 minutes. The Us issue TESTs and SETs to the up nodes, as described in §4.6.1. Half of the TESTs are for fresh stamps, and the other half are for a *reuse group*—843,750 reused stamps that are each queried 32 times during the experiment. This group of TESTs models an adversary trying to reuse a stamp. The Us count the number of "not found" replies for each stamp in the reuse group; each such reply counts as a stamp use. We set $n = 40$, and the number of bad nodes is between 6 and 10, so p varies between 0.15 and 0.25. For the replication factor (§4.4.1), we set $r = 3$.

[7] In this section, we often use "stamp" to refer to the key-value pair associated with the stamp.

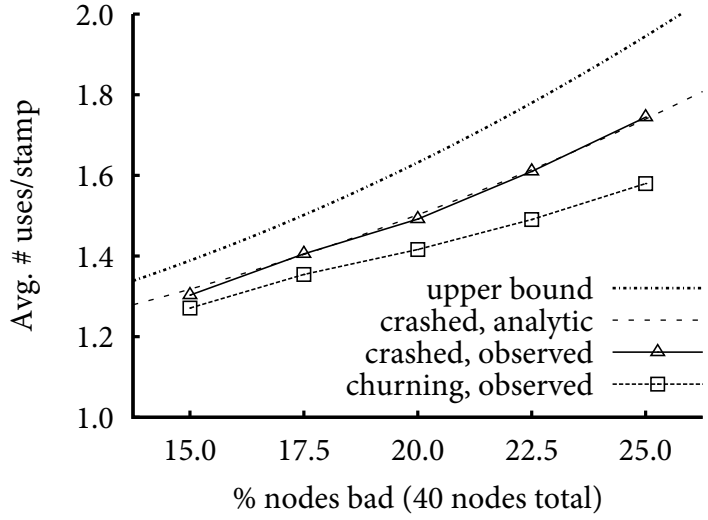


FIGURE 4.7—Effect of "bad" nodes on stamp reuse for two types of "bad". Observed uses obey the upper bound from the analysis (see §4.4.2). The crashed case can be analyzed exactly; the observations track this analysis closely.

The results are depicted in Figure 4.7. The two "observed" lines plot the average number of times a stamp in the "reuse group" was used successfully. These observations obey the model's least upper bound. This bound, from equation (4.1) in §4.4.2, is $1+\frac{3}{2}p+3p^2+p^3\left[40\,(1-p)-\left(1+\frac{3}{2}+3\right)\right]$ and is labeled "upper bound". (We take $n=40(1-p)$ instead of $n=40$ because, as mentioned above, the Us issue TESTs and SETs only to the "up" nodes.) The *crashed* experiment is amenable to an exact expectation calculation. The resulting expression[8] is depicted by the line labeled "crashed, analytic"; it matches the observations well.

4.6.3 Single-node Microbenchmarks

We now examine the performance of a single-node enforcer.

RAM. We begin by considering RAM and asking how it limits the number of PUTs. Each key-value pair consumes roughly 5.5 bytes of memory in expectation (§4.4.3), and each is stored for two days (§4.3.3). Thus, with one GByte of RAM, a node can store slightly fewer than 200 million key-

[8]The expression is as follows. Let $m = 40(1-p)$. The expression is $(1-p)^3(1) + 3p^2(1-p)\alpha + 3p(1-p)^2\beta + p^3 m\left(1-\left(\frac{m-1}{m}\right)^{32}\right)$. α is $\sum_{i=1}^{m} i\left(\frac{2}{3}\right)^{i-1}\frac{1}{m}\left(1+\frac{m-i}{3}\right)$, and β is $\sum_{i=1}^{m-1} i\left(\frac{1}{3}\right)^{i-1}\frac{m-i}{m(m-1)}\left(2+\frac{2}{3}(m-i-1)\right)$. See Appendix E.1 for a derivation.

value pairs, which, over two days, is roughly 1,100 PUTs per second. A node can certainly accept a higher average rate over any given period but must limit the total number of PUTs it accepts each day to 100 million for every GByte of RAM.

Disk. We next ask how the disk limits GETs. (The disk does not bottleneck PUTs because writes are sequential and because disk space is ample.) Consider a key k requested at a node d. We call a GET *slow* if d stores k on disk (if so, d has an entry for k in its index) and k is not in d's RAM cache (see §4.4.3). We expect d's ability to respond to slow GETs to be limited by disk random accesses. To verify this belief, an instance of U sends TESTs and SETs at a high rate to a single-node enforcer, inducing local GETs and PUTs. The node runs with its cache of key-value pairs disabled. The node responds to an average of 400 slow GETs per second (measured over 5-second intervals, with standard deviation less than 10% of the mean).

To understand this performance, we benchmarked the disk as follows. We wrote a utility that sits in a tight loop, doing random access disk reads within a contiguous "blob"; the size of this blob is configurable. To get disk parallelism, we run eight instances of the utility (which models the eight async I/O daemons used by our implementation; see §4.5). We find that when the size of the blob is small (e.g., 2 GBytes), a node can do 400 random accesses per second, which matches the single-node local GET rate observed above. But this number is quite high![9] The reason that the disk can do so many accesses per second is that, with only 2 GBytes of data, the "blob" is a narrow band on the disk, so the disk head (which is presumably following an elevator scheduling algorithm) can read multiple key-value pairs per rotation. Likewise, in the local GET experiment, U was not running for long, so the universe of reused keys was small, so nodes were not doing random accesses over a large chunk of the disk.

A more appropriate blob size for our purposes is 16 GBytes: this size models a node with 2 GBytes of RAM storing 400 million key-value pairs on a single disk. This blob size is pessimistic, first, because nodes could have multiple disks, thereby storing fewer key-value pairs per disk. Second, 400 million key-value pairs is likely far beyond what a node needs to store: our back-of-the-envelope in §4.6.5 finds 50 billion new key-value pairs per day. Divided over 1000 nodes (an under-estimate), the per-node total is only

[9] I am grateful to Bill Bolosky for observing that 400 accesses per second is much larger than would be expected, given a disk that spins at 10,000 RPM, or ∼167 rotations per second.

Operation	Ops/sec	Bottleneck
PUT	1,100	RAM
slow GET	320	disk
fast GET	38,000	CPU

TABLE 4.2—Single-node performance, assuming 1 GByte of RAM.

100 million key-value pairs in storage (because nodes store key-value pairs from two days). In any case, when the blob size is 16 GBytes, the benchmark is limited to 320 random accesses per second. We will use this more pessimistic disk capacity when estimating the required size of the enforcer later in this section.

CPU. We next consider *fast* GETs, which are GETs on keys k for which the node has k cached or is not storing k. In either case, the node can reply quickly. For this type of GET, we expect the bottleneck to be the CPU. To test this hypothesis, U again sends many TESTs and SETs. Indeed, CPU usage reaches 100% (again, measured over 5-second intervals with standard deviation less than 10% of the mean), after which the node can handle no more than 38,000 RPCs. A profile of our implementation indicates that the specific CPU bottleneck is `malloc()`.

Table 4.2 summarizes the above findings.

4.6.4 Capacity of the Enforcer

We now measure the capacity of multiple-node enforcers and seek to explain the results using the microbenchmarks just given. We define capacity as the maximum rate at which the system can respond correctly to the reused requests. Knowing the capacity as a function of the number of nodes will help us, in the next section, answer the dual question: how many nodes the enforcer must comprise to handle a given volume of email (assuming each email generates a TEST).

The measured capacity will depend on the workload. Specifically, which resource is the bottleneck—RAM or disk—depends on the ratio of fresh to reused TESTs. The reason is that fresh TESTs consume RAM (the SETs that follow these TESTs induce PUTs) while reused TESTs may incur a disk random access.

Note that the resources consumed by a TEST are different in the multiple-node case. A TEST now generates r (or $r-1$, if the portal is an assigned node)

GET RPCs, each of which consumes CPU cycles at the sender and receiver. A reused TEST still incurs only one disk access in the entire enforcer (since a portal issues GETs sequentially and stops once a node replies affirmatively). The resources consumed by a SET are also different in the multiple-node case: a SET now induces two PUTs (one remote and one local).

32-node experiments. We first determine the capacity of a 32-node enforcer. To emulate the per-node load of a several thousand-node deployment, we set $r = 5$. We set r to this value because, from §4.4.2, $r = 1 + \log_{1/p} n$; we take $p = 0.1$ and n to be several thousand, which is the upper bound in §4.6.5. Note that this reasoning is not circular: the upper bound on n is not determined by r but rather by disk or RAM capacity.

We run two groups of experiments in which 20 instances of U send half fresh and half reused TESTs at various rates to this enforcer. In the first group, called *disk*, the nodes' LRU caches are disabled, forcing a disk random access for every affirmative GET (§4.4.3). In the second group, called *CPU*, we enable the LRU caches and set them large enough that stamps will be stored in the cache for the duration of the experiment. The first group of experiments is fully pessimistic and models a disk-bound workload whereas the second is (unrealistically) optimistic and models a workload in which RPC processing is the bottleneck. We ignore the RAM bottleneck in these experiments but consider it at the end of the section.

Each node reports how many reused TESTs it served over the last 5 seconds (if too many arrive, the node's kernel silently drops). Each experiment run happens at a different TEST rate. For each run, we produce a value by averaging together all of the nodes' 5-second reports. Figure 4.8 graphs the positive response rate as a function of the TEST rate. The left and right y-axes show, respectively, a per-node per-second mean and a per-second mean over all nodes; the x-axis is the aggregate sent TEST rate. (The standard deviations are less than 9% of the means.) The graph shows that maximum per-node capacity is 400 reused TESTs/sec when the disk is the bottleneck and 1,875 reused TESTs/sec when RPC processing is the bottleneck; these correspond to 800 and 3,750 total TESTs/sec (recall that half of the sent TESTs are reused).

The microbenchmarks explain these numbers. The per-node disk capacity is exactly what we observed in the single-node case (in both the experiment just done and the single-node experiment, the "blob" size is not large, so the per-node capacity is high, relative to what one would expect from a 10,000 RPM disk). We now connect the per-node TEST-processing

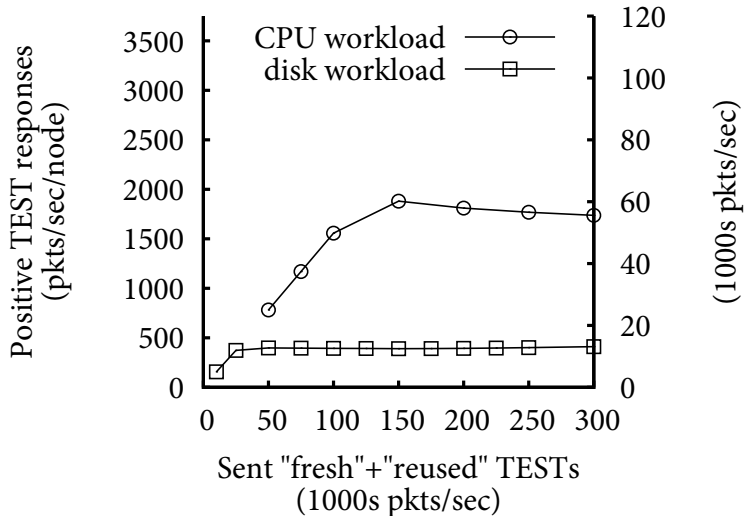


FIGURE 4.8—For a 32-node enforcer, mean response rate to TEST requests as function of sent TEST rate for disk- and CPU-bound workloads. The two y-axes show the response rate in different units: (1) per-node and (2) over the enforcer in aggregate. Here, $r = 5$, and each reported sample's standard deviation is less than 9% of its mean.

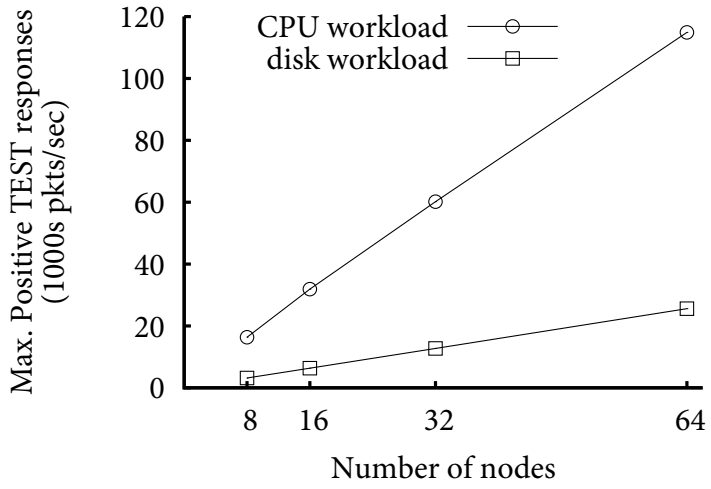


FIGURE 4.9—Enforcer capacity under two workloads as a function of number of nodes in the enforcer. The y-axis is the same as the right-hand y-axis in Figure 4.8. Standard deviations are smaller than 10% of the reported means.

rate (3,750 per second) to the RPC-processing microbenchmark (38,000 per second). Recall that a TEST generates multiple GET requests and multi-

ple GET responses (how many depends on whether the TEST is fresh). Also, if the stamp was fresh, a TEST induces a SET request, a PUT request, and a PUT response. Taking all of these "requests" together (and counting responses as "requests" because each response also causes the node to do work), the average TEST generates 9.95 "requests" in this experiment, as shown in Appendix E.2. Thus, 3,750 TEST requests per node per second is 37,312 "requests" per node per second, which is within 2% of the microbenchmark from §4.6.3 (last row of Table 4.2).

One might notice that the CPU line in Figure 4.8 degrades after 1,875 positive responses per second per node (the enforcer's RPC-processing capacity). The reason is as follows. Giving the enforcer more TESTs and SETs than it can handle causes it to drop some. Dropped SETs cause some future *reused* TESTs to be seen as *fresh* by the enforcer—but fresh TESTs induce r or $r - 1$ GETs while reused TESTs induce roughly $(r + 1)/2$ GETs on average since a portal stops querying when it gets a positive response. Thus, the degradation happens because extra RPCs from fresh-*looking* TESTs consume capacity. This degradation is not ideal, but it does not continue indefinitely.

Scaling. We now measure the enforcer's capacity as a function of the number of nodes, hypothesizing near-linear scaling. We run the same experiments as for 32 nodes but with enforcers of 8, 16, and 64 nodes. Figure 4.9 plots the maximum point from each experiment. (The standard deviations are smaller than 10% of the means.) The results confirm our hypothesis across this (limited) range of system sizes: an additional node at the margin lets the enforcer handle, depending on the workload, an additional 400 or 1,875 TESTs/sec—the per-node averages for the 32-node experiment.

We now view the enforcer's scaling properties in terms of its request mix. Assume pessimistically that all reused TEST requests cost a disk random access. Then, doubling the rate of spam (reused TEST requests) will double the required enforcer size.

Doubling the rate of *non*-spam, however, (i.e., fresh TEST requests) will only affect the required enforcer size if there is enough non-spam so that the enforcer's resource bottleneck is RAM. To be concrete, assume that each node in the enforcer has 1 GByte of RAM and receives 200 reused TESTs/second and 550 fresh TESTs/second. Then, the enforcer is correctly provisioned, but RAM is the resource that is driving the provisioning. For in this case, each node processes an average of 550 SETs/second (recall that each fresh TEST is followed by a SET) and 1,100 PUTs/second (recall

200 billion		emails daily (target from §4.2.2)
×	75%	spam [106, 107, 150]
150 billion		disk random accesses / day (pessimistic)
÷	320	disk random accesses / second / disk (§4.6.3)
÷	86400	seconds / day
	5425	disks needed
÷	4	disks / node (as one possibility)
	1356	**nodes needed**

TABLE 4.3—Estimate of enforcer size (based on average rates), and assuming each node has 4 disks.

that each SET induces a remote PUT and a local one), which is the single-node performance limit (see Table 4.2). More generally, RAM becomes a bottleneck—and the rate of non-spam drives the required enforcer size—when the ratio of fresh TESTs to reused TESTs is greater than or equal to the ratio of a single node's performance limits, namely 550 fresh TESTs/sec for every GByte of RAM to 320 reused TESTs/sec for every disk. (We assume 320, rather than 400, disk random accesses per node per second, because 320 better models a node that is under load; see §4.6.3.)

4.6.5 Estimating the Enforcer Size

We now give a rough estimate of the number of dedicated enforcer nodes required to handle current email volumes. We will assume that putting four disks in a node (a) is a reasonable thing to do; and (b) quadruples the node's capacity to handle disk random accesses. We believe that (a) is true based on server configurations and the cost of disks. We did not experiment to verify (b).

The calculation is summarized in Table 4.3. Our target is 200 billion messages daily. Current estimates suggest that the percentage of all email that is spam is roughly 75% [106, 107, 150].[10] We make the worst-case assumption that every reused TEST—each of which models a spam message—causes the enforcer to do a disk random access.

With this assumption and at this spam rate, the scaling bottleneck is disk capacity, not RAM. To show why, we follow the discussion at the end of the last section. For RAM to be the bottleneck, the node must have fewer

[10] The estimates vary somewhat. Some of them are as high as 90% (and in other months, the estimates have been as low as 60%).

than x GBytes of RAM, where x satisfies the following inequality: $1/3 > 550x/(320 \cdot 4)$. (This inequality says that the ratio 25% fresh to 75% reused TEST requests is greater than the ratio of a single node's RAM bottleneck to its random access disk capacity.) This inequality holds for $x < 0.78$. Most modern machines have much more RAM than 0.78 GBytes, so we conclude that disk, not RAM, is the scaling bottleneck (even with 4 disks per node).

The enforcer must do 150 billion disk random accesses per day and, since the required enforcer size scales linearly with the number of required disk accesses (§4.6.4), a straightforward calculation gives the required number of machines. For the disks in our experiments, the number is about 1,300 machines.

* * *

So far we have considered only average request rates. We must ask how many machines the enforcer needs to handle peak email loads while bounding reply latency. To answer this question, we would need to determine the peak-to-average ratio of email reception rates at email servers (their workload induces the enforcer's workload). As one data point, we analyzed the logs of our research group's email server, dividing a five-week period in early 2006 into 10-minute windows. The maximum window saw 4 times the volume of the average window. Separately, we verified with a 14-hour test that a 32-node enforcer can handle a workload of like burstiness with worst-case latency of 10 minutes. Thus, if global email is this bursty, the enforcer would need 5,400 machines (the peak-to-average ratio times the 1,300 machines derived above) to give the same worst-case latency.

However, global email traffic is likely far smoother than one server's workload. And spam traffic may be smoother still: the spam in Jung et al.'s 2004 data [81] exhibits—over ten minute windows, as above—a peak-to-average ratio of 1.9:1. Also, Gomes et al. [62] claim that spam is less variable than legitimate email. Thus, many fewer than 5,400 machines may be required. On the other hand, the enforcer may need some over-provisioning for spurious TESTS (§4.4.5). For now, we conclude that the enforcer needs "a few thousand" machines and leave to future work a study of email burstiness and attacker ability.

