# Controlling High Bandwidth Aggregates in the Network (Extended Version)

Ratul Mahajan, Steven M. Bellovin, Sally Floyd,
John Ioannidis, Vern Paxson, and Scott Shenker

AT&T Center for Internet Research at ICSI (ACIRI)
and AT&T Labs Research*

July 13, 2001- DRAFT

## Abstract

The current Internet infrastructure has very few built-in protection mechanisms and is therefore vulnerable to attacks and failures. In particular, recent events have illustrated the Internet's vulnerability to both denial of service (DoS) attacks and flash crowds in which one or more links in the network (or servers at the edge of the network) become severely congested. In both flash crowds and DoS attacks the congestion is not due to a single flow, nor to a general increase in traffic, but to a well-defined subset of the traffic – an *aggregate*. This paper discusses mechanisms for detecting and controlling such high bandwidth aggregates. Our approach involves both a local mechanism for detecting and controlling an aggregate at a single router, and a cooperative *pushback* mechanism in which a router can ask adjacent routers to control an aggregate upstream. These mechanisms, while certainly not a panacea, provide relief from flash crowds and flooding-style DoS attacks.

## 1 Introduction

In the current Internet, when a link is persistently overloaded all flows traversing that link experience significantly degraded service over an extended period of time. Protection mechanisms that could minimize the effects of such congestion would greatly increase the reliability of the Internet infrastructure. Persistent overloads can arise for several reasons, and each requires a different form of protection.

First, persistent overloads can result from a single flow not using end-to-end congestion control and continuing to transmit despite encountering a high packet drop rate. There is a substantial literature [DKS89, LM97, SSZ98, MF00] on mechanisms to cope with such *ill-behaved* flows (where, by *flow*, we mean a stream of packets sharing IP source and destination addresses, protocol field, and source and destination port numbers). Second, as was seen on the transatlantic links a few years ago, persistent overloads can also be due to a general excess of traffic [ILS99]. While better active queue management techniques [FJ93] may be of some use, there is little one can do to protect inadequately provisioned links.

However, even when all links are adequately provisioned, and all flows are using conformant end-to-end congestion control (or, equivalently, all routers have mechanisms to protect against ill-behaved flows), persistent congestion can still occur. Two examples of this are *denial of service* attacks (DoS) and *flash crowds*.

DoS attacks occur when a large amount of traffic from one or more hosts is directed at some resource of the network (*e.g.*, a link or a web server). This artificially high load denies or severely degrades service to legitimate users of that resource. The current Internet infrastructure has few protection mechanisms to deal with such DoS attacks, and is particularly vulnerable to distributed denial of service attacks (DDoS), in which the attacking traffic comes from a large number of disparate sites. A series of DoS attacks occurred in February 2000 to considerable media attention, resulting in higher packet loss rates in the Internet for several hours [Gar00]. DoS attacks have

*Ratul Mahajan is from University of Washington (work done while at ACIRI); Steven M. Bellovin and John Ioannidis are from AT&T Labs Research; Sally Floyd, Vern Paxson and Scott Shenker are from ACIRI. The email addresses are ratul@cs.washington.edu, smb@research.att.com, floyd@aciri.org, ji@research.att.com, vern@aciri.org, and shenker@aciri.org

also been directed against network infrastructure rather than against individual web servers [MVS01].

Flash crowds occur when a large number of users try to access the same server simultaneously. Apart from overloading at the server itself, the traffic from such flash crowds can overload the network links and thereby interfere with other, unrelated users on the Internet. For example, degraded Internet performance was experienced during a Victoria's Secret webcast [Bor99] and during the NASA Pathfinder mission.

While the intent and the triggering mechanisms are quite different for DoS attacks and flash crowds, from the network's perspective these two cases are quite similar. The persistent congestion is not due to a single well-defined flow, nor is it due to an *undifferentiated* overall increase in traffic. Instead, there is a particular set of packets causing the overload, and these offending packets – which we will call an *aggregate* – are spread across many flows. The resulting *aggregate-based congestion* cannot be controlled by conventional per-flow protection mechanisms. In this paper we propose control mechanisms that work on the granularity of aggregates. These Aggregate-based Congestion Control (ACC)[1] mechanisms fall between the traditional granularities of per-flow control (which looks at individual flows) and active queue management (which does not differentiate between incoming packets).

More specifically, an *aggregate* as a collection of packets from one or more flows that have some property in common. This property could be anything from destination or source address prefixes to a certain application type (streaming video, for instance). Other examples of aggregates are TCP SYN packets and ICMP ECHO packets. An aggregate could be defined by a property which is very broad, such as TCP traffic, or very narrow, such as HTTP traffic going to a specific destination.

To reduce the impact of congestion caused by such aggregates, we propose two related ACC mechanisms. The first, *local* aggregate-based congestion control (Local ACC), consists of an *identification* algorithm used to identify the aggregate (or aggregates) causing the congestion, and a *control* algorithm that then reduces the traffic sent by this aggregate to a reasonable level. As we will discuss, there are many situations in which local aggregate-based congestion control would, by itself, be quite effective in preventing aggregates from significantly degrading the service delivered to other traffic.

In some cases, however, it may be beneficial to control the aggregate closer to its source(s). The second ACC mechanism, *pushback*, allows a router to request adjacent upstream routers to rate-limit traffic corresponding to the specified aggregates. Pushback can prevent upstream bandwidth from being wasted on packets that are only going to be dropped later on in the network. In addition, for a DoS attack, if the attack traffic is concentrated at a few incoming links upstream, then pushback can be effective in protecting other traffic within the aggregate from the attack traffic.

ACC mechanisms are intended to protect the network from persistent and severe congestion due to rapid increases in traffic from one or more aggregates. We envision that these mechanisms would be invoked rarely, and we emphasize that these mechanisms are not substitutes for adequately provisioning links or for end-to-end congestion control. Nonetheless, we believe that introducing control mechanisms at this new level of granularity – aggregates – may provide important protection against flash crowds, DoS attacks, and other forms of aggregate-based congestion.

The organization of this paper is as follows. Section 2 gives an overview of ACC. In Section 3 we describe some related work done to tackle the problem of DoS attacks and flash crowds. Section 4 describes the local component of ACC in more detail. We discuss the pushback mechanisms in detail in Section 5, followed by some simulation results in Section 6. Section 7 evaluates the advantages and disadvantages of pushback, and discusses of several issues related to ACC.

## 2 Overview of ACC

This section gives an overview of our two proposed ACC mechanisms: Local ACC, in which a router deals with sustained overload by itself, and pushback, an extension to Local ACC in which a router signals other routers upstream to control a particular aggregate on its behalf. They are then explored in detail in §4 and §5.

We can think about an ACC mechanism running in a router (or possibly in an attached device) as consisting

[1] We note that the term "ACC" has been used in different contexts to denote "Active Congestion Control" and "ACK Congestion Control".

of the following sequence of decisions:

1. Am I seriously congested?
2. *If so,* can I identify an aggregate responsible for an appreciable portion of the congestion?
3. *If so,* to what degree do I limit the aggregate? Do I also ask upstream routers to limit the aggregate?
4. *And if I decide to deal with it,* when do I stop? When do I ask upstream routers to stop?

Each of these questions requires an algorithm for making the decision. Each is also a natural point to inject *policy* considerations into the decision making. The space of possible policies (e.g., who to treat better than whom, who to trust, what applications should get at most how much bandwidth, how to perhaps incorporate past history) is very large, and we do not attempt to explore it in this paper. Instead, we assume simple policies in order to focus on developing and understanding the mechanisms.

To answer the question "am I seriously congested?" our proposed mechanism periodically monitors each queue's packet drop rate to see if it exceeds a (policy-specific) threshold. A small jitter will be applied to the monitoring interval, both to avoid synchronization effects [FJ94] and to resist an attacker intent on predicting the response patterns of ACC in the presence of a DoS attack. Maintaining some longer-time history of the packet drop rate could help to detect the intermittent periods of heavy congestion that could result from DoS attacks.

When serious congestion is detected, the router attempts to identify the aggregate(s) responsible for the congestion. Identifying the offending aggregate(s) is a tricky problem to solve in a general fashion, for three reasons. First, the overload may be chronic, due to an under-engineered network, or unavoidable, *e.g.* as a shift in load caused by routing around a fiber cut. These lead to *undifferentiated congestion* not dominated by any particular aggregate. Second, there are many possible dimensions in which traffic might cluster to form aggregates: by source or destination address (e.g., a flash crowd attempting to access a particular server, or its replies back to them), address prefix (a flooding attack targeting a site or a particular network link), or a specific application type (a virulent worm that propagates by email, inadvertently overwhelming other traffic). Third, if the congestion is due to a DoS attack, the attacker may vary their traffic as much possible to complicate the router's detection of high-bandwidth aggregates.

We propose that routers identify aggregates by applying clustering to a sample of their high volume traffic, which they can attain by sampling drops from a randomized discard mechanism such as RED [FJ93]. We discuss the specifics of a possible clustering algorithm in Section 4.1. Note that if the clustering algorithm fails to find a narrowly defined aggregate, we conclude that the congestion is undifferentiated and take no action.

That the high bandwidth aggregates are in fact responsible for congestion is an assumption in our scheme. There are links in the network that are dominated by a particular aggregate(s), in the normal case. The ISP can use policy if it wants to protect such aggregates, resulting in ACC mechanisms looking for other aggregates or rate-limit these high bandwidth aggregates only when they exceed their policy defined limits. A possibility we have not explored yet is the use of history for identification.

Analogous to attack signature for describing various forms of malicious activities, we use the term *congestion signature* to denote the aggregate(s) identified as causing congestion. It is important to note that when constructing congestion signatures, the router does not need to make any assumptions about the malicious or benign nature of the underlying aggregate (which may not be possible in the face of a determined attacker). If the congestion signature is too broad, such that it encompasses additional traffic beyond that in the true high-bandwidth aggregate, then we refer to the signature as incurring *collateral damage*. In this case, restricting the bandwidth of the identified aggregate can increase the already high packet drop rate seen by the legitimate traffic within the aggregate, while easing the burden on the legitimate traffic that did not fall within the aggregate. Narrowing the congestion signature, and thus minimizing collateral damage, is one of the goals of our approach.

