DNS Congestion Control in Adversarial Settings

Huayi Duan
ETH Zürich
Switzerland

Jihye Kim
ETH Zürich
Switzerland

Marc Wyss
ETH Zürich
Switzerland

Adrian Perrig
ETH Zürich
Switzerland

Abstract

We instigate the study of *adversarial congestion* in the context of the Domain Name System (DNS). By strategically choking inter-server channels, this new type of DoS attack can disrupt a large user group's access to target DNS servers at a low cost. In reminiscence of classic network congestion control, we propose a DNS congestion control (DCC) framework as a fundamental yet practical mitigation measure for such attacks. With an optimized fair-queuing message scheduler, DCC ensures benign clients fair access to inter-server channels regardless of an attacker's behavior; with a set of extensible anomaly detection and signaling mechanisms, it minimizes collateral damage to innocuous clients. We architect DCC in a non-invasive style so that it can readily augment existing DNS servers. Our prototype evaluation demonstrates that DCC effectively mitigates adversarial congestion while incurring minor performance overheads.

CCS Concepts: • **Networks** → **Naming and addressing**; **Denial-of-service attacks**; **Public Internet**; • **Security and privacy** → **Denial-of-service attacks**.

Keywords: Name Resolution, DNS, DoS Attacks, Rate Limiting, Congestion Control, Fair Queuing Algorithm

ACM Reference Format:
Huayi Duan, Jihye Kim, Marc Wyss, and Adrian Perrig. 2024. DNS Congestion Control in Adversarial Settings. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA*. ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3694715.3695982

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11.
https://doi.org/10.1145/3694715.3695982

1 Introduction

The Domain Name System (DNS) plays a crucial role in the modern Internet. When DNS becomes unavailable, numerous online systems relying on it would be impacted. Unsurprisingly, Denial-of-service (DoS) attacks on DNS and the defenses against them have been in an arms race for decades. The recently discovered DNS amplification vulnerabilities [7, 14, 22, 41, 44, 64] give a substantial edge to an adversary, as they allow every single client request to trigger disproportionately many queries to be generated and processed by servers participating in the name resolution.

While existing DoS attacks exploit various *protocol-* or *implementation-level* vulnerabilities, we investigate attack surfaces on the *architecture level*. The DNS resolution architecture does not follow a simple client-server model but rather resembles a graph, where named data (i.e., resource records) flows from sources (i.e., authoritative nameservers) to sinks (i.e., end hosts), passing through and being cached at one or more intermediate nodes (i.e., recursive resolvers) which serve as both clients and servers.

With this observation, we devise a new type of attacks where an adversary, rather than overloading DNS servers themselves or the underlying networks, strategically induces congestion at *logical* inter-server channels and thereby impedes a target user group's access to the whole or part of the DNS namespace. Since many DNS servers implement rate limiting to throttle excessive messages they receive from or send to any IP address (or prefix), those inter-server channels usually come with limited capacities. Therefore, an attacker can create *adversarial congestion* with considerably fewer resources than conventional DoS attacks, especially when it also leverages application-layer amplification.

We have validated different forms of adversarial congestion using our own testbed and also carefully analyzed them on popular open resolvers. The results are concerning: even relatively low attack request rates—slightly above the target inter-server channel's capacity, or substantially lower in case of amplification—are sufficient to disrupt benign users' access to our test domains through the affected resolvers.

To mitigate this new type of attacks, we develop a *DNS congestion control* (DCC) framework inspired by classic network congestion control. At the heart of DCC is a novel *fair-queuing* scheduler, which allocates an inter-server channel

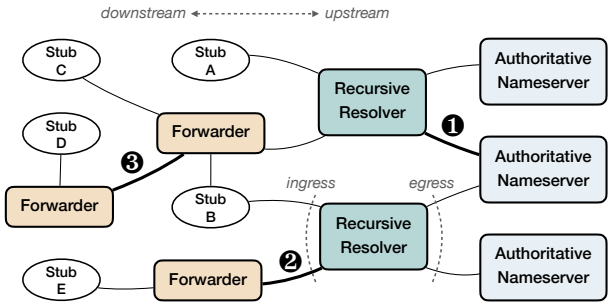


Figure 1. Real-world DNS resolution architecture. Dashed lines illustrate the directional terms used in this paper.

among the downstream resolver's clients. This establishes a worst-case guarantee for benign clients that they can always access the channel based on their fair shares, regardless of an attacker's behavior. Designing such a scheduler poses unique challenges, as DNS resolvers are fundamentally different from L2/L3 devices. With a thorough analysis of the scheduling problem, we develop an optimized scheduler called MOPI-FQ that achieves all desired fairness and performance properties.

In addition, DCC provides an *anomaly monitoring and policing* mechanism to counter attackers that exploit specially crafted query patterns to gain advantage over benign clients. Policing a resolver's suspicious client that keeps sending anomalous requests will cause collateral damage if the client itself is a resolver, and this creates another DoS attack vector. DCC is further armed with an *in-band signaling* mechanism that enables resolvers to exert fine-grained control along a resolution path while minimizing collateral damage to innocuous clients. The signaling mechanism also allows congestion information to be propagated downwards a resolution path for troubleshooting and performance tuning. By orchestrating these complementary mechanisms, DCC can offer comprehensive protection and performance benefits.

We have developed a prototype of DCC [1] and demonstrated its effectiveness in thwarting adversarial congestion in different scenarios. Our evaluation also confirms DCC's time and space efficiency through the minor performance overhead introduced to the DNS resolver that it augments.

2 Motivation

We start this section with an overview of real-world DNS resolution (Section 2.1), followed by a discussion of common DoS defenses deployed for DNS (Section 2.2) and the introduction of adversarial congestion (Section 2.3).

2.1 DNS Resolution Architecture

DNS is an Internet-scale distributed system comprising two general types of servers. The *authoritative nameservers* host resource records specified by domain owners. The *recursive resolvers* perform name resolution on behalf of clients (aka *stub resolvers*) by retrieving and serving those records.

Textbook examples of DNS often depict a recursive resolver that, upon receiving a client's name lookup request, iteratively finds the answer from authoritative nameservers. The reality is however more involved. Many recursive resolvers are *forwarders*, which do not conduct iterative resolution by themselves but simply forward DNS queries to upstream resolvers. Forwarders are pervasive and commonly integrated into residential routers, enterprise network gateways, and wireless access points [42, 55]; some of them may even remain hidden [46]. A forwarder's upstream server can also be another forwarder. Figure 1 depicts this complex architecture with different types of servers.

We define a DNS *resolution path* as an ordered list of entities, including an end host that initiates a request, possibly several forwarders, one egress recursive resolver, and one authoritative nameserver that provides relevant answers. A resolution path can terminate at any resolver that answers a query directly from its cache.

Recursive resolvers are traditionally operated by ISPs to serve their own customers. The modern DNS ecosystem features a growing number of public resolvers, which offer (typically free) resolution services to any Internet user. There are also semi-open resolvers operated by cloud providers and configured for their users by default. The resolver systems of large operators often contain multiple internal layers of caching and load balancing [9, 51].

Remark on Terminology. Hereafter, we refer to recursive resolvers and forwarders collectively as resolvers, distinguishing them from end hosts and authoritative servers. A resolver has a *client-facing side* that processes *requests* and returns *responses*, as well as a *server-facing side* that sends *queries* and receives *answers*. When using the common term QPS (queries per second), we refer to either client requests or resolver queries depending on the context.

2.2 DoS Defenses for DNS

DNS is one of the main enablers of reflection attacks. Meanwhile, it is also among the highest-value targets of DoS attacks. Since DNS servers are normally compute-bound [54], they can be overloaded by even moderate query volumes without their network bandwidth being saturated. In other words, network-layer DoS defenses for volumetric attacks are usually insufficient to handle application-layer DoS attacks on DNS. This becomes more concerning with the recent disclosure of various DNS amplification vulnerabilities [7, 14, 22, 39, 41, 44]: they allow a single malicious client request to elicit a large number of resolver queries, hence neutralizing common availability-enhancing techniques for DNS including over-provisioning and anycast [43, 54].

DNS operators often apply customized rules to filter suspicious queries. For example, Akamai DNS penalizes queries

that would result in NXDOMAIN (non-existing domain) answers and that come from spoofed addresses [54]. Bushart et al. recently proposed a DDoS defense system, which blocks a client if its query rate or spikes exceed predefined limits learned from historical data [15]. In general, filtering methods are subject to false positives and evasion by sophisticated and persistent attackers. They can also cause collateral damage if the client being policed is itself a resolver.

As a common DoS defense deployed for many Internet services, rate limiting (RL) caps the amount of load any (malicious) host can place on a server. In the case of DNS, RL comes in two generic forms. Ingress RL limits the requests from or responses to clients and applies to recursive and authoritative servers. Egress RL limits the outgoing queries to upstream servers and applies only to resolvers. With RL in place, an attacker must commandeer a sufficient number of network hosts to successfully overwhelm a target server.

2.2.1 RL for DNS. To understand how DNS servers implement RL in practice, we have conducted a measurement study on 45 popular public resolvers using our own domains and authoritative servers. Our measurements use different query patterns that can be exploited by real attackers: pseudo-random query names that elicit positive (NOERROR) or negative (NXDOMAIN) answers, and predefined query names that trigger two forms of compositional amplification [22], which allows us to reduce probing traffic and bypass a resolver's ingress limit in case it has a higher egress limit. We restrict the quantity and TTL of resource records, as well as probing duration, to avoid stressing the measured resolvers. More details can be found in Appendix A. Our measurement results are plotted in Figure 2.

For ingress RL, we find that some resolvers vary their rate limits for different IP prefixes, and we report the minimum values measured across our three geographically distributed probes. In addition, if we cannot decide a resolver's ingress rate limit up to a probing request rate of 5000 QPS, we mark it as uncertain. Of the 45 resolvers, over one third have an ingress rate limit below 100 QPS, while around 40 have a limit below 1500 QPS, regardless of response type. A few resolvers enforce lower ingress rate limits for NXDOMAIN responses, likely as a countermeasure against the pseudo-random subdomain or Water Torture attack [8].

For egress RL, we count a resolver as uncertain if we cannot decide its egress rate limit even when our probing rate reaches its ingress rate limit or 1000 QPS, whichever is lower. This is the case for around half of the measured resolvers. In those more certain cases, we observe clearer signs that the estimated egress QPS of the resolvers stop increasing at some point, which suggest possible egress RL, and the limits mostly lie between 100 and 1500 QPS.

We also observe that resolvers take different actions when the rate limit on a client or server is exceeded. Many appear to lower the limit, causing fluctuations in the measured QPS.

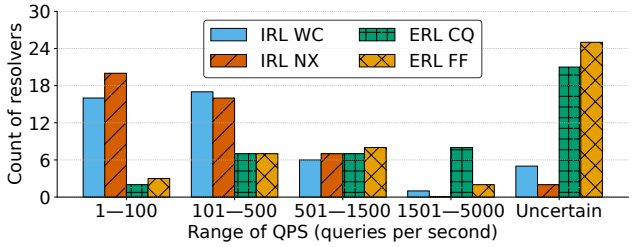


Figure 2. Rate limits measured on 45 open resolvers (listed in Table 3). We consider both ingress limits on clients (IRL) and egress limits on servers (ERL) using different query patterns: pseudo-random names that trigger wildcard synthesis (WC) or NXDOMAIN responses (NX), and predefined names that trigger amplification in the form of CNAME chain × QMIN (CQ) or NS-based fan-out × fan-out (FF) [22].

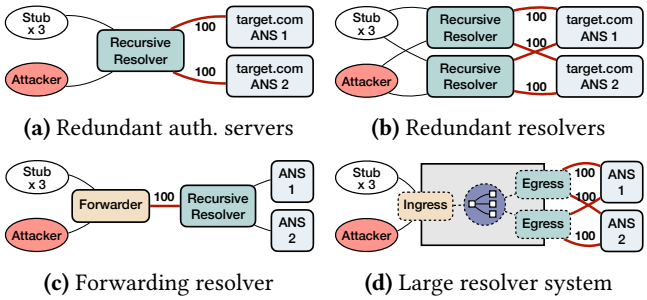
Some resolvers temporarily block our probes, while others return SERVFAIL or REFUSED responses.

2.3 Adversarial Congestion

Existing DoS attacks against DNS focus on depleting the target server's resources, exploiting vulnerabilities found in DNS protocols or implementations. We turn our attention to attack surfaces inherent in the modern DNS architecture, where a complex fabric of servers interact with each other to distribute named resource records. Our key observation is that each pair of DNS servers in this architecture forms a *logical channel*, and the congestion of such a channel can affect all clients of the downstream server. An attacker can strategically congest inter-server channels to disrupt a target user group's access to the affected domain name services.

To see the effect of such conceptual adversarial congestion, let us use the example architecture in Figure 1. If channel ❶ is congested, all its direct and indirect clients, including Stubs A, B, C and D, may lose access to the domains hosted by the affected authoritative nameserver in the middle. If channel ❷ is congested, Stub E's use of many Internet services may be disrupted, and so are any other clients of the forwarder at the bottom. This happens similarly to Stub D in case of channel ❸'s congestion.