4.6.6 Avoiding "Distributed Livelock"

We now briefly evaluate the scheme to avoid livelock (from §4.4.4). The goal of the scheme is to prevent the rate of correct TEST responses from degrading under high load. To verify that the scheme meets this goal, we run the

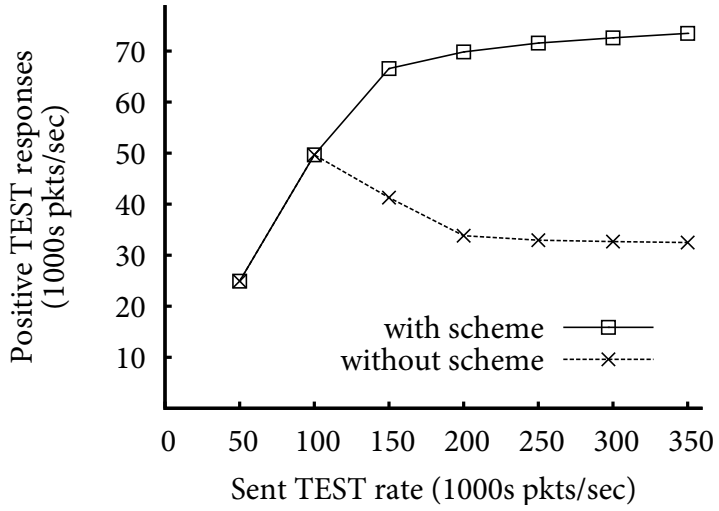


FIGURE 4.10—Effect of livelock avoidance scheme from §4.4.4. As the sent TEST rate increases, the ability of an enforcer without the scheme to respond accurately to reused TESTS degrades.

following experiment: 20 *U* instances send TEST requests (half fresh, half reused) at high rates, first, to a 32-node enforcer with the scheme and then, for comparison, to an otherwise identical enforcer without the scheme. Here, $r = 5$ and the nodes' caches are enabled. Also, each stamp is used no more than twice; TESTS thus generate multiple GETS, some of which are dropped by the enforcer without the scheme. Figure 4.10 graphs the positive responses as a function of the TEST rate. At high TEST rates, an enforcer with the scheme gives twice as many positive responses—that is, blocks more than twice as much spam—as an enforcer without the scheme.

4.6.7 Limitations

Although we have tested the enforcer under heavy load to verify that it does not degrade, we have not tested a flash crowd in which a *single* popular stamp *s* is TESTed at *all* (several thousand) of the enforcer nodes. However, as discussed in §4.4.6, we do not believe that this case will be problematic.

We have also not addressed heterogeneity. For *static* heterogeneity, i.e., nodes that have unequal resources (e.g., CPU, RAM), the bunker can adjust the load-balanced assignment of keys to values. *Dynamic* heterogeneity, i.e., when certain nodes are busy, will be handled by the enforcer's robustness to unresponsive nodes and by the application's insensitivity to latency.

4.7 QUOTA ALLOCATION

Recall that our top-level objective is to allocate human attention to senders in rough proportion to their numbers, meaning that no sender should be able to claim an outsized portion of aggregate human attention. To achieve this goal, we need to make validly stamped spam—i.e., the spam that reaches inboxes and claims human attention—a small fraction of all email. In this section, we consider how to do so. We begin with a basic analysis of what is required to limit spammers, then discuss the effect on legitimate senders, and finally address some policy questions briefly.

Limiting spammers. Stamps can only limit spammers if a stamp costs a scarce resource. For simplicity, we view stamps as costing money and do not discuss how to translate currencies like CPU [1], identity [13], or human attention [166] into money. We now ask what per-stamp monetary price would be required.

Assume that the spam industry has profit function $p(m)$, which it currently maximizes by sending $m = m^*$ emails. If DQE is deployed and induces a stamp cost of c per email, then setting $c \geq p(m^*)/m'$ will make it uneconomical for the industry to send more than m' emails, because for $m > m'$, the added costs $m \cdot p(m^*)/m'$ will exceed the entire possible profit $p(m^*)$. Thus, to reduce spam by a factor $f > 1$, i.e., to make $m' = m^*/f$, it suffices to set $c = \frac{p(m^*)}{m^*/f} = f \cdot p(m^*)/m^*$. That is, to reduce spam by a factor $f > 1$, a sufficient price per message is f times the profit-per-message.[11]

Although we will use this estimate of c below, the above analysis is very pessimistic. Roughly speaking, the analysis assumes that $p(m)$ attains the maximum, $p(m^*)$, for many $m < m^*$. This assumption implies an improbable shape for the function $p(m)$ and is likely false. For consider a variety of scams, each with a different profit-per-message when sent in the optimal amount. If, as we expect, most scams yield low profit, and few yield high profit, then setting a price c will prevent all scams with rate-of-return less than c. For example, if each scam sends the same amount, and if the number of scams returning more than a given amount q exponentially decays with q, then additive price increases in stamps result in multiplicative decreases in spam.

[11] I am grateful to James Grimmelmann for this paragraph. After observing that the previous version of this paragraph, in [168], was not fully explicit, he suggested the current wording, nearly verbatim (March, 2007).

We now give a rough estimate for c. First, consider spammers' profit per message. Goodman and Rounthwaite [65] survey media reports and find a wide range of purported values for spammers' revenues. The most conservative (i.e., the highest) per-message revenue that they survey is $300 for sending a million messages (.03 cents per message). Assuming (pessimistically) that this report applies to all spammers and that spammers have no costs, then reducing spam by a factor $f > 1$ requires a cost of $.03f$ cents per message. But what value of f is appropriate? As mentioned earlier, a rough estimate of current spam rates is 75% of all email [106, 107, 150]. To make spam 5% of the email that people receive, f needs to be approximately 50. (f needs to satisfy $\frac{75/f}{(75/f)+25} = 0.05$.) Thus, based on these pessimistic estimates, the per-email cost to spammers should be roughly 1.5 cents per message.

The analysis above holds even if DQE causes spammers to change tactics drastically (e.g., they might now create more appealing spams). The analysis assumes only that spammers are profit-maximizing and therefore incorporates all possible spammer strategies. (If creating more appealing spams actually resulted in more than $1/f$ as much spam, then this new strategy must be more profitable than their old one, contradicting our assumption that spammers are currently optimizing.)

Effect on legitimate senders. Although Laurie and Clayton [92] argue, in the context of computational puzzles, that *no* price exists that affects spammers while leaving legitimate users mostly unaffected, their analysis does not take into account "refunds" of computational work [1]. In our context, such "refunds" correspond to a social protocol in which receivers do not demand stamps from known senders. With such a social protocol, senders would need very few stamps (most people do not send large quantities of unsolicited email to strangers; doing so is the definition of spamming!)

If a legitimate sender directs one out of every 100 emails to a stranger and emails 100 recipients per day, then legitimate senders would have to pay on average 1.5 cents per day. This price is $5.50 per year per sender. Of course, we cannot prove that this rough estimate of cost would be negligible for legitimate senders, but perhaps the bundled allocation policy briefly mentioned below will be helpful.

Policy questions. One policy question is how to translate per-email prices into quotas, which are allocated in blocks. If the quotas have much "headroom" to account for days of heavy sending, then (1) quotas might be more

expensive than is necessary for legitimate senders (in effect, they will have to pay for more than they actually send); and (2) if adversaries compromise machines (see §4.8), they will be able to send more emails than the human owner of the machine would normally send, leading to extra emails. One possible answer is simply to plan for little headroom in users' quotas, say because quotas are actually allocated to organizations, not individuals, and organizations might send similar rates of email each day, even if their constituent senders are bursty. Another option is for the quota allocator to "bump up" a sender's quota temporarily by selling a quota that expires in the near future.

A more difficult policy question is: how can quota allocation give the poor fair sending rights without allowing spammers to send? We are not experts in this area and just mention one possibility. Perhaps a combination of explicit allocation in poor areas of the world, bundled quotas elsewhere (e.g., with an email account comes free stamps), and pricing for additional usage could impose the required price while making only heavy users pay.

4.8 SYNTHESIS: END-TO-END EFFECTIVENESS

To complete the argument that we previewed at the beginning of the chapter, we now consider DQE's end-to-end effectiveness, taking into account the combined effect of allocation, enforcement, and cheating. We aim to show that, under DQE, the spam that consumes human attention is, compared to all email that consumes human attention, a manageable fraction. To do so, we simply tally how much spammers can send.

First, as described in the previous section, the quota allocator should arrange for stamps to limit spammers to 5% of the total email volume. Second, spammers can steal stamps. As mentioned in §4.3.4, this effect is likely to be limited and is manageable via out-of-band communication between email providers and senders. However, even if we put aside these points and give the adversary tremendous power, stamp theft still has only a limited effect. Our reasoning is as follows. Assume pessimistically that spammers steal stamps from *every* bot. There are roughly 1 billion computers in the world [34], and a high estimate of the total number of bots is 20 million (see §2.2), or 2% of the world's computers. If we make an even more pessimistic assumption that spammers can steal all of the quota from 5% of the hosts on the Internet, they can still send only roughly 10% of the world's email. (We are assuming little headroom in quotas.)

The third and last contribution to the tally is cheating via stamp reuse. As we have argued, this cheating is limited (see §4.4.2 and §4.6.2). For example, if 15% of the nodes in the enforcer are bad, then each stamp can be used 1.3 times in expectation. And, as mentioned in §4.4.2 (page 92), the actual total use is almost certainly no more than 5% of the expected total, so we can bound the uses per stamp by $1.3 \cdot 1.05 = 1.37$. Thus, if spammers reuse all of the stamps to which they have access, then their 10% (5% legitimately allocated, 5% stolen) becomes $13.7/(13.7 + 90) = 13.2\%$ of all email. This fraction is still manageable.

4.9 ADOPTION & USAGE

In this section, we briefly discuss pragmatic concerns, including how DQE could gain adoption, how quota allocators could be established, and how DQE could be integrated with existing systems.

Adoption. We now speculate about paths to adoption. First, large email providers have an interest in reducing spam. A group of them could agree on a stamp format, allocate quotas to their users, and run the enforcer cooperatively. If each provider ran its *own, separate* enforcer, our design still applies: each enforcer must cope with a large universe of stamps.

Two other possibilities are organization-by-organization adoption and individual-by-individual adoption. In the former case, the incremental benefit is as follows. Currently, many organizations whitelist email from their organization (e.g., someone at MIT might accept all email with source address matching the pattern `*.mit.edu`). However, spammers take advantage of such whitelists by spoofing the source email address to match the recipient's domain. With DQE deployed in an organization, recipients would expect email from within their organization to be stamped, eliminating the need for these whitelists.[12]

In the latter case—individual-by-individual adoption—the incremental benefit is that stamping one's email and sending to another DQE-enabled user ensures that one's email will not be caught in a spam filter. In both of these cases, the deployment challenge is agreeing on a quota allocator and establishing an enforcer. The local changes (to email servers; email clients need not change) are less daunting.

[12] DQE does not uniquely bring this benefit, but our purpose here is only to show a benefit to incremental adoption.

Quota allocators. The DQE architecture relies on globally trusted allocators (§4.3.1), which raises some questions, including:

— Where do quota allocators come from? Who performs the function?

— How can we fulfill the requirement that quota allocators are globally trusted?

— How can we make sure that the quota allocators "do the right thing" (i.e., charge the right prices, do not cheat, do not become hacked)?

We do not have definitive answers but suggest possibilities. For each of the possibilities, we answer the questions above.

The first alternative is that a consortium of email providers establishes a non-profit allocator, with a charter that would legally bind the allocator to "do the right thing" (charging an appropriate quantity for stamps but not profiteering). The trust in this organization would derive from the fact of its legal charter and observers' belief that the organization was adhering to its charter.

A second alternative is that an organization like ICANN or Verisign operates the allocator (in ICANN's case, the allocator could be a non-profit). Although ICANN and Verisign are favorite targets of disapproval, the fact is that these organizations are implicitly trusted by almost all Internet users. As in the above case, the protection against the allocator cheating would be public scrutiny, mainly.

Another possibility is for each country to establish a quota allocator. Because some nations' allocators might "cheat" (by assigning huge quotas) recipients would need to apply reputation mechanisms to the set of allocators. Quota allocators would gradually lose trust if recipients received high volumes of spam with valid stamps from that allocator. Such a reputation scheme would reduce the general "detection" or "reputation" problem of "How can any given recipient trust any given sender?" to the problem of deciding which of a set of 200 allocators is trustworthy. This reduced problem seems tractable.

A final possibility is a competitive market in which *anyone* could establish an allocator. In such a market, allocators might have an incentive to cheat somewhat (e.g., charging "too much" for stamps). However, reputation systems are needed here anyway (as in the case above), and we hypothesize that these mechanisms could limit such abuse.

Usage. The amount of stamped spam will be tolerable, as we have argued. Thus, following the "no false positives" goal, stamped email should always be passed to the human user. For unstamped email: before DQE is widely deployed, this email should go through content filters (again risking false positives), and under widespread DQE deployment, this email can be considered spam. We expect that DQE will incorporate whitelists, where people agree not to cancel stamps from people they know (indeed, our rough analysis of the cost to good clients depends on such whitelists; see §4.7). With whitelists, senders should still stamp their emails to prevent spoofing, but these stamps should not "count" against the sender's quota. (This social protocol is similar to the refunds that are proposed by [1].) In §4.10.1, we further discuss how one may combine DQE with other defenses.

Mailing lists. For moderated lists, senders can spend a single stamp, and the list owner can then either sign the message or spend stamps for each receiver. Unmoderated, open mailing lists are problematic: spammers can multiply their effect while spending only one stamp. *Partially-moderated* lists might become more common under DQE. Here, messages from new contributors would be moderated (requiring only a glance to determine if the email is spam), and messages from known valid senders—based on past contributions and identified by the public key in the stamp—would be automatically sent to the list, again using either the list owner's public key or stamps for each recipient. In such lists, the moderation needed would be little (it would be proportional to the number of messages from new contributors), so more lists could convert to this form.

4.10 RELATED WORK

We first survey work on spam control to justify both our choice of quota-based spam control (or bankable postage) in general and the architecture of DQE in particular; we also say how DQE can be combined with other defenses. We then describe how DQE's architecture relates to micropayments (which share a similar purpose) and finally compare the enforcer to related distributed systems.

4.10.1 Spam Control

In this section, we cover only technical solutions, not legal ones. For surveys that include both kinds of solutions, see [68, 157].

We take as a starting point that labeling valid email as spam is unacceptable. To some extent, this stance is a matter of taste.

Filtering. The dominant defense to spam today is spam filters (e.g., [66, 144]), which analyze incoming email to classify it as spam or legitimate. A variant is collaborative filtering, in which the filtering logic has access to many users' email, allowing it to detect mass mailings. These tools certainly offer inboxes relief, but their decisions are error-prone and sometimes cause valid email to be labeled spam, as discussed in Chapter 1. Moreover, we believe that spammers have the motive and ability to fool filters reliably, thereby making filters ineffective in the long run—regardless of one's tolerance for errors. Given this weakness, many have proposed solutions that do not examine message content; we discuss some of them now.

Heuristics that do not examine message content. Blacklists (e.g., [158]) are collections of IP addresses that are suspected of originating spam. (The intent is that filters would consult these lists in deciding how to classify a message.) However, such lists are also prone to error. First, spammers can often acquire new IP addresses, as our threat model presumes (see Chapter 2 and §4.1). Second, blacklists routinely include innocent hosts. This "impugning" happens for various reasons, including blacklisting of an entire netblock, dynamic assignment of an IP address to a bot and subsequent reassignment to an innocent host, and a NAT situation in which an innocent host and a bot appear to have the same IP address.

Whitelisting, in contrast, explicitly seeks reliability. For example, the recently proposed (and brilliantly named) Re: [59] uses friend-of-friend relationships to let correspondents whitelist each other automatically, at which point all email that they send to each other will be delivered to the receiving human. However, whitelisting still allows *some* errors (for non-whitelisted senders).

Of course, whitelisting can be combined with other approaches. One example of a complementary approach is challenge/response. Here, the receiver's email server or client asks non-whitelisted senders to respond to a challenge, such as a proof-of-humanity test [166]; if the sender responds successfully, the sender's email will be delivered, and the sender will then be whitelisted. We believe that there are several problems with this approach. First, as mentioned in the previous chapter (§3.9.2), spammers might pay people to respond to proof-of-humanity challenges. Second, some non-human senders, (e.g., some software programs) are legitimate. Third, this

approach changes the "social dynamics": the sender now has to do work, even if the sender was the one "doing the favor".

A related tactic is for a receiver to make it hard for people to send him email, say by requiring potential senders to register on a Web page (e.g., [100]). Such an approach undoubtedly reduces the spam to the recipient (though it might not if everyone deployed it). However, it also changes the social dynamics of email drastically, and the recipient may not feel comfortable asking potential senders to exert more effort than is usual.

Another technique is exemplified by the Sender Policy Framework (SPF) [146]. With this approach, administrative domains publish in DNS a list of IP addresses that are allowed to send email from that domain. Receiving email servers can then query DNS to verify that the sending email server's IP address actually matches the envelope header on the email. Systems like Mail Avenger [102] use SPF (and other techniques) to detect email from bots that have installed "illegal" email servers on desktop machines. However, SPF provides only a heuristic and cannot stop spam on its own. One reason is that spammers can invent bogus envelope headers and can adopt IP addresses temporarily. Either or both of these things thwarts the purpose of SPF. A related check, also performed by Mail Avenger, is to verify that the sending email server can *receive* email, the purpose being to screen out illegal email servers that are running on bots behind firewalls and NATs. However, this check is again a heuristic: not all of spammers' hijacked machines are behind firewalls and NATs, and, moreover, spammers need not send from bots.

Most of the approaches in this category are helpful (and meet our requirement of not labeling legitimate email as spam). However, we seek a "backstop"—a solution that, if heuristics fail, limits volumes explicitly.

Explicit limits on volume. A strawman in this category is a world in which ESPs (Email Service Providers) become known as "good"; "good" means that they limit their users' outbound emails (e.g., using techniques like those in [65]). However, we believe that some kind of global accounting system—something like DQE's enforcer, for example—is required to guarantee that ESPs (of which there could be thousands) *actually* limit their users' outbound emails. Templeton [154] also focuses explicitly on volumes. He envisions an infrastructure formed by cooperating ISPs to handle worldwide email; the infrastructure throttles email from untrusted sources that send too much. Unlike DQE, this proposal assumes a trusted enforcement infrastructure. Moreover, it assumes that, given an email, the actual network

address of the source can be identified, yet in our threat model, such identifications are not reliable.

Given the above, we believe that limiting volumes explicitly requires attaching a cost to sending. Thus, we turn to the general category of *email postage*.

Postage

We first discuss the various proposals in this category and then compare pairwise postage to bankable postage; DQE is in the latter category.

Pairwise postage. The high-level idea of email postage is that senders should pay to send email, the hope being that no sender will be able to afford to send vast quantities of unsolicited email (see §4.7 for our basic economic arguments). There have been many variations of email postage and a lot of debate about how best to implement it and in what currency to set the price. Some have suggested that senders pay receivers in money, letting receivers set the price of their attention [50, 94, 131]. To implement such a scheme, one would presumably use micropayment systems, described below in §4.10.2. Zmail [91] is a variation of this idea: the parties, represented by their ESPs, settle once per day. This proposal avoids some infrastructure (namely a system to handle transactions online) but at the following cost: given a discrepancy between two parties' accounts, the bank cannot prove which of the two parties cheated. (In Zmail, ESPs are supposed to be certified as "compliant", but such certification does not guarantee good behavior.) The proposal contrasts with DQE in two ways: first, DQE is not in the business of certifying email participants as good or bad (anyone can obtain a quota), and second, DQE, as a system, is robust to cheating.

