Application-Level Distributed Denial of Service Prevention in a Replicated System

by

Alexander M. Vandiver

S.B. C.S., M.I.T., 2005

Submitted to the Department of Electrical Engineering

and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 2007

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 13 2008
LIBRARIES

Author .....
Department of Electrical Engineering and Computer Science
September 5, 2007

Certified by .....
Hari Balakrishnan
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Application-Level Distributed Denial of Service Prevention in a Replicated System

by

Alexander M. Vandiver

Submitted to the
Department of Electrical Engineering and Computer Science

September 5, 2007

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This paper presents the design and implementation of DFQ (Distributed Fair Queueing), a distributed system for defending a replicated set of HTTP servers against application-level distributed denial of service (DDoS) attacks. By using a modification of weighted fair queueing, all clients are guaranteed a fair share of the servers, no matter how many or which servers they connect to. DFQ continues to provide fair service even against malicious clients who are able to spoof additional IP addresses. It is also capable of accommodating HTTP proxies, which regularly provide many times more traffic than a single host. Such properties are desirable for package management servers and the like, whose responsiveness in the presence of flash crowds and malicious attackers is paramount to the security of the overall system.

Thesis Supervisor: Hari Balakrishnan
Title: Professor

Acknowledgments

I'd like to thank my thesis supervisor, Hari Balakrishnan, for his help and guidance throughout this project. I'd also like to thank my brother, Ben Vandiver, for being willing to listen and give suggestions when I ran into difficulties. Finally, thanks to my girlfriend, Laurel Bobrow, and my parents, Kathy and J. Kim Vandiver, for their support throughout this entire process.

Contents

Chapter 1

Introduction

Distributed denial-of-service (DDoS) attacks are an increasingly common problem on the public internet. To launch a DDoS attack, the attacker usually uses thousands of computers that they have previously compromised, most of which belong to unsuspecting cable modem users and the like. When instructed, each of the computers ("zombies") in these "botnets" suddenly turn all of their bandwidth and processing power on a single website or service, effectively denying all legitimate clients access to that service.

Initially, DDoS attacks targeted the most obvious resource in the network — the victim's network link. By simply flooding the target with packets, they could often overwhelm the bandwidth of the server, causing it to be unable to service requests as fast as they arrived. However, as defenses have evolved to counter these low-level attacks[21, 23], attackers have shifted their focus. Increasingly, to attack complex websites backed by database servers, they are crafting requests which attempt to target the database, rather than the network connection[11, 16, 28, 30]. In essence, this mode of operation shifts the bottleneck that they are attempting to exploit, to a layer where it is harder to detect[18].

By carefully crafting legitimate HTTP requests, which are mostly indistinguishable from the requests of real clients, attackers can achieve the same denial-of-service effects as before, at much lower bandwidth cost and chance of detection. These attacks are known as *application-level denial of service* attacks.

We wish to provide a way to protect a distributed set of replicated HTTP servers from application-level DDoS attacks. In order to do so, we provide fair service to all clients, taking into account the size of each client's requests. It is not sufficient that a single server be fair, for clients may connect to multiple servers; we must ensure the best fairness possible across the aggregate of all servers.

1.1 Power of attackers

In these attacks, we assume that the attackers are capable of crafting requests that are sufficiently similar to legitimate requests that they are indistinguishable. That is, we assume that the attacker has taken the time to remove the myriad of tell-tale signs that simple pattern-detection software would be able to detect — invalid headers, suspiciously fast inter-request arrival times, odd client names, etc.

Additionally, we assume the clients are capable of forging HTTP cookies, the canonical method of tracking individual users. By forging cookies, they are able to create new user profiles at will, evading most simple session tracking methods. A smart attacker could use this to impersonate a client-side proxy, for instance (see section 1.2).

An even more powerful tool that we must assume attackers have at their disposal is the ability to forge the originating IP address of their connection — this is a common occurrence with network level DDoS attacks. However, source address spoofing is considerably more difficult with application-level attacks; as most applications use TCP, the spoofed IP must be able to complete the entire TCP handshake in order to establish a connection. Thus, the spoofing computer must generally be able to intercept packets destined to the address it was spoofing, which limits them to spoofing IP addresses on the same physical link as them, which is generally the same subnet.

However, IP spoofing is a formidable tool. Combined with the ability to craft arbitrary requests with arbitrary cookies, it allows a single compromised host on a subnet to impersonate the *entire* subnet, if it so desires. This factor, combined with the several thousands of hosts in the botnet that the attacker already has at their

disposal, can yield a truly huge number of unique IPs with which to attack a host.

1.2 Internet Traffic Patterns

On the Internet, seeing abnormally high amounts of traffic from a single host is not always a sign of an attack. In fact, there are a great number of IP addresses on the Internet that legitimately send hundreds of times the bandwidth of a single client: HTTP proxies. For instance, all of America On-Line's 13 million clients pass through at most a few thousand machines, appearing to webservers as if they were requesting pages at hundreds of times the normal rate.

While many proxies add additional header information (canonically, an `X-Forwarded-For` header) concerning the original host that they are proxying for, this information cannot be relied on. Many proxies are proxies for private internal networks — thus, the IP address communicated by the proxy has no external meaning. More to the point, there is also no way to ensure that the proxy is not lying about the hosts it claims to be proxying for, particularly when they are in private IP space.