We find that adversarial congestion can be highly disruptive from two aspects. First, inter-server channels often have limited capacities due to the RL enforced by DNS servers, as discussed earlier. This leads to an *availability dilemma*: RL is an indispensable measure to mitigate DoS attacks in general, whereas it also enables an attacker to congest a rate-limited channel at a substantially lower cost than overloading an entire server or the underlying network. Second, unlike network packets that can be forwarded along many (dynamically updating) paths, DNS messages do not have as many resolution paths to choose from. An end host or forwarder is



(a) Redundant auth. servers **(b)** Redundant resolvers

(c) Forwarding resolver **(d)** Large resolver system

Figure 3. Different DNS resolution setups used for our empirical validation of adversarial congestion.

normally configured to use a few number of fixed upstream resolvers (e.g., 2 or 3 [4]), and most domains are delegated to two authoritative servers [2]. Hence, there is no much room for DNS messages to go around a congested channel.

At the onset of adversarial congestion on a resolver's channels to upstream servers, it can still answer queries from cache for a certain period of time. As cached records expire and the resolver must send queries to those congested channels, the attack's effect will intensify with an increasing number of clients or domains affected.

To sustain congestion on the target channels, an attacker must generate queries that bypass a resolver's cache. A common technique is querying random names that elicit NXDOMAIN responses [8]. Such queries can be suppressed by a resolver that implements DNSSEC-validated cache [24], but the adoption of DNSSEC still remains low, e.g., $< 5\%$ `.com` domains are signed as of September 2024 [5]. To evade any potential filtering of NXDOMAIN queries by the resolver, the attacker can always query existing names. This requires a large number of records from the queried zone which is achievable with a single *wildcard record* [36]. Cache bypassing is especially simple if the target is an *RR channel* between two resolvers, because the attacker can freely set up and query its own DNS zones. For an *RA channel* between a resolver and an authoritative nameserver, an attack is easy to set up if any zone hosted by the nameserver contains one wildcard record, or the attacker can install such a record on the nameserver, which is the case for the popular third-party DNS hosting services today [30].

2.3.1 Validation Methodology. To assess the practicality of adversarial congestion, we simulate attacks with different DNS resolution architectures using our own resolvers and public resolvers, as depicted in Figure 3.

(a) Two authoritative servers hosting our own domains, which are queried by one recursive resolver shared by three benign clients and one malicious client (attacker).

(b) One more resolver used by the clients compared to (a).

(c) One additional forwarder that sits between the recursive resolver and clients compared to (a).

(d) Similar to (c), but the forwarder (ingress) and recursive resolver (egress) are part of a large resolver system.

The inter-server channels to be congested are highlighted in red. All of them have a capacity of 100 QPS: for RA channels, we configure ingress RL at our authoritative servers; for RR channels, we either configure ingress RL at our own recursive resolver or choose public resolvers with an ingress rate limit close to 100 QPS. All our DNS servers using BIND 9 run on cloud VMs located in different data centers of DigitalOcean; so are the clients, except that we assign them to the same data center in order to increase the possibility that their DNS requests are delivered to the same point of presence if a public resolver deploys anycast.

In our simulations with setups (a), (b) and (d), the attacker exploits amplification with the same FF query pattern used for our measurements of resolvers' egress RL (Section 2.2.1); for setup (c) the attacker exploits the WC query pattern that triggers wildcard synthesis. The three benign clients use the WC pattern in all setups to simulate legitimate requests. We give examples of our zone configuration in Appendix A.

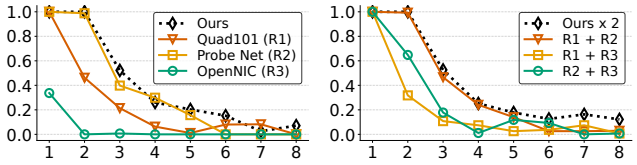
In each simulation, the attacker generates requests at a constant rate for 50 seconds and after it has started for 5 seconds, each benign client begins generating requests at the rate of 3 QPS for 30 seconds. We measure the their average request success ratio under varying attacker QPS.

2.3.2 Validation Results. Figure 4 reports the results.

For setup (a), most of the benign clients' requests already fail when the attacker sends as low as three requests per second to our resolver. This is explained by a message amplification factor (MAF) of around 50 the BIND resolver suffers from the FF pattern. That is, each attacker request elicits around 50 resolver queries to our authoritative servers. Among the three public resolvers that produce MAFs in the same ballpark, the benign clients experience a similar performance impact for the OpenNIC resolver and worse adversarial congestion effects for the other two resolvers.

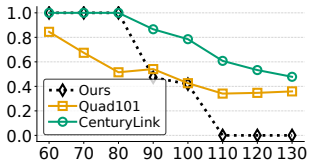
For setup (b), we observe only a slight increase in the legitimate request success ratio, which generally lies between the results for individual resolvers measured with setup (a). That is, introducing an additional resolver does not improve the situation too much. The underlying reason is that if the clients' initial requests fail due to the congestion at one resolver's RA channel, they will send roughly the same amount of retried requests to the other resolver and hence cause congestion there as well.

For setup (c) without amplification, we configure our forwarder to use three different upstream resolvers, one of them (Quad101) with an ingress RL of 60 QPS and the other two with the default 100 QPS. We find that the success ratio of legitimate request starts to drop once the attacker's QPS approximates the RR channel's capacity, and the success ratio declines as the attacker becomes increasingly aggressive.

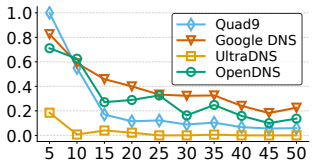


(a) Redundant auth. servers

(b) Redundant resolvers



(c) Forwarding resolver



(d) Large resolver system

Figure 4. Attack simulation results with the setups in Figure 3. The x-axes represent attacker's QPS and the y-axes represent the average request success ratio of benign clients.

For setup (d), we measure the public resolvers of four big DNS service providers. These resolver systems distribute client requests across multiple egresses, each serving as a recursive resolver that communicates directly with authoritative nameservers. We record the four resolvers' egresses and find that adversarial congestion's impact is reversely proportional to the size of a resolver system's egress set. UltraDNS dispatches queries to 4 egresses and appears to be the least resilient against our simulated attack, followed by Quad9 using 16 egresses and OpenDNS using 25 egresses. Although the Google public resolver using 60 egresses performs better than others, it is still significantly affected by the attack, resulting in a notable decrease in the success ratio of requests from benign clients.

2.3.3 Impact on DNS Infrastructures. Our attack simulation results demonstrate the feasibility and severity of adversarial congestion. The threats it poses to real-world DNS infrastructures are largely determined by the extent to which DNS servers implement RL policies. A prior study reported hundreds of thousands of authoritative nameservers implementing (ingress) response RL with a limit below 500 QPS [19]. Our own measurements also indicate that many open resolvers have ingress and egress RL in place. This is unsurprising as RL has been an essential measure to mitigate long-lasting security threats to DNS, including reflection attacks and cache poisoning. The stream of recently disclosed attack vectors are further pushing DNS vendors and operators to widen and tighten their RL policies [7, 22, 37, 64]. Hence, the threats of adversarial congestion are likely to remain, if not become an increasing concern, unless the DNS resolution architecture undergoes some radical change. Measuring RL policies adopted by DNS servers on an Internet scale is an important avenue for future research.

3 DNS Congestion Control

We develop a DNS congestion control framework (DCC) that guarantees end users fair access to domain name services, even when the servers are under severe adversarial congestion caused by sophisticated attack query patterns. In what follows, we state the problem of DNS congestion control in Section 3.1, describe our design for individual resolvers in Section 3.2, and explain how the control is extended to a full resolution path in Section 3.3. We elaborate on DCC's core FQ algorithm in Section 4.

3.1 Problem Statement

Adversarial DNS congestion exhausts inter-server channels. This necessitates proper traffic control at the *downstream* server of each susceptible channel. Discerning and blocking the culprit is not always possible: an attacker can mimic normal hosts but still inflict congestion (Section 2.3). To handle such worst-case scenarios, we must enforce fair sharing of the channel among the server's clients regardless of their nature. This idea is reminiscent of fair queuing (FQ) and in-network congestion control in general, but the case of DNS is fundamentally different from the following perspectives.

- **Fairness.** Upon receiving client requests, a resolver does not forward them in the same way as L2/L3 devices but creates its own queries to upstream servers when necessary. For each request, a resolver may generate no query at all in case of a cache hit, or potentially many queries otherwise. In general, the amount of resources consumed by requests varies, and therefore it is non-trivial to define and achieve fairness.
- **Output multiplicity.** Conventional FQ deals with one output interface. A DNS resolver, especially a recursive resolver, can have many logical output channels to upstream servers. Scheduling messages over a large number of channels with guaranteed fair use of each in a scalable manner (e.g., without creating an expensive FQ instance for each channel) is an open problem.
- **Visibility.** Network devices can apply traffic policing to end hosts based on the source and destination information retained in the packets they forward. However, DNS servers do not know a query's origin on a resolution path. If a server polices a client that itself is a resolver, collateral damage will be caused to the latter's clients and possibly their clients further downstream.

Assumptions. Our design deals with a practical adversary as described in Section 2.3. Specifically, the adversary is off the resolution path subject to congestion as well as the underlying network paths. It has control over up to N_A distributed hosts that can send requests to any on-path DNS server. We assume that anti-spoofing measures are deployed by all these DNS servers and hence the adversary cannot

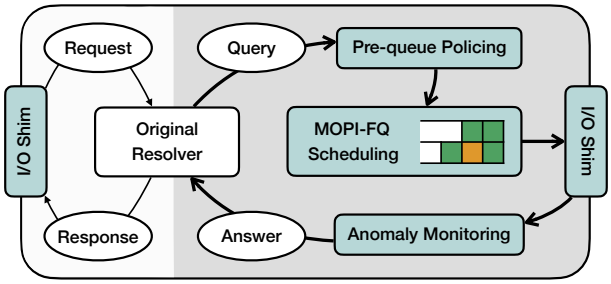


Figure 5. Overview of a DCC-enabled resolver with our non-invasive architecture. The left part is the resolver's *client-facing side* as a server, and the right part (shaded in gray) is its *server-facing side* as a client. DCC deals mainly with the server-facing side by controlling outgoing queries.

gain advantage by spoofing IP addresses. Such practical measures include DNS cookie [6], network ingress filtering [23], and source authentication [31, 52]. Our design should level the playing field such that the adversary amid legitimate clients can increase its overall query rate at a target inter-server channel only by increasing N_A.

We do not consider volumetric DoS attacks that congest the underlying networks or exhaust the resources of DNS servers themselves. These attacks are defended by orthogonal detection and mitigation techniques [40, 63, 69]. Targeting capacity-limited logical channels between servers, adversarial congestion can happen at a low request rate that is likely below the threshold triggering those defense systems.

3.2 Controlling Individual Resolvers

Retrofitting a new feature to DNS protocols or implementations, which are already highly complex, often result in unforeseen issues with existing components [39]. Therefore, we propose a non-invasive system architecture that can augment a resolver without requiring internal code changes. We call such an augmented resolver ***DCC-enabled.***

As Figure 5 shows, DCC wraps around a vanilla resolver by intercepting its I/O. The resolver's fast path of processing a client request upon a cache hit remains unchanged. In case of a cache miss, any resolver-generated outgoing queries will enter a control loop: (1) the queries are filtered by policies implemented in response to their senders' suspicious activities; (2) policy-compliant queries are buffered in a FQ data structure and scheduled for output; (3) the returned answers are monitored for anomalies, based on both locally collected statistics and signals issued by upstream servers, before they are passed to the resolver. We explain each of these technical aspects below.

3.2.1 Query Scheduling. Each upstream server that a resolver communicates with corresponds to one logical output channel. The goal of DCC's query scheduler is to fairly share every output channel among the resolver's clients. The fairness is defined over the number of queries *attributed to* a client, which neutralizes the amplification effects of malicious requests. DCC must track all queries derived from each request and link them to the responsible client. This process, which we refer to as *query attribution*, is already necessarily done by a resolver internally, and we describe how it can be realized in a portable way for DCC in Section 5.

The scheduler will try to insert an arriving query into a queue associated with the corresponding output channel. If the queue is already full, instead of discarding the query silently, DCC immediately returns a synthesized SERVFAIL answer to its accompanying resolver to avoid query timeout and waste of resources. The scheduler will pick and dequeue a query for output if the corresponding channel is not congested. Queries that fail to be dequeued in their turn will remain in the queue and may be dropped at some point as a result of fair scheduling. DCC uses a token bucket to control a channel's capacity, which is defined as the minimum between the rate limits imposed by the channel's two ends.[1]

Client Share Allocation. The scheduler ensures that the queries attributed to each client towards any channel can be sent out at a rate, which is determined dynamically by the channel's capacity and the number of active clients using it, according to its predefined share. DNS operators can allocate shares to clients in various ways. One simple strategy is to peg the share to the resolver's ingress rate limit: with a default per-client limit (e.g., 1500 for Google Public DNS), all clients are initially allotted the same share; clients admitted with higher limits get proportionally higher shares (e.g., Google allows large clients such as ISPs to request to raise their rate limits [3]). The share allocation can also be based on clients' query histories, which are commonly used by DNS operators to adjust system parameters [54].