Others have suggested that senders pay receivers in CPU cycles [11, 46] or memory cycles [2, 45] (the latter being fairer because memory bandwidths are more uniform than CPU bandwidths): to send an email viewed as valid, a sender must exhibit the solution to an appropriate computational puzzle (as in proof-of-work, discussed in the last chapter; see §3.9.1). Still others have suggested that senders pay human attention (e.g., [145]); this suggestion is related to challenge/response schemes, mentioned above.

Bankable postage. As a variation on postage, Abadi et al. pioneered *bankable* postage [1], in which senders buy tickets from a third party (called a "Ticket Server", or TS) and attach these tickets to emails. Receivers check with the TS to verify that tickets are fresh, cancel tickets with the TS, and

optionally refund them. The proposal is agnostic to the currency in which tickets are sold.

Though DQE and TS export a similar high-level interface to clients, they have very different realizations. TS does not meet most of our goals (see §4.2). Specifically, it relies on a trusted, central server for the enforcement function so does not meet our requirements that the enforcement function scale to the volume of the world's email and that it be fault-tolerant, attack-resilient, and untrusted. Also, TS does not separate allocation and enforcement or preserve sender-receiver email privacy.

Another bankable postage scheme, SHRED [89], also has a trusted, central cancellation authority. And Goodmail [63]—now used by two major email providers [32]—resembles TS. (See also Bonded Sender [22], which is not a postage proposal but has the same goal as Goodmail.) Goodmail accredits bulk emailers, trying to ensure that they send only *solicited* email, and tags their email as "certified". The providers then bypass filters to deliver such email to their customers directly. However, Goodmail does not eliminate the problem of valid email labeled spam because only "reputable bulk emailers" get this favored treatment. Moreover, like TS, Goodmail combines allocation and enforcement, does not preserve privacy, and presumably does not offer a large-scale enforcement solution.

Bankable Postage vs. Pairwise Postage

Advantages of bankable postage. Bankable postage has four advantages compared to pairwise postage; we use citations to indicate which authors first made the observation:

1. Asynchrony [1]. In a pairwise proposal, senders have to "purchase" the right to send every time they want to send email. But this requirement is problematic, as we now show by considering two different currencies. If the currency is money, then a micropayment infrastructure that can process requests "online" is needed, at which point the system would resemble DQE; see §4.10.2 below. If the currency is CPU cycles, then the payment will disrupt the sender's workflow. To see why, let us consider what price in CPU cycles might restrict spam. The answer, roughly speaking, is that the price has to be high enough so that a bot, even one whose CPU works around the clock, cannot send significantly more spam than a legitimate client. Most legitimate clients send fewer than 200 emails per day; to limit bots even to that number requires an average CPU price of (1440 minutes/day)(1 day/200 emails) $\approx$ 7 CPU minutes/email. This price is unacceptably high to be incurred "online". Human attention does not have the

problems just mentioned; however, it has others, as listed above during our description of challenge/response.

2. Separation of allocation & enforcement [13]. Bankable postage creates the *possibility* of separating allocation and enforcement. That possibility, which DQE seizes for reasons discussed in §4.2.1, leads to the following benefit: DQE can support a range of currencies and policies (in the pairwise case, in contrast, the currency and policy are "hard-coded" into the system). Thus, one way to view DQE is as a general platform that can accommodate the debate about which currency or platform is correct.

3. Stockpiling [1]. Senders can get tickets from various sources, e.g., their ISPs, rather than paying for every ticket.

4. Refunds [1]. If payment is incurred online and if senders use a non-recoverable resource like CPU cycles, then refunds are impossible. If payment is incurred offline, then receivers can let friendly senders reuse their tickets. Such refunds give a higher effective price to spammers, whose receivers would not refund.

Disadvantages of bankable postage. Against the above advantages, bankable postage has the following disadvantage:

Less infrastructure required (possibly). Bankable postage, and DQE in particular, require infrastructure that does not yet exist: globally trusted quota allocators, the enforcer, and the bunker. We have argued throughout that the enforcer is practical, and we have indicated in §4.7 and §4.9 why we think that a quota allocator could come into being. However, there is no question that it would be preferable to avoid this extra infrastructure.

Unlike DQE, many of the schemes described above (e.g., whitelisting, SPF, connecting back to the email server) can be adopted either in grassroots pairwise fashion or unilaterally. And pairwise postage likewise does not require infrastructure—*if* the currency is CPU cycles or human attention. However, if the currency is money, then a third party is needed, namely a bank to implement micropayments. And in that case, a system like the enforcer is called for, as we show below in §4.10.2. This fact means that bankable postage may be a way to *implement* pairwise postage.

Combining DQE with Other Defenses

DQE may work better in conjunction with other defenses. Indeed, our rough economic analysis in §4.7 presumes that senders and receivers make heavy use of whitelists, allowing them to see a far lower per-email price than spammers do.

More generally, DQE works with any algorithm in the "whitelisting family", meaning any algorithm that, given an email, outputs either "yes, this email is valid" or "I don't know". Except for the transition period, DQE ought not work with an algorithm that can output "no" (e.g., filtering or blacklisting)—such algorithms may sometimes label valid email as spam, an error that we view as unacceptable. Thus, here is how DQE can combine with other defenses: imagine that the recipient has deployed n defenses in the whitelisting family, $D_1, \ldots, D_n$. Given an email, the recipient's algorithm is as follows. Run each of the D_i. If any outputs "yes", stop and accept the email. (We are assuming that the D_i do not output "yes" wrongly.) If all output "I don't know", then invoke DQE's checks. Accept the email if and only if the email is stamped and all of the stamp checks (valid signatures, under quota, fresh stamp, etc.) pass.

This description makes explicit that DQE may function best as a "backstop" when other defenses fail.

4.10.2 Micropayments

The architecture and protocols of DQE, as described in §4.3, resemble a micropayment or digital cash system. In some of those schemes, a bank (corresponding to our allocator) allocates a block of "cash" (corresponding to our quota) to a user, who then spends the cash in units of "digital coins" (corresponding to our stamps). There has been a lot of work in the area of micropayments and digital cash (see [26, 60, 132] and citations therein).

One of the main problems in these systems is preventing users from spending a given digital coin twice; this attack is known as *double-spending*. The existing solutions for preventing double-spending at large scale do so in an *offline* fashion—at the end of the day, vendors turn over their coins to the bank, which detects double-spending by looking at the set of coins presented to it. However, for many applications (e.g., [18]), once-per-day timescales are too coarse-grained. Indeed, in our context, double-spending corresponds to stamp reuse, which of course DQE must detect as it happens.

Large-scale online detection of double-spending is currently viewed by the digital cash literature as technically infeasible: it is too computationally expensive for the bank to be involved in millions of transactions per second [26, 60]. Meanwhile, such online detection is the whole point of the enforcer! And because the enforcer is untrusted and presumed to fault, the bank needn't spend much to guarantee perfect correctness or even operate the infrastructure itself.

Thus, we believe that DQE makes a contribution to the digital cash literature: we show how to build an inexpensive, untrusted system that detects double-spending at intra-day timescales and handles millions of requests per second. To incorporate the enforcer, the designers of digital cash systems could arrange that, when digital coins are spent, vendors HASH, TEST, and SET them. These checks would be a hint, to prevent gross double-spending. Then, at the end of the day, the bank and vendors could run the protocols that actually exchange digital coins for cash, thereby detecting double-spending precisely.

4.10.3 Related Distributed Systems

Because the enforcer stores key-value pairs, distributed hash tables (DHTs) [12] seemed a natural substrate, and our first design used one. However, we abandoned them because (1) most DHTs do not handle mutually untrusting nodes and (2) in most DHTs, nodes route requests for each other, which can decrease throughput if request handling is a bottleneck. Castro et al. [27] address (1) but use considerable mechanism to handle untrusting nodes that route requests for each other. Conversely, one-hop DHTs [71, 72] eschew routing, but nodes must trust each other to propagate membership information. In contrast, the enforcer relies on limited scale to avoid routing and on a trusted entity, the bunker (§4.4), to determine its membership.

Such static configuration is common; it is used by distributed systems that take the replicated state machine approach [135] to fault-tolerance (e.g., the Byzantine Fault Tolerant (BFT) literature [28], the recently proposed BAR model [4], and Rosebud [133]) as well as by Byzantine Quorum Systems (e.g., [96, 97]) and by cluster-based systems with strong semantics (e.g., [67]).

What makes the enforcer unusual compared to the work just mentioned is that, to tolerate faults (Byzantine or otherwise), the enforcer does not need mechanism *beyond* the bunker: enforcer nodes do not need to know which other nodes are currently up (in contrast to replicated state machine solutions), and neither enforcer nodes nor enforcer clients try to protect data or ensure its consistency (in contrast to the Byzantine quorum literature and cluster-based systems with strong semantics). The reason that the enforcer gets away with this simplicity is weak semantics. It stores only immutable data, and the entire application is robust to lost data.

4.11 CRITIQUE & REFLECTIONS

We first critique DQE and reflect on its ability to handle spam, then give examples of where else DQE may apply, and finally reflect on the enforcer as an independently interesting system that may be a useful building block in other contexts.

DQE and spam. We have argued in this chapter that DQE can meet our goal of controlling spam via a roughly fair allocation of human attention; see §4.8 for a summary of the argument. At a high level, the way DQE is supposed to work is that (1) the economic mechanism of quotas will arrive at a proper allocation, and (2) a technical mechanism—the enforcer—will ensure that this allocation mostly holds in practice.

The principal critique of DQE is that the first part of this supposition, the non-technical part, may be improbable. The top-level challenge is adoption: to be fully effective, DQE must be in use everywhere, and achieving ubiquitous adoption is a tall order. Second, it may be unreasonable to suppose the existence of a small number of globally trusted quota allocators (though we discussed in §4.9 how such allocators might arise). Finally, we may be underestimating the difficulty of a quota allocation policy that meets the goal of reducing spam while leaving legitimate senders mostly unaffected (though we discussed this topic in §4.7).

We are more confident, however, about the second part of the supposition, the technical part. Based on our work, we believe that an enforcer comprising several thousand dedicated, mutually untrusting hosts can handle stamp queries at the volume of the world's email. Such an infrastructure, together with the other technical mechanisms in DQE, meets the design goals in §4.2—and shows that large-scale quota enforcement is practical. Indeed, the cost of the enforcer in hardware, single millions of dollars, is low and could be shared among many organizations and email providers that have an interest in reducing spam.

DQE and other applications. As Abadi et al. [1] note, the general approach of issuing and canceling stamps can apply to any computational service. In particular, one could imagine using DQE to regulate consumption of the resources of OpenDHT [130], Coral [55], or S3 [6], all of which are distributed systems that are intended to have large clienteles.

The enforcer. We believe that, apart from quotas, the enforcer is useful by itself in other contexts. As discussed in §4.10.2, the various digital cash sys-

tems could use the enforcer as a hint to prevent intra-day double-spending. Indeed, one of the authors of a recent e-cash proposal agrees that the enforcer could likely fill that void (i.e., online prevention of cheating) in his context [18, 79].

The enforcer's simplicity—particularly the minimal trust assumptions—encourages our belief in its practicality, both in the spam context and in others. Nevertheless, the enforcer was not an "easy problem". Though its external structure is now spare, it is the end of a series of designs—and a few implementations—that we tried. Its internal structure is, we believe, of independent interest. The enforcer is fault-tolerant but stores only one copy (roughly) of each key-value pair, it introduces a novel data structure that implements a dictionary interface, it avoids livelock (a problem that we conjecture exists in many other distributed systems if they are overloaded), and it resists a range of attacks. More generally, we believe that the enforcer is a novel design point: a set of nodes that implement a simple storage abstraction but avoid neighbor maintenance, replica maintenance, and mutual trust. And, the "price of distrust" in this system—in terms of what extra mechanisms are required because of mutual distrust—is zero.

5

Comparisons & Connections

In abstract terms, speak-up and DQE are two very different ways of allocating scarce resources. Their side-by-side inclusion in this dissertation raises several questions:

— What other abstract approaches to resource allocation are there, and what are the key differences and similarities among the various approaches?

— Why have we applied speak-up to the problem of DDoS and DQE to the problem of spam, rather than vice-versa?

— What are the fundamental connections between speak-up and DQE?

This chapter answers these questions in turn. To answer the first one, and to develop an informal vocabulary for discussing the others, we now consider a taxonomy of abstract resource allocation methods.

5.1 TAXONOMY

In any question of resource allocation, there is some set of scarce resources, some set of requesters of those resources, and some entity—an allocator—that decides which requests to fulfill. The entity's decisions may be explicit (e.g., "requester A gets 5 units") or implicit (e.g., "the resources go to the one who asks first"), and the entity may be the direct owner of the resources or a delegate that controls them. The taxonomy that we give in this section concerns (1) how the entity makes its decisions and (2) when the requester may consume the allocated resource. We will certainly not break new ground in economics; rather, our purpose is to establish a simple framework for comparing many possible approaches, including DQE and speak-up.

Admission Method	*Time of Consumption*		
	Moment of Allocation	Appointed Future Moment	During Some Window
First come, first served (∗)	over-subscribed server	free movie tickets	free gift certificates
Explicit, constant pricing	highway tolls	most sports tickets	**DQE**, snail-mail stamps
Explicit, variable pricing	grocery	airfares	stock options
Auction	**speak-up**	auction of any tickets	auction of stock options
Fair allocations (∗)	fair queuing	graduation tickets	disk quotas
Historical profiling (∗)	restaurant seating	club membership	whitelisting
Blocking undesirables (∗)	CAPTCHAS	club membership	blacklisting

(∗) Clients must be reliably identifiable for the discipline to be effective.

FIGURE 5.1—Taxonomy of abstract methods for resource allocation, with examples for each point. The taxonomy has two axes: the admission discipline (left) and the time at which the resources are consumed, relative to when they are allocated (top).

Our taxonomy is depicted in Figure 5.1 and has two axes. The first covers the actual admission rule, and the second concerns whether the allocated resource is consumed by requesters at the moment of allocation, at a specific point in the future, or at many possible points in the future. We have no proof that this taxonomy is complete, but it seems to capture many approaches used by computer systems and processes in "everyday life". Our elaboration of this taxonomy, below, will be abstract, but for each of the categories, we will give examples, both technical and "real world".

5.1.1 Axis 1: Admission Discipline

First come, first served. The easiest allocation policy is simply to give the resource to the requester that "gets there first". Examples include an over-subscribed Web server that does not aim for a fair allocation and someone giving out free movie tickets to the first takers. When other disciplines result in over-subscription, they may wind up incorporating this one. For example, if the allocator intends to charge a price but sets the price too low, then demand will be too high, and the resource will be allocated to the first requesters that pay.

Explicit, constant pricing. The allocator charges requesters in some currency. Of course, the price decided by the allocator may result in under- or over-subscription of the resource. In the latter case, some other discipline is also in effect implicitly (such as first come, first served, as mentioned above). One example in this category is consumer goods, sold at retail; in these cases, stores charge fixed prices for items but may run out of the item or be stuck with surplus. Other examples include highway tolls, snail-mail postage, and a pairwise email postage system (§4.10.1) in which the price

to send email, in CPU cycles or money, is fixed. A final example is quota allocation in DQE under the policy, discussed in §4.7, in which senders pay a fixed price to the quota allocator for the right to send a certain number of emails.

Explicit, variable pricing. This discipline is similar to the previous one, except that in this case the allocator adjusts the price frequently. "Everyday life" examples include trading posts, airline fares, and bookies in Las Vegas adjusting the price of a sports bet based on the existing "betting market". Technical examples include pairwise postage proposals for email (see §4.10.1) in which recipients adjust the price of sending them email based on how busy they are [50, 94].

Auction. Auctions have many incarnations (English auctions, Dutch auctions, Vickrey auctions, etc.). They free the allocator from having to "guess" or "estimate" a price because the auction mechanism finds an appropriate price. Two technical examples are as follows. First, recall that in one incarnation of speak-up (§3.4.3), the thinner uses a type of auction to decide which request to admit. Second, one could imagine a pairwise postage proposal for email in which recipients auction off the right to interrupt them (that is, the sender whose email gets through to the recipient is the one who is willing to pay the most). "Everyday life" examples of auctions are many (e.g., Treasury auctions to price new issues of U.S. government debt, selling famous works of art, etc.).

Fair allocations. In this discipline, the allocator gives every requester a "fair share" of the resources, meaning that if there are n requesters, each one should be able to claim a fraction $1/n$ of the resources (under max-min fairness, heavy requesters can get more than this fraction, by getting the resources not claimed by light requesters). Technical examples include Fair Queuing [42] and its many variants. An "everyday life" example is the way that universities allocate tickets to their graduation ceremonies: every student can claim several tickets.

Historical profiling. As mentioned in §3.9.2, profiling as a DDoS defense gives service to "regulars". Another technical example is whitelisting in the case of spam (previous legitimate clients of a given human's attention are allowed to consume that person's attention in the future). One "everyday life" example is club membership: "last year's" members typically have priority in becoming members the following year. Another is restaurant seating when the host chooses to give priority to regulars.

Blocking undesirables. As discussed in Chapter 1, §3.9.2, and §4.10.1, allocations in this category block "unwelcome" requests outright (e.g., CAPTCHAS [166] block requests from non-humans and spam filters block email that appears to be spam). Non-technical examples are many: admission to nightclubs on a busy night, accepting papers to conferences, etc.

Observe that some of the disciplines require that clients or requests be identifiable; if they are not, then the allocation method cannot work. These disciplines are first come first served, fair allocations, historical profiling, and blocking undesirables.

5.1.2 Axis 2: Permissible Consumption Times

This axis is orthogonal to the one above: one can incorporate any of the disciplines above into any of the approaches below.

During some window. We will call these approaches *during-window* approaches. Here, the allocator gives requesters a "right", and that right is good for a period of time. For example, under DQE, senders may mint a certain number of stamps (i.e., may consume a certain amount of human attention) at any point during the day. Another technical example is disk quotas for UNIX users. "Everyday life" examples include cellular telephone minutes, subway tokens (having purchased a token, a rider can use the token for as long as the fare hasn't increased), snail-mail postage stamps, gift certificates, and stock options (because, having purchased the stock option, the holder may convert it to the underlying equity at any time before the option expires).

At an appointed future time. We will call these approaches *appointed-future* approaches. Here, the allocator gives requesters the ability to consume a resource at an appointed time. For example, tickets to sporting events give the holder the right to attend at a specific time on a specific day. As another "everyday life" example, dormitory assignments generally happen before the academic year begins, but the assignment is good for a particular time (in this case, the following academic year). Technical examples include situations in which using scarce computing resources (e.g., a high performance cluster) requires an advance reservation for a particular time.

At the moment of allocation. We will call these approaches *moment-of-allocation* approaches. This case is similar to the one above, except that the allocator is making its decisions for the current moment. "Everyday life"

examples include restaurant seating, paying a toll to enter a highway, and situations in which a customer exchanges cash for ownership of a good.