We now turn to the question of to what degree the router should limit an aggregate's rate, and the mechanism by which it does so. We argue that there is no useful, policy-free equivalent to max-min fairness when applied to aggregates; no one would recommend for best-effort traffic that we give each destination prefix or application type an equal share of the bandwidth in a time of high congestion. Instead, the goal is to rate-limit the identified aggregate sufficiently to protect the other traffic on the

link from the congestion caused by the aggregate. Here, "sufficiently" is chosen such that, for all the aggregates we are currently rate-limiting, we restrict them so that their total arrival rate plus that of other traffic arriving at the queue maintains an ambient drop rate in the output queue of at most the configured target value (§4.2).

A more Draconian measure, like completely shutting off or imposing a very low bandwidth limit for identified aggregates, is not taken because of two reasons. First, the aggregate can be a flash crowd. Second, even if the aggregate is from a DoS attack, the congestion signature of the attack traffic will usually contain some innocent traffic too.

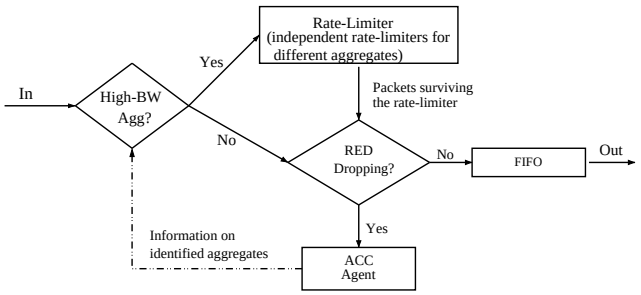


Figure 1: **The rate-limiting architecture.**

Figure 1 shows the rate-limiting architecture. There is a filter at the entry to the regular FIFO output queue. When a packet arriving at the output queue is identified as a member of the aggregate, it is passed to the rate-limiter, which decides whether to drop the packet or add the packet to the output queue. Once past the rate-limiter, the packet loses any identity as a member of the aggregate. Because packets that pass the rate-limiter are treated as regular arrivals to the output queue, rate-limiting cannot result in preferential treatment for the packets in the aggregate. In contrast, the rate-limited aggregates would get preferential treatment if they were allocated a fixed bandwidth share irrespective of the general congestion levels at the output queue.

We next turn to the possibility of using pushback to control an aggregate. But first, a more detailed description of pushback. Pushback works by the congested router requesting its adjacent upstream routers to rate-limit traffic corresponding to a given aggregate. This *pushback message* is only sent to immediate upstream routers that send the bulk of the traffic for that aggregate.[2] Routers receiving these messages can recursively propagate pushback upstream (closer to the sources). Throttling the high-bandwidth aggregate closer to the source prevents bandwidth being wasted on packets that are destined to be dropped later on in the network. In addition, by concentrating the rate limiting on the upstream links that carry the bulk of the traffic within the aggregate, pushback can restrict the degree to which a DoS attack denies service to legitimate traffic, since legitimate traffic on the other links will not suffer rate-limiting.

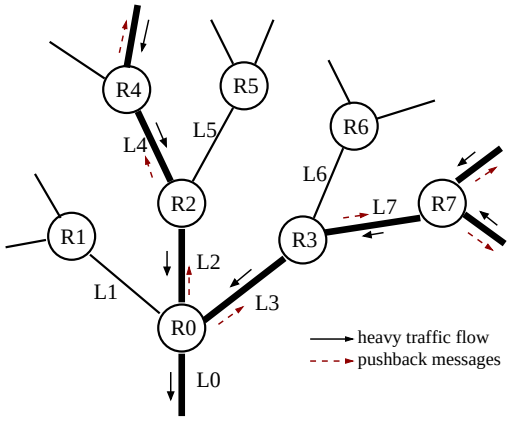


Figure 2: **Illustration of pushback**.

For example, consider the partial network topology shown in Figure 2. The paths used by most of the traffic in the high-bandwidth aggregate are shown in bold, and the direction is as indicated by the arrows. The destination of the aggregate is some host $D$ (not shown) which is reached using L0. Thus, most of the traffic covered by the attack signature comes from links L2 and L3, with very little coming from link L1.

Assume that the link L0 in Figure 2 is highly congested, and as a result R0 identifies the high bandwidth aggregate. By using Local ACC, R0 can protect the traffic not going to $D$. But with Local ACC only, traffic going from L1 to $D$ is not protected; pushback is needed to protect that traffic. Pushback in this case will propagate from R0 to R2 and R3. Subsequently, pushback will propagate upstream to R4 and R7. Pushback will not be invoked from R0 to R1. The path taken by pushback is the reverse of that taken by the high-bandwidth aggregate, and so pushback incidentally provides a form

[2]Clearly, pushback messages require authentication, lest they provide a powerful denial-of-service mechanism themselves!

of *traceback* if the source addresses in the aggregate are spoofed [FS00]. Pushback to upstream routers R2 and R3 helps protect the traffic to $D$ which comes in from L1. Similarly, pushing back further up to R4 from R2 and to R7 from R3 saves traffic coming along links L5 and L6 respectively.

The question "when to invoke pushback" is dealt with in §5.1. Briefly, pushback is useful when the router cannot find a narrow enough congestion signature (to minimize collateral damage), or when it is dealing with traffic that is known to be malicious (through some other information or by observing that traffic does not respond to increased drop-rate). Pushback can also be initiated by an overloaded server so that, in cases where a DoS attack was not causing congestion but was overloading a server, the benefits of pushback would still be available. In addition, the decision as to when to use pushback is likely to have a large policy component, which we do not address in this work.

The last question posed at the beginning of this section was: "when do I stop?" For Local ACC, the answer is simple: the router continues to monitor its congestion. If the router is no longer significantly congested, or if a particular aggregate being limiting is no longer one of the main responsible aggregates, then the router stops limiting the aggregate. (Clearly, we need to worry about an attacker predicting this decision in order to evade ACC.)

For pushback, however, the decision becomes more difficult, because the router must distinguish between no longer seeing much traffic from the aggregate because it is being limited upstream, versus because the aggregate has stopped sending much traffic. Disambiguating these two cases motivates the need for *feedback* messages that the upstream routers send out reporting on how much traffic from an aggregate they are still seeing (§5.4).

# 3 Related Work

In this section we discuss the various existing techniques to deal with flash crowds and DoS attacks. Some of the techniques for dealing with DoS attacks focus on protecting the network by dropping malicious packets; other techniques try to solve the *traceback* problem of tracing the attack back to the source(s). The traceback problem arises because the source IP addresses in IP packets are easily spoofed in the current Internet. When the source addresses are spoofed, a successful traceback would let the victim (and the network) find the immediate source of the attack. While traceback is important, if only as a prelude to the legal actions to discourage such attacks in the future, it alone will not stop the attacks.

Identifying the machines sending attack traffic does not necessarily lead to finding the ultimate originators of an attack. But it does allow the network to drop the attack packets near their source, before they damage the rest of the network. However, identifying the source machines is not a *requirement* for preventing the damage caused by an attack; all that's needed is to sufficiently localize the attack traffic in the topology.

In the presence of ACC mechanisms, we expect the damage control (by preferential dropping of the high-bandwidth aggregate) to trigger in much sooner than the time it takes to identify and stop the malicious sources.[3]

## 3.1 Identifying the Source of an Attack

One approach to the traceback problem is to reduce or eliminate the ability to spoof IP source addresses by some form of source filtering. In *ingress filtering* [FS00], an ISP filters out packets with illegitimate source addresses, based on the ingress link by which the packets enter the network. In contrast, *egress filtering* [SAN00] occurs at the exit point of a customer domain, where a router checks whether the source addresses of packets actually belong to the customer's domain. Packets with invalid source addresses are dropped.

While source filtering is increasingly supported as a necessary step in the protection against DoS attacks [ICS00], source filtering is not likely to completely eliminate the ability to spoof source IP addresses. For instance, if source filtering is done at the customer-ISP level, a single machine within the customer network can still disguise itself as any of the hundreds or thousands of machines in the customer domain. Even effective source filtering does not prevent attacks from compromised machines with valid source addresses.

[3] The fact that ACC, in both its local and pushback incarnations, gently restrains aggregates to the point where they are no longer causing congestion allows ACC to respond rather quickly because the downside of an inaccurate assessment of the offending aggregate is slight. DoS countermeasures that completely shut down the attacking traffic must be much more confident in their identification before they take action.

In contrast to source-based filtering, *traceback* assumes that source addresses can be spoofed, and tries to identify the source(s) of malicious traffic using the network itself. Recent proposals for traceback include a variety packet-marking schemes, i.e., Savage et al., [SWKA00], Song and Perrig [SP01], and Dean et al. [DFS01], as well as Bellovin's ICMP Traceback [Bel00].

In the absence of effective source filtering, some form of traceback would be required to identify the ultimate source of an attack. Limitations shared by all of the traceback proposals are that the damage done by the attack is not being controlled while the traceback is in progress, and the effectiveness of traceback schemes can be reduced when an attack is highly distributed. We see ACC mechanisms as complementary to both source filtering and to traceback.

Schnackenberg et al. [SDS00] suggest active control of infrastructure elements. Thus, a firewall or IDS that detected some sort of attack could request that upstream network elements block the traffic. There are obvious problems authenticating such requests in the inter-domain case, though work in the field is ongoing.

### 3.2 Identifying the Nature of an Attack

Some sites filter or rate-limit all traffic belonging to a certain category to evade particular kinds of attack. An example would be filtering ICMP ECHO messages to prevent the well-known smurf [CER98] attack. Such content-based filtering based on fixed filters can be of use, particularly in the short term, but is by definition limited to the fixed filters already defined. ACC and Pushback are based on using filters which are both dynamic and wider in range.

*Input debugging* uses attack signatures to filter out traffic at the routers. The victim identifies an attack signature and communicates it to its upstream ISP. The ISP installs a filter on its egress router to the victim, thus stopping the attack traffic. At the same time the ISP identifies the router's incoming interface of the attack, and recursively repeats the process upstream. Determining and controlling the attack traffic all the way to the sources requires cooperation between all entities controlling the routers on the paths from sources to victim. This is easier said than done, since the paths often cross administrative boundaries. The solution works on human timescales and is labor intensive. It requires the presence of skilled operators to successfully carry it out (though some ISPs have tools to do some of this work semi-automatically in their networks [Art97]).

Our proposal for Pushback is closely related to input debugging, except that instead of starting from an attack signature from a downstream victim, we would also start with a congestion signature from the congested router itself.