We also note that the simultaneous arrival of hundreds of new sessions and IP addresses is not necessarily a coordinated DDoS attack. Such simultaneous arrivals happen with non-malicious clients during so-called "flash crowds." For instance, in the "Slashdot effect," a widely-read website links to a website not nearly as well-provisioned; in the space of minutes, many thousands of users may make requests of the smaller website. In such cases, all of the clients are well-meaning, and should indeed be served if possible. As such, sufficiently advanced attackers will masquerade as a flash crowd.

1.3 Thesis

We present the design for DFQ (Distributed Fair Queueing), a distributed system of HTTP front-ends to harden existing replicated servers against application-level DDoS attacks. It is resistant to IP spoofing, as well as tolerant of existing proxies.

For the purposes of this research, we make several simplifying assumptions. We restrict ourselves to a model which protects generic webservers, which must be able to run without modification; many complex websites have large amounts of custom logic embedded into their servers, and altering this logic to support DFQ would be complicated and error-prone.

We also assume that requests require varying amounts of server resources to process; attackers may well attempt to generate requests that take longer to service. Part of the prior assumption that requests will run on unmodified webservers is the assumption that requests are atomic, and cannot be split into separate time slices in any manner. That is, once a server has committed to servicing a request, it runs it to completion.

A more limiting assumption is our assumption that the server is able to determine the amount of work a request will require, merely by examining the incoming request. This assumption is unfortunate; however, without support from the application, it is unavoidable: in general, determining the running time of a request before it executes is equivalent to solving the Halting Problem.

In addition, we assume that the service is comprised of multiple, functionally identical webservers. A client may decide to connect to an arbitrary set of servers, and does not care which servers it receives service from. A client is only guaranteed to recieve service from one server; connections to overloaded servers may never be serviced. Nonetheless, it is guaranteed a fair share of the total aggregate service in the system. It is also assumed that servers cannot redirect connections to other servers, nor answer requests asked of another server.

We also make the simplifying assumption that bandwidth between front-ends is not likely to be constrained when the front-end is under load. Though the bandwidth requirements between front-ends are relatively small, and could be further reduced, we recognize this as a limitation of DFQ; more work remains to be done on the *minimal* required information that distributed front-ends need to communicate in order to still ensure global fairness.

Finally, since automatic detection of proxies is difficult or impossible, we presume

that the algorithm has access to a traffic distribution, grouped by subnet, of request rates from a period when the service is not under attack. In section 3.3, we briefly address methods of gathering and maintaining such a traffic record, but, for the most part, the exact details are outside the scope of this research.

Chapter 2 discusses prior work on the subject, both in DDoS prevention, as well as queueing theory, which has notable similarities. Chapter 3 discusses the design choices in depth, and offers correctness proofs for the distributed case. Chapter 4 provides results from simulations which show the system in action, and Chapter 5 summarizes and offers conclusions from the research.

Chapter 2

Prior work

2.1 Fair queueing

The problem of fairly allocating service to a server, given requests from multiple clients, is isomorphic to a queueing theory problem. Instead of fairly scheduling variable-sized packets on multiple links onto one fixed-bandwidth outgoing link, the scheduler must schedule variable-workload requests from multiple clients on one fixed-capacity server.

In queueing theory, optimal scheduling for any centralized scheme is achieved by approximating Generalized Processor Sharing (GPS). Routers themselves cannot implement GPS directly, as this would involve simultaneously servicing multiple queues at once, and interleaving output bit-by-bit. A protocol called Weighted Fair Queueing[12] is a work-conserving scheduling algorithm that approximates GPS, with a delay bound proven to be within one packet transmission time of that which would be provided by GPS.

This work was expanded in [8], in a modification known as Worst-Case Fair Weighted Fair Queueing (WF^2Q), which additionally guarantees that packets will not be serviced more than one packet size *before* GPS would schedule them. This was further expanded in [7]; DFQ uses their virtual clock model for its implementation of weighted fair queueing.

The hierarchical aspects of [7] are also notable, as they examine the implications

of violating some assumptions of standard weighted fair queueing – aspects which DFQ's subnet decomposition (detailed in section 3.2) also violates. However, due to a greater focus on running time performance, they draw different conclusions.

Ours is not the first work to consider using weighted fair queueing for the problem of scheduling load on back-end servers; [9] is notable for having a similar problem space. However, they do not attempt to use it to solve any of the problems imposed by malicious attackers.

2.2 DDoS prevention

Link-level DDoS prevention is a well-studied topic. Solutions range from server-side solutions, either via detecting and blocking traffic[4, 10, 20] or over-provisioning the link[26], to edge-network solutions [23, 21]. However, these kinds of solutions do not attempt to detect or address application-level attacks.

Though application-level DDoS prevention is not as well-studied, ours is by no means the first system to tackle the problem. [32] uses "trust tickets" encoded in HTTP cookies to de-prioritize requests from malicious clients, but the exact method of determining who the malicious clients are is left under-specified. [37] and [27] attempt to detect attackers automatically, using semi-Markov Models and statistical analysis respectively, both of which can be defeated by a sufficiently dedicated attacker, and do not address variable request workloads. Several systems make the attackers pay in some currency, be it CPU or memory[1, 5, 6, 13, 14, 17, 19, 35], or bandwidth[34].

Chapter 3

System design

DFQ is composed of a number of unmodified HTTP servers for a replicated service; each server sits behind a front-end. Clients can connect to arbitrary front-ends, as well as arbitrary *numbers* of proxies (Figure 3-1). In order to maintain fairness, each front-end runs a modified version of weighted fair queueing, which ensures that the aggregate over all servers is fair, and copes with proxies and IP spoofing.