Technical examples are as follows. First, consider speak-up: when the server is overloaded, it simultaneously charges clients, makes its allocation decisions, and fulfills requests. Second, in capability systems (see §3.9.3), clients use the requested capabilities in the near future. Also, consider how providers like Gmail and Yahoo give out free email accounts: after having decided that the client is acceptable (usually because the client correctly answered a CAPTCHA), the provider immediately issues a new email account. Last, consider pairwise postage for email (see §4.10.1) when senders incur the cost in "real time", for example, by paying CPU cycles.

5.2 REFLECTIONS ON THE TAXONOMY

For each of the axes, we now discuss which scenarios call for which choices.

5.2.1 Axis 1 (Admission Discipline)

One of the main questions here is whether to charge requesters, that is whether to use one of the pricing methods (constant, variable, or auction) or whether to use one of the disciplines that requires identifying clients or their requests. Which decision is appropriate of course depends on context. Given the definition of the abstract problem in §1.1, and the specific instantiations that we have addressed in this dissertation, identifying clients is not reliable,[1] so we turn to pricing methods. In other contexts, one might prefer not to charge clients (e.g., if one is offering a "free" service and can filter out "bad" requests). On the other hand, in some contexts, charging clients is obviously desirable (e.g., if one is selling a product).

5.2.2 Axis 2 (Permissible Consumption Times)

The advantages of the during-window approaches are as follows. First, they might be a natural fit, given the social context. For example, it would be awkward to give a gift certificate that denied its recipient control over when to spend it. Second, these approaches often permit fungibility, because a given requester's right to claim future resources is often reflected in a "ticket" or "token" that the requester can transfer to another entity. Third, these ap-

[1] We discuss this point in §1.2, Chapter 2, and Chapter 3.

proaches, being essentially options, give the requester more flexibility compared to the other approaches.

The advantages of appointed-future approaches are as follows. First, there is no uncertainty about when the resource is consumed, so the allocator does not have to worry that too many "claims" or "rights to consume" are outstanding. Second, separating consumption from allocation might be far more convenient for both the allocator and requesters than decisions made "in the heat of the moment". For example, if the Red Sox were to sell tickets to their World Series games right before the game started, the result would be mayhem.

Moment-of-allocation approaches require less overhead so are appropriate when there is no need for the other two approaches. The reason that they require less overhead is that, with the other two approaches, the allocator must give a requester a token—a ticket, stamp, etc.—that is valid in the future (either at a particular moment or for a period of time). Implementing this token and ensuring that it is not counterfeit requires some mechanism. For example, consider nightclubs: they take guests' money and then let them inside, whereas using advance tickets would require a ticket office. Moreover, moment-of-allocation approaches may be required if the demand becomes manifest shortly before consumption occurs. For example, if human Web users want access to a Web site right *now*, not in the future, then the server has no choice but to allocate for the current moment.

5.2.3 Other Considerations

There are of course other considerations besides the ones above. For example, with respect to the first axis, social convention may dictate which pricing method—auction or constant pricing—is acceptable. Another example is with respect to the second axis: if an allocator cannot make decisions until all of the demand has become manifest (e.g., deciding which 1,000 people gain admission out of 10,000 people who have expressed interest), then moment-of-allocation does not work.

5.3 OUR CHOICES

In this section, we reflect on why we chose (1) speak-up to defend against denial-of-service (2) DQE to defend against spam, and (3) speak-up again to defend the enforcer against resource exhaustion attacks (see §4.4.5).

Why speak-up for DDoS defense? In the taxonomy in §5.1, speak-up is given by (auction, moment-of-allocation). We now explain why this design point is appropriate for our context. Consider the second axis in the taxonomy. In our context, the server needs to allocate itself instantaneously; nothing is gained by separating allocation and consumption. Now, consider the admission discipline (axis 1). For this function, we require one of the "pricing" methods because we presume that clients are not identifiable (see §2.3). We chose auctions (as opposed to variable or constant prices) because they are a simple way to match the aggregate demand to the server's capacity. Of course, the preceding does not explain why we chose speak-up out of all possible (auction, moment-of-allocation) defenses; for this reasoning, see §3.9.1.

Why DQE for spam defense? In the taxonomy in §5.1, DQE is given by (constant pricing, during-window). For the first axis, we require *some* pricing scheme because the other approaches do not meet our requirement from §1.2 of not examining the contents of messages. Of the pricing schemes, we choose fixed per-email prices because it permits a simple argument that the total volume of spam would decrease to a manageable level (see §4.7). However, other policies for axis 1 would also work; indeed, DQE works with any quota allocation policy.

For axis 2, our reason for choosing during-window is as follows. Consider the alternatives. First, appointed-future approaches do not apply because senders cannot predict exactly when they are going to send email. The second alternative is to have senders pay when they want to send email, corresponding to moment-of-allocation approaches. Yet, we already addressed the disadvantages of senders incurring costs in real time; see §4.10.1. As we argued in that section, senders buying stamps "in advance"—which is a during-window approach—gives more flexibility.

Another option that might appear to be a moment-of-allocation approach is senders paying receivers in digital cash. However, this option is actually a during-window approach because the digital cash is valid for a some (possibly long) period of time. Indeed, because of this extended validity, digital cash schemes, like DQE, need to prevent double-spending. The enforcer is one way to do so; see §4.10.2.

Why speak-up to protect DQE's enforcer? As discussed in §4.4.5, we suggest defending the enforcer against resource exhaustion attacks by using speak-up or a variant. Our reasons are as follows; we again argue in terms of

the taxonomy in §5.1. First, consider axis 1. The discussion in §4.4.5 calls for one of the pricing approaches because we presume that clients are not identifiable. Moreover, even if clients *were* identifiable, the distributed nature of the enforcer would make it difficult to apply a discipline that depends on tracking how many requests each client sends to the enforcer in aggregate.

For axis 2, we call for a moment-of-allocation approach. The reasons are as follows. First, appointed-future approaches do not apply because DQE clients do not know in advance the specific moments when they need to contact the enforcer. Second, consider during-window approaches. Here, some entity would have to allocate tokens—tickets or stamps or certificates—giving DQE clients the right to make TEST and SET requests. *Yet, what would prevent clients from reusing these tokens?* This problem is the same as preventing stamp reuse for email. Thus, one would need *another* enforcer, and what would protect *that* enforcer against resource exhaustion attacks? To put this reasoning another way, a during-window approach would require a mechanism to prevent double-spending, and that mechanism would tax precisely the resources that we are trying to defend. For this reason, we must defend the enforcer with an approach in which clients "pay" for service "directly" without intermediating tokens, tickets, stamps, certificates, etc.

But then why use DQE at all? Given our argument that speak-up (or a variant) is required to protect DQE's enforcer, one might wonder, "Why not cut out the middleman and use speak-up to defend against spam directly?" For example, email servers could require prospective senders to attach dummy bytes to the end of messages. The reason that we do not favor this approach is that it would be similar to receivers charging senders pairwise in CPU cycles, the disadvantages of which we cover in §4.10.1. (As a side point, our arguments do not rule out a scenario in which a quota allocator doles out certificates based on bandwidth payments from clients. In that scenario, bandwidth, or some other resource, would be used twice—once to procure the quota and once to protect the enforcer.)

5.4 CONNECTIONS

We now discuss the fundamental connection between DQE and speak-up. More generally, we illustrate a connection between during-window approaches and moment-of-allocation approaches.

Any-time approaches require a mechanism that mediates between the

act of allocation and the act of consumption. This mechanism can be tokens, tickets, stamps (in the case of DQE), certificates, signed attestations, etc. Because requesters have latitude in when they spend these tokens—that is, because the tokens are valid at many points in time—the system must prevent double-spending. As examples, retail stores collect gift certificates after the customer spends them, and DQE's enforcer controls stamp reuse. If the scale of the system is limited, then tracking clients' spending does not require a separate infrastructure. For example, a "real world" store can keep track of the gift certificates that it issues, and an operating system can keep track of its users' disk quotas.

However, if the scale of the system is very large, as in the case of the world's email load, then a distributed system for preventing token reuse is very likely required. Once such a system is called for, the resources of *that* system must be allocated properly. And for this latter function, a *moment-of-allocation* approach is required. Why? Because, as illustrated in the previous section, a during-window approach would reintroduce the problem: it would require another set of tokens, which would require an infrastructure to prevent double-spending, which would have to be protected, etc. (Appointed-future approaches do not apply because they would require that participants know in advance when they need to make requests of the distributed system.)

In summary, then, a during-window allocation method of sufficient scale needs an enforcement mechanism, and such an enforcement mechanism must protect its resources using a moment-of-allocation approach. This high-level relationship explains why DQE's enforcer depends on speak-up or something like it.

6

Critiques & Conclusion

A recapitulation of this dissertation's narrative is as follows. We defined an abstract problem, namely, good and bad clients making requests for some scarce resource, with the good and bad requests indistinguishable. We argued that the abstract problem is motivated by criminals perpetrating attacks that are hard to filter. We advanced a philosophy to guide any solution to the abstract problem, namely, that solutions should not try to differentiate among clients and should instead aim to allocate the scarce resources in proportion to clients' numbers. We presented solutions to two instances of the abstract problem and argued that they uphold the philosophy. Specifically, we presented speak-up as a defense against DDoS and DQE as a defense against spam. We justified our choices of speak-up and DQE by comparing them within the same framework. This framework also allowed us to articulate a fundamental connection between the two defenses or, more accurately, between their general approaches.

* * *

This narrative brings to mind two high-level critiques. (We critiqued speak-up and DQE individually in §3.9 and §4.11, respectively.)

The first critique is as follows: our philosophy says to avoid solutions that examine the contents of requests because such heuristics err, yet *any allocation discipline can err*. For example, under speak-up, a low-bandwidth legitimate client may get less of the server than it demands; this shortfall is a form of error. And under DQE, if the quota allocator sets the wrong price, unsavory clients can purchase too many stamps. Moreover, both systems permit some gaming (though, as we proved, only to a limited extent).

Our response to this first critique concerns the degree of the error. If bad requests look *exactly* like good ones, then heuristics are no better than

random guessing and thus introduce a vast amount of error. In that case, the only defenses that work are those based on over-provisioning or on charging clients (like speak-up and DQE). Of course, one might well ask whether bad and good requests actually are hard to differentiate, which brings us to the second critique.

This next critique is about the abstract problem itself and about our view of the present and the future. The threat that this dissertation defends against is one in which adversaries issue totally convincing, legitimate-looking requests. Yet, what if this view is too pessimistic? Indeed, as we discussed at the end of Chapter 3, today's application-level attacks are primitive. Perhaps tomorrow's adversaries will be no more advanced than today's. Or perhaps the economics of the Internet underworld will shift so that spam and denial-of-service no longer afflict the Internet.

Our response to this second critique is to ask: what if the pessimal future that motivated DQE and speak-up *does* become fact? What if, as we suspect, bad requests continue to evolve until they are indistinguishable from good requests? What if spam and denial-of-service increase in frequency and intensity? We need to be prepared for that world. In that world, the best that we can hope for is rough proportionality: if 10% of the clients are bad, we hope to limit them to 10% of the scarce resources.

This ethos is appealing, for it is egalitarian. A seeming weakness is that if 90% of the clients are bad, then they get 90% of the scarce resources. However, this fact is independent of our philosophy. Indeed, if the bad clients outnumber the good ones ten to one, and the two populations are indistinguishable, the only way to ensure that all good clients get service is heavy over-provisioning (so that the "slice" that the good clients can claim meets their demand) together with proportional allocation (so that the bad clients can't deny the good ones even this "slice").

6.1 LOOKING AHEAD

Such predictions about the future as we make above invite questions about how the dynamics between adversaries and "good guys" will evolve. For example, even if we grant ourselves that future defenses need to strive for rough proportionality, as argued just above, it does not follow that the *particular* required defenses will include DQE and speak-up: couldn't the "good guys" change the landscape so that some *other* proportional allocation defense suffices? We now discuss this question and others about the future.

Speak-up and DQE are motivated by a specific threat. If future responses from various "good guy" communities—academics, security professionals, law enforcement, etc.—mitigate that threat, are these two defenses still needed? We first consider how the threat could be mitigated. For our current purposes, the threat in §1.1 has three components: (1) adversaries issue legitimate-looking requests; (2) the Internet's notion of identity is fuzzy; and (3) adversaries use bots as a low-cost computing platform. It is of course impossible to predict future solutions, but we believe that (1) is unavoidable (a smart adversary will always be able to imitate a good client) whereas (2) and (3) could be substantially mitigated by the community.

With respect to (2), changes to the Internet architecture could eliminate address hijacking and the need for proxies and NATs (in fact, if all providers applied today's "best practices", address hijacking would be more difficult than it is right now). For (3), the bot population could be curtailed substantially by a combination of two thrusts. The first is "botnet hunting", which law enforcement and other security professionals do today but which could and would be amplified if the political will were in place. The second is to continue the trend of software architecture changes that make operating systems and applications harder to compromise.

But would such responses eliminate the need for DQE and speak-up? Our belief is that, regardless of these responses, DQE, or some other form of email postage, would still be required. The reason is that mitigating (2) and (3) would not fundamentally restrict spammers from sending bulk email. In this scenario, they might need to send email from their own computers and to spend more on computing resources, but those changes are not the kind of inherent restrictions that result from per-email costs.

Speak-up, in contrast, would likely *not* be needed to defend Web servers against application-level DDoS. The reason is that with (2) mitigated, the server would be able to map each request to some client identifier. Thus, when overloaded, the server could achieve a fair allocation explicitly, without needing to use bandwidth payments as a proxy for identity.

However, speak-up or some other resource-based defense (see §3.9.1) would still be needed in at least one context—defending a distributed resource like DQE's enforcer (as discussed in §4.4.5 and §5.3). If each enforcer node allocated itself "fairly" across all requesting clients but did not charge clients, then an attacker could claim a piece of every enforcer node, resulting in a globally unfair allocation. In contrast, making clients "pay" for service in some currency ensures that any client's share of the *total* enforcer is bounded by that client's share of the total client currency.

Assuming that some *proportional allocation defense is deployed and effective, how are the adversaries likely to respond?* If the author were in the adversary's position, he would attack speak-up and DQE in particular by trying to amass as many resources as possible (Chapters 3 and 4 show that these systems resist many other attacks). More generally, if a proportional allocation defense is deployed, then an attacker's power is given by the number of hosts that he controls, so the attacker's response will be to try to acquire more hosts. Thus, attackers may concentrate less on crafting legitimate-looking requests and even more on compromising machines.

And how will the "good guys" respond to those efforts? They will respond by trying to minimize the number of machines available to adversaries. As mentioned above, they can do so via "botnet hunting" and architectural changes to operating system and application software. Their efforts will not be perfect: even if they could eliminate compromised machines altogether, adversaries could still use their own computers to mount attacks.

Given all of this back-and-forth, how can we argue that DQE and speak-up will have a positive effect? We cannot predict the future. However, if the current economics of computer crime—the cost to compromise a machine, the profit from spamming, etc.—remain roughly the same, then we *can* in fact make predictions about adversaries *even if we cause them to change tactics*. For example, we showed in §4.7 that a per-email cost of f times the profit per-email would limit validly stamped spam to a fraction $1/f$ of today's spam volume. Our only assumption in that (highly pessimistic) analysis is that spammers are profit-maximizing. Under that assumption, DQE could "disrupt the market" for spam or push spammers to new strategies, but their total volume would still be limited to the $1/f$ fraction. (If spammers' new strategies actually allowed them to buy more stamps, then the new strategies yield more profit than the old ones, contradicting our assumption that spammers are currently following the optimal strategy.)

For speak-up, the case is less clear because we do not understand the economics of DoS as clearly as the economics of email. However, we can still consider how speak-up affects attackers' costs. Recall that speak-up forces attackers to acquire many more machines to conduct the same level of service-denial as they can under the status quo (as stated in §3.2). Thus, the question becomes: how hard is it for an adversary to compromise or control orders of magnitude more machines?

We believe that such compromises would in fact be costly for attackers,

if not downright crippling. Our reasons are twofold. First, *compromising machines is already a competitive activity; bot herders compete with each other based on how many machines they control*. Thus, any given adversary today *already* has an incentive to compromise as many machines as he can. Second, *compromising machines happens automatically* (e.g., two ways that worms spread are by scanning and as users open attachments that email themselves to the users' contact lists). Thus, we can assume that for any known exploit, all of the vulnerable machines are already compromised (roughly speaking). The implication is that compromising further machines requires identifying further vulnerabilities. Yet, doing so cannot be easy: if a vulnerability were easy to find, it would have been found already, given the competition mentioned above.

To sum up our predictions about speak-up: while we cannot predict attackers' next steps, we do know that speak-up has increased their costs—and that the increase is likely to be significant.

Of course, the economics of computer crime could change, causing adversaries to adopt new tactics—and this point is independent of whether DQE and speak-up are adopted as defenses. For this reason, our reflections above should be recognized as speculations.

6.2 LOOKING BACK

At the end of the day, we do not wish to sell short speak-up and DQE. Even if the pessimal scenarios that motivated these systems do not materialize, we believe that they are interesting for concrete technical reasons.

Speak-up introduces the idea that bandwidth could be a computational currency and presents several mechanisms for charging in this currency. These mechanisms are simple (both conceptually and in implementation), resist gaming, find the correct price automatically without requiring explicit server-client communication, and likely apply to other currencies. And, speak-up admits a practical implementation.

DQE illustrates that large-scale, distributed quota enforcement is practical, in part because its enforcer can handle the volume of the world's email with just several thousand machines. As we discussed at the end of Chapter 4, DQE can apply to other contexts, and the enforcer by itself is likely to be a useful building block when one needs a system to prevent double-spending of "tokens"—stamps, digital coins, etc. The enforcer's spe-

cific techniques (summarized in §1.4 and §4.11, and at the beginning of Chapter 4) are interesting. And, more generally, it occupies a novel design point for distributed systems. Specifically, its weak semantics permit it to shed many mechanisms—including neighbor maintenance, replica maintenance, and heavyweight cryptography—that are required of other systems. The result is a system that can scale to a workload of hundreds of billions of requests per day.

Appendix

A

Questions about Speak-up

A.1 THE THREAT

How often do application-level attacks occur?
We do not know how often they occur. They seem to be less common than other forms of DDoS but, according to anecdote, the trend is toward such attacks. See §3.10.1 for more detail.

Why would an adversary favor this type of attack?
Such attacks are harder to filter (because existing DDoS-prevention tools already have the ability to filter other kinds of attacks). Moreover, application-level attacks require less bandwidth (because the request rate needed to deplete an application-level resource is often far below what is needed to saturate a server's access link). This latter characteristic would attract adversaries who have access to small botnets or who need to conserve resources, and indeed there are reports that botnets are becoming smaller [17, 33, 35, 49, 78, 105, 125].

If such attacks are not yet common, why bother thinking about the defense?
We answer this question in §3.10.1 and Chapter 6. Briefly, there are a few reasons. First, the *vulnerability* still exists and thus could be exploited in the future. Second, we believe that the trend is toward such attacks. Last, we believe that it is important to be proactive, that is, to identify and defend against weaknesses before they are exploited.

How often are the attacking clients' requests actually indistinguishable from the legitimate clients' requests?
We do not know. According to anecdote, many application-level attacks are primitive and hence easily filtered. However, as argued just above, we believe that the trend is toward smarter attacks and smaller botnets. See §3.10.1.