Instead of hop-by-hop input debugging, [Sto00] proposes building an overlay consisting of all edge routers and one or more tracking routers. In case of attacks the input debugging procedure would be carried out along the overlay tunnels. The scheme requires an overlay connecting all the edge routers of an ISP, with appropriate authentication between routers, and changes to global routing tables. Each ISP would use its own overlay system to find the entry and exit points of the traffic in its domain, using human intervention when crossing ISP boundaries.

### 3.3 Related Work on ACC and Network Congestion

In this section we discuss briefly related bodies of work on web-caching and content distribution infrastructures, scheduling mechanisms, and Quality of Service, and their relationship to ACC.

Web-caching infrastructures and Content Distribution Networks (CDNs) [Dav00] like Akamai[Aka] and Digital Island [Dig] are powerful mechanisms for preventing flash crowds from congesting the network. IP Multicast and application-level multicast are additional tools for accommodating flash crowds without creating congestion in the network, for a different set of applications. However even the combination of multicast, caching infrastructures, and CDNs may not be sufficient to completely prevent network congestion from flash crowds. For example, flash crowds could occur for traffic not carried by CDNs, or for traffic marked as uncacheable by the origin server, or for traffic that is not suitable for multicast distribution. Internet slowdowns could still be caused by an event or site that witnesses an unprecedented "success" for which neither it nor the related infrastructure is prepared.

There is a considerable body of work on scheduling and preferential dropping mechanisms that have some rela-

tionship to ACC but operate at a different granularity. Per-flow scheduling mechanisms include Fair Queuing [DKS89] and Deficit Round Robin [SV95]. There is a growing body of work on using drop preference to approximate per-flow scheduling [SSZ98] or to protect conformant flows from flows that do not use end-to-end congestion control [FF97, LM97, MF00]. However, flow-based congestion control and scheduling mechanisms are not solutions for aggregate-based congestion control, since an aggregate could be composed of many flows which are conformant individually. CBQ [FJ95] is a class-based scheduling mechanism in which aggregates can be limited to a certain fraction of the link bandwidth in a time of congestion. However, CBQ is discussed largely for fixed definitions of aggregates, and does not include mechanisms for detecting particular high-bandwidth aggregates in times of congestion.

There is also a substantial body of work on QoS mechanisms like Integrated Services [CSZ92] and Differentiated Services [BBC+98] to protect a designated body of traffic from congestion caused by lower-priority or best-effort traffic. Such QoS mechanisms could be a critical component in protecting designated traffic from congestion caused by best-effort flash crowds or DoS attacks.

# 4 Local ACC

We now describe the architecture and the algorithms used by the router to detect and control high-bandwidth aggregates. Pseudocode for the algorithms can be found in Appendix C. This section focuses on Local ACC, and the next on pushback.

Local ACC can be broken down into detection and control. In Figure 1, the ACC Agent is responsible for identifying aggregates and computing a rate limit for them. The actual rate-limiting (by dropping packets) is done by the *Rate-Limiter*. The ACC Agent is not in the fast path used for packet forwarding, and might not even be on the same machine. Packets arriving to the output queue are checked to determine if they belong to a rate-limited aggregate. Packets belonging to a rate-limited aggregate may be dropped by the Rate-Limiter depending on the arrival rate of that aggregate and the rate limit imposed on it. Packets that survive are forwarded to the output queue. Dropping also takes place at the output queue because of normal congestion. Relevant information (headers) about packets dropped at the output queue is fed into the ACC Agent which uses these packet drops for identifying high-bandwidth aggregates. Alternately, random samples from the output queue can be used in the identification process.

The identification process in the ACC Agent is triggered when the output queue experiences sustained high congestion. We define sustained congestion as a drop rate of more than $p_{high}$ over a period of $K$ seconds. During sustained congestion, using the packet drop history (or random samples) of the last $K$ seconds, the ACC Agent tries to identify a small number of aggregates responsible for the high congestion. If some aggregates are found, the ACC Agent computes the limit to which these aggregates should be restricted. The limit is computed such that the *ambient* drop rate, that is the drop rate at the output queue (not taking into account the drops in the Rate-Limiter), is brought down to below $p_{target}$. At the same time this limit cannot be less than the highest arrival rate among aggregates which are not being rate-limited. The ACC Agent then installs the necessary filters at the Rate-Limiter to rate-limit the identified aggregates. The ACC Agent is also responsible for modifying the limit imposed on various rate-limited aggregates based on changes in demand from background traffic.

The following subsections discuss the algorithms for identifying aggregates to be rate-limited, determining the rate, and implementing the rate-limiting. Later sections show how the pushback of rate-limiting to upstream nodes could be combined with local aggregate-based congestion detection and control to help make finer distinctions between the legitimate and the malicious traffic within an aggregate.

## 4.1 Identification of High Bandwidth Aggregates

In principle, an aggregate could be defined only in terms of the protocol field or port number; all DNS packets, for instance. However, almost all DoS attacks and flash crowds have either a common source or a common destination prefix. As a result, we expect that most aggregate definitions will be based on either a source or destination address prefix. As is discussed later in the paper, pushback is invoked only for aggregates whose definition includes a destination address prefix.

We present a technique to identify high-bandwidth aggregates based on the destination address. The same technique could be used to identify aggregates based on the source address (though we acknowledge that source addresses cannot necessarily be trusted). This is only one of many possible algorithms for identifying high-bandwidth aggregates; more accurate and flexible algorithms are a subject of further research. We would note that more complex definitions of aggregates would require an appropriate language for expressing the aggregate definition and for passing the aggregate definition to upstream routers during pushback.

The identification technique presented below was designed with the observation that most Web sites operate in a small range of IP addresses[4]. If one were to specify a prefix which characterized all the IP addresses in use by a server, this prefix would be longer than 24 bits in most cases. Even the sites that need fewer than 24 bits in their prefix envelopes can be better characterized by multiple 24+ bit envelopes.

Based on the drop history[5] (or random samples) draw out a list of high-bandwidth addresses (32-bit); for example, addresses with more than twice the mean number of drops. Now cluster these addresses into 24-bit prefixes. For each of these clusters try obtaining a longer prefix that still contains most of the drops. This can be easily done by walking down the prefix tree having this 24-bit prefix at the root. At each step a heavily biased branch would give a longer prefix with most of the weight. We also try to merge prefixes that are closely related to each other. For example, two adjacent 24-bit prefixes can be described by a single 23-bit prefix. Multiple clusters can also be formed for sites with spaced-out IP addresses. All these clusters are then sorted in decreasing order based on the number of drops associated with them. The number of drops also gives us an arrival rate estimate for each cluster. The algorithm to decide how many clusters should be rate-limited in order to decrease the ambient drop rate to below $p_{target}$ is described in the next section.

[4]Use of CDNs can result in a flash crowd near many caches; all routers that get congested will invoke Local ACC independently. Attacks, on the other hand, are likely to use IP addresses of the primary installation.

[5]With active queue management scheme (like RED) that distributes drops fairly, drops can be considered random sample of incoming traffic [FFT98].

Since access links have much less capacity than backbone links, they are more likely to be congested during DoS attacks and flash crowds. The identification of high-bandwidth aggregates is easier in such cases. For instance, the aggregates for the congested router could correspond to prefixes present in its routing table.

We note that different aggregates have quite different definitions, even in such basic characteristics as the number of IP addresses included in each destination-based aggregate. Thus, we reiterate that the notion of max-min fairness among aggregates is not viable as a general policy objective.

### 4.2 Determining the Rate Limit for Aggregates

Using the list of high-bandwidth aggregates obtained above during a period of high congestion, the ACC Agent determines if any aggregates should be rate-limited, and if so, what the rate-limit should be. The ACC Agent has a sorted list of aggregates, starting with the aggregate with the most drops from the drop history. The ACC Agent calculates the total arrival rate at the output queue, and uses this and the drop history to estimate the arrival rate from each aggregate over the most recent $K$ seconds.

The ACC Agent next calculates $R_{excess}$, the excess arrival rate at the output queue. This is the amount of traffic that would have to be dropped before the output queue (at the Rate-Limiter) to bring the ambient drop rate down to $p_{target}$, in the worst case.

The procedure next determines the minimum number of aggregates that could be rate-limited to sufficiently reduce the total arrival rate. One constraint is that the rate-limit for rate-limited aggregates must be greater than the arrival rate of the largest non-rate-limited aggregate. A second constraint is that the total number of rate-limited aggregates must be at most $MaxSessions$.

If the ACC Agent has determined that it can rate-limit the $i$ top aggregates, it next computes the rate-limit $L$ to be applied to each aggregate. The limit $L$ is computed such that

$$\Sigma_{k=1}^{i}(Aggregate[k].arr - L) = R_{excess},$$

where $Aggregate[k].arr$ is the arrival rate estimate of aggregate $k$.

The two constraints listed above ensure the $L$ is less

than the arrival rate estimate of $i+1$-th aggregate, and that $i$ is at most $MaxSessions$. Ideally, the Local ACC mechanisms should not rate-limit *any* aggregate during times of undifferentiated congestion caused by under-provisioned links or hardware failures. In the absence of effective methods for distinguishing between aggregate-based and undifferentiated congestion, we use the upper bound $MaxSessions$ on the number of aggregates that are rate-limited simultaneously. With better understanding of the traffic composition and behavior during DoS attacks and flash crowds, we can tune the Local ACC mechanism such that it does not identify any aggregate in times of undifferentiated congestion.

In the presence of policy constraints, the computation of the rate-limit $L$ would have to be modified slightly. For instance, the router could be configured never to rate-limit a specified aggregate to less than $B$ Mbps. Such policy level decisions have to be honored in the rate limit calculation.

The ACC Agent revisits its rate-limiting decisions periodically, revising the rate limit $L$, and determining if some aggregate no longer needs to be rate-limited. The ACC Agent measures the arrival rate for rate-limited aggregates, so for the refreshes, the ACC Agent has more precise information about these arrival rates. Aggregates that have had an arrival rate less than the limit for some number of refresh intervals are no longer rate-limited. Similarly, if congestion persists, more aggregates may be added to the list of rate-limited aggregates. There is no harm done if rate-limiting continues for some time after the DoS attack or flash crowd has subsided, because the rate-limiter only drops packets if the arrival rate is more than the specified limit. However, some care is needed that the rate-limit for an aggregate does not change abruptly as another aggregate is added or removed from the list of rate-limited aggregates.