3.1 Distributed fair queueing

While weighted fair queueing has many favorable properties, by itself it is insufficient to counter DDoS attacks. Specifically, being fair at a single server is not sufficient; the aggregate of *all* servers must be fair. To do this, we must communicate service

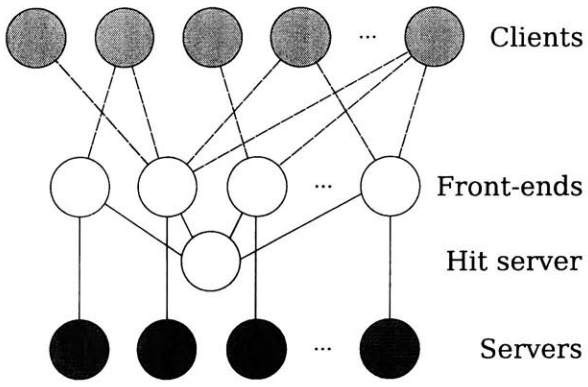

Figure 3-1: Overall design of DFQ

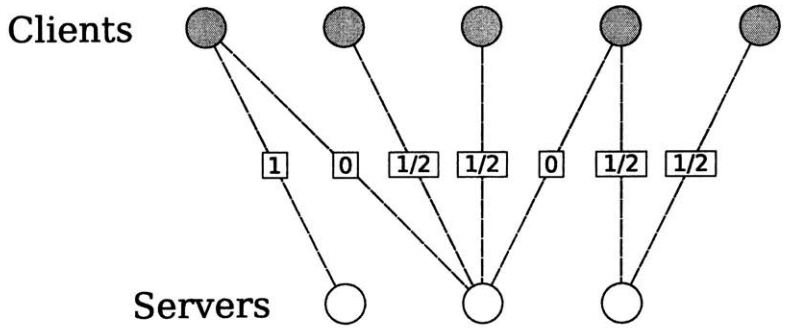

Figure 3-2: An example of a bipartite connectivity graph, with 5 clients and 3 servers. The weightings (service rates) which maximize the minimum client allocation for this particular set of connections are given.

information between front-ends.

Each front-end maintains a connection to a "hitserver," whose sole purpose is to re-distribute information about new requests to all front-ends. Each front-end reports to the hitserver when a client makes a new request, when a new request is forwarded to the back-end server, or when a request is completed. The hitserver, in turn, relays this information to every other front-end in the system.

In the context of the distributed system, we define a "fair system" as one that maximizes the minimum service allocation, over all possible service allocations. We assume that clients connect to whichever server or servers they wish, and that the servers are incapable of adjusting the connections once made. Put differently, the system takes as input a bipartite graph of clients and servers, and must choose edge weights (service rates) so as to maximize the total weight that the minimum rate client gets (see Figures 3-2, 3-3). Additionally, we wish the servers to be *work-conserving*, meaning that they are only idle if they have no requests to serve. Thus, servers also have the additional constraint that the sum of their edge weights must equal one.

To accomplish this goal, we must charge clients for service which they have received at other servers. Standard WF^{2}Q+, from [7], defines simulated times at which packets would begin and end service, were the packets served in a bit-by-bit round robin. It then schedules the packets in the order in which they would complete service under GPS. For every request which is served remotely, we simulate having served it locally by delaying the expected start and finish times of the top-most request in the

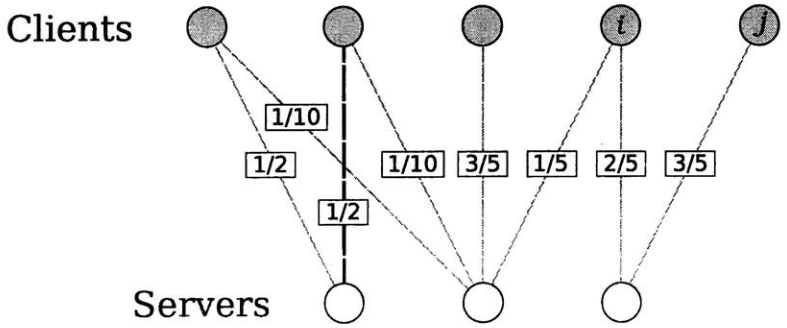

Figure 3-3: The same connectivity graph as Figure 3-2, with one edge added. Note that all clients now receive the same total allocation.

S	The set of all servers
C_s	The set of all clients with a connection to server s
$\alpha_{c,s}$	The fraction of time that client c is eligible for local service at server s
$\beta_{c,s}$	The fraction of time that server s spends serving client c
R_c	The total rate which client c is receiving service; $R_c = \sum_{i \in S} \beta_{c,s}$
T	The current virtual clock time
L	The set of local sessions
Q_i	The queue for session i i
S_i	The start time for first request in Q_i
F_i	The finish time for first request in Q_i
R_i	The local rate for queue i, as a fraction < 1; $\sum_i R_i = 1$
W_i^N	Work of the N^{th} request in Q_i

Table 3.1: Notations used in this section

SELECT-REQUEST

1 $T \leftarrow \max(T, \min_{j \in L}(S_j))$
2 $i \leftarrow$ argmin F_j where $(j \in L, S_j \leq T)$
3 $req \leftarrow$ first request in Q_i
4 **if** $|Q_i| > 1$
5 **then** $S_i \leftarrow F_i$
6 $F_i \leftarrow S_i + (W_i^2)/R_i$
7 $T \leftarrow T + W_i^1$
8 report *req* to hitserver

Figure 3-4: SELECT-REQUEST pseudocode; called by FINISHED-REQUEST or NEW-REQUEST when it must select a new request for the front-end's server to process next.