What makes you think that bad clients send requests at higher rates than legitimate clients do?
If bad clients *weren't* sending at higher rates, then, as long as their numbers didn't dominate the number of good clients, the server could restore service to the good clients with modest over-provisioning. (If the number of bad clients is vastly larger than the number of good clients, and requests from the two populations are indistinguishable, then *no* defense works.)

Aren't there millions of bots? Aren't current DDoS attacks 10 Gbits/s? How can speak-up possibly defend against that?
See §3.2, §3.3, and §3.10.2. In short: first, only a few botnets are of this size and only a minuscule fraction of attacks are 10 Gbits/s (§3.10.2). Second, speak-up (or any resource-based defense) works best when the adversarial and good populations are roughly the same order of magnitude. As mentioned in the answer to the previous question, if the adversarial population vastly outnumbers the good population, and if requests from the two populations are indistinguishable, then *no* defense works.

Can speak-up defend tiny Web sites?
Yes and no. Speak-up helps no matter what. However, speak-up cannot leave the legitimate clientele unharmed by an attack unless the legitimate population and the attacking population are of the same order of magnitude (or unless the server is highly over-provisioned). See §3.2.

A.2 THE COSTS OF SPEAK-UP

Doesn't speak-up harm the network, a communal resource?
See §3.2, §3.5.1, and §3.11. Our brief answer is that speak-up introduces extra traffic only when a server is attacked, that speak-up's traffic is congestion-controlled, that the network core appears to be over-provisioned, and that one should regard speak-up as simply a heavy user of the network. However,

as with any application (e.g., BitTorrent), there might be collateral damage from the extra traffic introduced by speak-up.

But bad guys won't control congestion!
True. However, a bad client refusing to control congestion is carrying out a link attack, which speak-up does not defend against (a bad client can carry out such an attack today); see §3.3. And, if a bad client does flood, the bad client won't get much more of the server than it would if it obeyed congestion control (see §3.4.4). Thus, speak-up does its job regardless of whether bad clients control congestion.

What is the effect of speak-up when links are shared?
See §3.5.2.

Bandwidth is expensive, so why would a site want to use speak-up, given that speak-up requires the site to allocate a lot of inbound bandwidth?
The economics of every site are different. For some sites, speak-up is certainly less economical than over-provisioning the server's application-level resources to handle every good and bad request. However, we believe that there are other sites for which bandwidth is not terribly expensive; see condition C2 in §3.3 and §3.5.3.

Doesn't speak-up introduce opportunity costs for end-users?
Yes, but such costs are introduced by any network application and, indeed, by any resource-based defense. See §3.2 and §3.11.

A.3 THE GENERAL PHILOSOPHY OF SPEAK-UP

Won't speak-up cause adversaries to acquire more resources (i.e., compromised hosts)? For example, if speak-up has nullified a 100-node botnet, won't an adversary just build a 10,000-node botnet? Thus, won't speak-up inflame the bot problem?
It is true that speak-up (or any resource-based defense) creates additional incentive for adversaries to compromise machines. However, the cost to doing so is likely quite high: we believe that many of the computers worldwide that could be compromised *cheaply* already have been. Thus, speak-up increases the adversary's costs, thereby resulting in a "higher fence". We discuss this point in §3.2 and §6.1.

Doesn't speak-up give ISPs an incentive to encourage botnets as a way to increase the bandwidth demanded by good clients?
Such misalignment of incentives can happen in many commercial relationships (e.g., investment managers who needlessly generate commissions), but society relies on a combination of regulation, professional norms, and reputation to limit harmful conduct.

If the problem is bots, then shouldn't researchers address that mess instead of encouraging more traffic?
Our answer to this philosophical question is that cleaning up bots is crucial, but even if bots are curtailed by orders of magnitude, a server with scarce computational resources must still limit bots' influence. Speak-up is a way to do so.

A.4 ALTERNATE DEFENSES

Instead of charging clients bandwidth and allocating the server in a way that is roughly fair, why not allocate the server in a way that is explicitly fair by giving every client the same piece?
Doing so requires that the server be able to identify its clients. However, our threat model presumes that, given a request, the server cannot be sure which client originated it. The reasons for such uncertainty are address hijacking, proxies, and NAT. For more detail, see §1.2, §2.3, §3.3, and page 26.

Why does speak-up make clients* consume *bandwidth? Why doesn't speak-up simply determine how much bandwidth each of its clients has (say, by using a bandwidth-probing tool) and then allocate the server according to this determination?
Doing so would again require that the server be able to identify its clients. For example, if an adversary adopts several IP addresses, each of these addresses would appear to have the same bandwidth, thereby giving the adversary a bandwidth advantage.

Instead of charging clients bandwidth, why not charge them CPU cycles?
Such a defense would be a reasonable alternative. For a detailed comparison of bandwidth and CPU cycles as computational currencies, see §3.9.1.

Sites need to protect themselves against link attacks (and speak-up does not serve this purpose, as you state in §3.3). So why not regard the application as being connected to a virtual link, and use existing link defenses to protect the application?
Depending on the scenario, this approach may work. However, deploying link defenses often requires network modification or help from ISPs; it may be easier to deal with application-level attacks on one's own. Also, many defenses against link attacks work by detecting very high amounts of aggregate traffic, and an effective application-level attack needs far less bandwidth so may not trigger these defenses. Finally, adversaries may move away from link attacks, removing the need for link defenses.

How does speak-up compare to ... ?
Please see §3.9. In that section, we compare speak-up to many other defenses.

A.5 DETAILS OF THE MECHANISM

Can bandwidth be faked? For example, can a client compress its bandwidth payment to give the illusion that it is paying more bits? Or, could a client get a proxy to pay bits on its behalf?
The thinner counts the bits that arrive on behalf of a request, so a client connecting directly to the thinner cannot fake its bandwidth payment. And, proxies generally relay what clients send, so if a client compresses its payment en route to the proxy, then the proxy will submit a compressed request to the thinner.

By how much does speak-up increase bandwidth consumption?
Assume that the bad clients were flooding independent of speak-up. In the worst case, the good clients need to spend all of their bandwidth. In this case, the extra bandwidth consumption is $G/(gn)$, where G is the total bandwidth of the good clients expressed in bits/s, g is the good clients' legitimate demand expressed in requests/s, and n is the size of a request, in bits. However, if the good clients do not need to spend all of their bandwidth, the bandwidth consumption may be far smaller; see the discussion of "price" in §3.4.2.

Does speak-up allow bad clients to amplify their impact? For example, if bad clients attack a site, they can trigger speak-up, causing the* good *clients

to pay bandwidth. Don't adversaries therefore have a disproportionate ability to increase traffic under speak-up?
At a high level, the answer is no. The "price"—that is, the extra traffic introduced by speak-up—varies with the attack size. Roughly, if the bad clients do not spend much bandwidth, then they do not make the good clients spend much bandwidth. Like the previous question, this one is related to the discussion of "price" in §3.4.2.

Can the bad clients amplify their impact by cycling through a population of servers, driving each into overload but spending only a little bit of time at each server?
No. The purpose of speak-up is exactly to give clients service in proportion to *the bandwidth that they spend*. Thus, if the bad clients go from site to site, never spending much bandwidth at any site, then they will not get much service. Moreover, if they follow this pattern, then the price at each of the sites will be low, so the good clients will not spend much bandwidth either. If the bad clients *do* spend many bits and temporarily drive up the price at a site, that price will subside once those bad clients leave.

A.6 ATTACKS ON THE THINNER

What happens if the thinner gets a lot of clients or connections at once? Can it run out of file descriptors?
Yes, but the implementation protects itself against this possibility. The implementation recycles file descriptors that correspond to requests that have not been "active" for some period. Moreover, we have configured the thinner to allocate up to hundreds of thousands of file descriptors; see §3.7.

Is it possible to build a production-quality thinner, given that the box would have to sink bits at high rate and maintain a lot of state?
Yes, we believe it is possible. The state for each request is very little—a TCP control block, a counter for the number of bytes that have been paid on behalf of that request, and a small amount of other per-request data. Moreover, even our unoptimized implementation can sink bits at a high rate; see §3.8.1.

A.7 OTHER QUESTIONS

What happens if a server defended by speak-up experiences a flash crowd, that is, overload from legitimate clients?
Such a server will conduct a bandwidth auction, just as if it were under attack. Though this fact might seem distasteful, observe that if the server is overloaded it still has to decide which requests to drop. Deciding based on bandwidth is not necessarily worse than making random choices. We discuss this question in more detail when critiquing resource-based schemes (see §3.9.1).

Does your implementation of speak-up work if the Web client does not run JavaScript?
No. See §3.7.

Under speak-up, all clients are encouraged to send at high rates. So how is one supposed to tell the difference between the legitimate clients and the bots?
An attacked Web site cannot tell; indeed, part of the motivation for speak-up is that it can be difficult for Web sites to identify clients. However, the *ISP* of a given bot should be able to tell, based on the computer's traffic consumption: for a good client, most of the sites that it visits are not under attack, whereas a bot consumes much more of its access link. Thus, if it were economically attractive for ISPs to identify and eliminate bots, they could do so.

Appendix

B

Questions about DQE

B.1 GENERAL QUESTIONS ABOUT DQE

I don't get much spam anymore. Isn't the spam problem solved?
No. It is true that the big email providers prevent a lot of spam from showing up in their customers' inboxes, but these providers have huge resources to devote to the problem of identifying spam. Many people use regional ISPs that do not have the resources to be as effective. Moreover, the big email providers may be causing "false positives" (legitimate email in the spam folder). Finally, even the big email service providers (ESPs) dislike spam, suggesting that it is still a problem.

Do you think that it will ever be solved?
We do not believe that spam will stop being sent (we believe that it will always be easy for someone to harvest email addresses and send vast amounts of email to them). However, we believe that the *symptoms*—wasted human attention—can be mitigated. DQE is one way to do so.

Why do receivers* TEST *a stamp with the enforcer and then* SET *it? Why doesn't the enforcer expose a* TEST-AND-SET *primitive?
The enforcer is not trusted by the other participants in DQE (see §4.2.1). Thus, the TEST call must "test" the enforcer to verify that it really has seen the stamp (see §4.3.2). Such verification prevents the enforcer from labeling valid email as spam. A TEST-AND-SET primitive would defeat this purpose by presenting the "answer" along with the "test". The enforcer would then be able to lie to receivers.

How is DQE's enforcer different from a DHT?
The enforcer and DHTs each store key-value pairs, but the enforcer has a very different design. For example, the enforcer is allowed to "lose data". Also, the membership of the enforcer is fixed. See §4.10.3 for further comparison.

B.2 ATTACKS ON DQE

Can stamps be stolen? What is the effect of such theft?
Yes, our threat model presumes that stamps can be stolen from end-hosts (though such theft would probably be difficult in practice). Nevertheless, as we show in §4.3.4 and §4.8, such theft is unlikely to thwart DQE.

What if an email server is compromised?
In this case, the adversary can steal stamps from email that passes through the email server. However, if email servers were easy to compromise, we would see much more spam coming from legitimate email servers. Moreover, the human owners of the email server would be able to detect that such theft was occurring. See §4.3.4 for further discussion of this attack.

What prevents adversaries from counterfeiting stamps?
The cryptography in stamps combined with the trust that everyone places in the quota allocator. See §4.3.1. Briefly, an adversary cannot forge a valid certificate; doing so requires forging the quota allocator's signature. Because adversaries cannot forge certificates, they cannot forge stamps.

What happens if a stamp is reused in bulk, all at once? Will that overload the enforcer and/or allow a lot of spam through?
We address this attack in §4.4.6.

What if the portal is adversarial? Can't it endlessly give the wrong answer to requesting clients?
Yes, but clients choose portals that they trust, and if the portal is adversarial, the client will begin to suspect it; see §4.4.7. Also, recall that the portal can only lie in one direction: it can only declare that a given stamp is fresh; it cannot label a fresh stamp as reused.

B.3 ALLOCATION, DEPLOYMENT, & ADOPTION

How are you going to get DQE adopted? Doesn't it require everyone to start using it before it is useful?
Adoption is definitely a challenge. However, DQE can be useful even before everyone begins using it; see §4.9.

How does quota allocation work? What prevents the spammers from getting large quotas? If quotas cost money, how can quotas be assigned equitably? How do you know that the quotas would not harm legitimate senders?
We answer these questions when discussing quota allocation policy in §4.7.

Where does the quota allocator come from? What entity performs this function? How can you ensure that quota allocators are globally trusted? What prevents the allocators from cheating?
We answer these questions in §4.9.

Are quotas assigned per-person, per-machine, or per-organization?
From a technical perspective, any of these options works. However, we imagine that, for ease of deployment and management, organizations would acquire quotas for their users and then dole out pieces of the quotas to them.

So human users have to understand stamps?
No. Email servers can handle all of the DQE-related work.

How do mailing lists work under DQE?
See §4.9.

B.4 MICROPAYMENTS & DIGITAL POSTAGE

What is the relationship between DQE and digital postage schemes?
DQE is a *bankable postage* [1] scheme. In §4.10.1, we discuss how DQE relates to digital postage generally and bankable postage in particular.

But aren't digital postage schemes known not to work?
No. There are two sets of issues raised by digital postage schemes. The first regards pricing. The question here is how to impose a price on spammers without affecting legitimate clients. However, refunding stamps can address such concerns; see §4.7. The second set of issues is technical: there has not

been a concrete proposal for a system that could prevent double-spending at the volume of the world's email (tens or hundreds of billions of emails per day) in an online fashion. DQE fills this void.

How does DQE relate to micropayments?
See §4.10.2. DQE implements a lightweight variant of micropayments. And, as mentioned just above, the micropayment literature does not describe a system that could handle millions of requests per second and prevent double-spending of generic currency in an online fashion. (The existing micropayment systems can detect such double-spending either off-line or with vendor-specific currency.) Thus, existing micropayment systems could use DQE's enforcer for this purpose.

B.5 ALTERNATIVES

As an alternative to DQE, why not have ISPs rate-limit each of their users? This approach would impose a quota on users but would not require the allocator and the enforcer.
Such an approach would still require a mechanism to make sure that ISPs were not cheating. That mechanism would need to keep track of every email and would probably end up looking something like DQE's enforcer.

Instead of DQE, why not have receivers charge CPU cycles pairwise?
See the postage section of §4.10.1.

Shouldn't senders pay receivers directly instead of paying some third-party?
This policy is reasonable. How should one implement it? As mentioned in §4.10.2, current micropayment proposals do not scale to the volume of the world's email. As it happens, we think that DQE might be able to implement this policy (receivers would exchange their stamps for money at the end of the day or year). If we are right, then DQE is agnostic to whether senders pay receivers or a third party.

Why not defend against spam using legal means?
One can certainly do so. However, the current volume of spam is an indication that legal strategies are not enough on their own. Our interest is in finding a technical solution that works independent of the legal system.

What about controlling spam by ... ?

Please see §4.10. In that section, we survey many defenses and compare DQE to them.

Appendix

C

Address Hijacking

In this appendix, we describe how an adversary can temporarily adopt IP addresses that it does not own. The attack can take various forms.

Spurious BGP Advertisements

An attacker can issue spurious Border Gateway Protocol (BGP) advertisements, thereby declaring to the Internet's routing infrastructure that the attacker owns a block of IP addresses to which he is not entitled. The result is that the attacker may now be reachable at potentially millions of IP addresses. This attack works because many ISPs (a) accept BGP routes without validating their provenance and (b) propagate these routes to other ISPs [52]. Such spurious adoption of IP addresses has been observed before [23] and correlated with spam transmissions [126].

Stealing from the Local Subnet

A host can steal IP addresses from its subnet [51]; this attack is particularly useful if the thieving host is a bot in a sparsely populated subnet. The attack works as follows:

— The thieving host, H, cycles through the IP addresses in the subnet. For each IP address X, H broadcasts an ARP (Address Resolution Protocol) request for X.

— If H does not receive a reply to this ARP, H infers (assumes) that X is currently unused. At this point, H undertakes to steal X, using one of two methods:

* *H* sends packets to remote destinations, and the packets have source IP address *X*. Any reply packet will be addressed to *X*. When these reply packets arrive at the subnet, the router in front of the subnet will issue an ARP request for *X*. *H* simply responds to this request, declaring to the subnet that it is reachable at IP address *X*.
* Another option is for *H* to preempt the process above by broadcasting an ARP *reply* associating *X* with *H*'s hardware address. The local router now believes that *H* is reachable at IP address *X*. At this point, *H* can place source IP address *X* in its outbound packets and receive replies that are sent to *X*.

Of course, if the router in front of the subnet is properly configured, this attack is precluded, but not all routers are properly configured.

Stealing from Remote Networks

A host can adopt IP addresses from *another* network, assuming that the adversary controls hosts in two networks. The goal of this attack is to have a high-bandwidth host (say, one that is owned by the attacker) appear to have many IP addresses (say, that were harvested from bots on low-bandwidth dial-up connections). The attack works as follows:[1]

— The high-bandwidth, thieving host, *H*, sends packets with source IP address *X*; this time, *X* is the IP address of, for example, a bot with low bandwidth.

— Replies to this packet will go to the bot (these replies, in the case of TCP traffic, will usually be ACKs of small packet size).

— To complete the attack, the bot sends the ACKs to the high-bandwidth host, at which point it has become reachable at IP address *X*.

In fact, *H* could combine this attack with the previous one, as follows. The bot could steal *X* from its subnet, using the previous attack. Then, *H* and the bot would execute the attack just described.

[1] Nick Feamster told me about this attack and is my source for it.

Appendix

D

Bounding Total Stamp Reuse

Our aim is to show that, with what is essentially certainty, the actual total stamp use in DQE is close to the expected total stamp use, regardless of which subset of *np* nodes fails. To establish this result, we now prove Theorem 4.2 from §4.4.2:

Theorem 4.2. *Let K be the number of stamps that are active in a given day. If $K > (6n^2 + 300n)/\epsilon^2$, then, with probability at least $1 - e^{-100}$, there is no subset of size np whose failure leads to more than $(1 + \epsilon)$ times the expected total use across all stamps.*

Proof: To establish the theorem, we will first use a Chernoff bound to show that, for a given set of *np* nodes that fail, the probability is very small that the actual use across all stamps is more than $(1 + \epsilon)$ times the expected use across all stamps. We will then use a union bound to show that, out of all $\binom{n}{np}$ subsets of size *np*, the probability is still small that *any* subset's failure would result in significant deviation.

For each stamp $i \in \{1, 2, \ldots, K\}$, let X_i be a random variable equal to the number of uses of the stamp. Each X_i takes a value in $[1, n]$ (recall that the worst case is a stamp being reused at each portal). Also, from Theorem 4.1 in §4.4.2, each X_i has mean $\mu < 1/(1-p)^2 + p$, assuming $r = 1 + \log_{1/p} n$.

Now, fix a subset of *np* nodes that have failed. Once this subset is chosen, each of the X_i is independent. The reason is that X_i depends only on the assigned nodes for stamp i (see §4.4.2) and that stamps choose their assigned nodes independently (see §4.4.1). The total use across all K stamps is

$\sum_{i=1}^{K} X_i$ (which has mean $K\mu$), and this sum is what we are trying to bound.