3.2.2 Anomaly Monitoring. The FQ scheduling ensures fair use of a resolver's output channels, but attackers can still exploit specially crafted query patterns to gain advantages over benign clients: requests that trigger amplification effects with disproportionately many queries, requests for pseudo-random names that bypass resolver caching, requests inducing exceptionally high computational costs, etc.

DCC provides a generic and extensible module to monitor such anomalous requests. It keeps track of a collection of anomaly metrics, e.g., the amount, the rate, or the percentage of anomalous requests, for each client over a sliding window (e.g., 2 seconds). At the end of each window, an anomaly alarm will be generated if any metric goes beyond a predefined threshold. Upon the first alarm, DCC will put the client in a suspicious state for intensified monitoring. If the

[1] A DCC-enabled resolver can obtain the ingress rate limit of an upstream server in different ways: sending regular probing queries, using system parameters publicized by or negotiated between DNS operators, or leveraging DCC's in-band signal mechanism.

Table 1. Summary of the additional state introduced by DCC in comparison with existing resolver state.

	Per-Client	Per-Server	Per-Request
Resolver	Policing state	NS info, RL state	Resolution state
DCC	Monitoring metrics Pre-queue policies (policed clients only)	Queueing state (e.g., depth of subqueue)	Query statistics Signal status (e.g., anomaly)

number of alarms reaches a threshold (e.g., 10) within a period of suspicion (e.g., 60 seconds), defensive actions will be taken as described below. Otherwise, DCC will release the client from the suspicion. For accurate anomaly detection, resolver operators can define abnormalities and monitoring parameters based on their operational profiles.

3.2.3 Pre-Queue Policing. Once a suspicious client is convicted, DCC will activate and enforce a control policy on it. Possible policies include but not limited to temporarily rate limiting or blocking subsequent queries attributed to the client. The query policing is applied before the FQ scheduling. This avoids complicating the queuing design, as non-compliant queries will not be unnecessarily queued and delayed or dropped later, which undermines both fairness and performance. Note that the pre-queue policing here differs from a vanilla resolver's ingress policing on client requests (cf. Section 2.2). The former does not affect requests that are answered from cache and treated as normal by DCC.

3.2.4 State Management. DNS resolvers are inherently stateful. DCC also maintains state at different granularity levels, as summarized in Table 1. The creation, update, and deletion of DCC state goes in tandem with the corresponding resolver state. Per-client state entries are created upon the first requests from clients and per-server state entries upon the first queries to servers. These entries are maintained until the associated clients or servers become inactive for a while. Per-request state is more transient and is maintained only during a request's life span at the resolver. The overall state of DCC is asymptotically no larger than that of a vanilla resolver and is indeed concretely smaller as shown in our empirical evaluation (Section 5.2).

3.3 Controlling Full Resolution Path

DCC should be applied to all resolvers, as adversarial congestion can happen at any hop on a resolution path. Every resolver is controlled independently most of the time. In abnormal situations, however, the prompt notification of each other's actions becomes necessary for avoiding collateral damage. If a resolver does not know that its queries are marked anomalous at the upstream and take action in time, it will be soon policed with all its clients affected—this creates a DoS attack vector for a malicious client.

We arm DCC with an *in-band* signaling mechanism. At the upstream, it attaches control signals to outgoing DNS responses whenever needed. At the downstream, it processes such signals and removes them before passing the incoming answers to the augmented resolver. Hence, the signaling mechanism is transparent to resolvers and requires no extra control message. It can also be used to communicate system parameters between DCC instances. We encode the signals using EDNS [18]; they are semantically similar to and can be specified as Extended DNS Errors [33].

DCC-Awareness. DCC-enabled resolvers will function as intended without any coordination from regular DNS clients and servers. But these entities can enjoy performance and security benefits if they become *DCC-aware*—that is, being able to recognize and process certain DCC signals.

Below we explain how different types of signals are generated and processed along a resolution path.

3.3.1 Anomaly Signal. DCC attaches an anomaly signal in the response to every anomalous request from a client marked as suspicious. The signal contains the reason for suspicion, the current period of suspicion, the policy to be enforced, and a countdown to policing defined as the remaining number of alarms to convict the client. This allows the downstream resolver to react and find the real culprit.

To protect itself from the impending policing from upstream, a DCC-enabled resolver receiving an anomaly signal will control its client responsible for the anomaly. If the signaled countdown goes below a predefined threshold, the resolver starts policing the suspicious client right away. Otherwise, it relays the signal in its response to the suspect, optionally with the countdown value lowered so that the suspect is stressed to react more rapidly. By propagating anomaly signals downwards the resolution path, DCC enables precise control at the resolver closest to the attacker and thereby minimizes the collateral damage to benign hosts, as illustrated in Figure 6.

A DCC-aware entity can also process any received anomaly signals to avoid itself being policed, using its own logic that is not necessarily compatible with DCC. For an end host, the signals enable it to identify local malicious or compromised applications that generate anomalous DNS requests.

3.3.2 Policing Signal. If a client has been policed, DCC issues policing signals for all the client's requests that fail due to queries being dropped by pre-queue policing. Specifying an enforced policy's type and expiration time among other parameters, such signals are informative for a DCC-aware client to adjust its behavior, e.g., reducing request rate or switching to another resolver. A DCC-enabled resolver will propagate received policing signals to its own clients. It can

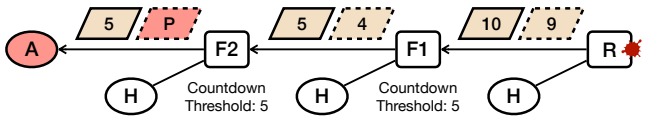


Figure 6. Illustration of DCC's signaling mechanism. The resolver (R) generates two anomaly signals with an initial countdown of 10. One forwarder (F1) lowers the countdown by 5 when relays the signal whereas the other (F2) does not. The dashed parallelograms represent the second round of signals, which cause F2 to start policing the suspicious client (A) and generates a policing signal (P), without causing collateral damage to other hosts (H).

also temporarily increase the sensibility of anomaly monitoring by lowering thresholds, as it failed to identify and control the culprit earlier.

3.3.3 Congestion Signal. DCC's scheduler will fail to enqueue a query if the corresponding output channel is congested, and the augmented resolver may resend the query multiple times. If the client request eventually fails, DCC generates a congestion signal containing the count of queries dropped and the current query rate allocated to the client, among other information. Upon receiving a congestion signal, a DCC-aware entity can reduce its request rate for the same domain, increase its backoff time, or try a different resolver as requests to the same resolver will likely fail again due to upstream congestion. A DCC-enabled resolver will propagate the signal to its own clients.

Despite the names, congestion signals are intended to *inform* rather than *control* congestion. It is DCC's query scheduling that prevents an aggressive client from using an inter-server channel more than its fair share in any event.

3.3.4 Co-Existence of Signals. One response can carry multiple signals, one for each type, that are generated locally by the answering resolver or originated from further upstream. When both an upstream signal and a local signal of the same type are available to be included in a response, DCC prefers the former because it has a bigger impact on the resolver as a whole. When processing such a response, a DCC-aware entity prioritizes signals based on their severity, i.e., in the order of policing, anomaly, and congestion.

4 MOPI Scheduler

The core query scheduler of DCC should satisfy three properties: (1) fair sharing of each output channel of a resolver among its clients; (2) practical space overheads comparable to the resolver's own runtime state; and (3) low scheduling delay added to the overall DNS resolution latency. Designing such a scheduler turns out to be non-trivial. In this section, we first analyze the underlying fair scheduling problem and its subtle differences from similar problems (Section 4.1). We then develop a solution that achieves all desired properties (Section 4.2).

4.1 Problem Formulation

Consider a scheduler that dispatches messages from a set of input sources $\mathcal{S}$ to a set of destinations $\mathcal{D}$ through a set of output channels O each with limited capacity. Let $O(p) \subset O$ be the set of channels that a message p can be sent out of, and a_{ij} be the message rate allocated to source $i \in \mathcal{S}$ for channel $j \in O$. Our goal is to achieve a *max-min fair* allocation $\{a_{ij}\}_{i\in\mathcal{S}}$ for any channel j (more details are given in Appendix B.2). Below we explain how the problem setting varies in different fair queuing (FQ) variants.

Conventional FQ deals with a single output channel embodied as a network interface (i.e., $|O| = 1$). It aims to achieve a max-min fair allocation $\{a_{io}\}_{i\in\mathcal{S}}$ of the only channel o. If a device has several interfaces, each of them requires an independent scheduler instance. In our context, it is infeasible to maintain one expensive FQ scheduler for each (logical) output channel because the number of such channels can be very large (i.e., $|O| = |\mathcal{D}|$).

Multi-server FQ [11] schedules traffic to multiple servers (equivalent to output channels) for load-balancing. These servers are indistinguishable in that any of them can service any traffic flow (i.e., $O(p) = O$). This is in contrast to our setting where every DNS message has one predetermined destination and output channel (i.e., $|O(p)| = 1$). Similarly, multi-queue FQ [26, 57] dispatches traffic flows to the internal queues (equivalent to output channels) of an I/O device or OS, and the queues are essentially identical. As another essential distinction, this category of FQ variants aim for fairness in the aggregated traffic rate of all channels (i.e., $\{\sum_{j\in O} a_{ij}\}_{i\in\mathcal{S}}$), whereas we seek fair sharing of each and every channel. These two goals are indeed incompatible: fairly allocating the aggregated rate can undermine the fairness of individual channels and vice versa, especially when different sources use different subsets of channels.

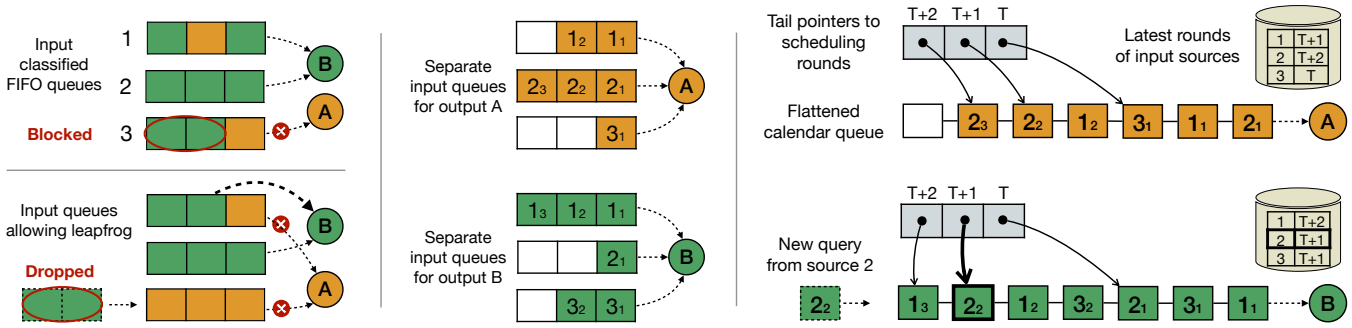
Multi-interface FQ [66] generalizes multi-server FQ by restricting the interfaces usable by a flow to a subset of the network interfaces available to a device. When restricting each flow to a single interface, the solution reduces to the aforementioned naive approach of applying FQ independently to each interface or output channel.

Multi-resource FQ [25] considers a fairness notion defined over different resources consumed by flows. Regarding each output channel in our context as a distinct resource does not make sense, as a message is sent to only one channel.

In summary, we are not aware of any previous formulation of the fair scheduling problem that we study here, and to the best of our knowledge, FQ algorithms proposed in the literature do not apply to the problem.

4.2 Scheduler Design

We explore the design space progressively, starting with simple intuitive ideas, explaining the challenges at each design point, and eventually arriving at our solution.



(a) Input-centric queuing. Source 3 gets reduced msg rate to B due to A's congestion.

(b) IO-isolated queuing w/ separate per-source queues for each output.

(c) Output-centric queuing, with the same buffered messages as (b). The changes after enqueuing a message 2_3 are highlighted in bold.

Figure 7. Illustration of MO-FQ design space. A symbol X_y in (b) and (c) indicates source X's y-th message to a given output.

Input-Centric Queuing. First proposed by Nagle [45], the textbook FQ algorithm creates a separate first-in-first-out (FIFO) queue for each source and schedules messages from the queues in a round-robin manner. Such input-centric queuing ensures max-min fairness for equal-sized packets in the conventional setup with a single output interface.