REMOTE-SELECT-REQUEST(req)

1 $i \leftarrow req.sessionid$
2 **if** $i \in L$
3 **then** $S_i \leftarrow S_i + (req.work\,/R_i)$
4 $F_i \leftarrow F_i + (req.work\,/R_i)$

Figure 3-5: REMOTE-SELECT-REQUEST pseudocode; called when the front-end is notified by the hitserver that a request has been selected by one of the other front-ends.

appropriate queue by the amount of work done remotely.

Figure 3-4 shows the pseudocode for selecting which request to service next. This code is identical to the WF^{2}Q+ code from [7], with the sole addition of notifying the hitserver of the algorithm's choice (line 8). In standard WF^{2}Q+, R_i is known as the "guaranteed rate" for a link, and an immutable property of the link. In DFQ, the rate is variable, and determined by the number of simultaneous connections from the subnet; section 3.2 discusses this in more detail.

Figure 3-5 lists the pseudocode for charging local sessions for remote service. When a remote front-end selects a request, it notifies the hitserver (line 8 of Figure 3-4). The hitserver, in turn, relays this information to all other front-ends in the system, which call REMOTE-SELECT-REQUEST. If the request in question was for a queue which also exists locally, then the starting (line 3) and finishing (line 4) times are

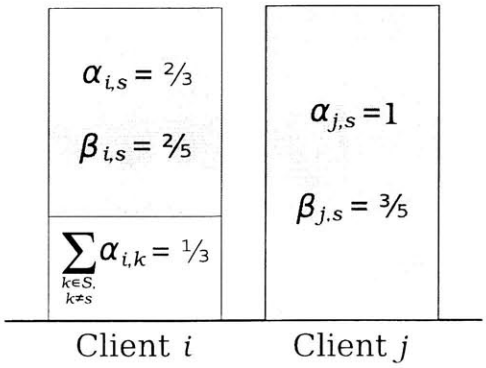

Figure 3-6: Pictorial representation of time that queues are eligible for local service, for the rightmost server in Figure 3-3.

delayed by and amount equal to the time that the request would have taken to serve locally under GPS. Note that R_i is the same across all servers for any i (see section 3.2), guaranteeing that the local delay is the same length as it was remotely.

This delay ensures that the rate at which a queue is delayed is proportional to the rate at which it is receiving service at all other servers, as shown in Figure 3-6. Thus, the system holds the following properties:

$$\alpha_{i,s} = \frac{\beta_{i,s}}{R_i} \tag{3.1}$$

$$\beta_{i,s} = \frac{\alpha_{i,s}}{\sum_{p \in C_s} \alpha_{p,s}} \tag{3.2}$$

A third property of the system is that delays are allowed to accrue faster than the rate at which the local clock ticks. Doing so, however, implies that the queue will not receive local service. Conversely, if a queue is receiving local service, the sum of the local and remote service delays must be less than all queues which do *not* receive local service:

$$\min_{\substack{p \in C_s, \\ \alpha_{p,s}=0}} \left(\sum_{q \in S} \alpha_{p,q} \right) > \max_{\substack{p \in C_s, \\ \alpha_{p,s}>0}} \left(\sum_{q \in S} \alpha_{p,q} \right) \tag{3.3}$$

In order to prove that the above scheme maximizes the minimum amount of service, as constrained by the connections which are already in place, we must prove

that for all i, R_i cannot be increased without reducing R_j for some j which shares a server s with i, where $R_j \leq R_i$. That is, we wish to prove it impossible to find:

$$\begin{aligned} R_j &> R_i \\ \beta_{j,s} &> 0 \end{aligned}$$

That is, a client j which gets more service than i, but who has non-zero service from server s which could be passed to j. From equation 3.1, assuming $\alpha_{j,s} > 0$, then:

$$\begin{aligned} R_j &= \frac{\beta_{j,s}}{\alpha_{j,s}} \\ &= \frac{\frac{\alpha_{j,s}}{\sum_{p \in C_s} \alpha_{p,s}}}{\alpha_{j,s}} \\ &= \frac{1}{\sum_{p \in C_s} \alpha_{p,s}} \end{aligned}$$

... which is independent of j. That is, all clients receiving non-zero service from a server s are guaranteed to have the same global service rate. This proves impossible that $R_j > R_i$ if $\beta_{j,s} > 0$ and $\beta_{i,s} > 0$. Due to equation 3.3, is not possible that $R_j > R_i$ if $\beta_{j,s} > 0$ and $\beta_{i,s} = 0$. Together, these are sufficient to prove the system is, in aggregate, max min fair.

3.2 Subnet-level aggregation

DFQ is also resistant to IP spoofing, subject to the constraints described in section 1.1. In the case of TCP, the three-way handshake serves to verify that the client is capable of receiving packets addressed to the IP address that it originally claimed to be. Note that this does *not* entirely prevent spoofing — if the spoofed IP address is on the same physical link as the real IP address, a careful attacker will be able to observe the response packet anyways. Thus, their spoofing is limited to IP addresses

on the same physical link.

By assuming that no physical link layer will contain more than 255 distinct hosts (a "Class C" network), DFQ can remove the advantage of spoofing. Instead of aggregating sessions by originating IP address, we instead aggregate them by originating class C subnet. Thus, a host which establishes two sessions, from two IP addresses on the same subnet, gains nothing over a host which establishes two sessions from the same computer; in both cases, the bandwidth allocated to them by the front-end is the same.