For each X_i, define a new random variable $Y_i = X_i/n$. Each of the Y_i is independent and take values in $[1/n\,, 1]$, so we can apply a Chernoff bound to $\sum_{i=1}^{K} Y_i$ to show that this sum does not deviate from its mean, $K\mu/n$, by more than $(1+\epsilon)$:

$$\Pr\left(\sum_{i=1}^{K} X_i > K\mu(1+\epsilon)\right) = \Pr\left(\sum_{i=1}^{K} Y_i > \frac{K\mu}{n}(1+\epsilon)\right) \tag{D.1}$$

$$< \exp\left(-\epsilon^2\frac{K\mu}{3n}\right). \tag{D.2}$$

The justification for applying this inequality is, first, a Chernoff-Hoeffding bound on Bernoulli random variables that is stated in [8]. Second, we can apply such bounds to random variables with arbitrary distributions over $[0, 1]$, not just to Bernoulli random variables (see [110, Problem 4.7]).

Now, we consider the probability that *any* subset of size np deviates as above. Let T be the event "there exist one or more subsets of size np whose failure would produce more than $(1+\epsilon)$ times the total expected stamp use". T is the union of $\binom{n}{np}$ different events, one for each subset; each event has probability bounded as above. Applying the union bound,[1] we get:

$$\begin{aligned}
\Pr(T) &< \binom{n}{np}\exp\left(-\epsilon^2\frac{K\mu}{3n}\right) \\
&< \left(\frac{ne}{np}\right)^{np}\exp\left(-\epsilon^2\frac{K\mu}{3n}\right) \quad \text{(from Stirling's approximation)} \\
&= \exp\left(np + np\ln(1/p) - \epsilon^2\frac{K\mu}{3n}\right) \\
&< \exp\left(np + np(1/p) - \epsilon^2\frac{K\mu}{3n}\right) \quad \text{(because } \ln x < x \text{ for all } x\text{)} \\
&< \exp\left(2n - \epsilon^2\frac{K}{3n}\right) \quad \text{(because } p < 1 \text{ and } \mu > 1\text{)}.
\end{aligned}$$

If we take $K > (6n^2 + 300n)/\epsilon^2$, then we get:

$$\Pr(T) < \exp(-100)\,,$$

so the probability that no subset deviates is greater than $1 - e^{-100}$, as claimed. □

[1] The union bound applies the fact that the probability of the union of a group of events is no greater than the sum of the probabilities of the individual events. In this case, each of the events is of the form, "the failure of subset 243 causes more than ϵ deviation from the expected stamp use".

Appendix

E

Calculations for Enforcer Experiments

This appendix gives the details of two calculations that were mentioned in the evaluation of the enforcer. In §E.1, we analyze the "crashed" experiment of §4.6.2 (see page 107). In §E.2, we calculate the average number of RPCs induced by a TEST for the 32-node experiments of §4.6.4 (see page 112).

E.1 EXPECTATION IN "CRASHED" EXPERIMENT

In this section, we derive an exact expression for expected stamp use in the "crashed" experiment in §4.6.2. (The expression is stated in footnote 8 of §4.6.2, on page 107.) Recall from that section that n is the number of nodes in the system, p is the probability a machine is "bad" (i.e., does not respond to queries), $m = n(1-p)$ is the number of "up" or "good" machines, stamps are queried 32 times, and r, the replication factor, is 3.

Claim E.1 *The expected uses per stamp in the "crashed" experiment is:*

$$(1-p)^3(1) + 3p^2(1-p)\alpha + 3p(1-p)^2\beta + p^3 m\left(1 - \left(\frac{m-1}{m}\right)^{32}\right),$$

$$\textit{where} \quad \alpha = \sum_{i=1}^{m} i\left(\frac{2}{3}\right)^{i-1}\frac{1}{m}\left(1 + \frac{m-i}{3}\right) \quad \textit{and}$$

$$\beta = \sum_{i=1}^{m-1} i\left(\frac{1}{3}\right)^{i-1}\frac{m-i}{m(m-1)}\left(2 + \frac{2}{3}(m-i-1)\right).$$

Proof: We consider 4 cases: none of a stamp's 3 assigned nodes is good; 1 is good; 2 are good; and all 3 are good.

Let $U(s)$ be the number of times a stamp s is used. We calculate the expected value of $U(s)$ in each of the four cases. The first case is trivial: if all of the assigned nodes for s are good (which occurs with probability $(1-p)^3$), the stamp will be used exactly once.

Next, to determine $\mathbb{E}[U]$ for stamp with no good assigned nodes (probability p^3), we recall the facts of the experiment: stamps are queried 32 times at random portals, and once a stamp has been SET at a portal, no more reuses of the stamp will occur *at that portal*. Thus, the expected number of times that s will be used, if *none* of its assigned nodes is good, is the expected number of distinct bins (out of m) that 32 random balls will cover. Since the probability that a bin isn't covered is $\left(\frac{m-1}{m}\right)^{32}$, the expected value of $U(s)$ in this case is:

$$m\left(1-\left(\frac{m-1}{m}\right)^{32}\right).$$

We now compute the expected number of stamp uses for stamps with one or two good assigned nodes. In either case:

$$\mathbb{E}[U] = 1\cdot \Pr(\text{exactly 1 use}) + 2\cdot \Pr(\text{exactly 2 uses}) + \cdots$$

For stamps with one good assigned node (probability $(1-p)p^2$) there are two ways for the stamp to be used exactly once: either, with probability $\frac{1}{m}$, the stamp is TEST and then SET at the one good assigned node, or, with probability $\left(\frac{m-1}{m}\right)\frac{1}{3}$, the PUT generated by the SET is sent to the good assigned node. (The latter probability is the product of the probabilities that the TEST and SET are sent to a node *other than* the good assigned node and that the resulting PUT is sent to the good assigned node.) Thus,

$$\Pr(\text{exactly 1 use}) = \frac{1}{m} + \left(\frac{m-1}{m}\right)\frac{1}{3}.$$

If the stamp is used exactly twice, then the stamp was not stored at its good assigned node on first use; this occurs with probability $\left(\frac{m-1}{m}\right)\frac{2}{3}$. To calculate the probability that the second use is the last use, we apply the same logic as in the *exactly 1 use* case. Either, with probability $\frac{1}{m-1}$, the stamp is TEST and SET at the good assigned node ($m-1$ because there has already been one use, so one of the m nodes already stores the stamp, and thus a TEST at that node would not have resulted in this second use), or, with probability

$\left(\frac{m-2}{m-1}\right)\frac{1}{3}$, the PUT generated by the SET is sent to the good assigned node. Thus,

$$\Pr(\text{exactly 2 uses}) = \left(\frac{m-1}{m}\right)\frac{2}{3}\left[\frac{1}{m-1} + \left(\frac{m-2}{m-1}\right)\frac{1}{3}\right].$$

By the same logic, a third use only happens if the first and second uses do not store the stamp on the good node, and the third use is the last use if it results in the stamp being stored on its good assigned node:

$$\Pr(\text{exactly 3 uses}) = \left(\frac{m-1}{m}\right)\frac{2}{3}\left(\frac{m-2}{m-1}\right)\frac{2}{3}\left[\frac{1}{m-2} + \left(\frac{m-3}{m-2}\right)\frac{1}{3}\right].$$

A pattern emerges; cancellation of terms yields an expression for the general case:

$$\Pr(\text{exactly } i \text{ uses}) = \left(\frac{2}{3}\right)^{i-1}\frac{1}{m}\left(1 + \frac{m-i}{3}\right).$$

Thus, we have an expression for the expected number of uses for stamps with one good node:

$$\begin{aligned} E_1 &\stackrel{\text{def}}{=} \mathbb{E}[U \mid \text{one assigned node is good}] \\ &= \sum_{i=1}^{m} i\left(\frac{2}{3}\right)^{i-1}\frac{1}{m}\left(1 + \frac{m-i}{3}\right). \end{aligned} \tag{E.1}$$

A similar argument applies to stamps with two good nodes (probability $(1-p)^2p$), except we begin with

$$\Pr(\text{exactly 1 use}) = \frac{2}{m} + \left(\frac{m-2}{m}\right)\frac{2}{3}.$$

The $2/m$ term replaces $1/m$ because a TEST and SET to either of the (now two) good assigned nodes will result in exactly one use, and $2/3$ replaces $1/3$ because the SET's PUT now has a $2/3$ chance of reaching a good assigned node.

To get Pr (exactly 2 uses), we follow similar logic as before. The first use is not the last with probability $\left(\frac{m-2}{m}\right)\frac{1}{3}$, because the stamp is SET to a non-assigned node with probability $(m-2)/m$ and PUT to a bad node with probability $1/3$. Then, the second use is the last with probability $\frac{2}{m-1} + \left(\frac{m-3}{m-1}\right)\frac{2}{3}$, and

$$\Pr(\text{exactly 2 uses}) = \left(\frac{m-2}{m}\right)\frac{1}{3}\left[\frac{2}{m-1} + \left(\frac{m-3}{m-1}\right)\frac{2}{3}\right].$$

Continuing,

$$\Pr(\text{exactly 3 uses}) = \left(\frac{m-2}{m}\right)\frac{1}{3}\left(\frac{m-3}{m-1}\right)\frac{1}{3}\left[\frac{2}{m-2} + \left(\frac{m-4}{m-2}\right)\frac{2}{3}\right].$$

A pattern again emerges, and cancellation gives us

$$\Pr(\text{exactly } i \text{ uses}) = \left(\frac{1}{3}\right)^{i-1}\frac{m-i}{m(m-i)}\left(2 + \frac{2}{3}(m-i-1)\right).$$

Thus, we have an expression for the expected number of uses for a stamp that has exactly two good assigned nodes:

$$\begin{aligned} E_2 &\stackrel{\text{def}}{=} \mathbb{E}[U \mid \text{two assigned nodes are good}] \\ &= \sum_{i=1}^{m-1} i\left(\frac{1}{3}\right)^{i-1}\frac{m-i}{m(m-1)}\left(2 + \frac{2}{3}(m-i-1)\right) \end{aligned} \tag{E.2}$$

Note that for equations E.1 and E.2, the summation begins with the first use ($i = 1$) and ends with the stamp being on as many nodes as possible ($i = m$ or $i = m - 1$).

Letting $\alpha = E_1$ (from equation E.1) and $\beta = E_2$ (from equation E.2), we establish the claim. □

E.2 AVERAGE NUMBER OF RPCS PER TEST

In §4.6.4, on page 112, we rely on a calculation of how many RPCs are induced per TEST in the 32-node enforcer experiments. We now give the details of that calculation.

Claim E.2 *In those experiments, the average number of RPCs per* TEST *is 9.95.*

Proof: Recall from §4.6.4 that the 32-node enforcer is configured with replication factor $r = 5$. On receiving a fresh TEST, the portal must contact all 5 assigned nodes for the stamp. With probability 5/32, the portal is an assigned node for the stamp, and one of the GETs will be local. Thus, we expect a fresh TEST to generate $\frac{5}{32}\cdot 4 + \frac{27}{32}\cdot 5 = 4.84$ GET requests and GET responses. (Note that a request and a response both cause the CPU to do roughly the same amount of work, and thus an RPC response counts as an RPC in our calculations.) A fresh TEST will also be followed by a SET that will in turn cause both a PUT and a PUT response with probability $31/32 = 0.97$. (With

RPC Type	# RPCs from fresh TEST	# RPCs from reused TEST	Average
TEST	1.0	1.0	1.0
GET	4.84	2.64	3.74
GET resp.	4.84	2.64	3.74
SET	1.0	0	0.5
PUT	0.97	0	0.485
PUT resp.	0.97	0	0.485
Total RPCs/TEST			**9.95**

TABLE E.1—Number of RPCs of each type generated by fresh and reused TESTs. To calculate the average number of RPCs of each type (last column), we assume that half of the TESTs are fresh and half are reused.

probability 1/32, the portal is one of the assigned nodes and chooses itself as the node to PUT to, generating no remote PUT.)

A reused TEST generates no subsequent SET, PUT request, or PUT response. In addition, for reused TESTs, the number of induced GETs is less than in the fresh TEST case: as soon as a portal receives a "found" response, it will not issue any more GETs. The exact expectation of the number of GETs caused by a reused TEST, 2.64, is established by Claim E.3, below.

The types and quantities of RPCs generated are summarized in Table E.1; the average number of RPCs generated per TEST assumes that 50% of TESTs are fresh and 50% are reused, as in the experiment from §4.6.4. Thus, the expected number of RPCs generated by a single TEST is:

$$\begin{aligned} 1.0 &+ \frac{1}{2}\left[\left(\frac{5}{32}\cdot 4 + \frac{27}{32}\cdot 5\right) + 2.64\right] \\ &+ \frac{1}{2}\left[\left(\frac{5}{32}\cdot 4 + \frac{27}{32}\cdot 5\right) + 2.64\right] \\ &+ \frac{1}{2}\left[1 + \frac{31}{32} + \frac{31}{32}\right] \\ &= 9.95. \end{aligned}$$

□

Claim E.3 *A reused* TEST *generates 2.64* GETs *in expectation.*

Proof: The number of GETs generated by a TEST for a reused stamp depends on the circumstances of the stamp's original SET: did the SET occur at an assigned node, and if so, did it induce a remote PUT? Note that, for any

Event	$\Pr(A_i)$	stamp originally SET at …
A_1	27/32	…a non-assigned node
A_2	1/32	…an assigned node, no further PUTS
A_3	4/32	…an assigned node, 1 additional PUT

TABLE E.2—Possible SET circumstances.

Event	stamp queried (TESTed) at …
B_1	…a node storing the stamp
B_2	…an assigned node not storing the stamp
B_3	…a non-assigned node not storing the stamp

TABLE E.3—Possible reused TEST circumstances.

stamp, 27 of the 32 enforcer nodes will not be assigned nodes. Thus, with probability $\frac{27}{32}$, a SET will be to a non-assigned node, and the stamp will be stored at both an assigned node and a non-assigned node (event A_1). If the SET occurs at an assigned node (with probability $\frac{5}{32}$), then $\frac{1}{5}$ of the time the node will choose itself as the recipient of the PUT (event A_2, with overall probability $\frac{1}{5} \cdot \frac{5}{32} = \frac{1}{32}$), and the stamp will only be stored at that single, assigned node; $\frac{4}{5}$ of the time, the node will choose another assigned node (event A_3, with overall probability $\frac{4}{5} \cdot \frac{5}{32} = \frac{4}{32}$), and the stamp will be stored at two assigned nodes. We summarize the three possible circumstances in Table E.2. Note that the events A_i partition their sample space.

The number of GETs caused by a TEST for a reused stamp also depends on the circumstances of the TEST: is the queried node storing the stamp, and if not, is the node one of the stamp's assigned nodes? There are again three possible circumstances: the TEST is sent to some node storing the stamp (event B_1); the TEST is sent to an *assigned* node not storing the stamp (event B_2); the TEST is sent to a *non-assigned* node not storing the stamp (event B_3). These events are summarized in Table E.3; they partition their sample space as well.

Now, let $C(A_i, B_j)$ count the number of GET RPCs that occur when events A_i and B_j are true. Values of $C(A_i, B_j)$ are easy to determine. First consider event B_1: the TEST is sent to a node already storing the stamp. In this case, there will be no remote GETs regardless of the original SET's results.

Next, consider event B_2: the TEST is sent to an assigned node not storing the stamp; now, events A_1 and A_2 both cause a single assigned node to store the stamp, and thus, in either case, we expect the portal to send 2 (of $r - 1 =$

$C(A_i, B_j)$	A_1	A_2	A_3
B_1	0	0	0
B_2	2.5	2.5	(1+2/3)
B_3	3	3	2

TABLE E.4—Values of $C(A_i, B_j)$, the expected number of RPCs generated by a TEST when A_i and B_j are true.

$\Pr(B_j \mid A_i)$	A_1	A_2	A_3
B_1	2/32	1/32	2/32
B_2	4/32	4/32	3/32
B_3	26/32	27/32	27/32

TABLE E.5—Conditional probabilities $\Pr(B_j \mid A_i)$.

4 possible) GETs. However, event A_3 causes the stamp to be stored on two assigned nodes, and in that case, we expect the portal to send $\left(\frac{1}{2}\right) \cdot 1 + \left(1 - \frac{1}{2}\right)\left(\frac{2}{3}\right) \cdot 2 + \left(1 - \frac{1}{2}\right)\left(1 - \frac{2}{3}\right)(1) \cdot 3 = 1 + \frac{2}{3}$ GETs.

Finally, consider event B_3: the TEST is set to a non-assigned node not storing the stamp. If the stamp is stored on a single assigned node (events A_1, A_2), we expect the portal to send 3 (of 5 possible) GETs; if the stamp is stored on two assigned nodes (A_3), we expect the portal to send $\left(\frac{2}{5}\right) \cdot 1 + \left(1 - \frac{2}{5}\right)\left(\frac{2}{4}\right) \cdot 2 + \left(1 - \frac{2}{5}\right)\left(1 - \frac{2}{4}\right)\left(\frac{2}{3}\right) \cdot 3 + \left(1 - \frac{2}{5}\right)\left(1 - \frac{2}{4}\right)\left(1 - \frac{2}{3}\right)(1) \cdot 4 =$ 2 GETs. We summarize the values of $C(A_i, B_j)$ in Table E.4.

Now we can construct an expression for the expected number of RPCs generated by a reused TEST, which we call C:

$$C = \sum_{j=1}^{3} \sum_{i=1}^{3} C(A_i, B_j) \cdot \Pr(A_i \wedge B_j). \tag{E.3}$$

To calculate this expression, we use $\Pr(A_i \wedge B_j) = \Pr(B_j \mid A_i) \cdot \Pr(A_i)$. We already calculated the value of each $\Pr(A_i)$ at the beginning of this proof, so we only need to calculate each $\Pr(B_j \mid A_i)$. We begin by considering the stamps originally SET at a non-assigned node (event A_1), which are now stored at one assigned node and one non-assigned node. Given event A_1, there are 2 nodes storing the stamp, 4 assigned nodes not storing the stamp, and 26 non-assigned nodes not storing the stamp. The probabilities of sending a TEST to nodes in these three classes—which correspond to events B_1, B_2, and B_3, respectively—are simply 2/32, 4/32, and 26/32. The same method can be used to find the conditional probabilities given A_2 and A_3; we present these values in Table E.5.

Combining the values of $C(A_i, B_j)$ with the joint probabilities, we compute, from equation (E.3), $C = 2.64$. □

Appendix

F

Revisiting the Enforcer's Design

(This appendix is an addendum to the submitted version of the dissertation.)

In §4.4.3, we described nodes' key-value storage: an index in RAM, with the actual keys and values on the disk. As mentioned in that section, we chose this design over the default approach of storing keys and values in RAM because we wanted to conserve RAM.

In this appendix, we revisit that choice.

First, we perform a detailed comparison of our design to the default (§F.1). Our main point of comparison will be hardware costs. Second, we briefly mention an alternate design for the enforcer: storing all keys and values in flash memory (§F.2).