However, the fairness property no longer holds in our multi-output setup, as illustrated in Figure 7a. In the example at the top, the queue for one input source (3) is blocked by a congested output channel (A), which prevents the scheduler from dequeuing messages to other available channels and hence renders those channels' allocation unfair to the source. This is commonly known as the problem of *head-of-line (HOL) blocking*. A plausible fix is to relax the FIFO property and allow the scheduler to leap over the blocking message, but it does not fix the issue: as shown by the example at the bottom of Figure 7a, messages to other channels are still dropped when a blocked queue gets filled up as an inevitable consequence of the channel's congestion. Attackers can exploit these forms of unfairness to block messages from a target source especially when it is a shared resolver.

IO-Isolated Queuing. The limitations of input-centric queuing stem from the lack of isolation among output channels. An immediate improvement is to create separate per-source FIFO queues for each output channel (or equivalently, separate per-output queues for each source). We refer to this paradigm as IO-isolated queuing as depicted in Figure 7b.

With the isolation, no message of a queue will be blocked or dropped due to congestion at other queues. An obvious drawback however arises: the substantial costs of maintaining $O(|\mathcal{S}|\cdot|\mathcal{O}|)$ queues, and the concomitant vulnerability to resource exhaustion attacks where attackers trick the scheduler to create a myriad of queues. Moreover, how to fairly schedule over these queues becomes unclear, e.g., whether to apply round robin first over sources and then destinations or vice versa. Regardless of the design, now that there are multiplicatively more queues, the queuing delay of messages can be $|\mathcal{O}|$ times larger than input-centric queuing.

While IO-isolated queuing solves the MO-FQ problem with the fairness guarantee, it is hardly practical due to these performance drawbacks.

Output-Centric Queuing. The bit-by-bit round robin algorithm (BBRR) [20] improves over Nagle's FQ algorithm by buffering messages in a *single* priority queue, where variable-sized messages are sorted according to their virtual finish time. Applying BBRR to each output channel leads to what we refer to as output-centric queuing.

In our problem setting, the message size is immaterial and messages are always treated as atomic units for scheduling. Since messages' finish time depends only on their arrival time, we can avoid BBRR's logarithmic enqueue cost. This is achieved by tracking the boundaries between rounds in the queue and always inserting a message to the end of its corresponding round, as depicted in Figure 7c. The queue should be implemented as a linked list to allow efficient insertion with the help of round-tracking pointers. With such design, the queuing data structure coincides with a calendar queue [13] but in a *flattened* form.

Output-centric queuing reduces the storage overhead of IO-isolated queuing, but it is not yet ideal. First, the costs of maintaining $|\mathcal{O}|$ queues are still undesirable, especially when they need to be pre-allocated for efficiency reasons. Second, the problem of queuing delay inflation remains: enqueued messages may be reordered (with respect to their arrival order) as a result of the fair scheduling on each output queue as well as *across all queues*.

MOPI-FQ. We further improve the above design in two aspects: (1) allocating recyclable entries for all queues from a single resource pool of fixed size, without wasting space for the pre-allocation of individual queues; (2) preserving the arrival order of messages for output across all queues, up to in-queue reordering for fair scheduling and channel congestion. The first aspect is relatively straightforward. For the second, we attach to each queued message its arrival time and always pick the queue whose channel is not congested,

Table 2. A summary of client settings for evaluating DCC under different adversarial congestion scenarios. The 2nd and 3rd columns indicate when a client starts and stops sending queries in the 60-seconds measurement window. The last column indicates the client's query patterns (Figure 2) used in the corresponding scenarios (Figure 8).

Client	Start	End	QPS	Query Pattern
Heavy	0	60	600	WC (a,b,c) or NX (b)
Medium	0	50	350	WC (a,b,c)
Light	20	60	150	WC (a,b,c)
Attacker	10	60	1100 or 50	WC (a), NX (b) or FF (c)

and whose message at the front has the earliest arrival time among all queues, to be sent out.

We call the resulting scheduling algorithm MOPI-FQ, which is short for *multi-output pseudo-isolated fair queuing*, since all output channels are scheduled independently in that their fairness and output order are preserved but their queues are allocated from a shared resource pool (hence not fully isolated from each other). MOPI-FQ is both *space-efficient*, requiring $O(|O| + q)$ storage where q is the total number of queued messages, and *time-efficient*, supporting $O(log(|O|))$ enqueue and dequeue operations. We provide a detailed algorithm specification and analysis in Appendix B.

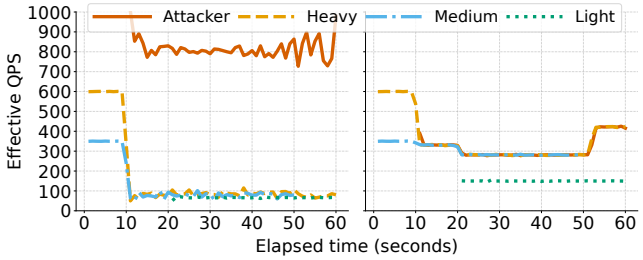
5 Implementation and Evaluation

We have developed a prototype of DCC in C++. The non-invasive architecture of DCC allows it to be retrofitted into existing DNS software or implemented as a standalone program that intercepts the original resolver's DNS traffic. We chose the latter approach because it allows DCC to be deployed on the same host running the resolver or on a separate host as a middlebox. This requires a portable way for DCC to track outgoing resolver queries and associate them with the responsible clients.

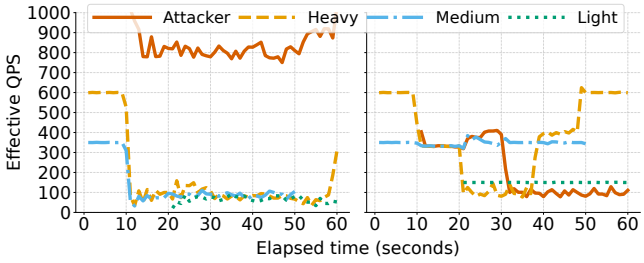
We realize this query attribution function (Section 3.2.1) by repurposing the EDNS Client Subnet option [17] to include the client's IP address, source port, and DNS request ID. In particular, we modify the BIND resolver to include this EDNS option in every outgoing query generated by the resolution of a client request, and let DCC strip off this option when sending out queries to upstream DNS servers.

For traffic interception, we use `libnetfilter_queue` and set up two separate queues for incoming and outgoing DNS messages with the corresponding `iptables` rules. To avoid an interception loop—the outgoing queries processed by DCC are captured by `libnetfilter_queue` again—we use the raw socket `AF_PACKET` to send out DCC queries, which comes with the extra performance benefit of kernel bypassing.

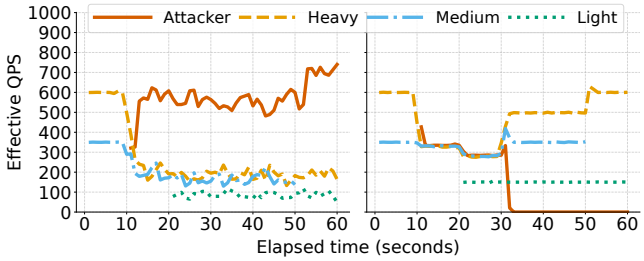
All runtime state of DCC (see Table 1) can be efficiently managed by hash tables and we realize them using the standard C++ `std::unordered_map`. For MOPI-FQ scheduling,



(a) Attacker exploiting the WC query pattern.



(b) Attacker and heavy client exploiting the NX query pattern.



(c) Attacker exploiting the FF amplification pattern.

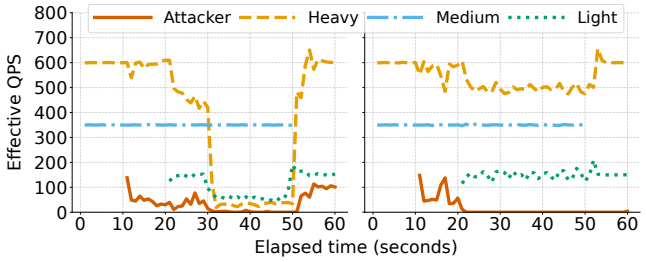
Figure 8. Client dynamics in different adversarial congestion scenarios summarized in Table 2. The effective QPS is measured by the ratio of successful responses (NOERROR or NXDOMAIN), with the exception that it is calculated from the actual queries received by our nameserver in case the attacker uses FF amplification pattern. For each scenario, the left and right subplots show results for a vanilla BIND resolver and a DCC-enabled resolver, respectively.

we implement the data structures specified in Appendix B with customized memory management.

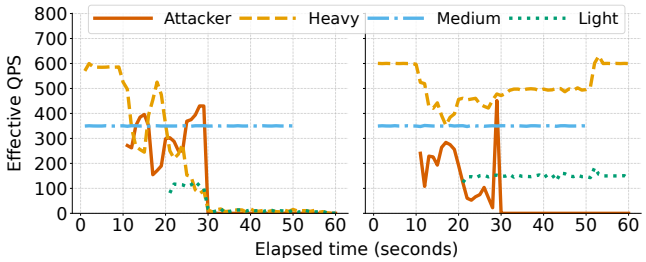
In addition to the main thread driven by `libnetfilter`, our prototype uses a separate thread for MOPI-FQ dequeue operations and another thread for periodically purging the state of inactive clients or servers.

Evaluation Setup. With the prototype, we evaluate a DCC-enabled resolver's attack resilience and performance overheads. All our DNS servers and clients are deployed in the DigitalOcean cloud using VMs with dedicated virtual CPUs (two cores at 2.6GHz) and 8GB RAM. Our resolver and authoritative nameservers run BIND 9.

For MOPI-FQ configuration, we set the capacity of each output queue as 100, the maximum number of per-queue



(a) Attacker exploiting the NX query pattern.



(b) Attacker exploiting FF amplification pattern.

Figure 9. Evaluation of DCC's anomaly monitoring, policing, and signaling mechanisms in two scenarios (Table 2). For each of them, the left and right subplots show results when the signaling mechanism is turned off and on, respectively.

scheduling round as 75 (which makes fair scheduling meaningful when there are at least two clients), and the overall capacity of the resource pool as 100000 (=100K). We set no limit on the number of concurrently tracked clients or servers. The state of an entity will be removed if it has been inactive for 10 seconds. All clients are configured with identical shares for the MOPI-FQ scheduling.

5.1 Attack Resilience

We examine DCC's effectiveness against adversarial congestion using an architecture similar to the setup (a) in our attack validation experiments (Section 2.3.1). It consists of four clients sending queries of different patterns and at varying rates to a resolver towards an authoritative nameserver, as summarized in Table 2. The first three of them represent benign clients with different degrees of aggressiveness and the last one represents an attacker. The inter-server channel is limited to 1000 QPS.

We configure DCC's anomaly monitoring window as 2 seconds, and the anomaly alarm threshold to convict a suspicious client as 10 within a suspicion period of 60 seconds. The policy for a client convicted with NXDOMAIN anomalies is rate limiting it to 100 QPS for 20 seconds. The policy for a client convicted with amplification anomalies is blocking all its queries for 30 seconds.

We measure the clients' effective QPS over time under these settings and plot the results in Figure 8, with side-by-side comparisons between a vanilla BIND resolver (version 9.19) and our prototype of a DCC-enabled resolver.

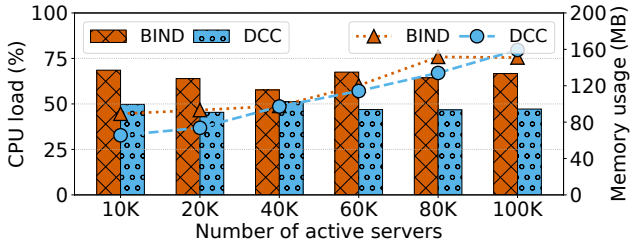
Scenario 1: Wildcard. This represents a worse-case scenario where we cannot distinguish the attacker from benign clients. As depicted in Figure 9a, for the vanilla resolver, the heavy and medium clients send queries at their full speed in the initial 10 seconds, but their QPS drastically reduces to a range below 100 once the attacker kicks in. In comparison, DCC fairly and dynamically allocates the channels among active clients in a work-conserving manner: evenly among the three higher-rate clients during seconds 10–20, then among them during seconds 20–50 after fulfilling the low-rate client, and so on.

Scenario 2: NXDOMAIN. This scenario illustrates DCC's dynamics when handling the common query pattern used by pseudorandom subdomain attacks. In addition to the attacker, we also let the aggressive client use the NX pattern for the first 20 seconds and switch to the benign WC pattern afterwards. From Figure 9a we can see that the vanilla resolver is subject to adversarial congestion as in the previous scenario. When DCC detects clients abusing this query pattern (with the ratio of NXDOMAIN responses above 0.2), it rate limits the senders and instantaneously allocates the freed channel space to benign clients. Since the heavy client ceases to send anomalous requests right after it is policed, it regains its share of the channel after the policy expires at second 50. On the other hand, the attacker who persists the malicious query pattern is rate limited until the end.