## 4.3 Rate-limiter

The rate-limiter is responsible for classifying packets, rate-limiting those belonging to a rate-limited aggregate, and measuring the arrival rate of the rate-limited aggregates. This section discusses the properties of the rate-limiting architecture shown in Figure 1 and describes a mechanism for implementing the rate-limiter.

Because it sits in the forwarding fast path, the rate-limiter needs to be light-weight and efficient to implement. Because the rate-limiter is a pre-filter before the output queue that merely decides whether or not to drop each arriving packet in the aggregate, it is consistent with FIFO scheduling in the output queue. Unlike strict lower priority queues, it will not starve the identified aggregates. As noted earlier in §2 rate-limited aggregates are never protected from the normal congestion occurring in the output queue. To ensure that the rate-limited aggregates are protected from each other, the drop decision for each aggregate is taken independently based on the state for that aggregate.

### 4.3.1 Virtual Queue

In this section we discuss the virtual queue, the mechanism that we use for rate-limiting. Appendix A describes preferential dropping, an alternate mechanism for the rate-limiter. Preferential dropping and virtual queues differ in the procedure for making a drop (rate-limiting) decision. However, both mechanisms only drop packets from the aggregate when the aggregate's arrival rate to the rate-limiter is above the specified limit.

A virtual queue can be thought of as simulating a queue, without actually queuing any packets. The service rate of the simulated queue is set to the specified bandwidth limit for the aggregate, and the queue size is set to the tolerated burst size. When a packet arrives at the rate-limiter, the rate-limiting mechanism simulates that packet arriving at the virtual queue. Packets that would have been dropped at the virtual queue are dropped by the rate-limiter, and packets that would have been queued at the virtual queue are forwarded to the real output queue.

A virtual queue can simulate either a simple tail drop queue or a queue with active queue management. A virtual queue that simulates tail drop behavior can be implemented as a token bucket, with the fill rate of the token bucket set to the bandwidth limit of the rate-limiter, and the bucket size of the token bucket set to the tolerated burst size for the rate-limiter.

### 4.3.2 Narrowing the Congestion Signature

In the discussion above, the aggregates identified by the ACC Agent are based only or source or destination addresses. In fact, the rate-limiter can do more sophisticated narrowing of the congestion signature that, in

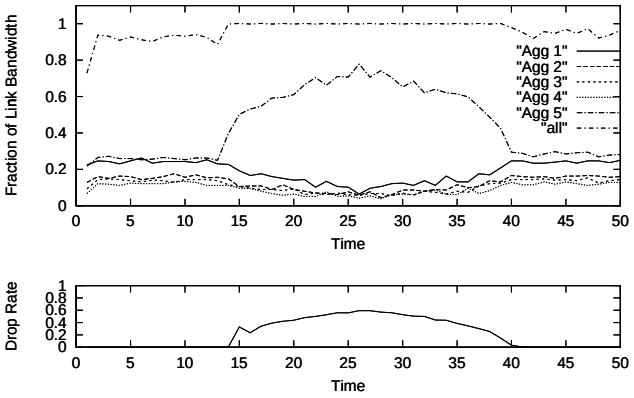


Figure 3: **A simulation without ACC.**

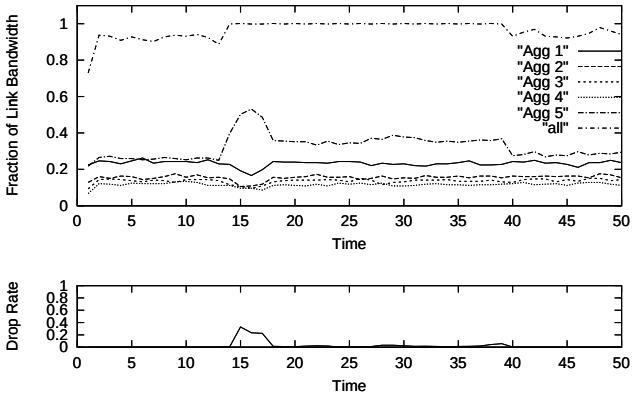


Figure 4: **The same simulation with Local ACC.**

times of specialized attacks, can result in dropping more of the attack traffic within the aggregate. For example, a suitably sophisticated ACC Agent would detect a more specific dominant signature within the aggregate, based on other packet characteristics (such as port number or ICMP type code), and drop more heavily from this subset. Narrower rate-limiting could be achieved by placing another virtual queue, with a smaller service rate, in front of the aggregate's virtual queue.

This hierarchical rate-limiting is safe in scenarios where attacker frequently changes her attack signature, as the total bandwidth available to the aggregate is still bound. Such specialized rate-limiting can be very useful in cases of attacks like the SYN attack [CER96] or the smurf attack [CER98].

One might perhaps argue that during flash crowds the routers should do some form of flow-aware rate-limiting, for example, dropping more heavily from SYN packets to provide better service to connections that manage to get established. However, this can be dangerous if applied for a DoS attack rather than a flash crowd. The attacker could simply send packets in the category being favored by flow-aware rate-limiting (TCP data packets in the above example). Flow-aware rate-limiting is different from narrow rate-limiting mentioned above. While the latter punishes the dominant (relative to usual levels) packet type in the aggregate, the former favors a particular packet type, a strategy that can be gamed.

### 4.4 Simulations

We use a simple simulation to illustrate the effect of Local ACC. Figure 3 shows a simple simulation without ACC, with six aggregates, each composed of multiple CBR flows, with the sending rate of the fifth aggregate varying over time.[6] Because this simulation is of CBR flows, rather than of flows using end-to-end congestion control, it has very simple dynamics; its purpose is to illustrate the underlying functionality of Local ACC.

The two graphs in Figure 3 show that, without ACC, the high-bandwidth aggregate is able to capture most of the link bandwidth. The bottom graph of Figure 3 shows the ambient packet drop rate in the output queue. At time 13 the sending rate of the fifth aggregate gradually increases, increasing the drop rate and decreasing the bandwidth received by the other four aggregates.

Figure 4 shows the same simulation repeated with Local ACC enabled. When the ambient drop rate exceeds the configured value of 10%, the ACC Agent attempts to identify an aggregate or aggregates responsible for the high congestion. Within a few seconds the ACC Agent identifies the fifth aggregate, and rate-limits that aggregate sufficiently to control the drop rate in the output queue. The bottom graph of Figure 4 shows the ambient drop rate in the output queue, but does not show the drop rate in the rate-limiter for the fifth aggregate.

## 5 The Pushback Mechanism

Section 4 described Local ACC; let us now discuss the pushback mechanism in detail. Pushback for an aggregate can be visualized as a tree, where the congested

[6] These simulations can be run with the commands "./test-all-pushback slowgrow" and "./test-all-pushback slowgrow-acc" in the tcl/test directory in the NS simulator.

router initiating the pushback is the root, and the upstream routers rate-limiting the aggregate are the interior nodes and the leaves of the tree. For example, in Figure 2, node R0 is the root of the pushback tree, and nodes R4 and R7 are the leaves.

### 5.1 Deciding when to Invoke Pushback

After detecting aggregate-based congestion, the ACC Agent must decide whether to invoke pushback by calling the Pushback Agent at the router. The ACC Agent has information only about its own output queue, while the Pushback Agent coordinates information from diverse input and output queues, and sends and receives pushback messages from neighboring routers.

Two situations warrant the invocation of pushback. The first is when the drop rate for an aggregate in the rate-limiter remains high for several seconds (because the arrival rate for the aggregate remains much higher than the limit imposed on it).[7] The second is when the Pushback Agent has other information that a DoS attack is in progress. In some cases packet drop history can help the router differentiate between DoS attacks and flash crowds. For instance, if most of the packets within the aggregate are destined for a notorious UDP port, the router can be fairly certain that it is witnessing a DoS attack and not a flash crowd. Another source of information can be the downstream server[8] itself. For example, pushback could be invoked by a router at the behest of a server directly connected to it, if allowed by the policy at the router. This would also be helpful to the server in situations when considerable traffic is being sent to the server, but at a level not high enough for the ACC Agent at the adjacent router to invoke Local ACC or pushback.

### 5.2 Sending the Pushback Requests Upstream

When the Pushback Agent at the congested router invokes Pushback for an aggregate, it has to divide the rate limit for the aggregate among the upstream links. This requires that the Pushback Agent have some estimate of the amount of aggregate traffic coming from each upstream link. The upstream links sending only a small fraction of the aggregate traffic are termed as *non-contributing links*, and we call the other upstream links *contributing links*. Because one of the motivations of pushback is to concentrate the rate-limiting on the links sending the bulk of the traffic within the aggregate, the Pushback Agent does not send a *pushback request* to non-contributing links. The assumption is that if a DoS attack is in progress, the aggregate traffic on the contributing links is more likely to include the attack traffic, while the aggregate traffic on the non-contributing links is more likely to be legitimate traffic.

In the general case, contributing links do not all contribute the same amount of bad traffic. A link carrying more traffic belonging to the aggregate is more likely to be pumping in attack traffic. One of many possible algorithms, and the one used in our simulations, is to first determine how much traffic in the aggregate each link contributes. We then divide the desired limit $L$, reduced by the amount of traffic coming from non-contributing links, among the contributing links in a max-min fashion. For example, assume that we have three contributing links with arrival rates of 2, 5, and 12 Mbps, and that the desired limit, after the non-contributing traffic has been subtracted from it, is 10 Mbps. The limits sent to each of the three contributing links would then be 2, 4, and 4 Mbps respectively.

| Congestion Signature |
|---|
| Bandwidth Limit |
| Expiration Time |
| RLS-ID |
| Depth of Requesting Node |
| Pushback Type |

Figure 5: **Contents of a pushback request**

After the Pushback Agent determines the limit to request from neighboring upstream routers, it sends a pushback *request* message[9] to those routers. As shown in Figure 5, a pushback request contains the congestion signature characterizing the aggregate, the requested upper bound for the amount of traffic sent belonging to the aggregate, the time period after which the pushback request expires, the Rate-Limit Session ID (RLS-ID), the depth of the re-

[7] The high drop rate implies that the router has not been able to control the aggregate locally by preferential dropping, in an attempt to encourage increased end-to-end congestion control.