F.1 CURRENT DESIGN COMPARED TO DEFAULT

The default of storing keys and values in RAM wastes RAM. For this reason, our hypothetical alternative actually uses a modification of this default. We assume that nodes maintain only the *values*, and not the *keys*, in a "map" in RAM. In this hypothetical, a request of PUT(k, v) causes a node to store v at a location in the map given by k. On a request of GET(k), a node performs a lookup in the map for the key k. If the lookup returns multiple candidate values, the node applies HASH to each candidate to determine which matches k. We call this modification the ALL-IN-RAM design. We call the current design, which is described in §4.4.3, the INDEX-IN-RAM design.

In the rest of this section, we first perform a back-of-the-envelope calculation of the total RAM and total disk cost in the two designs (§F.1.1). We then consider the cost of processors (§F.1.2) and then compare the to-

tal costs (§F.1.3). The key parameters are two ratios: (1) the ratio of the rate of daily spam to the rate of non-spam (currently roughly 3:1, as mentioned in §4.6.5); and (2) the ratio of the cost per disk to the cost of one gigabyte of RAM. The intuition is that if the rate of non-spam is high, then the SETs of legitimate stamps consume much RAM, and it might be more efficient to use the index. If the rate of spam is high, then not much RAM will be consumed, and the enforcer can save disk accesses by using ALL-IN-RAM instead of INDEX-IN-RAM.

F.1.1 RAM and Disk Costs

Let ϕ_f be the number of fresh, legitimately stamped emails sent per day. Also, as discussed in §4.4.5 (page 100), a spam email with a reused stamp has the same effect on the enforcer as a spurious TEST request. For this reason, our calculations here do not incorporate the number of daily spams explicitly. Instead, we represent "adversarial activity" with a variable ϕ_a, defined as the number of spurious TEST or SET requests that adversaries can generate each day. Then, the analysis is unaffected by any choice by adversaries about whether to send spams, spurious TESTs, spurious SETs, or a combination of the three.

We first consider ALL-IN-RAM. We begin with SET requests. Each of the ϕ_f daily legitimately stamped emails induces a SET request. And, adversaries can issue ϕ_a spurious SET requests. All of these SET requests consume RAM. TEST requests consume neither disk accesses nor space in RAM.[1] They do cost processor cycles, which we consider below.

We can calculate the RAM cost of ALL-IN-RAM as follows. Each SET request induces two PUT requests (see Figure 4.4 on page 89 in §4.4.1). Each PUT request consumes ∼24 bytes of RAM (20 bytes for the value and ∼4 bytes of data structure overhead). Each node must store PUTs for two days. Thus, the total RAM required by the enforcer to handle two days' worth of SET requests is about $24 \cdot 2 \cdot 2 \cdot (\phi_f + \phi_a)$ bytes, and the total dollar cost of RAM is

$$\frac{96\left(\phi_f + \phi_a\right) c_R}{2^{30}}, \tag{F.1}$$

where c_R is the cost in dollars of 1 GByte of RAM.

We now consider the RAM and disk cost of INDEX-IN-RAM. Under this design, adversaries could put all of their "firepower" to work exhausting

[1] Note that, under ALL-IN-RAM, each node needs a single disk for logging and crash recovery. We account for this cost below, in §F.1.2.

RAM or exhausting the disk bandwidth. The enforcer must be provisioned for both possibilities; otherwise, its resources could be exhausted.

The dollar cost of INDEX-IN-RAM is as follows. We begin with RAM. There can be up to $\phi_a + \phi_f$ SET requests per day. In this case, each PUT request costs only 5.5 bytes of RAM, as discussed in §4.4.3. Thus, the enforcer's dollar cost from RAM is $5.5 \cdot 2 \cdot 2 \cdot (\phi_f + \phi_a) \cdot c_R/2^{30}$. We now consider the disk cost. Recall that TESTs for fresh stamps almost never induce a disk access. Thus, the disk cost is driven only by the number of spams sent per day or, equivalently, by the number of spurious "reused" TEST requests issued by adversaries. There can be up to ϕ_a spurious TEST requests per day. We assume that one disk can handle 320 requests per second (see §4.6.3). With 86,400 seconds in the day, the enforcer needs $\phi_a/(320 \cdot 86400) = \phi_a/(2.8 \cdot 10^7)$ disks. Its total dollar cost for RAM and disk is

$$\frac{22\left(\phi_f + \phi_a\right) c_R}{2^{30}} + \frac{\phi_a \cdot c_D}{2.8 \cdot 10^7}, \tag{F.2}$$

where c_D is the cost in dollars of a single disk.

F.1.2 Adding Processor Costs

Processor costs are driven by two factors. First, there must be enough processors to handle all of the RPCs; for details of how processors bottleneck the enforcer's ability to process RPCs, see §4.6.3, page 109 and §4.6.4, page 110. Second, the total RAM and disk calculated above must be divided over some number of machines, each of which needs a processor.

To account for the first factor, we assume that a node can process 40,000 RPCs/second (see the sections and pages just referenced). Then, over a whole day, a node can process $40,000 \cdot 86,400 = 3.5 \cdot 10^9$ RPCs. We now estimate how many RPCs the enforcer must handle, for $r = 5$. Each of the ϕ_f legitimate emails generates 3 RPCs for SETs (the SET, 1 PUT request, and 1 PUT response) and 11 RPCs for TESTs (five GET requests, five GET responses, plus the TEST). Each of the ϕ_a adversarial requests generates, in the worst case, 11 RPCs also. For more detail on this reasoning, see §4.6.4, page 112 and Appendix E.2. Thus, the enforcer needs a number of processors that is at least

$$\frac{14\phi_f + 11\phi_a}{3.5 \cdot 10^9}. \tag{F.3}$$

To account for the second factor, we note that the enforcer also needs enough machines to house all of the RAM and disks. Let n_R equal the number of GBytes of RAM per machine, and let n_D equal the number of disks

per machine. Appropriate values for these variables depend on which design we are considering.

Taking both factors together, the enforcer needs, for ALL-IN-RAM, a number of processors equal to

$$\max\left(\frac{14\phi_f + 11\phi_a}{3.5 \cdot 10^9}, \frac{96\left(\phi_f + \phi_a\right)}{2^{30} \cdot n_R}\right). \tag{F.4}$$

For INDEX-IN-RAM, the calculation is similar, except that we need to incorporate a term that accounts for the fact that processors are needed to drive the disks:

$$\max\left(\frac{14\phi_f + 11\phi_a}{3.5 \cdot 10^9}, \frac{22\left(\phi_f + \phi_a\right)}{2^{30} \cdot n_R}, \frac{\phi_a}{2.8 \cdot 10^7 \cdot n_D}\right). \tag{F.5}$$

We can now account for the cost of processors, disk, and RAM under the two alternatives. For ALL-IN-RAM, we need to set n_R so that the two terms in (F.4) are equal; otherwise, processors or RAM will be wasted. The exact value of n_R depends on the ratio ϕ_f/ϕ_a, but the range is between 22 and 29 GBytes. By choice of n_R, we eliminate the max operator in (F.4). Letting c_P be the cost of an appropriate processor, we get that the total cost of ALL-IN-RAM is

$$\frac{96\left(\phi_f + \phi_a\right)c_R}{2^{30}} + \frac{\left(14\phi_f + 11\phi_a\right)\left(c_P + c_D\right)}{3.5 \cdot 10^9}. \tag{F.6}$$

The c_D term enters because each machine in the enforcer needs a disk for logging and crash recovery, as mentioned in footnote 1 in §F.1.1.

For INDEX-IN-RAM, we again eliminate n_R by setting the first two terms in (F.5) equal to each other. The resulting value of n_R is between 5 and 7 GBytes. We can do likewise to eliminate n_D in (F.5). We get that n_D could be as low as 8 (for $\phi_a = 3\phi_f$, corresponding to the 3:1 ratio of spam to legitimate email) or as high as 11 (for $\phi_a \gg \phi_f$). These values of n_D are quite high, so in practice we might be forced to use a lower value for n_D, meaning that the enforcer would have to waste processors. However, for simplicity, we assume that n_D can be as high as necessary, which allows us to eliminate the max operator in (F.5). The total cost of INDEX-IN-RAM is

$$\frac{22\left(\phi_f + \phi_a\right)c_R}{2^{30}} + \frac{\phi_a \cdot c_D}{2.8 \cdot 10^7} + \frac{\left(14\phi_f + 11\phi_a\right)c_P}{3.5 \cdot 10^9}. \tag{F.7}$$

F.1.3 Comparison

We can now determine which of ALL-IN-RAM and INDEX-IN-RAM is cheaper by comparing (F.6) and (F.7). Rearranging and comparing those two expressions, we get that (F.6), the dollar cost of ALL-IN-RAM, is smaller when

$$\left(\frac{\phi_f}{\phi_a} + 1\right)\left(\frac{c_R}{c_D} + 0.058\right) < 0.53 \tag{F.8}$$

Today, $c_R \approx$ \$60, and the cost of the SCSI disks that we use in our experiments, c_D, is $\approx$ \$140 (these numbers are determined by appropriate searches on `newegg.com`). As mentioned above, the ratio ϕ_f/ϕ_a is currently 1:3. Plugging in these values, we find that the right-hand side of the expression above is slightly smaller, that is, the cost of INDEX-IN-RAM is smaller, slightly.

However, this calculation is likely too generous to INDEX-IN-RAM, for several reasons. First, as mentioned above, commodity rack-mounted servers may not be able to house 11 disks. Second, adversaries' power, represented by ϕ_a, might be significantly greater than their activities today indicate (today, their activities cause the ratio ϕ_f/ϕ_a to equal 1/3); this point is discussed in §4.4.5. Indeed, if $\phi_a > 11.2 \cdot \phi_f$, then the left-hand side of (F.8) is smaller than the right-hand side. Third, the calculations above did not take into account the LRU cache in RAM (discussed in §4.4.3) that nodes maintain in the INDEX-IN-RAM case; this cache adds costs to INDEX-IN-RAM. Finally, the INDEX-IN-RAM design likely requires more power since it relies on more disks.

Our conclusion is that the two designs, under today's ratio of $\phi_f/\phi_a = 1/3$, are roughly equal in cost. However, under a more "energetic" adversary, the ALL-IN-RAM design would actually be cheaper. Moreover, as the price of RAM continues to decline, ALL-IN-RAM will become even cheaper.

F.2 FLASH MEMORY

Another set of possibilities involves flash memory. Specifically, one could replace the disks in INDEX-IN-RAM with flash memory.[2] Or one could replace the *RAM* in INDEX-IN-RAM with flash memory and continue to use the disks. Or one could replace the bulk of RAM in ALL-IN-RAM with flash memory. The trade-offs are as follows. First, flash memory is cheaper, but

[2] This option was suggested by Jakob Eriksson.

slower, than RAM for the same amount of space. Second, compared to disks, flash memory stores fewer bytes for the same number of dollars but offers much higher random access throughput.

F.3 SUMMARY

The summary of this appendix's musings and back-of-the-envelope calculations is as follows. First, given today's ratio of spam to legitimate email, our current design is cheaper than a design that places all keys and values in RAM. Second, if adversaries would attack the enforcer with much more firepower than they use to send spam today, then the enforcer could save resources (money and perhaps machines) by storing keys and values fully in RAM. Finally, a future possibility for the enforcer is to replace some of its RAM or disk storage with flash memory.

References

All of the URLs listed here are valid as of November, 2007.

[1] M. Abadi, A. Birrell, M. Burrows, F. Dabek, and T. Wobber. Bankable postage for network services. In *Proc. Asian Computing Science Conference*, Dec. 2003. (Referenced on pages 76, 116, 117, 121, 124, 125, 126, 129, and 155.)

[2] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology (TOIT)*, 5(2), May 2005. (Referenced on pages 17, 26, 63, and 124.)

[3] S. Agarwal, T. Dawson, and C. Tryfonas. DDoS mitigation via regional cleaning centers. Sprint ATL Research Report RR04-ATL-013177, Aug. 2003. (Referenced on page 33.)

[4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005. (Referenced on page 128.)

[5] Alexa Internet, Inc. `http://www.alexa.com`. (Referenced on page 72.)

[6] Amazon Web Services. `http://aws.amazon.com`. (Referenced on page 129.)

[7] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2003. (Referenced on pages 68 and 69.)

[8] D. Angluin and L. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, (19), 1979. (Referenced on page 161.)

[9] Arbor Networks, Inc. http://www.arbornetworks.com. (Referenced on pages 25 and 68.)

[10] T. Aura, P. Nikander, and J. Leiwo. DoS-resistant authentication with client puzzles. In *Proc. International Workshop on Security Protocols*, 2000. (Referenced on pages 17, 26, 63, and 64.)

[11] A. Back. Hashcash—a denial of service counter-measure, Aug. 2002. http://www.cypherspace.org/adam/hashcash/hashcash.pdf. (Referenced on pages 17, 26, 63, and 124.)

[12] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, Feb. 2003. (Referenced on pages 88 and 128.)

[13] H. Balakrishnan and D. Karger. Spam-i-am: A proposal to combat spam using distributed quota management. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2004. (Referenced on pages 76, 77, 78, 79, 80, 81, 84, 116, and 126.)

[14] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *Proc. ACM SIGCOMM*, Sept. 1999. (Referenced on page 44.)

[15] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999. (Referenced on page 68.)

[16] P. Barford and V. Yegneswaran. An inside look at botnets. In *Special Workshop on Malware Detection, Advances in Information Security, Springer Verlag*, 2006. http://pages.cs.wisc.edu/~pb/botnets_final.pdf. (Referenced on page 21.)

[17] Barrett Lyon. Private conversation, Aug. 2006. (Referenced on pages 70 and 146.)

[18] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, and E. Rachlin. Making P2P accountable without losing privacy. In *Proc. Workshop on Privacy in the Electronic Society (WPES)*, Oct. 2007. (Referenced on pages 127 and 130.)

[19] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, Nov. 1993. (Referenced on page 84.)

[20] M. Bellare and P. Rogaway. The exact security of digital signatures—how to sign with RSA and Rabin. In *Proc. EUROCRYPT*, May 1996. (Referenced on page 84.)

[21] D. J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`. (Referenced on page 13.)

[22] Bonded Sender Program. `http://www.bondedsender.com/info_center.jsp`. (Referenced on page 125.)

[23] P. Boothe, J. Hiebert, and R. Bush. Short-lived prefix hijacking on the Internet, Feb. 2006. Presentation to NANOG. `http://www.nanog.org/mtg-0602/pdf/boothe.pdf`. (Referenced on page 158.)

[24] D. Brown. Gangsters hijack home PCs to choke internet with spam. *The Times*, Nov. 2006. `http://business.timesonline.co.uk/tol/business/law/public_law/article649541.ece`. (Referenced on pages 22, 30, and 70.)

[25] BT Counterpane. DDoS prevention offerings. `http://www.counterpane.com/ddos-offerings.html`. (Referenced on pages 25, 32, 45, and 68.)

[26] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact E-Cash. In *Proc. EUROCRYPT*, May 2005. (Referenced on pages 82 and 127.)

[27] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002. (Referenced on page 128.)

[28] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002. (Referenced on pages 78 and 128.)

[29] Cisco Guard, Cisco Systems, Inc. `http://www.cisco.com`. (Referenced on pages 25 and 68.)

[30] ClickZ News. Costs of blocking legit e-mail to soar, Jan. 2004. http://www.clickz.com/news/article.php/3304671. (Referenced on page 12.)

[31] ClickZ News. Spam blocking experts: False positives inevitable, Feb. 2004. http://www.clickz.com/news/article.php/3315541. (Referenced on page 12.)

[32] ClickZ News. AOL to implement e-mail certification program, Jan. 2006. http://www.clickz.com/news/article.php/3581301. (Referenced on page 125.)

[33] CNET News. Bots slim down to get tough, Nov. 2005. http://news.com.com/Bots+slim+down+to+get+tough/2100-7355_3-5956143.html. (Referenced on pages 70 and 146.)

[34] Computer Industry Almanac, Inc. PCs in-use surpassed 900M in 2005, May 2006. http://www.c-i-a.com/pr0506.htm. (Referenced on pages 23 and 118.)

[35] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting and disrupting botnets. In *Proc. USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, July 2005. (Referenced on pages 70, 71, and 146.)

[36] J.-S. Coron. On the exact security of full domain hash. In *Proc. CRYPTO*, Aug. 2000. (Referenced on page 84.)

[37] L. F. Cranor and B. A. LaMacchia. Spam! *Communications of the ACM*, 41(8), Aug. 1998. (Referenced on pages 12 and 79.)

[38] Criminal complaint filed Aug. 25, 2004, United States v. Ashley et al., No. 04 MJ 02112 (Central District of California). http://www.reverse.net/operationcyberslam.pdf. (Referenced on pages 13 and 22.)

[39] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004. (Referenced on page 104.)

[40] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using time zones. In *Proc. Network and Distributed System Security Symposium (NDSS)*, Feb. 2006. (Referenced on pages 22, 30, and 70.)

[41] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *Proc. USENIX Security Symposium*, Aug. 2001. (Referenced on pages 26, 63, and 64.)

[42] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *CCR*, 25(1), Jan. 1995. (Referenced on pages 68 and 133.)

[43] J. Douceur. The sybil attack. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002. (Referenced on page 74.)

[44] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 1996. (Referenced on page 97.)

[45] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *Proc. CRYPTO*, 2003. (Referenced on pages 17, 26, 63, and 124.)

[46] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proc. CRYPTO*, 1992. (Referenced on pages 17, 26, 63, and 124.)

[47] Emulab. `http://www.emulab.net`. (Referenced on pages 50 and 105.)

[48] Enterprise IT Planet. False positives: Spam's casualty of war costing billions, Aug. 2003. `http://www.enterpriseitplanet.com/security/news/article.php/2246371`. (Referenced on page 12.)

[49] eWEEK. Money bots: Hackers cash in on hijacked PCs, Sept. 2006. `http://www.eweek.com/article2/0,1895,2013957,00.asp`. (Referenced on pages 70 and 146.)

[50] S. E. Fahlman. Selling interrupt rights: A way to control unwanted e-mail and telephone calls. *IBM Systems Journal*, 41(4):759–766, 2002. (Referenced on pages 124 and 133.)

[51] E. Falk. New host cloaking technique used by spammers, Feb. 2006. `http://thespamdiaries.blogspot.com/2006/02/new-host-cloaking-technique-used-by.html`. (Referenced on page 158.)

[52] N. Feamster, J. Jung, and H. Balakrishnan. An empirical study of "bogon" route advertisements. *CCR*, 35(1), Jan. 2005. (Referenced on page 158.)

[53] W. Feng. The case for TCP/IP puzzles. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, Aug. 2003. (Referenced on pages 26, 63, and 64.)

[54] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 17(6), 2003. (Referenced on pages 30 and 44.)

[55] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004. (Referenced on page 129.)

[56] F. C. Freiling, T. Holz, and G. Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, Sept. 2005. (Referenced on pages 65 and 71.)

[57] L. Frieder and J. Zittrain. Spam works: Evidence from stock touts and corresponding market activity. Berkman Center Research Publication No. 2006-11, Mar. 2007. `http://ssrn.com/abstract=920553`. (Referenced on page 22.)

[58] You might be an anti-spam kook if... `http://www.rhyolite.com/anti-spam/you-might-be.html`. (Referenced on page 79.)

[59] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazières, and H. Yu. Re: Reliable email. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006. (Referenced on page 122.)

[60] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. The Millicent protocol for inexpensive electronic commerce. In *Proc. International World Wide Web Conference (WWW)*, Dec. 1995. `http://www.w3.org/Conferences/WWW4/Papers/246`. (Referenced on pages 82 and 127.)