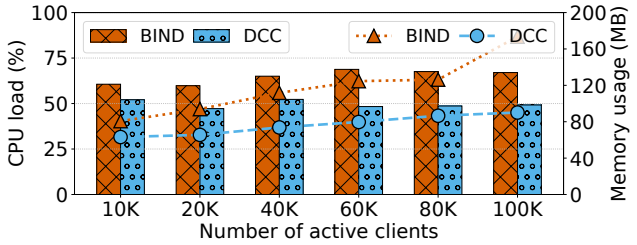
Scenario 3: Amplification. Here the attacker exploits message amplification and sends requests at a rate of 50 QPS, which is multiplied at the target channel. As shown in Figure 9b, congestion still happens with significant impact on the benign clients of the vanilla resolver. In contrast, DCC effectively controls the attacker's aggressive queries and blocks them after confirming the suspicion, while always allocating the channel fairly among non-policed clients.

Efficacy of Signaling. To understand DCC's dynamics in a full resolution path and assess the efficacy of its signaling mechanism, we consider an architecture similar to the setup (c) in Figure 3c. Both the forwarder and recursive resolver are DCC-enabled with our prototype, and the channel between them is limited to a capacity of 1000 QPS. The attacker, heavy client and light client send requests to the forwarder and the medium client send requests directly to the recursive resolver. The heavy client always uses the WC pattern. We reduce the attacker's request rate to 200 QPS for the NX pattern and 20 QPS for the FF pattern. All other settings remain the same as above. We compare two scenarios with the signaling functions enabled or disabled.

Figure 9 depicts the results. Now the resolver-nameserver channel is shared between the medium client and the forwarder. The former can always send traffic at its expected rate 350 (<1000/2) QPS, with the remaining 650 QPS allocated to the forwarder and indirectly its clients.



(a) Fixed 1K clients and varying numbers of servers



(b) Fixed 1K servers and varying numbers of clients

Figure 10. DCC's performance overheads under different workloads dictated by the number of entities tracked. The bars indicate CPU load and lines indicate memory usage.

Absent the signaling mechanism, the forwarder's two benign clients are fate-sharing with the attacker: either competing for the limited upstream channel in case of the NX pattern, or being completely blocked in the case of the FF pattern, as a result of the resolver's policy enforced on the forwarder. With DCC signals enabled, the forwarder blocks (configured as the default policy for signal-triggered policing) its own suspicious client in time (with the anomaly countdown threshold configured as 5), thus effectively saving the two innocuous clients from collateral damage.

5.2 Performance Overhead

In real deployments a resolver can maintain state for many clients and servers concurrently. To investigate the performance of DCC under varying workloads, we simulate large numbers of entities by mapping random query names to client and server ID spaces of given sizes. In the experiments each of our four clients sends queries using the WC pattern at 750 QPS. With an aggregate rate of 3000 QPS, we can sustain the vanilla resolver's CPU load below 100% and rule out potential side effects due to overloading.

Figure 10 reports the CPU and memory usage of DCC compared with its accompanying BIND resolver. Each reported data point is averaged over consecutive per-second measurements for 1 minute, and we start to collect data when DCC has tracked approximately the expected number of clients or servers. Overall, DCC consumes less resources than the vanilla BIND resolver. DCC's computational cost is insensitive to the number of entities tracked, since most of its operations are constant-time and the MOPI-FQ operations with logarithmic complexity are also highly efficient. A large fraction of DCC's CPU time is actually spent on busy waiting for queries to be enqueued by MOPI-FQ.

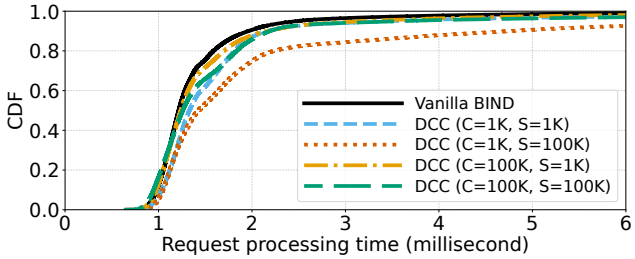


Figure 11. The processing delay incurred by DCC under varying numbers of active clients (C) and servers (S).

DCC's memory usage is more sensitive to the number of servers than clients, due to the larger per-server state including round pointers and counters of active clients MOPI-FQ maintains for each output queue. Nonetheless, our results show that DCC always has a smaller memory footprint than the vanilla resolver. In fact, we observed that BIND consumes substantially more memory if our measurements use amplification query patterns, which force the vanilla resolver to maintain more state for recursive queries.

We also measured the extra processing delay incurred by DCC. In particular, we consider the time a vanilla or DCC-enabled resolver takes to process a client request without cached answer using the WC query pattern. Figure 11 depicts the distribution of processing time for 1 million client requests. The results indicate that DCC introduces marginal delay under varying workloads. The processing time is indeed dominated by network delay as the RTT between our resolver and ANS is around 1 millisecond.

6 Discussion

Incremental Deployment. The deployment of DCC is facilitated by its noninvasive design, which requires no modification to today's DNS infrastructure and only minimum change to existing resolver software. It is also incrementally deployable, meaning that not all DNS servers must adopt it simultaneously for it to be effective or for the servers to maintain interoperability. Every individual resolver augmented by DCC obtains the availability benefits without the need to coordinate with others. Deploying DCC at a resolver incentivizes its adoption at the downstream resolvers, as this prevents them from being framed by malicious clients and policed by the upstream resolver. Even without deploying DCC, resolvers can opt to process DCC signals as Extended DNS Errors to improve their resilience and performance.

Encrypted DNS. Secure transport protocols for DNS, such as DoT [47], DoH [27] and DoQ [28], are widely deployed nowadays. DCC is compatible with them because it deals with application-layer DNS messages without touching the underlying connection methods. DoT/DoH/DoQ servers can adopt DCC to protect their queries from adversarial congestion at upstream channels.

Oblivious DNS. Solutions with even stronger privacy properties are also gaining traction. Oblivious DNS protocols introduce additional proxies to obfuscate the linkage between queries and their originators [32, 53]. In our framework, such a proxy is essentially a forwarder that communicates with an upstream oblivious resolver (operated by a different entity), and they are independently augmented by DCC. The proxy can perform query attribution without the need to see queries in plaintext, and it is treated as a regular client by the resolver for congestion control. For certain design that encrypts the entire DNS message rather than only the domain and address information, DCC's signaling mechanism may be hindered as the EDNS options may not be accessible to a DCC-enabled resolver. In this case, the signals as non-sensitive data can be taken out from DNS messages and stored as separate payload of lower-layer protocols.

7 Related Work

A prevalent strategy for mitigating DDoS attacks is *traffic filtering*, which aims to identify and discard or deprioritize malicious traffic. As in-network filtering solutions, Xatu relies on attack preparation signals and history [65], ACC-Turbo uses online clustering for attack identification at line rate [10], and SENSS allows victims to request attack monitoring and filtering from ISPs in exchange for payments [50]. Filtering techniques are also employed for application-level DoS protection. SkyShield [62] utilizes sketches to detect and counter application-layer DDoS attacks, FineLame [21] builds a model of resource utilization to identify and handle offending requests in real-time, and Leader [61] learns legitimate per-application usage patterns to detect irregularities. In general, filtering-based solutions face the difficulty in reliably distinguishing between malicious and benign traffic [29, 34, 38, 59]. Moreover, as explained in Section 2.3, they can be abused by attackers to inflict collateral damage on innocuous clients in the case of DNS.

Routing-based solutions like Nyx [56] or CoDef [35] reroute traffic around congestion points. DNS's architectural redundancies also allow entities to send requests along different resolution paths. But this can mitigate adversarial congestion only to a limited degree, because of the relatively small sets of candidate resolution paths and the fact that upon failure retried requests are indeed duplicated multiple times.

DCC's signaling mechanism draws inspiration from conventional end-to-end congestion control algorithms (CCAs). Unlike explicit congestion notification (ECN) [49] and solutions alike, which designate endpoints as entities responding to congestion, DCC signaling takes on a recursive downstream process. The different communication patterns—linear network paths from source to destination versus a recursive resolution tree structure formed by query branching to multiple upstream servers—pose challenges for seamlessly integrating existing CCAs into the DNS architecture.

Another mechanism closely aligned with DCC is *traffic isolation*. PSP [16], for instance, seeks to mitigate the collateral damage of DDoS attacks on benign traffic by implementing bandwidth isolation between traffic flows originating from different network origin-destination pairs. But the required network visibility is lacking in the DNS architecture. A prominent avenue explored in this direction is fair queuing (FQ), and we have discussed why existing FQ variants cannot be adapted to solving our problem in Section 4.1. Cebinae [67], by redistributing a fraction of flows' bandwidth, adjusts link allocations towards achieving max-min fairness. In the context of home networks, CRAB [60] estimates downlink capacity and flow demands, using this information to compute and throttle flows towards max-min weighted fair share rates. Core-stateless fair queuing [58], along with its hierarchical variant [68], accomplishes fair bandwidth allocation and ensures isolation without maintaining per-flow state in the network. However, their applicability to DNS is limited as they assume the operator's complete control over the network.

8 Conclusion

The importance of DNS to the Internet cannot be overstated. In this paper, we identify and experimentally validate a new class of DoS attacks that enable an adversary to disrupt DNS services without large volumes of attack traffic. Adapting principles from classic network congestion control, we design, implement, and evaluate a framework DCC that can effectively mitigate such attacks while minimizing collateral damage to benign users. DCC can enhance the availability and fairness of today's DNS infrastructure with every additional resolver that adopts it, without requiring a global setup. We hope DCC can contribute to making DNS and the Internet as a whole more resilient to DoS attacks.

Ethics statement. Our measurements use our own domains and nameservers and we send small volumes of requests to the measured public resolvers with negligible impact on their normal operation. Since adversarial congestion is not pertinent to any particular protocol or implementation vulnerabilities, we are discussing it with entities within the DNS community to identify effective ways to broadly disclose it and implement practical fixes including DCC.

Acknowledgments

We thank the anonymous reviewers and our shepherd Nickolai Zeldovich for their valuable feedback. We also thank Zechao Cai for helping with instrumenting the BIND 9 code and Simon Scherrer for providing early feedback on this work. We gratefully acknowledge support from ETH Zürich, ZISC, from SNSF for project 200021_215318, and from Hasler Stiftung via the ETH Zurich Foundation.

References

[1] DCC Artefact. https://gitlab.ethz.ch/netsec/dcc-artefact.

[2] DNS Nameserver Counts for Top Million Websites (2020-08). https://dnsinstitute.com/research/2020/top-million-202008.html.

[3] Google Public DNS for ISPs. https://developers.google.com/speed/public-dns/docs/isp, 2024.

[4] resolv.conf(5) — linux manual page. https://man7.org/linux/man-pages/man5/resolv.conf.5.html, Janurary 2024.

[5] TLD Zone File Statistics. https://www.statdns.com, Janurary 2024.

[6] D. Eastlake 3rd and M. Andrews. Domain Name System (DNS) Cookies. RFC 7873, IETF, May 2016.

[7] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. Nxnsattack: Recursive DNS inefficiencies and vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2020.

[8] Akamai. Whitepaper: DNS Reflection, Amplification, & DNS Watertorture. Technical report, 2019.

[9] Rami Al-Dalky and Kyle Schomp. Characterization of Collaborative Resolution in Recursive DNS Resolvers. In *Proceedings of the International Conference on Passive and Active Measurement (PAM)*, 2018.

[10] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. Aggregate-based congestion control for pulse-wave ddos defense. In *Proceedings of the ACM SIGCOMM Conference*, 2022.

[11] Josep M Blanquer and Banu Özden. Fair Queuing for Aggregated Multiple Links. *ACM SIGCOMM Computer Communication Review*, 31(4):189–197, 2001.

[12] S. Bortzmeyer, R. Dolmans, and P. Hoffman. DNS Query Name Minimisation to Improve Privacy. RFC 9156, IETF, November 2021.

[13] Randy Brown. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[14] Jonas Bushart and Christian Rossow. DNS unchained: Amplified application-layer dos attacks against DNS authoritatives. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[15] Jonas Bushart and Christian Rossow. Anomaly-based filtering of application-layer ddos against dns authoritatives. In *2023 IEEE 8th European Symposium on Security and Privacy*, 2023.

[16] Jerry Chou, Bill Lin, Subhabrata Sen, and Oliver Spatscheck. Proactive surge protection: a defense mechanism for bandwidth-based attacks. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, USA, 2008. USENIX Association.

[17] Carlo Contavalli, Wilmer van der Gaast, David C Lawrence, and Warren "Ace" Kumari. Client Subnet in DNS Queries. RFC 7871, May 2016.

[18] Joao da Silva Damas, Michael Graff, and Paul A. Vixie. Extension Mechanisms for DNS (EDNS(0)). RFC 6891, April 2013.

[19] Casey Deccio, Derek Argueta, and Jonathan Demke. A Quantitative Study of the Deployment of DNS Rate Limiting. In *2019 International Conference on Computing, Networking and Communications (ICNC)*, 2019.