[8] The server can have some higher level or application-specific attack detection mechanism

[9] The pushback protocol, including timing and format of messages, is described in [FBI+01].

quester in the pushback tree, and the type of pushback. In our simulations the attack signature consists of the destination prefix or prefixes characterizing the aggregate. The RLS-ID is returned in the feedback messages (see Section 5.4) to enable the Pushback Agent to map the feedback to the corresponding pushback request. In the pushback tree, the depth of the root is zero, and a child's depth is one more than the depth of its parent. Depth information is useful in setting timers for sending feedback. The type of pushback influences the decision of an upstream router about whether to propagate pushback upstream. The router is more likely to propagate when the type corresponds to a malicious attack (*e.g.*, server-initiated pushback).

The rate-limit specified in the pushback request is only a requested *upper* bound for the bandwidth to be given to the aggregate. If the upstream router itself becomes heavily congested, then it may give less bandwidth to the aggregate than the specified limit. Because the pushback request only specifies an upper bound, it will not end up shielding the aggregate from local congestion at the upstream router in the guise of rate limiting (see Section 4.3). That is, the aggregate will not necessarily receive bandwidth at the upstream router equal to the upper bound; the upstream router is simply requested not to give *more* than the upper bound to the specified aggregate.

We also note that the congested router could receive more than the desired amount of traffic in the aggregate if the non-contributing upstream neighbors (which were not sent pushback requests) start sending more traffic in the aggregate. However, since the rate-limiting is also being done at the congested router, more than desired amount of aggregate traffic never goes over the congested link.

## 5.3 Propagating Pushback

On receiving a pushback request, the upstream router starts to rate-limit the specified aggregate just as it does for Local ACC, using the rate limit in the request message. The router's decision whether to further propagate the pushback request upstream uses similar algorithms to those described in Sections 5.1 and 5.2 above.

When propagating a pushback request upstream, the destination prefixes in the congestion signature have to be narrowed, to restrict the rate-limiting to traffic headed for the downstream congested router only. The means that pushback can only be invoked for congestion signatures that include a destination prefix. This is discussed in more detail in Appendix B.1.

## 5.4 Feedback to Downstream Routers

The upstream routers rate-limiting some aggregate in response to a pushback request send pushback *status* messages to the downstream router, reporting the total arrival rate for that aggregate from upstream. The total arrival rate of the aggregate upstream is a lower bound on the arrival rate of that aggregate that the downstream router would receive if upstream rate-limiting were to be terminated. Because the upstream rate-limiting (dropping) may have been contributing to end-to-end congestion control for traffic within the aggregate, terminating the upstream rate-limiting may result in a larger arrival rate for that aggregate downstream. Pushback status messages enable the congested router to decide whether to continue rate-limiting (and pushback). The timing of the pushback status messages is described in more detail in Appendix B.2.

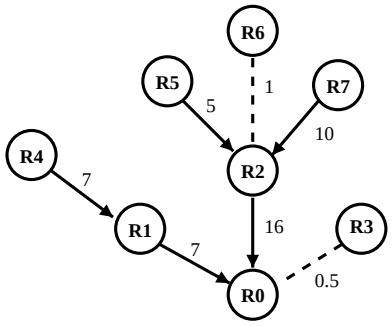


Figure 6: **Pushback status messages reporting the aggregate's arrival rate from upstream.**

The arrival rate reported in the pushback status message is the sum of the arrival rates in all the status messages received from upstream, plus the arrival rates from the upstream non-contributing nodes. For example, in Figure 6, *RO* is the root of the pushback tree, shown by the solid lines. The labels for each solid line show the arrival rate estimate contained in the pushback status message. The dashed lines connect the non-contributing nodes that did not receive pushback request messages, and the labels show the aggregate's arrival rate as estimated by the

downstream neighbor. From the pushback status messages, $R0$ can estimate the total arrival rate for the aggregate as 23.5 Mbps. If $R0$ were to terminate the rate-limiting upstream, and invoke an equivalent rate-limiting locally, this would be roughly the arrival rate that $R0$ could expect from that aggregate.

### 5.5 Pushback Refresh Messages

The Pushback Agent at the router uses soft state, so that rate limiting will be stopped at upstream routers unless refresh messages are received from downstream. In determining the updated rate limit in the refresh messages, the downstream router uses the status messages to estimate the arrival rate from the aggregate, and then uses the algorithms in Section 4 to determine the bandwidth limit. The arrival rates reported in the pushback status messages are also used by the downstream router in determining how to divide the new bandwidth limit among the upstream routers.

## 6 Simulations with Pushback

This section shows a number of simulations using the NS [NS] simulator testing the effect of Local ACC and pushback in a variety of aggregate-based congestion scenarios. These simulations do not pretend to use realistic topologies or traffic mixes, or to stress Local ACC and pushback in difficult or highly dynamic environments; the simple simulations in this scenario are instead intended to illustrate some of the basic underlying functionality of Local ACC and pushback.

Before going into the details of the simulations we introduce some informal terminology here that would help us in describing the simulations. For the scenarios with DoS attacks, the *bad* sources send attack traffic to the victim destination $D$, and the *poor* sources are innocent sources that happen to send traffic to the destination $D$ when it is under attack. In other words, packets from the poor sources represent the unmalicious traffic in the congestion signature. For all of the scenarios, the *good* sources send traffic to destinations other than $D$.

### 6.1 A Simple Simulation

Figure 7 shows the topology for a simple simulation intended to show the dynamics of pushback. The good and the poor sources each send traffic generated by seven infinite demand TCPs. The bad source sends UDP CBR traffic, with the sending rate varied from one simulation to the next.

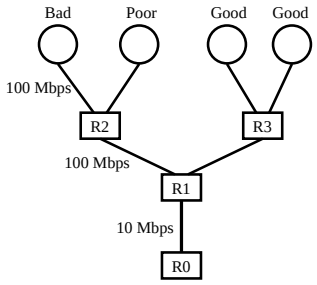


Figure 7: **The topology for a simple simulation.** $R1 - R0$ is the congested link.

The results of the simulation are shown in Figure 8. Each column of marks represents the results from a single simulation, with the $x$-axis indicating the sending rate of the bad source. When the bad source sends 8 Mbps or more, the drop rate at the output queue exceeds 10%, the configured value of $p_{high}$, and Local ACC and Pushback are initiated for the aggregate consisting of the bad and poor traffic. As a result of the rate-limiting, the arrival rate to the output queue is reduced, and the good traffic is protected from the bad.

For this scenario, the use of pushback is also effective in concentrating the rate-limiting on the bad traffic and protecting the poor traffic within the aggregate. The simulations with Local ACC without pushback (not shown) produced approximately the same result for the good traffic; however, the poor source received almost

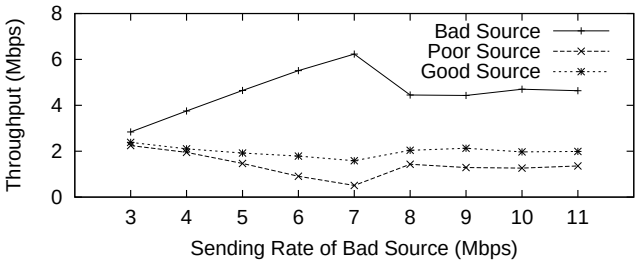


Figure 8: **The effect of pushback in a small topology.**

no bandwidth in that situation.

## 6.2 DoS Attacks

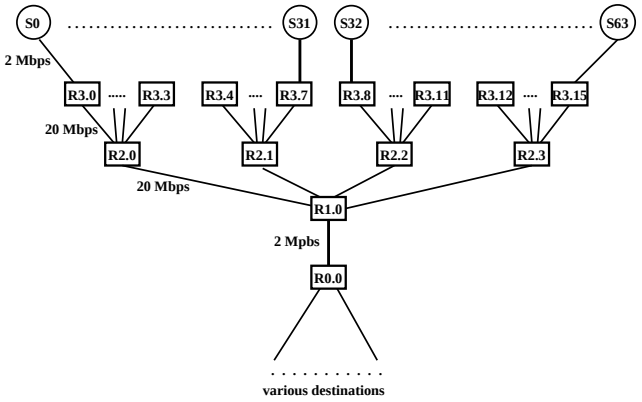


Figure 9: **The topology used in simulations**. $R1.0 - R0.0$ is the congested link

The simulations in this section illustrate Local ACC and Pushback with both sparsely-spread and with highly diffuse DoS attacks. These simulations use the topology shown in Figure 9, consisting of four levels of routers. There is one router each in the bottom two levels and 4 and 16 routers, respectively, in the upper two levels. Except for the router at the lowest level, each router has a fan-in of four. The top-most routers are attached to four sources each. The link bandwidths are shown in the figure, and have been allocated such that congestion is limited to the access links at the top and bottom.

The first simulation scenario, with a sparsely-spread DoS attack, includes four bad sources, four poor sources, and ten good sources, randomly distributed among the 64 source nodes. Each of the four bad sources sends 1 Mbps of UDP CBR traffic, half the link capacity. The good and the poor sources send Web traffic, using the Web traffic generator in NS.

Figure 10 shows the results of these simulations. "Default" denotes a simulation with the router not doing any form of ACC. The two lines in the graph denote the quantity of good and poor traffic in the absence of any attack traffic. Both the Local ACC and pushback simulations bring down the bandwidth consumed by the attacker, leading to a significant bandwidth gain for the good traffic. However, as expected, Local ACC leaves the poor hosts starved because the congested router, $R1.0$, cannot differentiate between the poor and the bad

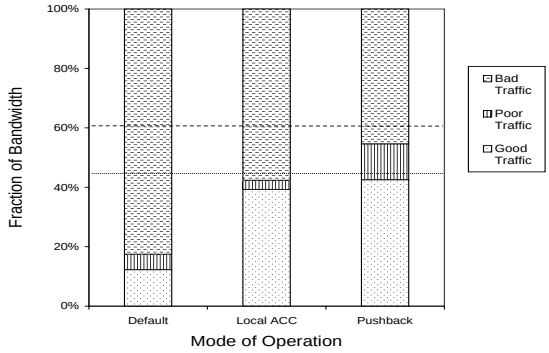


Figure 10: **Bandwidth allocation at the congested link during a sparse DoS attack.**

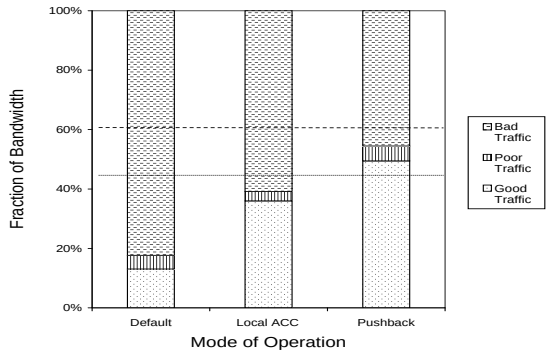


Figure 11: **Bandwidth allocation at the congested link during a diffuse DoS attack.**

traffic in the aggregate. We obtained similar results for simulations with different amounts of attack traffic.