[61] V. D. Gligor. Guaranteeing access in spite of distributed service-flooding attacks. In *Proc. International Workshop on Security Protocols*, 2003. (Referenced on pages 25, 31, and 64.)

[62] L. H. Gomes, C. Cazita, J. M. Almeida, V. Almeida, and W. Meira Jr. Charaterizing a spam traffic. In *Proc. ACM Internet Measurement Conference (IMC)*, Oct. 2004. (Referenced on page 114.)

[63] Goodmail Systems. `http://www.goodmailsystems.com`. (Referenced on page 125.)

[64] J. Goodman, G. V. Cormack, and D. Heckerman. Spam and the ongoing battle for the inbox. *Communications of the ACM*, 50(2), Feb. 2007. (Referenced on page 79.)

[65] J. Goodman and R. Rounthwaite. Stopping outgoing spam. In *Proc. ACM Conference on Electronic Commerce (EC)*, May 2004. (Referenced on pages 117 and 123.)

[66] P. Graham. Better bayesian filtering. `http://www.paulgraham.com/better.html`. (Referenced on page 122.)

[67] S. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2000. (Referenced on page 128.)

[68] J. Grimmelmann and B. Bolin. Policy responses to spam. `http://works.bepress.com/james_grimmelmann/11/`, Mar. 2005. (Referenced on pages 79 and 121.)

[69] R. F. Guilmette. ANNOUNCE: MONKEYS.COM: now retired from spam fighting. Newsgroup posting: `news.admin.net-abuse.email`, Sept. 2003. (Referenced on page 98.)

[70] C. A. Gunter, S. Khanna, K. Tan, and S. Venkatesth. DoS protection for reliably authenticated broadcast. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2004. (Referenced on pages 27 and 63.)

[71] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004. (Referenced on page 128.)

[72] I. Gupta, K. Birman, P. Linka, A. Demers, and R. van Renesse. Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2003. (Referenced on page 128.)

[73] M. Handley. In a presentation to Internet architecture working group, DoS-resistant Internet subgroup, 2005. (Referenced on pages 13, 22, 23, 30, 31, 70, and 72.)

[74] M. Handley and A. Greenhalgh. Steps towards a DoS-resistant Internet architecture. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, Aug. 2004. (Referenced on page 69.)

[75] Honeynet Project and Research Alliance. Know your enemy: Tracking botnets. Mar. 2005. `http://www.honeynet.org/papers/bots/`. (Referenced on pages 21, 22, 30, and 70.)

[76] N. Ianelli and A. Hackworth. Botnets as a vehicle for online crime. CERT Coordination Center, Dec. 2005. `http://www.cert.org/archive/pdf/Botnets.pdf`. (Referenced on page 21.)

[77] IDC. Worldwide email usage forecast, 2005-2009: Email's future depends on keeping its value high and its cost low. `http://www.idc.com/`, Dec. 2005. (Referenced on pages 18 and 80.)

[78] *Information Week*. Botnets: Small is in. Oct. 2007. (Referenced on pages 70 and 146.)

[79] J. Jannotti. Private communication, July 2007. (Referenced on page 130.)

[80] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 1999. (Referenced on pages 17, 26, 63, and 64.)

[81] J. Jung and E. Sit. An empirical study of spam traffic and the use of DNS black lists. In *Proc. ACM Internet Measurement Conference (IMC)*, Oct. 2004. (Referenced on page 114.)

[82] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005. (Referenced on pages 13, 25, 31, 63, and 68.)

[83] D. Karger. Private communication, Apr. 2007. (Referenced on page 74.)

[84] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. ACM Symposium on the Theory of Computing (STOC)*, May 1997. (Referenced on pages 88 and 89.)

[85] D. E. Knuth. *The Art of Computer Programming*, volume 2, chapter 3.4.2. Addison-Wesley, third edition, 1998. (Referenced on page 42.)

[86] D. E. Knuth. *The Art of Computer Programming*, volume 3, chapter 6.4. Addison-Wesley, second edition, 1998. (Referenced on page 94.)

[87] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *Proc. ACM SIGCOMM*, Sept. 2006. (Referenced on pages 44 and 104.)

[88] H. Krawzyk, M. Bellare, and R. Canetti. HMAC: *Keyed-Hashing For Message Authentication*. RFC 2104, Internet Engineering Task Force, Feb. 1997. `http://www.faqs.org/rfcs/rfc2104.html`. (Referenced on page 88.)

[89] B. Krishnamurthy and E. Blackmond. SHRED: Spam harassment reduction via economic disincentives. `http://www.research.att.com/~bala/papers/shred-ext.ps`, 2004. (Referenced on pages 76 and 125.)

[90] M. Krohn. Building secure high-performance Web services with OKWS. In *Proc. USENIX Technical Conference*, June 2004. (Referenced on page 47.)

[91] B. J. Kuipers, A. X. Liu, A. Gautam, and M. G. Gouda. Zmail: Zero-sum free market control of spam. In *Proc. 4th International Workshop on Assurance in Distributed Systems and Networks*, June 2005. (Referenced on page 124.)

[92] B. Laurie and R. Clayton. "Proof-of-Work" proves not to work; version 0.2, Sept. 2004. `http://www.cl.cam.ac.uk/users/rnc1/proofwork2.pdf`. (Referenced on pages 67 and 117.)

[93] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004. (Referenced on pages 94 and 104.)

[94] T. Loder, M. V. Alstyne, and R. Wash. An economic response to unsolicited communication. *Advances in Economic Analysis & Policy*, 6(1), Mar. 2006. `http://www.bepress.com/bejeap/advances/vol6/iss1/art2`. (Referenced on pages 12, 124, and 133.)

[95] A. Mahimkar, J. Dange, V. Shmatikov, H. Vin, and Y. Zhang. dFence: Transparent network-based denial of service mitigation. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007. (Referenced on page 68.)

[96] D. Malkhi and M. K. Reiter. Byzantine quorum systems. In *Proc. ACM Symposium on the Theory of Computing (STOC)*, 1997. (Referenced on page 128.)

[97] D. Malkhi and M. K. Reiter. Secure and scalable replication in Phalanx. In *Proc. IEEE Symposium on Reliable Distributed Systems*, Oct. 1998. (Referenced on page 128.)

[98] D. Mankins, R. Krishnan, C. Boyd, J. Zao, and M. Frentz. Mitigating distributed denial of service attacks with dynamic resource pricing. In *Proc. IEEE Computer Security Applications Conference*, Dec. 2001. (Referenced on pages 17, 26, and 63.)

[99] J. Markoff. Attack of the zombie computers is growing threat. *The New York Times*, Jan. 2007. `http://www.nytimes.com/2007/01/07/technology/07net.html`. (Referenced on page 21.)

[100] David Mazières. (Referenced on page 123.)

[101] D. Mazières. A toolkit for user-level file systems. In *Proc. USENIX Technical Conference*, June 2001. (Referenced on pages 47 and 104.)

[102] D. Mazières. Blocking unwanted mail with Mail Avenger. *Virus Bulletin*, July 2005. See `http://www.mailavenger.org` or `http://www.scs.stanford.edu/~dm/home/papers/mazieres:avenger-virusbtn.pdf`. (Referenced on page 123.)

[103] Mazu Networks, Inc. `http://mazunetworks.com`. (Referenced on pages 25 and 68.)

[104] L. McLaughlin. Bot software spreads, causes new worries. *IEEE Distributed Systems Online*, 5(6), June 2004. http://csdl2.computer.org/comp/mags/ds/2004/06/o6001.pdf. (Referenced on pages 22, 30, 70, and 71.)

[105] D. McPherson and C. Labovitz. Worldwide infrastructure security report, volume II. Arbor Networks, Inc., Sept. 2006. http://www.arbor.net/downloads/worldwide_infrastructure_security_report_sept06.pdf. (Referenced on pages 70 and 146.)

[106] MessageLabs Ltd. MessageLabs intelligence: January 2007. http://www.messagelabs.com/mlireport/messagelabs_intelligence_report__january_2007_5.pdf. (Referenced on pages 12, 76, 113, and 117.)

[107] Messaging Anti-Abuse Working Group (MAAWG). Email metrics program, first quarter 2007 report. http://www.maawg.org/about/MAAWG20071Q_Metrics_Report.pdf, June 2007. (Referenced on pages 12, 76, 113, and 117.)

[108] J. Mirkovic and P. Reiher. A taxonomy of DDoS attacks and DDoS defense mechanisms. *CCR*, 34(2), Apr. 2004. (Referenced on pages 63 and 98.)

[109] W. Morein, A. Stavrou, D. Cook, A. Keromytis, V. Mishra, and D. Rubenstein. Using graphic turing tests to counter automated DDoS attacks against Web servers. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, Oct. 2003. (Referenced on pages 25, 31, 63, and 68.)

[110] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. (Referenced on page 161.)

[111] *Network World*. Extortion via DDoS on the rise. May 2005. http://www.networkworld.com/news/2005/051605-ddos-extortion.html. (Referenced on pages 22, 25, and 68.)

[112] *Network World*. How big is the botnet problem? July 2007. http://www.networkworld.com/research/2007/070607-botnets-side.html. (Referenced on page 22.)

[113] K. Park, V. S. Pai, K.-W. Lee, and S. Calo. Securing Web service by automatic robot detection. In *Proc. USENIX Technical Conference*, June 2006. (Referenced on page 31.)

[114] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: Protecting connection setup from denial-of-capability attacks. In *Proc. ACM SIGCOMM*, Aug. 2007. (Referenced on pages 26, 63, 64, 69, and 74.)

[115] V. Paxson and S. Floyd. Wide area traffic: the failure of Poisson modeling. *ACM/IEEE Transactions on Networking*, 3(3):226–244, 1995. (Referenced on page 105.)

[116] PC World. Spam-proof your in-box, June 2004. `http://www.pcworld.com/reviews/article/0,aid,115885,00.asp`. (Referenced on page 12.)

[117] The Penny Black Project. `http://research.microsoft.com/research/sv/PennyBlack/`. (Referenced on page 76.)

[118] *Pittsburgh Post-Gazette*. CMU student taps brain's game skills. Oct. 5, 2003. `http://www.post-gazette.com/pg/03278/228349.stm`. (Referenced on page 68.)

[119] J. Postel. *Internet Control Message Protocol*. RFC 792, Internet Engineering Task Force, Sept. 1981. `http://www.faqs.org/rfcs/rfc792.html`. (Referenced on page 13.)

[120] Prolexic Technologies, Inc. `http://www.prolexic.com`. (Referenced on pages 25, 32, 45, and 68.)

[121] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002. (Referenced on page 97.)

[122] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, Jan. 2002. (Referenced on page 94.)

[123] Radicati Group Inc.: Market Numbers Quarterly Update Q2 2003. (Referenced on pages 18 and 80.)

[124] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proc. ACM Internet Measurement Conference (IMC)*, Oct. 2006. (Referenced on page 71.)

[125] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In *Proc. 1st USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*, Apr. 2007. `http://www.usenix.org/events/hotbots07/tech/full_papers/rajab/rajab.pdf`. (Referenced on pages 22, 70, 71, and 146.)

[126] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proc. ACM SIGCOMM*, Sept. 2006. (Referenced on pages 79 and 158.)

[127] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *Proc. ACM SIGCOMM*, Aug. 2004. (Referenced on page 28.)

[128] S. Ranjan, R. Swaminathan, M. Uysal, and E. W. Knightly. DDoS-resilient scheduling to counter application layer attacks under imperfect detection. In *Proc. IEEE INFOCOM*, Apr. 2006. (Referenced on pages 25 and 68.)

[129] E. Ratliff. The zombie hunters. *The New Yorker*, Oct. 10, 2005. (Referenced on pages 13 and 22.)

[130] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. ACM SIGCOMM*, Aug. 2005. (Referenced on pages 15 and 129.)

[131] F.-R. Rideau. Stamps vs spam: Postage as a method to eliminate unsolicited commercial email. `http://fare.tunes.org/articles/stamps_vs_spam.html`, Sept. 2002. (Referenced on page 124.)

[132] R. L. Rivest and A. Shamir. PayWord and MicroMint: Two simple micropayment schemes. In *Proc. International Workshop on Security Protocols*, Apr. 1996. (Referenced on pages 82 and 127.)

[133] R. Rodrigues and B. Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, Dec. 2003. (Referenced on page 128.)

[134] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992. (Referenced on page 94.)

[135] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990. (Referenced on page 128.)

[136] SecurityFocus. FBI busts alleged DDoS mafia. Aug. 2004. `http://www.securityfocus.com/news/9411`. (Referenced on pages 13 and 22.)

[137] V. Sekar. Private communication, Sept. 2007. (Referenced on pages 71 and 72.)

[138] V. Sekar, N. Duffield, O. Spatscheck, J. van der Merwe, and H. Zhang. LADS: Large-scale automated DDoS detection system. In *Proc. USENIX Technical Conference*, June 2006. (Referenced on pages 7, 8, 45, and 71.)

[139] Shadowserver Foundation. Bot counts. `http://www.shadowserver.org/wiki/pmwiki.php?n=Stats.BotCounts`. (Referenced on page 71.)

[140] Shadowserver Foundation. Private communication, Jan. 2007. `http://www.shadowserver.org`. (Referenced on page 70.)

[141] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002. (Referenced on page 98.)

[142] M. Sherr, M. Greenwald, C. A. Gunter, S. Khanna, and S. S. Venkatesh. Mitigating DoS attack through selective bin verification. In *Proc. 1st Workshop on Secure Network Protocols*, Nov. 2005. (Referenced on pages 27 and 63.)

[143] K. K. Singh. Botnets—An introduction. Course Project, CS6262, Georgia Institute of Technology, Spring, 2006. http://www-static.cc.gatech.edu/classes/AY2006/cs6262_spring/botnets.ppt. (Referenced on pages 72 and 99.)

[144] SpamAssassin. http://spamassassin.apache.org/. (Referenced on page 122.)

[145] Spambouncer. http://www.spambouncer.org. (Referenced on page 124.)

[146] Sender Policy Framework. http://spf.pobox.com/. (Referenced on page 123.)

[147] M. Srivatsa, A. Iyengar, J. Yin, and L. Liu. A middleware system for protecting against application level denial of service attacks. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, Nov. 2006. (Referenced on pages 25 and 68.)

[148] A. Stavrou, J. Ioannidis, A. D. Keromytis, V. Misra, and D. Rubenstein. A pay-per-use DoS protection mechanism for the Web. In *Proc. International Conference on Applied Cryptography and Network Security*, June 2004. (Referenced on pages 26 and 63.)

[149] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *ACM/IEEE Transactions on Networking*, 11(1):17–32, Feb. 2003. (Referenced on page 88.)

[150] B. Stone. Spam doubles, finding new ways to deliver itself. *The New York Times*, Dec. 2006. http://www.nytimes.com/2006/12/06/technology/06spam.html. (Referenced on pages 12, 76, 113, and 117.)

[151] Stupid Google virus/spyware CAPTCHA page. http://www.spy.org.uk/spyblog/2005/06/stupid_google_virusspyware_cap.html. (Referenced on pages 25, 31, and 68.)

[152] W. Sturgeon. Denial of service attack victim speaks out. May 2005. http://management.silicon.com/smedirector/0,39024679,39130810,00.htm. (Referenced on page 72.)

[153] TechWeb News. Dutch botnet bigger than expected. Oct. 2005. http://informationweek.com/story/showArticle.jhtml?articleID=172303265. (Referenced on pages 22, 30, 70, and 71.)

[154] B. Templeton. Best way to end spam. http://www.templetons.com/brad/spam/endspam.html. (Referenced on page 123.)

[155] B. Templeton. Origin of the term "spam" to mean net abuse. http://www.templetons.com/brad/spamterm.html. (Referenced on page 12.)

[156] B. Templeton. Reaction to the DEC spam of 1978. http://www.templetons.com/brad/spamreact.html. (Referenced on page 12.)

[157] B. Templeton. The spam solutions. http://www.templetons.com/brad/spam/spamsol.html, 2003. (Referenced on pages 79 and 121.)

[158] The Spamhaus Project. http://www.spamhaus.org. (Referenced on pages 15 and 122.)

[159] The Spamhaus Project. Spammers release virus to attack spamhaus.org. http://www.spamhaus.org/news.lasso?article=13, Nov. 2003. (Referenced on page 98.)

[160] *The Register*. East European gangs in online protection racket. Nov. 2003. http://www.theregister.co.uk/2003/11/12/east_european_gangs_in_online. (Referenced on page 22.)

[161] *The Register*. Phatbot arrest throws open trade in zombie PCs. May 2004. http://www.theregister.co.uk/2004/05/12/phatbot_zombie_trade. (Referenced on page 23.)

[162] D. Thomas. Deterrence must be the key to avoiding DDoS attacks, June 2005. http://www.vnunet.com/computing/analysis/2137395/deterrence-key-avoiding-ddos-attacks. (Referenced on pages 45 and 72.)

[163] R. Thomas and J. Martin. The underground economy: Priceless. *;login:*, 31(6), Dec. 2006. http://www.usenix.org/publications/login/2006-12/openpdfs/cymru.pdf. (Referenced on page 21.)

[164] R. Vasudevan, Z. M. Mao, O. Spatscheck, and J. van der Merwe. Reval: A tool for real-time evaluation of DDoS mitigation strategies. In *Proc. USENIX Technical Conference*, June 2006. (Referenced on page 44.)

[165] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1), Mar. 1985. (Referenced on page 42.)

[166] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *Communications of the ACM*, 47(2), Feb. 2004. (Referenced on pages 25, 31, 68, 116, 122, and 134.)

[167] M. Walfish, H. Balakrishnan, D. Karger, and S. Shenker. DoS: Fighting fire with fire. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2005. (Referenced on pages 10 and 27.)

[168] M. Walfish, J.D. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Distributed quota enforcement for spam control. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006. (Referenced on pages 8 and 116.)

[169] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *Proc. ACM SIGCOMM*, Sept. 2006. (Referenced on pages 8, 10, 26, 27, and 63.)

[170] X. Wang and M. K. Reiter. A multi-layer framework for puzzle-based denial-of-service defense. *International Journal of Information Security*, 2007. Forthcoming and published online, `http://dx.doi.org/10.1007/s10207-007-0042-x`. (Referenced on pages 17, 26, 63, and 64.)

[171] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Proc. CRYPTO*, Aug. 2005. (Referenced on page 84.)

[172] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for DoS resistance. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, Oct. 2004. (Referenced on pages 26, 63, and 64.)

[173] L. Weber. Wikimedia request statistics. `http://tools.wikimedia.de/~leon/stats/reqstats`. (Referenced on page 72.)

[174] L. Weber. Wikimedia traffic statistics. `http://tools.wikimedia.de/~leon/stats/trafstats`. (Referenced on page 32.)

[175] M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In *Proc. USENIX Symposium on Internet Technologies and Systems (USITS)*, Mar. 2003. (Referenced on page 97.)

[176] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001. (Referenced on page 97.)

[177] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *Proc. IEEE Symposium on Security and Privacy*, May 2004. (Referenced on pages 32, 68, and 69.)

[178] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *Proc. ACM SIGCOMM*, Aug. 2005. (Referenced on pages 32, 63, 68, and 69.)