[20] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[21] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer Denial-of-Service attacks In-Flight with FineLame. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[22] Huayi Duan, Marco Bearzi, Jodok Vieli, David Basin, Adrian Perrig, Si Liu, and Bernhard Tellenbach. CAMP: Compositional Amplification Attacks against DNS. In *Proceedings of the USENIX Security Symposium*, 2024.

[23] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, IETF, May 2000.

[24] K. Fujiwara, A. Kato, and W. Kumari. Aggressive Use of DNSSEC-Validated Cache. RFC 8198, IETF, July 2017.

[25] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

[26] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. Multi-Queue Fair Queuing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[27] P. Hoffman and P. McManus. DNS Queries over HTTPS (DoH). RFC 8484, IETF, October 2018.

[28] C. Huitema, S. Dickinson, and A. Mankin. DNS over Dedicated QUIC Connections. RFC 9250, IETF, May 2022.

[29] Min Suk Kang, Soo Bum Lee, and Virgil D. Gligor. The crossfire attack. In *IEEE Symposium on Security and Privacy*, 2013.

[30] Aqsa Kashaf, Vyas Sekar, and Yuvraj Agarwal. Analyzing Third Party Service Dependencies in Modern Web Services: Have We Learned from the Mirai-Dyn Incident? In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.

[31] S. Kent. IP Authentication Header. RFC 4302, IETF, December 2005.

[32] E. Kinnear, P. McManus, T. Pauly, T. Verma, and C.A. Wood. Oblivious DNS over HTTPS. RFC 9230, IETF, June 2022.

[33] W. Kumari, E. Hunt, R. Arends, W. Hardaker, and D. Lawrence. Extended DNS Errors. RFC 8914, IETF, October 2020.

[34] Yi-Hsuan Kung, Taeho Lee, Po-Ning Tseng, Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Soo Bum Lee, Yue-Hsun Lin, and Adrian Perrig. A practical system for guaranteed access in the presence of ddos attacks and flash crowds. In *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*, 2015.

[35] Soo Bum Lee, Min Suk Kang, and Virgil D. Gligor. Codef: collaborative defense against large-scale link-flooding attacks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13. Association for Computing Machinery, 2013.

[36] E. Lewis. The Role of Wildcards in the Domain Name System. RFC 4592, IETF, July 2006.

[37] Xiang Li, Dashuai Wu, Haixin Duan, and Qi Li. DNSBomb: A New Practical-and-Powerful Pulsing DoS Attack Exploiting DNS Queries-and-Responses. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024.

[38] Yuanjie Li, Hewu Li, Zhizheng Lv, Xingkun Yao, Qianru Li, and Jianping Wu. Deterrence of intelligent ddos via multi-hop traffic divergence. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.

[39] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. A Formal Framework for End-to-End DNS Resolution. In *Proceedings of the ACM SIGCOMM Conference*.

[40] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *Proceedings of the USENIX Security Symposium*, 2021.

[41] Florian Maury. The "indefinitely" delegating name servers (idns) attack. https://indico.dns-oarc.net/event/21/contributions/301/attachments/272/492/slides.pdf, 2015. Accessed 2022-04-30.

[42] Andrew McGregor, Phillipa Gill, and Nicholas Weaver. Cache Me Outside: A New Look at DNS Cache Probing. In *Proceedings of the International Conference on Passive and Active Measurement (PAM)*, 2021.

[43] G. Moura, W. Hardaker, J. Heidemann, and M. Davids. Considerations for Large Authoritative DNS Server Operators. RFC 9199, IETF, March 2022.

[44] Giovane C. M. Moura, Sebastian Castro, John S. Heidemann, and Wes Hardaker. Tsuname: exploiting misconfiguration and vulnerability to ddos DNS. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2021.

[45] J. Nagle. On Packet Switches With Infinite Storage. RFC 970, IETF, December 1985.

[46] Marcin Nawrocki, Maynard Koch, Thomas C Schmidt, and Matthias Wählisch. Transparent Forwarders: An Unnoticed Component of The Open DNS Infrastructure. In *Proceedings of the International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2021.

[47] S. Proust. Additional WebRTC Audio Codecs for Interoperability. RFC 7875, IETF, May 2016.

[48] Bozidar Radunovic and Jean-Yves Le Boudec. A Unified Framework for Max-Min and Min-Max Fairness With Applications. *IEEE/ACM Transactions on Networking*, 15(5):1073–1083, 2007.

[49] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF, September 2001.

[50] Sivaramakrishnan Ramanathan, Jelena Mirkovic, Minlan Yu, and Ying Zhang. Senss against volumetric ddos attacks. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, 2018.

[51] Audrey Randall, Enze Liu, Gautam Akiwate, Ramakrishna Padmanabhan, Geoffrey M Voelker, Stefan Savage, and Aaron Schulman. Trufflehunter: Cache Snooping Rare Domains at Large Public DNS Resolvers. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.

[52] Benjamin Rothenberger, Dominik Roos, Markus Legner, and Adrian Perrig. PISKES: Pragmatic Internet-scale key-establishment system. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2020.

[53] Paul Schmitt, Anne Edmundson, Allison Mankin, and Nick Feamster. Oblivious DNS: practical privacy for DNS queries. In *Proceedings on Privacy Enhancing Technologies*, 2019.

[54] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K Sitaraman. Akamai DNS: Providing Authoritative Answers to the World's Queries. In *Proceedings of the ACM SIGCOMM Conference*, 2020.

[55] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. On Measuring The Client-Side DNS Infrastructure. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2013.

[56] Jared M Smith and Max Schuchard. Routing around congestion: Defeating ddos attacks and adverse network conditions via reactive bgp routing. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[57] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[58] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1998.

[59] Ahren Studer and Adrian Perrig. The coremelt attack. In *Computer Security – ESORICS*, 2009.

[60] Ammar Tahir and Radhika Mittal. Enabling users to control their internet. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[61] Rajat Tandon, Haoda Wang, Nicolaas Weideman, Shushan Arakelyan, Genevieve Bartlett, Christophe Hauser, and Jelena Mirkovic. Leader: Defense against exploit-based denial-of-service attacks on web applications. 2023.

[62] Chenxu Wang, Tony T. N. Miu, Xiapu Luo, and Jinhe Wang. Skyshield: A sketch-based defense system against application layer ddos attacks. *IEEE Transactions on Information Forensics and Security*, 2018.

[63] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *Proceedings of the USENIX Security Symposium*, 2021.

[64] Wei Xu, Xiang Li, Chaoyi Lu, Baojun Liu, Haixin Duan, Jia Zhang, Jianjun Chen, and Tao Wan. TsuKing: Coordinating DNS Resolvers and Queries into Potent DoS Amplifiers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2023.

[65] Zhiying Xu, Sivaramakrishnan Ramanathan, Alexander Rush, Jelena Mirkovic, and Minlan Yu. Xatu: boosting existing ddos detection systems using auxiliary signals. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '22, 2022.

[66] Kok-Kiong Yap, Te-Yuan Huang, Yiannis Yiakoumis, Sandeep Chinchali, Nick McKeown, and Sachin Katti. Scheduling packets over multiple interfaces while respecting user preferences. In *Proceedings of the International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2013.

[67] Liangcheng Yu, John Sonchack, and Vincent Liu. Cebinae: Scalable in-network fairness augmentation. In *Proceedings of the ACM SIGCOMM Conference*, 2022.

[68] Z. Yu, J. Wu, V. Braverman, I. Stoica, and X. Jin. Twenty years after: Hierarchical core-stateless fair queueing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[69] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2020.

Materials presented in the appendix have not been peer-reviewed

A Measurement Details

To understand whether and how real-world DNS servers implement rate limiting (RL), we conduct a measurement study on a list of 45 popular public resolvers as summarized in Table 3. They cover DNS operators of varying scale and geo-locations. Some of the resolvers implement IP-based access control and are non-responsive to some of our probing clients (probes), but all of them stay responsive to at least one probe throughout our measurements.

We configure two authoritative BIND 9 servers to host our own domains with carefully designed resource records, and use three probes to generate requests for these records. All our servers and probes run on cloud VMs from DigitalOcean. The two servers and one probe locate at a data center in Europe and the other two probes in North America and Asia. Our measurements do not violate the cloud provider's user policy nor raise any security alarm from the cloud platform. We took different precautions described below to ensure that the measurements cause no harm to those public resolvers. Our study consists of the following two parts.

Table 3. The 45 public resolvers used in our measurements. The list is compiled from multiple sources*.

Resolver	IP	Resolver	IP
AdGuard (AG) DNS	94.140.14.14	InfoServer GmbH	212.89.130.180
AliDNS	223.5.5.5	Level 3 DNS	209.244.0.3
AMAZON-02	54.93.169.181	Liteserver	5.2.75.75
Baidu Public DNS	180.76.76.76	NTT America	129.250.35.250
CIRA Canadian	149.112.121.10	Neustar	64.6.64.6
CNNIC-SDNS	1.2.4.8	NextDNS	45.90.30.193
CenturyLink	205.171.3.65	Nextgi LLC	134.195.4.2
CleanBrowsing	185.228.168.9	Norton-ConnectSafe	199.85.126.10
Cloudflare	1.1.1.1	OVH SAS	217.182.198.203
Cogent Comm.	66.28.0.61	OneDNS	117.50.10.10
Comodo Secure DNS	8.26.56.26	OpenDNS Home	208.67.222.222
Control D	76.76.2.0	OpenNIC	51.77.149.139
Cyberlink AG	89.249.44.73	Probe Networks	82.96.65.2
DNS for Family	94.130.180.225	Quad101	101.101.101.101
DNS.WATCH	84.200.69.80	Quad9	9.9.9.9
DNSForge	176.9.93.198	ScanPlus GmbH	212.211.132.4
DNSpai	101.226.4.6	Swisscom	195.186.4.110
Deutsche Telekom	194.25.0.68	TEFINCOM S.A.	103.86.96.100
Dyn	216.146.35.35	TREX	195.140.195.21
Fortinet	208.91.112.53	Vodafone	195.27.1.1
Freenom World	80.80.80.80	xTom	77.88.8.8
GCore Free	95.85.95.85	114DNS	114.114.114.114
Google DNS	8.8.8.8		

* https://stats.labs.apnic.net/rvrs, https://www.dnsperf.com, https://publicdnsserver.com/fastest/, https://www.publicdns.xyz

A.1 Ingress RL

Many DNS server implementations provide native functions for rate limiting responses to clients based on IP address/prefix. This is regardless of whether a DNS server works in the authoritative or recursive mode, and whether client requests hit cache or not. Ingress RL can be applied at the granularity of response type. For instance, BIND allows the configuration of individual limits for responses with RCODE as NOERROR, NXDOMAIN, error, etc. We consider two query patterns that generate two typical types of response:

- **P1 (WC):** pseudo-random names that trigger wildcard-synthesized NOERROR responses;
- **P2 (NX):** pseudo-random names that trigger NXDOMAIN responses.

We use a standard DNS performance testing tool `dnsperf` on our probes to measure possible ingress rate limits implemented by the resolvers in our dataset. It estimates a DNS server's query processing throughput in a self-pacing manner, up to a maximum QPS specified via the `-Q` option. One issue with `dnsperf` is that it sends requests in bursts, which causes the estimated QPS to fluctuate drastically and fail to converge for some DNS servers implementing RL. Rather than setting a fixed large value for `-Q` and relying solely on `dnsperf` to find out the actual QPS, we dynamically adjust the value until we observe that the measured QPS remains stable. The QPS estimation counts only NOERROR or NXDOMAIN responses and excludes SERVFAIL or REFUSED responses. Each measurement lasts for 30 seconds by default, which we find sufficient to produce stable results in most cases, and we prolong it to 60 seconds if the measured QPS does not converge within 30 seconds. We start with an initial probing rate of 100 QPS and proceed with a binary search for a possible rate limit up to 5000 QPS, which is an upper bound we set to avoid stressing the resolvers.

We set the TTL of our resource records (including wildcard records for positive answers and SOA records for negative answers) as 600 seconds so that they will not stay in the resolvers' cache for an unnecessarily long time. While the two query patterns can be exploited to bypass caching with almost infinitely many names (only bounded by the maximum length of a DNS name), we restrict the number of unique query names used in our measurements to match the probing QPS and let `dnsperf` loop over the names; as a result, most requests are answered from resolvers' cache. This avoids wasting resources of the measured resolvers and allows us to isolate the impact of resolvers' egress RL and other potential limiting factors on cache-missed pending requests that create outgoing queries.