The second simulation scenario, with a highly diffuse DoS attack, uses 32 bad sources, four poor sources, and ten good sources. In this scenario each of the 32 bad sources sends 0.125 Mbps of UDP CBR traffic, for the same total bad traffic as in the previous scenario. This setup is intended to simulate a DoS attack where a large number of sources spread throughout the network are used to generate the attack traffic. Each bad source by itself generates a small amount of traffic, making it hard to detect such sources at their access links.

As Figure 11 shows, with diffuse attacks pushback loses the ability to differentiate between the bad and the poor traffic, though it still reduces the bandwidth consumed by the bad sources. In fact, in an attack in which a lot of sources are used, an individual bad source might be generating less traffic than a valid poor source. When these bad sources are spread throughout the network, the at-

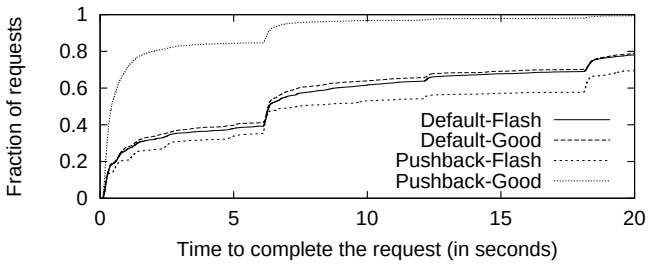


Figure 12: **Time to complete a request during a flash crowd.**

tack looks more like a flash crowd, making it harder to distinguish between the bad and the poor sources. The bandwidth obtained by the good traffic with pushback goes slightly above the no-attack case (lower line) because of reduced competition from the poor traffic.

### 6.3 Flash Crowds

This section shows simulations with flash crowds instead of DoS attacks, with the "flash" traffic from 32 sources sending Web traffic to the same destination. The good traffic comes from ten other sources sending Web traffic to various other destinations, accounting for about 50% link utilization in absence of any other traffic.

Figure 12 shows the distribution of the times to complete the transfers for the good and the flash traffic respectively in the Default and Pushback mode. The distribution for Local ACC mode (not shown) was similar to the Pushback one. With Pushback, 80% of the good transfers complete within a few seconds, compared to less than 40% completed in less than six seconds in the Default case. While the performance gain for the good traffic is significant, the degradation seen by the flash traffic is not that much. The time to complete a Web request can be directly correlated to the drop rate experienced. The drop rate for the good traffic comes down from 30% to just 6% ($p_{target}$=5%) and that of the flash traffic goes up only by 3% to about 33%. Because the flash traffic is much more than the good traffic, even a slight increase in its drop rate frees up a lot of link capacity.

The hump around the 6-second mark represents short web transfers whose first SYN or SYN/ACK packet was lost, resulting in the transfer completing slightly more than six seconds later, after the retransmit timer expires. The magnitude and the location of the hump along the y-axis is a good indication of the packet drop rates in the network for that aggregate. Recall that Local ACC and Pushback are only invoked in scenarios of extreme congestion where the packet drop rate exceeds the configured threshold, set to 10% in our simulations, and at these levels of congestion a large fraction of transfers will have the first SYN or SYN/ACK packet dropped.

Though the graph did not show major differences between the transfer time distribution of web requests for Local ACC and pushback, the good traffic receives roughly 37% of the link bandwidth with Local ACC, compared to 50% with pushback. Because pushback rate-limits the bad traffic upstream, this leads to a decrease in the amount of bad traffic reaching the congested router, relative to Local ACC (absence of statistical multiplexing), which in turn enables more good traffic to go through. The transfer time distribution on the other hand is a function of the drop rate. Thus, more good traffic gets through with pushback than with Local ACC while keeping the same ambient drop rate at the output queue.

## 7 Discussion

### 7.1 Advantages and Limitations of Pushback

Pushback is not a panacea for flooding attacks. In fact, if not used carefully, it can make matters worse. This section discusses the advantages and limitations of adding pushback to ACC.

One advantage of pushback is to prevent scarce upstream bandwidth from being wasted on packets that will be dropped downstream.

When attack traffic can be localized spatially, pushback can effectively concentrate rate-limiting on the malicious traffic within an aggregate. This is very useful when source addresses cannot be trusted because then the congested router cannot narrow the congestion signature by itself.[10] In addition, if the offending traffic within an aggregate is heavily represented on some upstream link in the network, but the congested router cannot identify this subset of the aggregate based on the source IP

[10] If source addresses could be trusted, then in some cases the congested router could narrow the attack signature itself, by identifying both the source and the destination address prefixes responsible for the bulk of the traffic in the identified aggregate.

addresses alone (i.e. the attack can be localized spatially but no concise description in terms of source prefixes exists), then pushback is necessary to narrow the attack signature, *even if source addresses are genuine.*

For example, if a DoS attack on www.whitehouse.gov using legitimate source addresses were coming largely from computers at MIT, this could be identified either by source address prefixes at the congested router at www.whitehouse.gov, or by pushback to the wide-area link leaving MIT. In contrast, for a DoS attack from various places in the US on a machine in the UK, the congested router might only be able to define the aggregate containing the attack by destination prefix, unable to narrow the attack signature to some subset of the IP source addresses because they are too diverse. Pushback to a transoceanic link would more precisely identify the attack, and focus preferential dropping only on the traffic within that aggregate that is carried over the transoceanic link, sparing traffic to the UK site from elsewhere in the UK and other parts of the world. In this case, pushback focuses in on the source of the attack. We emphasize that there is nothing special about crossing a transoceanic link in this example—the point holds for any case where the attack is concentrated on some upstream link, possibly a number of hops upstream, but where the downstream congested routers cannot isolate this traffic using source addresses. This can happen for flash crowds, too, if, for example, the sources of the flash crowd all come from a particular provider or region.

For some DoS attacks, pushback will not be effective in concentrating rate-limiting on the malicious traffic within an aggregate. For example, this would be the case for an attack uniformly distributed across the inbound links. Consider, for example, a reflector attack [Pax00] based on DNS [CER00]. If sufficiently many reflectors are used from all portions of the network, the aggregate bandwidth will swamp the victim's link. During such an attack pushback will not be able to differentiate between the poor and the bad DNS traffic going to the destination, and will drop from both equally.

Pushback may overcompensate, particularly when it is invoked for non-malicious events such as flash crowds. If the overall demand from other traffic is reduced before the pushback refresh period expires (Section 5.5), then the upstream routers could unnecessarily drop packets from the high-bandwidth aggregate even when the downstream link becomes underutilized. In Local ACC link underutilization is more easily avoided, as rate-limiting does not drop packets when the output queue is itself low. We reduce the possibility of overcompensation (and lower link utilization) by calculating the rate-limit of an aggregate so that the total traffic coming to the congested router is still greater than the capacity of the congested link (see the discussion of $p_{target}$ in §4.2). Performing some of the rate-limiting just at the congested router can also help to prevent overcompensation.

In some cases, the use of pushback can increase the damage done to legitimate traffic from a source close to the attacking host. As pushback propagates upstream towards the attack sources, the drop rate for the aggregate is increased. If pushback fails to reach a point where it can differentiate between the attack sources and the nearby legitimate traffic within the same aggregate, for instance, when the two sources are in the same edge network which is not pushback-enabled, the legitimate traffic at that point will share the same high drop rate as the attack traffic. This property of pushback could lead to potential DoS attacks in which the attacker's aim is to hinder a source from being able to send to a particular destination. To be successful, an attacker would need to launch the attack from a host close to the victim source. However, the ability to compromise a machine that shares a downstream bottleneck link with the victim enables many other forms of attack anyway.

## 7.2 Implementation and Operational Issues

In this section we address some implementation and operational issues concerning deployment of ACC mechanisms in the Internet.

### 7.2.1 Implementation Complexity

The identification of aggregates can be done as a background task, or in a separate machine entirely, so the processing power required to identify aggregates should not be an issue. However, the presence of a large number of rate-limited aggregates could pose a design challenge. When a packet arrives at the output queue, the router has to determine if that packet belongs to one of the rate-limited aggregates, and if so, place it in the correct virtual queue. The time required for this lookup may increase with an increasing number of aggregates. We do not expect the limitation on the number of rate-limited

aggregates to be a problem, as we envision Local ACC and Pushback as mechanisms to be instantiated sparingly, in times of high congestion, for a handful of aggregates. But a deployed system needs to be robust against new attacks that could generate many rate-limited aggregates. One possible approach would be to use the routing table of the router for detecting membership in a rate-limited aggregate; however, this would restrict the definition of aggregates to destination prefixes.

#### 7.2.2 Control Message Overhead

The network bandwidth requirement for pushback messages is minimal. During each refresh round of each pushback session, on the order of a few seconds, one message is sent over a link in the pushback tree in each direction.

#### 7.2.3 Estimating the Upstream Link's Contribution

The distribution of the rate-limit among upstream links depends on the downstream router's ability to estimate what fraction of the aggregate comes from each upstream router. This is simple for point-to-point links; only one router is attached to each interface. However, for routers joined by LANs, VLANs, or frame relay circuits, there are multiple routers attached to an interface. Similarly, some routers may only be able to determine from which line card traffic originated, rather than which port on the card. The downstream router in this situation might not be able to distinguish between multiple upstream routers.

One way of dealing with this problem is to send a *dummy* pushback request to all upstream neighbors. The dummy request is similar to the real request, but the recipient does not actually rate-limit the aggregate. The only impact of this request is that the recipient will estimate the arrival rate of the specified aggregate and report it to the downstream router in status messages. These messages help the downstream router to send pushback requests with the appropriate rate-limits to contributing routers.

#### 7.2.4 Incremental Deployment

The pushback mechanism described so far works in contiguous areas only. Pushback messages sent to an upstream router that does not understand pushback will simply be ignored.

Pushback can be deployed incrementally by deploying it only on the edges of an *island* of routers, where an island is a set of connected routers. An autonomous system (AS) is a natural island. Assume that the island has $N$ edge routers. When one of these routers gets congested and decides to invoke pushback, it could consider the remaining $N - 1$ edge routers as its upstream links. Using dummy pushback messages it could ascertain the aggregate's arrival rate at each of these routers, and send authenticated pushback request messages accordingly. Thus, even without universal deployment the island can selectively throttle traffic coming in from certain directions.