This scheme is in some ways similar to hierarchical fair queueing, as described in [7]. Much like in H-WF^{2}Q+, the addition of a new queue may drastically change the allocation of rates in pre-existing queues. This means that the relative finish times of existing queues may be affected by future packet arrivals, which complicates the algorithm. This violates the simplifying assumption that allows WF^{2}Q to be implemented in $\mathcal{O}(\log N)$ time.

H-WF^{2}Q+ solves this problem, as its name implies, using a hierarchical arrangement of WF^{2}Q+ servers. DFQ is less concerned with efficiency, since the processing times that it deals with are much longer. As such, we have opted to incur the $\mathcal{O}(N)$ overhead of updating the start and finish time of all pre-existing queues when the situation demands it — an option discussed in [7], but dismissed because of performance implications. DFQ does this recalculation whenever a queue is added or removed. We do enforce that a single cookie may only be used at a single subnet, so as to not have to possibly recalculate subnet allocation on every new request. This is sufficient to deal with the case of dynamic proxies, such as AOL uses, which often redirects each request through a random AOL proxy.

Figure 3-7 shows the code for NEW-REQUEST, which is run when a front-end receives a new request from a client. Lines 1–6 are according to standard WF^{2}Q+; the request is added to the appropriate queue, and if the queue is new, the start and finish times of the first packet in the queue are updated. Lines 7–13 step through each local session which existed before. We calculate the time remaining after the current virtual clock time for each session, and rescale that for the new rate, adjusting start

```
NEW-REQUEST(req)
 1  i ← req.sessionid
 2  add i to L
 3  push Q_i, req
 4  if Q_i = req
 5     then S_i ← max(T, F_i)
 6          F_i ← S_i + req.work / R_i
 7          for j ∈ L, j ≠ i
 8               do if S_j < T
 9                    then left ← (F_j − T) * (R_j as if req had not arrived)
10                         F_j ← T + left / R_j
11                    else delay ← (S_j − T) * (R_j as if req had not arrived)
12                         S_j ← T + delay / R_j
13                         F_j ← S_j + (W_j^1)/R_j
14          T ← max(T, min_{j∈L}(S_j))
15  if no request currently running on back-end server
16     then SELECT-REQUEST
```

Figure 3-7: NEW-REQUEST pseudocode; called whenever a client sends a request to the front-end.

and finish times as necessary.

A similar process is undergone for when a client finishes a request, as it could affect the rates of other clients on the same subnet. Figure 3-8 lists the pseudocode for FINISHED-REQUEST, when a request finishes – either when a back-end server services it, or the client voluntarily closes the connection. This code contains a nearly identical loop to NEW-REQUEST, on lines 5–11, which similarly adjusts the start and finish times for the new rates.

3.3 Proxy prioritization

As discussed in section 1.2, the situation is further complicated by the presence of proxy servers. These appear to be single IP addresses which source many times the amount of traffic as any other client. DFQ addresses this problem by using historical information about the request rates of each class C subnet. Such a historical traffic record could be generated by examining past webserver logs from periods when the

```
FINISHED-REQUEST(req)
 1  i ← req.sessionid
 2  remove req from Q_i
 3  if Q_i = ∅
 4     then delete i from L
 5          for j ∈ L, j ≠ i
 6               do if S_j < T
 7                     then left ← (F_j − T) * (R_j as if req had not finished)
 8                          F_j ← T + left / R_j
 9                     else delay ← (S_j − T) * (R_j as if req had not finished)
10                          S_j ← T + delay / R_j
11                          F_j ← S_j + (W_j^1)/R_j
12  if there are non-empty queues
13     then SELECT-REQUEST
14     else T ← 0
15          for j ∈ all sessions which have existed
16               do S_j ← 0
17                  F_j ← 0
```

Figure 3-8: FINISHED-REQUEST pseudocode; called when the front-end's server finishes the request it was given.

```
NEW-REQUEST(req)
1  i ← req.sessionid
2  add i to M
3  push Q_i, req

NEW-REMOTE-REQUEST(req)
1  NEW-REQUEST(req)
```

```
FINISHED-REQUEST(req)
1  i ← req.sessionid
2  remove req from Q_i
3  if Q_i = ∅
4      then delete i from M

FINISHED-REMOTE-REQUEST(req)
1  FINISHED-REQUEST(req)
```

Figure 3-9: Pseudocode needed to maintain M, the set of all sessions in the system. This code is run alongside that in Figures 3-7 and 3-8.

service was not under attack, and regularly updated to detect and adjust for consistent over- or under-allocation. However, the exact method of gathering this historical information, as well as decisions about how and when to update it, are complex and beyond the scope of this research; for our purposes, we will simply assume that such a profile is provided as input to the algorithm.

We then use this historical profile in order to inform the weightings provided to the Weighted Fair Queueing implementation. Specifically, the historical request rate for the subnet is divided by the number of sessions (in the set of all servers) originating from that subnet. This number is additionally capped by the historical average request rate over all subnets. Figure 3-9 shows the pseudocode needed to maintain a set M of all active sessions in the system. Using this, Figure 3-10 iterates through all of the sessions in the system, and determines the number of sessions which originate from the same subnet as the request. Because each front-end has total information, a result of the RATE function is thus identical across all servers, for any given session.

This method ensures that a subnet which is weighted at ten times the normal traffic rate will allow ten concurrent sessions to connect without penalizing them. Should the 11^{th} connect, however, all sessions from that subnet will receive only $\frac{10}{11}$ of normal rate. Additionally, if but a single session is active from the subnet, it will receive only the standard traffic rate, not ten times the normal rate.

```
RATE(req)
1  subnet ← first 3 parts of req.ip
2  concurrent ← 0
3  for i ∈ M
4        do j ← first request in Q_i
5           if first 3 parts of j.ip = subnet
6              then concurrent ← concurrent + 1
7  rate ← global average request rate, from historical profile
8  subnet-rate ← (average request rate of subnet, from historical profile)/concurrent
9  return min(rate, subnet-rate)
```

Figure 3-10: RATE pseudocode; this code defines R_i, based on the first request in Q_i, which is passed as input.