A.2 Egress RL

Measuring egress RL is more challenging because large public resolvers normally distribute client requests across multiple egresses and sometimes do so even for outgoing queries triggered by a single request. We also observe that some resolvers replicate queries to multiple egresses, and some of them change their egress set even for different requests from the same client. In general, large resolvers can implement customized and complex load balancing strategies that

```
>zone target-domain @ 127.0.0.1
// Amplification instance 1
15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r1-1     CNAME     15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r2-1
15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r2-1     CNAME     15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r3-1
...
15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r15-1    CNAME     15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r16-1
15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r16-1      A       127.0.0.1

// Amplification instance 2
15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r1-2     CNAME     15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r2-2
15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r2-2     CNAME     15.14.13.12.11.10.9.8.7.6.5.4.3.2.1.r3-2
...
```

(a) CNAME chain × QNAME minimization.

```
>zone attacker-com @ 127.0.0.2
// Amplification instance 1                  |  // Amplification instance 2
q-1          NS      ns-a1-1                 |  q-2          NS      ns-a1-2
q-1          NS      ns-a2-1                 |  q-2          NS      ns-a2-2
q-1          NS      ns-a3-1                 |  q-2          NS      ns-a3-2
...                                          |  ...
ns-a1-1      NS      ns-t11-1.target-domain  |  ns-a1-2      NS      ns-t11-2.target-domain
ns-a1-1      NS      ns-t12-1.target-domain  |  ns-a1-2      NS      ns-t12-2.target-domain
ns-a1-1      NS      ns-t13-1.target-domain  |  ns-a1-2      NS      ns-t13-2.target-domain
...                                          |  ...
ns-a2-1      NS      ns-t21-1.target-domain  |  ns-a2-2      NS      ns-t21-2.target-domain
ns-a2-1      NS      ns-t22-1.target-domain  |  ns-a2-2      NS      ns-t22-2.target-domain
ns-a2-1      NS      ns-t23-1.target-domain  |  ns-a2-2      NS      ns-t23-2.target-domain
...                                          |  ...
```

(b) Fan-out × fan-out with large NS RRSets

Figure 12. Examples of the zone files for query patterns **P3** and **P4** with compositional amplification effects.

require dedicated measurement methods to analyze. Applying the previous two query patterns **P1** and **P2** to egress RL measurements may require high request rates that put undue load on the measured resolvers, and this approach does not work if a resolver's egress rate limit is higher than its ingress rate limit, which we find to be the case for many resolvers in our dataset.

To this end, we adopt two additional query patterns that leverage amplification to trigger more resolver queries than the requests sent by our probes [22].

- **P3 (CQ):** a predefined name that initiates a chain of CNAME records formed by names with many labels.
- **P4 (FF):** a predefined name that owns a large set of NS records, each of which owns yet another large set of NS records.

P3 causes a resolver to chase a CNAME chain while performing query name minimization [12]. **P4** causes a resolver to recursively look up excessive nameserver names. Figure 12 provides examples of the zone files for both patterns.

There are several additional changes in our measurement methods. Since measuring egress RL requires a resolver to generate queries, we should evade caching as much as possible. For each amplification query pattern we set up 5000 distinct instances that can be queried in parallel. We also set the resource records' TTL as 1 so that they can be quickly evicted from resolvers' cache and re-queried during our measurements. To reduce the load put on public resolvers, we shorten the duration of each measurement to 15 seconds.

We cannot rely on `dnsperf` to estimate a resolver's egress QPS, as it has no visibility into the queries generated by resolvers. Moreover, the theoretical message amplification factor of a query pattern is not always achievable by a resolver, and there is no consistent relationship between how many requests the resolver receives and how many queries it sends. We instead calculate egress QPS from the query log at our authoritative nameservers. For each public resolver, we single out its egress that sends the most queries to our authoritative servers, calculate the QPS within a 1-second sliding window over a measurement window, and report the most stable value that lasts over consecutive windows. We conduct the measurements sequentially, one at a time, and wait 60 seconds between them, in order to obtain clean data for each measurement without the complexity to isolate resolvers based on their egresses from the query log.

The probing process for egress rate limits is similar to that for ingress rate limits. We start with a probing request rate

of 10 QPS and proceed in a binary-search style. A possible egress rate limit is reached if the measured egress QPS stops to increase with the probing request rate. We observe that once this happens, some resolvers starts allocating more egresses to process our requests.

It should be noted that the measurement results for egress RL are best-effort and not as reliable as ingress RL. This is due to the existence of other potential limiting factors implemented by resolvers: the limited number of pending requests per client, the overall number of outstanding queries, the depriorization of authoritative servers that return SERVFAIL or REFUSED responses, etc. We leave the design of more accurate measurement methods for future work.

B MOPI-FQ Scheduling

Figure 13 describes the pseudo code implementing our MOPI-FQ scheduler. The syntax is in the C/C++ style: for example, we use `struct` to define data structures, the `*` symbol to represent pointer types, and `container<elem>` to represent abstract containers where `elem` is the type of the underlying elements. Constants are represented in the upper case, e.g., `MOPI_CAPACITY`.

Our specification assumes equal shares for all input sources, and we will show how this can be easily extended to unequal shares (Appendix B.1.3). We also omit non-essential details such as initialization functions, configurable parameters, error handling, and various performance optimizations.

In this section, we first elaborate on MOPI-FQ's design with a complexity analysis (Appendix B.1) and then prove its fairness property (Appendix B.2).

B.1 Functionality and Complexity

The backbone of MOPI-FQ is a pool (`out_pool`) of linkable entries (`q_entry`) that can be allocated to per-output queues (`poq`). The available entries themselves form a list whose head is stored in `avail_slots`, which is updated every time an entry is allocated or recycled. The scheduler also maintains a collection of queuing state (`poq_state`) for each active output channel[2], which is identified by a destination address, using a standard unordered associative array or hash table (`poq_tracker`). Each queue is logically divided into multiple scheduling rounds each containing messages from unique input sources (cf. Figure 7c). We track the boundaries between rounds with pointers stored in a ring buffer (`round_tails`). To decide which round to insert messages from an input source, it is also necessary to track individual sources' latest rounds (`source_latest`). Other important state needed for correct implementation includes the queue's head and depth, its current round to output, and the latest round scheduled (i.e., the highest among all sources). In addition, the scheduler associates rate limiters implemented as token buckets with all active outputs (`rate_lim`). It also maintains a sequence of active channels scheduled for output (`out_seq`), which are ordered by the arrival time of the front messages in their queues and the time when congested channels become available.

MOPI-FQ exposes two major operations for scheduling, `enqueue` and `dequeue`, which rely on a collection of auxiliary functions. For efficiency, these two operations should be executed by different threads in synchrony. Here we focus on the essential algorithmic aspects without diving into optimized synchronization design. Unlike FIFO queues or queuing structures used for classic FQ scheduling [13, 20, 45], MOPI-FQ does not always push messages to one end of a queue nor pop messages from the other end. Messages can be inserted or removed anywhere in the corresponding queue depending on the scheduler's state. This will become clear below as we explain the functions in greater details.

In what follows, let n be the number of active input sources and m be the number of active output channels.

B.1.1 Space Complexity.

The main entry pool `out_pool` has a fixed capacity (e.g., one million), which should be substantially larger than the capacity (`MAX_POQ_DEPETH`) of individual queues in order to accommodate sufficient queues. Each of the three containers `poq_tracker`, `rate_lim`, and `out_seq` has exactly m elements. The latter two containers have fixed-size elements. As for the other, all data fields in its element `poq_state`, including the small ring buffer `round_tails` (e.g., with a size `MAX_ROUND` equal to 5), have fixed sizes except `source_latest`, whose size equals to the number of active input sources corresponding to the output: an entry is created as soon as the first message with the source and destination addresses is enqueued, and the entry is removed as soon as the last such message is dequeued. Hence, the aggregated size of all `source_latest`s is $s \le q :=$ `total_depth` (i.e., the number of queued messages), and MOPI-FQ has a space complexity of $O(m+q)$.

For `DCC` as a whole with MOPI-FQ, per-client anomaly monitoring and pre-queue policing, its overall space complexity is $O(n+m+p)$ where $p \ge q$ is the number of pending resolver queries waiting for answers. This is asymptotically the same as the state maintained by a vanilla DNS resolver (see Table 1).

B.1.2 Time Complexity.

We first analyze the `enqueue` function, which inserts a message at the right position in the queue such that the message source's fair share of the output link is maintained. In particular, the message should be placed at the next round of the source's latest round currently scheduled in the corresponding queue. This involves updating different internal data structures and dealing with

[2] A channel is active if at least one message to the output destination is in the corresponding queue.

```
struct message {                                    |  message dequeue() {
  addr    get_source();                             |    dst = peek_next_available_output();
  addr    get_destination();                        |
  /* the data is store somewhere else */           |    if ( dst == NULL )
  bytes*  raw_data;                                 |      return FAIL_NO_DATA_OR_ALL_CONGESTED;
}                                                   |    else
                                                    |      entry = remove_head_poq( dst );
struct mopi_fq {                                    |
  struct q_entry {                                  |    return entry.msg;
    q_entry*    next;                               |  }
    q_entry*    prev;                               |
    message     msg;                                |  int enqueue( msg ) {
    /* when the message is enqueued */              |    src = msg.get_source();
    time        arr_time;                           |    dst = msg.get_destination();
  };                                                |
                                                    |    crt_r = get_poq_current_round( dst );
  struct poq_state {                                |    lat_r = get_poq_latest_round( dst );
    int                     queue_depth;            |    src_nxt = get_src_next_round ( dst, src );
    q_entry*                queue_head;             |
    ring_buffer<q_entry*>   round_tails;            |    if ( src_nxt == crt_r + MAX_ROUND )
    int                     current_round;          |      return FAIL_CLIENT_OVERSPEED;
    /* latest round for the entire poq */           |
    int                     latest_round;           |    if ( get_poq_depth( dst ) == MAX_POQ_DEPTH
    /* latest rounds for individual sources */      |         && src_nxt >= lat_r )
    unorder_map<addr,int>   source_latest;          |      return FAIL_CHANNEL_CONGESTED;
  };                                                |
                                                    |    if ( total_depth == MAX_CAPACITY
  /* pre-allocated queue storage */                 |         && src_nxt >= lat_r )
  array<q_entry>                  out_pool;         |      return FAIL_QUEUE_OVERFLOW;
  /* list of available entries at  any time */      |
  q_entry*                        avail_slots;      |    activate_output_if_new( dst );
  int                             total_depth;      |    append_poq_round( dst, src_nxt, msg );
  unorder_map<addr, poq_state>    poq_tracker;      |
  unorder_map<addr, token_bucket> rate_lim;         |    return SUCCESS;
  /* where logarithmic costs come from */           |  }
  ordered_map<time, addr>         out_seq;          |}
```

Figure 13. The pseudo code for MOPI-FQ scheduling described in a C-style syntax. The auxiliary functions used by the `enqueue` and `dequeue` functions are self-explaining and further analyzed in Appendix B.1.

several corner cases. We analyze the involved auxiliary functions in the order of their appearance.

The first 5 getter functions, from `get_source()` through `get_poq_depth()` to `get_src_next_round()`, are simple and self-evident. Only `get_src_next_round()` requires accessing a hash table `source_latest` which takes constant time. If the message's output destination is currently not active, `activate_output_if_not_exists()` will create and insert new items to `poq_tracker` and `rate_lim` in constant time. It also inserts the destination address to `out_seq` according to the message's arrival time, which has a $O(log(m))$ cost as explained below. If the current queue is not full and the current round of the message's source does not exceed the maximum round for the destination, the scheduler will insert the message at the end of the expected next round with the `append_poq_round` function, which entails accessing `poq_tracker`, `source_latest`, and `round_tails` and updating at most three `q_entry`s, all in constant time. In summary, the time complexity of `enqueue` is $O(log(m))$.

The `dequeue` function takes messages out from active queues. The central question here is which queue to choose for each dequeue operation. Simple round robin over the

queues will break the arrival order of messages across different queues and so increase queuing delay. The same holds for random selection or any scheduling policy that sticks to a queue until it becomes empty or the channel becomes congested. To this end, we introduce an ordered map data structure `out_seq` to track the scheduling order for queues based on the arrival time of their head messages. The scheduler always chooses the queue indicated by the front element of `out_seq` subject to the availability of the corresponding output channel. Since the arrival order of messages across different queues can be arbitrary, the elements of `out_seq` should be relocated whenever necessary with $O(log(m))$ costs. Another technicality arises from channel congestion, in which case the scheduler must skip the congested queue and retry it later. We handle the situation by relocating the congested channel's corresponding element in `out_seq` according to a predicated future time when the channel becomes available again. Putting everything together, the time complexity of `dequeue` is also $O(log(m))$.

Figure 15 illustrates the state of MOPI-FQ, the queues and `out_seq` in particular, with a series of dequeue operations.

B.1.3 Unequal Shares. In the algorithm described so far, the scheduler enqueues exactly one message in each round for every source sending to the same destination. If sources have unequal shares, we can adjust the numbers of messages that they are allowed for each round accordingly. To implement this, we track per-source *round quota*, which is initialized to a normalized source share, in each `poq_state`. This additional state can be merged with `source_latest` to save space and lookup time. The quota of a source will be decremented by 1 every time a message from the source is enqueued to its latest round. The quota will be reset to its maximum when it reaches zero and hence the source's latest round will advance by 1. The update of a source's round quota for an output channel is independently of other sources for the same channel. Except this minor modification, all other operations of MOPI-FQ remain unchanged.