#### 7.2.5 Parallel Paths

The presence of parallel/multiple paths in the Internet does not introduce any complication. If they occur, pushback will propagate along both (all) if both have a high volume of traffic, and where they merge upstream two different pushback limiters will be instantiated on two different interfaces. The only extra step is merging the limits as pushback recurses, which should be straight-forward.

### 7.3 Policy Knobs

Like traffic engineering, Local ACC and Pushback are areas that will take significant guidance from local policy databases. This is unlike many other router mechanisms such as per-flow or FIFO scheduling, active queue management, or Explicit Congestion Notification, which are generally best with a modest number of policy hooks. It will take many simulations and a good deal of operational experience to get a better understanding of the right policies for ACC.

In principle, there is nothing stopping a pushback tree from extending across ASs. However, real networks make extensive use of policy databases in making decisions involving other ASs. There are issues related to trust in accepting a pushback request from a neighboring AS. Moreover, a pushback request might be in contradiction with a contractual obligation that says "provide transit to this much traffic". In this case an edge router could discover conflicts and inform its parent (and the

congested router). This distribution of limits along the pushback tree might be heavily policy-driven in such cases. In absence of any policy conflicts, we believe that there is sufficient incentive for ASs to honor pushback requests from neighboring ASs. Not only does it enable getting to the sources of attack, but also saves bandwidth spent on carrying traffic that will be dropped downstream.

### 7.4 Empirical Data on Traffic Behavior and Topologies

One issue for Local ACC is the detection of sustained congestion. What are the drop rates, over what period of time, that merit invoking ACC? We expect the answer to be different for different places in the network. For understanding this, it would help to have as much measurement data as possible on the pattern of packet drop rates at different routers in the Internet. How frequently do different routers have periods of sustained high congestion? How long does this sustained congestion last? And how often is it due to special events like flash crowds and DoS attacks, as opposed to the more diffuse congestion from hardware failures or routing changes for which Local ACC and Pushback would be less appropriate? More measurement data could also help in considering the issue of when to invoke pushback in addition to Local ACC. Sites such as the Internet Traffic Report [ITR] and the Internet Weather Report [IWR] have some data on packet drop rates, as well as reports on specific fiber cuts, flash crowds, and DoS attacks, but we are not aware of any systematic characterization and identification of the high-congestion periods at a specific router. As a further complication, past measurements are not necessarily good predictors of future conditions for such volatile occurrences as flash crowds and DoS attacks, either at an individual router or for the Internet as a whole.

As noted before, the effectiveness of Pushback in differentiating between malicious and non-malicious traffic within an aggregate during a DoS attack is dependent on the distribution of attack sources and the paths that the attack traffic takes to reach the congested router. More information of typical attack topologies from specific past DoS attacks, analytical topology models, or Internet topology databases would help us to evaluate the potential effectiveness of Pushback in protecting non-malicious traffic within an aggregate.

## 8 Conclusions

Congestion caused by aggregates differs in some fundamental aspects from that caused by individual flows, and hence requires different protection mechanisms in the network. We have proposed both local and cooperative mechanisms for aggregate-based congestion control to answer this need. While a lot remains to be investigated, simulations carried out till now have shown that they are very promising directions. As part of this work, we are in the process of implementing an experimental testbed using software routers.

Future work will include more thorough investigations of the underlying dynamics and possible pitfalls of these mechanisms. The effectiveness of these mechanisms are deeply tied in with their operating environment. For effective evaluation we need measurement-based answers to questions like “how frequently is sustained congestion caused by aggregates, and not by failures”, and “what do attack traffic and topologies look like”.

## Acknowledgments

The original idea for pushback came from an informal DDoS research group consisting of Steven M. Bellovin, Matt Blaze, Bill Cheswick, Cory Cohen, Jon David, Jim Duncan, Jim Ellis, Paul Ferguson, John Ioannidis, Marcus Leech, Perry Metzger, Vern Paxson, Robert Stone, Ed Vielmetti, and Wietse Venema. We also thank Randy Bush, Eddie Kohler, Neil Spring and Ed Vielmetti for feedback on an earlier draft of this paper.

## References

[Aka] Akamai home page. http://www.akamai.com.

[Art97] Artsci.net Web Pages: Dostrack. http://www.artsci.net/~jlinux/security/dostrack, October 1997. URL for acknowledgement purposes only.

[BBC+98] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[Bel00] Steve M. Bellovin. ICMP Traceback Messages. Internet Draft: draft-bellovin-itrace-00.txt, March 2000.

[Bor99] John Borland. Net Video Not Yet Ready for Prime Time. CNET news, February 1999. http://news.cnet.com/news/0-1004-200-338361.html.

[CER96] CERT Web Pages: CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks. http://www.cert.org/advisories/CA-1996-21.html, September 1996.

[CER98] CERT Web Pages: CERT Advisory CA-98.01 "smurf" IP Denial-of-Service Attacks. http://www.cert.org/advisories/CA-98.01.smurf.html, January 1998.

[CER00] CERT. Cert incident note in-2000-04, 2000. http://www.cert.org/incident_notes/IN-2000-04.html.

[Cis98] Cisco Web Pages: Committed Access Rate. http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/car.htm, February 1998.

[CSZ92] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network Architecture and Mechanism. In *ACM SIGCOMM*, 1992.

[Dav00] Brian D. Davidson. Content delivery and distribution services. http://www.web-caching.com/cdns.html, August 2000.

[DBCP97] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM*, September 1997.

[DFS01] Drew Dean, Matt Franklin, and A. Stubblefield. An algebraic approach to ip traceback,. In *Proceedings of NDSS '01*, February 2001.

[Dig] Digital Island home page. http://www.digitalisland.com.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM SIGCOMM*, 1989.

[FBI+01] Sally Floyd, Steve Bellovin, John Ioannidis, Kireeti Kompella, Ratul Mahajan, and Vern Paxson. Pushback Messages for Controlling Aggregates in the Network. Work in progress. Internet-draft: draft-floyd-pushback-messages-00.txt, July 2001.

[FF97] Sally Floyd and Kevin Fall. Router mechanisms to support end-to-end congestion control. Technical report, LBL, February 1997.

[FFT98] Sally Floyd, Kevin Fall, and Kinh Tieu. Estimating Arrival Rates from the RED Packet Drop History, April 1998. http://www.aciri.org/floyd/end2end-paper.html.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, Vol. 1(4):pp. 397–413, August 1993.

[FJ94] Sally Floyd and Van Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking*, Vol. 2(2):pp. 122–136, April 1994.

[FJ95] Sally Floyd and Van Jacobson. Link-sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking*, Vol. 3(4):pp. 365–386, August 1995.

[FS00] Paul Ferguson and Daniel Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, May 2000.

[Gar00] Lee Garber. Denial-of-Service Attacks Rip the Internet. *IEEE Computer*, vol. 33(4):pp. 12–17, April 2000.

[ICS00] ICSA Web Pages: Alliance for Internet Security. http://www.icsa.net/html/communities/ddos/alliance/guide.shtml, 2000.

[ILS99] A. S. Induruwa, P. F. Linington, and J. B. Slater. Quality of Service measurements on SuperJANET - the UK academic information highway. In *Proc INET'99*, June 1999.

[ITR] The Internet Traffic Report. http://www.internettrafficreport.com/.

[IWR] The Internet Weather Report. http://www.mids.org/weather/.

[LM97] Dong Lin and Robert Morris. Dynamics of Random Early Detection. In *ACM SIGCOMM*, 1997.

[MF00] Ratul Mahajan and Sally Floyd. Controlling High-Bandwidth Flows at the Congested Router, November 2000. http://www.aciri.org/red-pd/.

[MVS01] David Moore, Geoffrey Voelker, and Stefan Savage. Inferring Internet Denial of Service Activity. *USENIX Security Symposium*, August 2001.

[NS] NS Web Page: http://www.isi.edu/nsnam.

[Par96] Craig Partridge. Locality and Route Caches. NSF Workshop on Internet Statistics Measurement and Analysis, February 1996.

[Pax00] Vern Paxson. An analysis of using reflectors to defeat DoS traceback, August 2000. ftp://ftp.ee.lbl.gov/.vp-reflectors.txt.

[SAN00] Egress Filtering. SANS Web Pages, February 2000. http://www.sans.org/y2k/egress.htm.

[SDS00] Dan Schnackenberg, Kelly Djahandari, and Dan Sterne. Infrastructure for intrustion detection and response. In *Proceedings of the DARPA Information Survivability Conference and Exposition 2000*, March 2000.

[SP01] Dawn Song and Adrian Perrig. Advanced and authenticated techniques for ip traceback. In *Proceedings of Infocomm 2001*. IEEE, 2001. http://paris.cs.berkeley.edu/ dawnsong/papers/iptrace.ps.

[SSZ98] Ion Stoica, Scott Shenker, and Hui Zhang. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *ACM SIGCOMM*, 1998.

[Sto00] Robert Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods. In *9th Usenix Security Symposium*, August 2000.

[SV95] M. Shreedhar and George Varghese. Efficient Fair Queuing using Deficit Round Robin. In *ACM SIGCOMM*, 1995.

[SV98] V. Srinivasan and George Varghese. Faster IP Lookups using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, November 1998.

[SV00] Sandeep Sikka and George Varghese. Memory-Efficient State Lookups with Fast Updates. In *ACM SIGCOMM*, August 2000.

[SWKA00] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical Network Support for IP Traceback. In *ACM SIGCOMM*, August 2000.

[WVTP97] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable High Speed IP Routing Lookups. In *ACM SIGCOMM*, September 1997.

# A Preferential Dropping as a Rate-limiting Mechanism

Section 4.3 described the virtual queue as the mechanism used for rate-limiting aggregates. This section describes at alternate mechanism, preferential dropping, and contrasts this with the virtual queue.

When preferential dropping is used as the rate-limiting mechanism, the aggregate's arrival rate $r_a$ is estimated. Given the specified bandwidth limit $r_l$ for the aggregate, the rate-limiter drops each arriving packet in the aggregate with probability $1 - r_l/r_a$, so that the expected rate of traffic leaving the rate-limiter is the desired limit $r_l$. For example, if the arrival bandwidth $r_a$ is 10 Mbps and the target bandwidth $r_l$ is 7.5 Mbps, the incoming packets will be dropped with a probability $1 - r_l/r_a = 1/4$. When $r_l \geq r_a$, no packet will be dropped by the rate limiter.