Should an attacker have control of a host behind a proxy or NAT, it is possible for them to create extra sessions, and in so doing take over any unused bandwidth which the proxy has. That is, in the case of the proxy in the example above, if only five sessions were active, an attacker could trivially create an additional five sessions — and in so doing, procure five times the bandwidth that they would be able to procure from one host in most other subnets. However, this attack does not damage other clients any more than if the subnet were completely filled with non-malicious attackers.

Chapter 4

Implementation and validation

DFQ was implemented as a number of modules in Perl, comprising about 4000 lines of code and API documentation. The scheduling front-end was implemented using a single-threaded event-driven model, with a pluggable scheduler. Multiple different scheduling implementations were tried before arriving at the scheme described in chapter 3. For instance, we attempted a number of variations of deficit round robin queueing; though this leads to a more clear model for charging local clients for remote service, it does not have nearly as desirable scheduling bounds, and is harder to reason about correctness.

Figure 4-1 presents a test layout. We ran a 125 second test, during which clients 1 and 2 were continually submitting requests. Client 3 was active in the time between 25–75 seconds, and client 4 was active between 50–100 seconds. Additionally, clients 2 and 4 originate from the same IP subnet. For the purposes of testing, the back-end HTTP server was modeled as a single-threaded server which, instead of actively doing

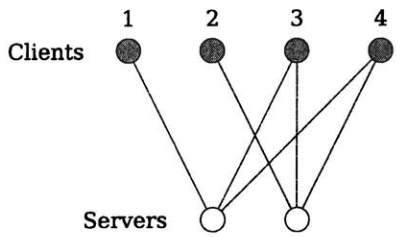

Figure 4-1: Connectivity graph for testing; clients are shown at the top, servers at the bottom.

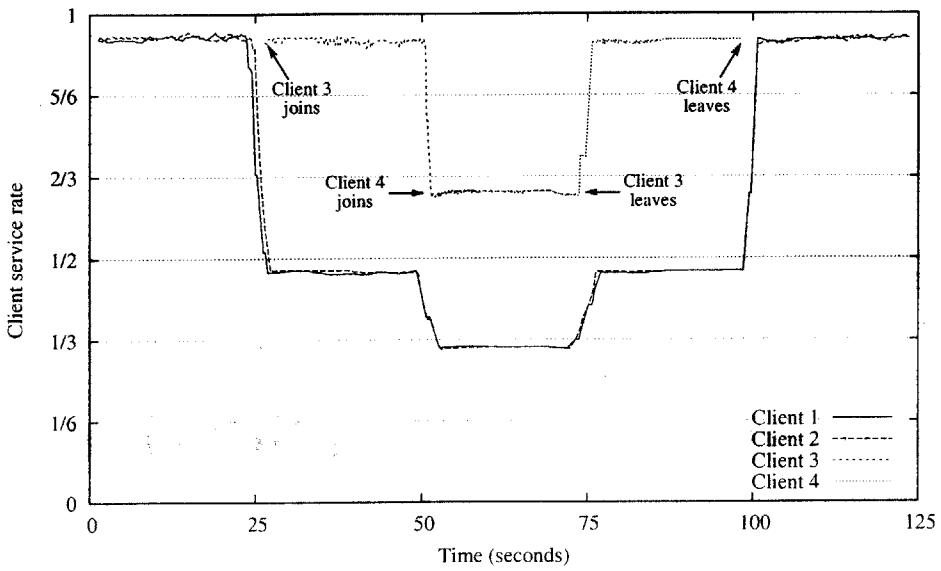

Figure 4-2: Results of a FIFO scheduler.

work, contained statements to pause for an appropriate number of seconds.

For comparison, the results using a simple FIFO scheduler are seen in Figure 4-2. This scheduler does not communicate any information between front-ends, nor does it attempt to achieve fairness between the two servers. Note that client 3 receives twice the service as clients 1 and 2 when it joins, due to being connected to two servers, as does client 4 when it joins later. For the middle portion of the test, the subnet which clients 2 and 4 belong to is receiving half of the service in the system – and, indeed, would continue to do so even if the clients were from the exact same IP address. This is clearly not fair.

We note a 5% performance loss for all requests, observable by the rates in Figure 4-2 always being slightly below the predicted values. Given that the system was running at a total aggregate rate of eight requests per second, this equates to an overhead of 0.00625 seconds per request in this test. Given that the implementation is untuned, and written in an interpreted language not particularly known for its speed, this is quite an acceptable overhead.