B.2 Fairness

We refer to the problem formulated in Section 4.1 as multi-output fair queuing (MO-FQ). It aims to simultaneously allocate all output channels among sources according to their predefined shares. In particular, we require the allocation to be *max-min fair* (MMF) as formalized below.

Definition B.1 (Output Allocation). For an output channel j and its set of active sources $\mathcal{S}_j$ (with loss of generality, let $\mathcal{S}_j := [1, k]$ where $k \leq n$), an allocation is defined as a vector $\mathbf{A} := \langle a_{1j}, a_{2j}, \ldots \rangle \subset \mathbb{R}^{|\mathcal{S}_j|}$ subject to $0 \leq a_{ij} \leq r_{ij}, \forall i \in [1, k]$ and $\sum_{i=1}^{k} a_{ij} \leq C_j$, where r_{ij} is source i's message sending rate towards destination j for which the corresponding output channel has capacity C_j.

Clearly, an allocation is subject to the demands of individual sources as well as the output channel's capacity. If the aggregate demand is below the channel capacity, all sources can send at their desired rates and there is no congestion. We are interested in how fair sharing is achieved in the case of congestion. This will be the focus of the rest of this section. For ease of presentation, we use $\mathbf{A}_i$ to denote the i-th element of $\mathbf{A}$.

Definition B.2 (MMF Allocation). An allocation $\mathbf{A}$ is max-min fair if and only if for any other allocation $\mathbf{B}$, if there exists $\mathbf{B}_x > \mathbf{A}_x$ then there must also exist $\mathbf{B}_y < \mathbf{A}_y \leq \mathbf{A}_x$.

Intuitively, we cannot increase an MMF allocation's element A_i to obtain a new allocation, without decreasing another element A_j already smaller than or equal to A_i. Max-min fairness is among the most adopted notions of fair resource sharing in computer systems and networks.

The definition above assumes equal shares for all sources. Extending it to a weighted version is straightforward [48]. Recall that π_i is the share of source i. An MMF allocation $\mathbf{A}$ w.r.t. the shares (or weights) $\{\pi_i\}_{1 \leq i \leq k}$ is an allocation such that for any allocation $\mathbf{B}$ if there exists $\mathbf{B}_x > \mathbf{A}_x$ then there must also exist $\mathbf{B}_y/\pi_y < \mathbf{A}_y/\pi_y \leq \mathbf{A}_x/\pi_x$.

An interesting question is whether an MMF allocation exists for any MO-FQ instance. In general, the existence of MMF vector over a set depends on whether the set is convex and compact [48]. For MO-FQ this is indeed the case: its linear constraints (see Definition B.1) always define such a feasible set. Moreover, there is always a unique MMF allocation for any MO-FQ instance. This conclusion extends to the weighted version by Theorem 1 of the seminal work of Radunovic and Boudec [48].

The theorem below states MOPI-FQ's fairness property.

Theorem B.1 (MOPI-FQ Fairness). *MOPI-FQ achieves a unique MMF allocation U_j for each active output channel j among its active input sources $\mathcal{S}_j$.*

From a high level, MOPI-FQ maintains a separate flattened calendar queue for each active output channel. The actual scheduling for fairness takes place at the enqueue operations: messages are inserted in a way that for each scheduling round no source can have more messages allocated than its (normalized) share. The messages in a queue are simply dequeued in the order determined by the enqueue operations. Every queue is fairly scheduled independently in this way. One technicality here is that the queues compete for the shared resource pool with their real capacities (no greater than the nominal capacity `MAX_POQ_DEPTH`) dynamically adjusted. If a queue's capacity is too small, fairness can be affected (e.g., slow senders may be blocked by fast senders) depending on the workload, traffic pattern, and the number of active entities. Since this issue is common to any FQ designs, we do not discuss further the subtleties therein. We instead assume that once created each queue is

guaranteed a minimum capacity that can accommodate all its active senders given the channel capacity without affecting fair scheduling. With this consideration, we can then analyze each output channel independently and we prove the above theorem as follows.

Proof. We aim to derive an allocation $\mathbf{A}$ of an output channel's capacity C by MOPI-FQ and show that it is MMF. Consider a period of time T during which a fixed number of sources $[1, k]$ send messages to a destination at their respective constant rates r_i. The period T should be large enough to allow every source to send full messages, i.e., their message counts are integers rather than fractions.

Let us consider the sources' output traffic volumes during T. Because of the differences in sending rates, it is likely that not all sources' messages appear in the queue at any point in time. Whenever a message from a not-in-queue source arrives, the scheduler will insert the message to the output channel's current round, evicting out a message of some other source from the latest round if the queue is full. Hence, despite the rate differences and queue fullness, a source can always have its messages scheduled for output at its allocated rate. The number of messages from source i that ever arrive at the scheduler but not necessarily sent out is simply $r_i T$. The overall number of messages MOPI-FQ sends out is CT. The way it sends messages round by round corresponds exactly to the Water Filling (WF) procedure [48]. Figure 14 gives an illustration.

Following WF, we can calculate the number of messages output for every source during T and hence their allocated rate a. Assuming identical shares, this can be analytically derived as $a_i := f(C, r_i, \mathcal{R} := \{r_i\}_{1 \le i \le k})$, where $f(\cdot)$ is a recursive function defined as follows:

$$f(C, r, \mathcal{R}) = \begin{cases} r, & \text{if } r \le \frac{C}{|\mathcal{R}|}; \quad (1) \\ \frac{C}{|\mathcal{R}|}, & \text{if } r_* > \frac{C}{|\mathcal{R}|}, \text{ s.t. } r_* := \min(\mathcal{R}); \quad (2) \\ r_* \quad + & \text{otherwise, s.t. } C' := C - |\mathcal{R}| \cdot r_*, \\ f(C', r - r_*, \mathcal{R}'), & \mathcal{R}' := \{r_i - r_* | r_i \in \mathcal{R} \setminus \{r_*\}\}. \quad (3) \end{cases}$$

In case (1), a source's sending rate is no greater than the average rate that can be allocated in theory, and therefore its demand can be safely satisfied (e.g., source 2 in Figure 14). In case (2), even the least demanding source requires more than the average rate and therefore everyone is assigned the average rate. In case (3), every source is allocated the lower-than-average rate of the least demanding source, and the remaining bandwidth is allocated again in the WF manner among the unsatisfied sources. It is easy to see that $f(\cdot)$ is a monotonically increasing function on r for a given configuration C and $\mathcal{R}$. Moreover, whenever the recursions in case (3) terminate at case (1), we have $f(r) = r$ up to a point $S > C/|\mathcal{R}|$. We call S a *satisfaction threshold*, which is uniquely determined for a given configuration. Beyond the satisfaction threshold, the recursions will terminate at case (2) and $f(r) = M$ for some constant $M \in (S, C]$. We can

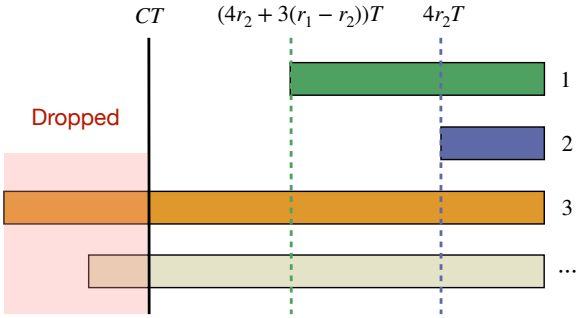


Figure 14. Illustration of the water filling (WF) procedure for max-min fair allocation in the context of MOPI-FQ. The horizontal bars represent messages arriving at the scheduler from different sources within a time period T. The vertical lines indicate the aggregate numbers (marked on the top) of messages from all sources represented by the partial bars to the right. All extra messages beyond the output channel's capacity C are dropped.

then rewrite the function as

$$f_{C,\mathcal{R}}(r) = \begin{cases} r, & r \le S; \\ M, & r > S. \end{cases}$$

Now we examine $\mathbf{A} := \langle f_{C,\mathcal{R}}(r_1), f_{C,\mathcal{R}}(r_2), \ldots, f_{C,\mathcal{R}}(r_k) \rangle$. Let $\mathcal{S}$ and $\mathcal{U}$ be the set of sources that are satisfied and unsatisfied under $\mathbf{A}$, respectively. That is, $\mathcal{S} := \{i \mid r_i \le S, 1 \le i \le k\}$ and $\mathcal{U} := \{i \mid r_i > S, 1 \le i \le k\}$. Let $\mathbf{B}$ be any alternative allocation. Note that it is impossible to have $\mathbf{B}_i > \mathbf{A}_i$ for any $i \in \mathcal{S}$ because of the individual allocation constraints (Definition B.1). Let $\mathbf{B}_i > \mathbf{A}_i$ for some $i \in \mathcal{U}$. Because of the aggregate allocation constraint $\sum_{i=1}^{k} a_i = C$, we know there must be $\mathbf{B}_j < \mathbf{A}_j$ for some j. If $j \in \mathcal{U}$, we have $\mathbf{A}_j = M = \mathbf{A}_i$, otherwise $j \in \mathcal{S}$ and we have $\mathbf{A}_j < M = \mathbf{A}_i$. Hence, by Definition B.2 $\mathbf{A}$ is an MMF allocation. The case of unequal shares can be proved in the same way.

□

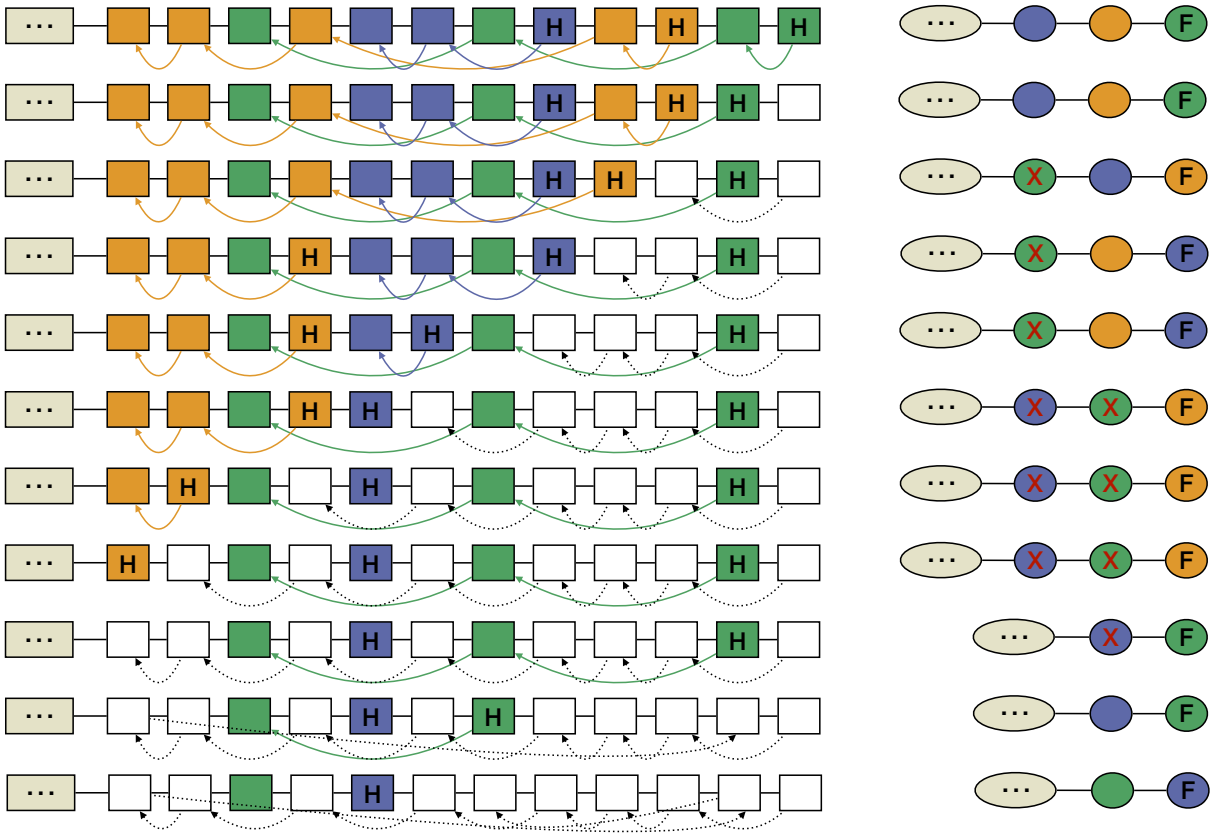


Figure 15. Illustration of MOPI-FQ with consecutive dequeue operations. On the left are per-output queues allocated over a flat array of linkable entries with the queue heads marked as 'H'. The colored entries contain messages for different outputs and the white entries linked by dashed arrows are recycled for future allocation. On the right is the output sequence with the front element marked as 'F' and congested outputs marked as 'X'. We assume that the currently queued messages arrive in the order from right to left and that no new message is enqueued. The scheduler dequeues messages according to their arrival order while skipping and re-scheduling messages for congested output channels.