Next we compare the preferential dropping mechanims with the virtual queue. The preferential drop mechanism and the virtual queue both test arriving packets, forwarding some and dropping others, so that the forwarded packets are roughly limited to the specified bandwidth. However, there are behavioral differences between the two mechanisms.

Both mechanisms must maintain some state: the preferential drop mechanism maintains state for rate estimation, and the virtual queue maintains state for the virtual queue simulation.

The virtual queue is more sensitive than the preferential drop mechanism to the short-term burstiness of the arrival process. For example, once the virtual queue is full, the virtual queue is likely to drop a burst of incoming packets, while the preferential dropping mechanism spreads out its drops more evenly.

The preferential drop filter does not guarantee that its output is no greater than the specified bandwidth; the estimate of the arrival rate is not necessarily accurate, and, because each packet is dropped with a certain probability, only the expected behavior of the preferential drop filter can be guaranteed. In contrast, the virtual queue precisely controls the exit rate, as a function of the queue size and service rate of the virtual queue. The preferential drop filter could keep dropping for a while after the arrival rate of the aggregate has been reduced, because it might take some time for the estimate of the arrival rate to reflect the actual reduction in the arrival rate. In contrast, the virtual queue will respond promptly to a slow-down in the arrival rate of the aggregate.

A deployment advantage of virtual queues is that they are already available in commercial routers [Cis98].

# B Pushback in more Detail

## B.1 Propogating Pushback Upstream

.

When propagating a pushback request upstream, the destination prefixes in the congestion signature might have to be narrowed, to restrict the rate-limiting to traffic headed for the downstream congested router only.

Consider the following scenario. Suppose the congested router X identifies a certain aggregate A with destination prefix 128.95.0.0. X will ask its upstream router Y (among others) to rate-limit traffic from aggregate A (128.95.0.0). However, Y cannot use the same specification directly because while Y could be forwarding 128.95.1.0 to X, it might not be forwarding the rest of 128.95.0.0 to X. If Y (and routers upstream of Y) started rate-limiting all of 128.95.0.0, the network would drop traffic which would not have reached the congested router.

To avoid this unnecessary packet-dropping, it is important that Y look at its routing table to find which prefixes within 128.95.0.0 are forwarded to X. Y has to check all extensions of the given prefix in the routing table. For example, if X specifies a prefix bbbb, Y would check bbbb0 and bbbb1. Any branch that does not exist is covered by the request, so no further searching is needed. If the branch exists, the algorithm is applied recursively. This check is reasonably cheap. Existing IP lookup schemes are based on either multibit tries or binary search of hash tables [SV00]. Both multibit tries [DBCP97, SV98] and hashing schemes [WVTP97] enable fast prefix lookups (in fact they lookup prefixes extracted from the destination address). Prefix lookups cannot be done in caches as caches usually store complete addresses, but this is not a limitation as even in the normal forwarding process cache misses are not an uncommon occurence [Par96]. It should also be noted that most of the times upstream routers will not have a longer prefix in the routing table, because it is not very

common for an upstream router to have longer prefixes than the downstream one (longer prefixes tend to occur closer to destinations).

### B.2 Pushback Request Messages

Pushback status messages are sent one hop downstream. Leaf nodes use timers to send status messages. A non-leaf node sends status message when it has received status messages from all its children. In case a child fails to send a status message in a round, the parent router will eventually timeout and send the status message downstream using the last value received from this child or its own estimate. The timer values for status messages are set based on the node's depth in the Pushback tree so that failure of one node does not trigger timeouts in all its ancestors and force them to use stale values.

### B.3 Pushback Refresh Messages

On receiving the Pushback refresh the upstream routers update the expiration time for the rate-limit session, the limit imposed on the aggregate and the aggregate specification (if it has changed). Timers, whose value depends on the depth in the tree, are set for status messages at this point. Routers which are not leaves in the Pushback tree send a refresh message further upstream after dividing the limit and checking with the routing table as to what prefix should be sent.

## C Local ACC

This section gives the pseudocode for the algorithms used for Local ACC. We must note that it does not specify the algorithms in minute detail; for this, the only reference, at the moment, is the pushback code in the NS simulator.[11] The sole purpose of presenting this skeletal pseudocode here is to give the reader a sense of the underlying algorithms.

Figure 13 gives the parameters needed by Local ACC. These include the time period $K$ and packet drop rate $p_{high}$ for defining a period of high congestion, the target packet drop rate $p_{target}$ for determining the rate limit, the maximum number *MaxSessions* of aggregates to be rate-limited at once, and the refresh time $T_{refresh}$ for reviewing the rate limits. In our simulations, we use a time period $K = 2$ seconds and packet drop rate $p_{high} = 0.1$, the target packet drop rate $p_{target} = 0.05$, *MaxSessions* of 3, and refresh time $T_{refresh}$ of 5 seconds.

**Parameters**

| | |
|---|---|
| $p_{high}$ | drop rate to trigger aggregate-based congestion control |
| $p_{target}$ | target ambient drop rate at OQ |
| $K$ | time period for checking high drop rate |
| $T_{refresh}$ | time period for reviewing the limit imposed on the aggregates |
| *MaxSessions* | maximum number of aggregates to rate-limit simultaneously |

**Variables**

| | |
|---|---|
| $R_{estimate}$ | arrival rate estimate |
| $R_{target}$ | (Link BW)/(1 - $p_{target}$) |
| *DropLog* | drop history |
| *LowerBound* | Arrival rate of biggest non-rate-limited aggregate (dynamically updated) |

Figure 13: **Some defintions used in the pseudocode**

Figure 14 describes the steps taken when the $K$-second timer to detect sustained congestion expires. The procedure executed on packet arrival to the queue is shown in Figure 15.

Figure 16 shows the procedures used to identify aggregates. The helper method *get_clusters()* analyses the *DropLog* and returns a sorted list of aggregates based on their arrival rates. It uses destination based clustering discussed in the Section 4.1, but a different defintion of aggregate can also be plugged in. The *identify_aggregate* procedure takes this sorted list and determines how many aggregates should be rate-limited and what the rate-limit should be. It then invokes the procedure *limit()* shown in Figure 17 to start rate-limiting the identified aggregates.

[11] The pushback code in the NS simulator is in "~ ns/pushback", and the validation test is in file "~ ns/tcl/test/test-all-pushback".

```
/* invoked on K-second timer expiry */
timeout()
   1.  Estimate p
       (drop rate at the queue)
   2.  if (p > p_high)
         identify_aggregate()
   3.  updateLowerBound()
   4.  Reset DropLog
   5.  Set up timer with K second delay

/* invoked to update lowerBound */
updateLowerBound()
   1.  Clusters = get_clusters()
   2.  lowerBound = arrival rate of first
       aggregate which is not rate-limited.
   3.  average lowerBound exponentially.
   (to get rid of temporary fluctuations)
```

Figure 14: $K$**-second Timer Expiry**

```
/* invoked when a new packet arrives */
enque(pkt)
   1.  Check if pkt belongs to a
         rate-limited aggregate.
   2.  if (yes)
         Update the arrival rate estimate
         of the aggregate
         Check if pkt needs to be dropped
         based on virtual queue state
         if (dropped) return
   3.  Update R_estimate
   4.  Check if pkt is to be dropped based
         on RED state
   5.  if (dropped) Log pkt in DropLog
       else Insert pkt in queue
```

Figure 15: **Packet Arrival**

The procedure *refresh()* in Figure 18 is executed every $T_{refresh}$ seconds to review rate-limiting; it modifies the rate-limit imposed on the aggregates based on the current conditions, and also stops rate-limiting of aggregates that have reduced their arrival rates.

```
/* invoked to get a sorted list of
   aggregates */
get_clusters()
   1.  High32 = List of destinations with
       more than mean number of drops in
       DropLog
   2.  High24 = List of 24 bit prefixes
       in High32
   3.  Clusters0 = List of prefixes after
       merging close prefixes in High24
   4.  Clusters1 = List of prefixes longer
       than those in Clusters0 but
       contaning most drops in the sub-tree
   5.  return sorted Clusters1
       (decreasing order of drops)

/*invoked when drop rate goes above p_high*/
identify_aggregate()
   1.  Clusters = get_clusters()
   2.  Estimate arrival rate of each
       prefix in Clusters using
       agg_arr =
         (R_estimate)* (agg_drops/total_drops)
   3.  R_excess = R_estimate - R_target
   4.  i=1, done=0, sum_rate=0
   5.  while not done:
       sum_rate+=Clusters[i].agg_arr
       L = (sum_rate - R_excess)/i
       if (L ≥ Clusters[i+1].agg_arr)
         done=1, break
       else
         if (i ≥ MaxSessions)
            break
         else
            i++
   6.  limit(Clusters[1..i],L)
```

Figure 16: **Identifying Aggregates**

```
/* invoked to install new aggregates
   for rate-limiting */
limit(List_new, L)
   1.  List_old = Aggregates already being
               rate-limited
   2.  Remove aggregates in List_old from
       List_new and change their limit to
       min(oldLimit,L)
   3.  Pick the top sending MaxSessions
       aggregates from union of List_old
       and List_new
   (Normally, the total in two lists would
   be less than MaxSessions, so all of them
   will be picked)
   4.  Install filters for new ones with
       limit L
   5.  Release (after some time) the old
       aggregates that were not picked
```

Figure 17: **Rate-limiting Identified Aggregates**

```
/* invoked every T_refresh seconds to
   review the limit on aggregates */
refresh()
   1.  N = Number of rate-limited
           aggregates
   2.  Limit_total = total limit on
                 aggregates
   3.  Arr_total = total arrival rate of
               aggregates
   4.  R_incoming = R_estimate − Limit_total + Arr_total
   5.  R_excess = R_incoming − R_target
   6.  L = (Arr_total − R_excess)/N
   7.  if (L < LowerBound)
          L = LowerBound
   8.  foreach A in (aggregates sorted in
         increasing order of arrival rate)
         if (A.arr < L)
         /* actual release happens after
         some time if A continues to have
         a low arrival rate */
            release A
            N--;
            L += (L - A.arr)/N
         else
            A.limit = (L << A.limit)?
                      L: (L+A.limit)/2
   9.  Set timer for next refresh
```

Figure 18: **Refresh**