The same situation, with DFQ instead, is shown in Figure 4-3. Client 3 receives no more service than the other two clients in the system when it connects. When client 4 joins, it divides the service previously allocated to client 2. This property is

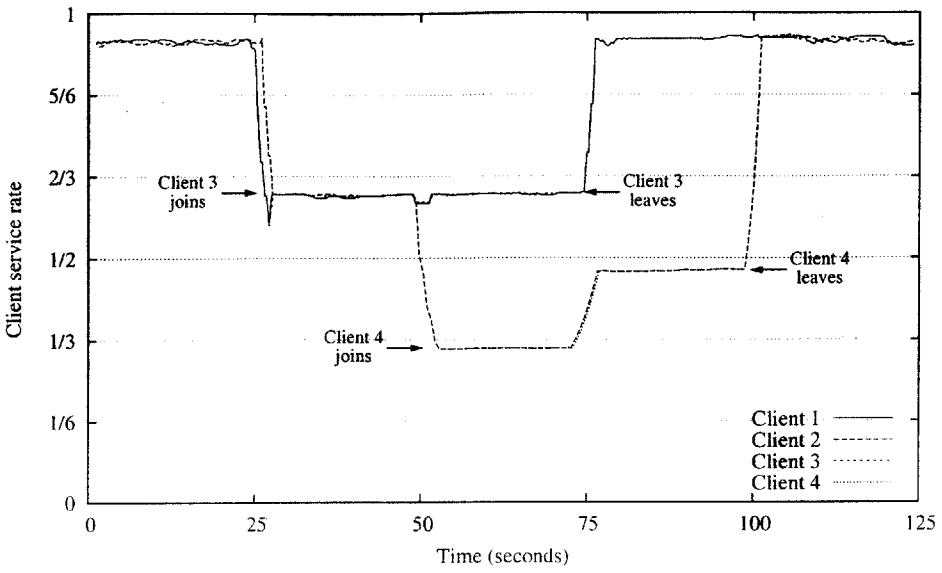

Figure 4-3: Results of DFQ scheduling

maintained when client 3 leaves the system at 75 seconds. All of this true, despite clients 3 and 4 making requests of both servers. Figures 4-4 and 4-5 show the service rates at server 1 and 2 respectively. Note that while client 4 makes requests to both servers, it only receives service from server 2. While odd, this is according to the original specifications laid out in section 1.3.

We also note a 5% performance loss, similar to the degradation experienced by the FIFO scheduler. That is, the additional complexity of the system has a negligible impact on the efficiency of the system.

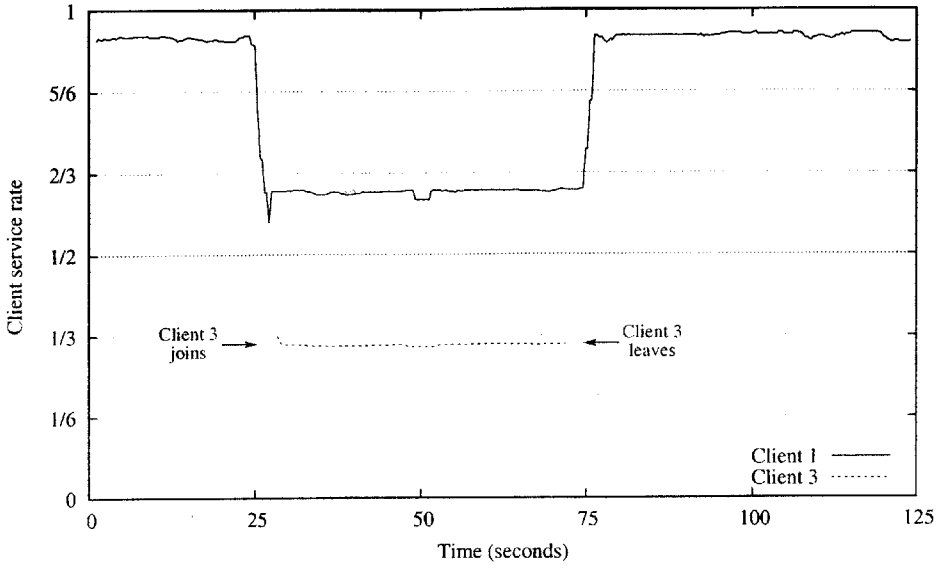

Figure 4-4: Service under DFQ at server 1

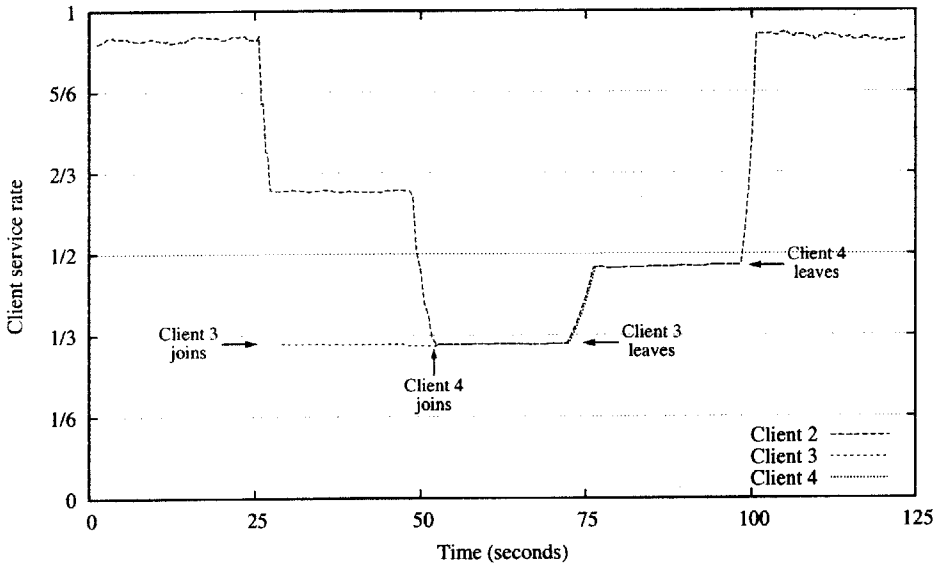

Figure 4-5: Service under DFQ at server 2

Chapter 5

Conclusions

We have presented the algorithm and implementation of a distributed, proxy-aware scheduling front-end, dubbed DFQ (Distributed Fair Queueing). It is capable of fairly scheduling clients which open connections to multiple replicated servers, even in the face of malicious enemies capable of spoofing IP addresses. Given these properties, it is capable of defending a distributed set of replicated HTTP servers from application-level DDoS attacks.

As not much work has been found regarding distributed fair scheduling, we find the results to be interesting. However, their utility is limited by the number of slightly odd properties which evolve; namely, that no single request has a bounded service interval if requests from the same client are already being served elsewhere.

This property, however, is not detrimental if the requests are idempotent. Thus, a possible application is any in which multiple servers are capable of servicing the request, and the client has many more requests to make than there are servers. Specifically, in the case of package management servers, which are responsible for distributing up-to-date packages to many clients at once, every server contains all of the packages. A client capable of connecting to multiple servers to request a large set of packages can be assured that they will never overload a single server, and, indeed, is guaranteed no more than their fair share. Protecting such servers from DDoS attack is particularly important, as such an attack could lengthen the vulnerability of other systems by denying them access to security patches.

In conclusion, distributed fair scheduling is an interesting topic, which merits further research, particularly as it pertains to DDoS prevention. DFQ proves the feasibility of implementing such a system, as well as uncovering several areas for further research.

Bibliography

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *NDSS*, 2003.

[2] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck characterization of dynamic web site benchmarks. Technical Report TR02-388, Department of Computer Science, Rice University, February 2002.

[3] T. Anderson, T. Roscoe, and D. Wetherall. Preventing internet denial-of-service with capabilities. In *HotNets*, November 2003.

[4] Arbor Networks, Inc. `http://www.arbornetworks.com`.

[5] T. Aura, P. Nikander, and J. Leiwo. DoS-resistant authentication with client puzzles. In *Intl. Wkshp. on Security Prots.*, 2000.

[6] A. Back. Hashcash. `http://www.cypherspace.org/adam/hashcash/`.

[7] J. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ ACM Transactions on Networking*, 5(5):675–689, 1997.

[8] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996.

[9] J. M. Blanquer and B. Ozden. Fair queuing for aggregated multiple links. In *Proc. ACM SIGCOMM*, pages 189–198, 2001.

[10] Cisco Guard, Cisco Systems, Inc. `http://www.cisco.com/`.

[11] Criminal Complaint: USA v. Ashley, Hall, Schictel, Roby, and Walker, August 2004. `http://www.reverse.net/operationcyberslam.pdf`.

[12] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM Press.

[13] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *CRYPTO*, 2003.

[14] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

[15] V. D. Gligor. Guaranteeing access in spite of distributed service-flooding attacks. In *Intl. Wkshp. on Security Prots.*, 2003.

[16] M. Handley. Internet architecture WG: DoS-resistant Internet subgroup report, 2005. `http://www.communicationsresearch.net/object/download/1543/doc/mjh-dos-summary.pdf`.

[17] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS*, 1999.

[18] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *USENIX NSDI*, May 2005.

[19] D. Mankins, R. Krishnan, C. Boyd, J. Zao, and M. Frentz. Mitigating distributed denial of service attacks with dynamic resource pricing. In *Proceedings of the IEEE ACSAC*, December 2001.

[20] Mazu Networks, Inc. `http://mazunetworks.com`.

[21] P. Mirkovic and P. Reiher. D-WARD: A source-end defense against flooding deniale of service attacks. In *IEEE Transactions on Dependable and Secure Computing*, volume 2, pages 216–232, July 2005.

[22] 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 *ACM CCS*, October 2003.

[23] C. Papadopoulos, R. Lindell, J. Mehringer, A. Hussain, and R. Govindan. COSSACK: Coordinated suppression of simultaneous attacks. In *Proceeding of Discex III*, April 2003.

[24] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the single node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[25] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994.

[26] Prolexic Technologies, Inc. `http://www.prolexic.com`.

[27] S. Ranjan, R. Swaminathan, M. Uysal, and E. Knightly. DDoS-resilient scheduling to counter application layer attacks under imperfect detection. In *INFOCOM*, pages 1–13, April 2006.

[28] E. Ratliff. The zombie hunters. *The New Yorker*, October 10 2005.

[29] The Register. East European gangs in online protection racket, November 2003. `http://www.theregister.co.uk/2003/11/12/east_european_gangs_in_online/`.

[30] SecurityFocus. FBI busts alleged DDoS mafia, August 2004. `http://www.securityfocus.com/news/9411`.

[31] Stupid Google virus/spyware CAPTCHA page. `http://spyblog.org.uk/2005/06/stupid_google_virusspyware_cap.html`.

[32] M. Srivatsa, A. Iyengar, J. Yin, and L. Liu. A middleware system for protecting against application level denial of service attacks. In *Proceedings of 7th ACM/IFIP/USENIX International Middleware Conference*, 2006.

[33] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. In *Communications of the ACM*, volume 47, pages 56–60, February 2004.

[34] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *ACM SIGCOMM*, volume 36, pages 303–314, October 2006.

[35] X. Wang and M. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *IEEE Symp. on Security and Privacy*, May 2003.

[36] Network World. Extortion via DDoS on the rise, May 2005. `http://www.networkworld.com/news/2005/051605-ddos-extortion.html`.

[37] Y. Xie and S. Yu. A novel model for detecting application layer DDoS attacks. In *Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences*, volume 2, pages 56–63, 2006.

[38] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *IEEE Symp. on Security and Privacy*, May 2004.

[39] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, August 2005.

4807-69