RC24080 (W0610-071) October 16, 2006
Computer Science

IBM Research Report

On the Role and Controllability of Persistent Clients in Traffic Aggregates

Hani Jamjoom
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Kang G. Shin
University of Michigan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

On the Role and Controllability of Persistent Clients in Traffic Aggregates

Hani Jamjoom
IBM Watson Research
jamjoom@us.ibm.com

Kang G. Shin
University of Michigan
kgshin@eecs.umich.edu

ABSTRACT

Flash crowd events (FCEs) present a real threat to the stability of routers and end-servers. Such events are characterized by a large and sustained spike in client arrival rates, usually to the point of service failure. Traditional rate-based drop policies, such as Random Early Drop (RED), become ineffective in such situations since clients tend to be persistent, in the sense that they make multiple retransmission attempts before aborting their connection. As it is built into TCP's congestion control, this persistence is very widespread, making it a major stumbling block to providing responsive aggregate traffic controls. This paper focuses on analyzing and modeling the effects of client persistence on the controllability of aggregate traffic. Based on this model, we propose a new drop strategy called *persistent dropping* to regulate the arrival of SYN packets and achieves three important goals: (1) it allows routers and end-servers to quickly converge to their control targets without sacrificing fairness, (2) it minimizes the portion of client delay that is attributed to the applied controls, and (3) it is both easily implementable and computationally tractable. Using a real implementation of this controller in the Linux kernel, we demonstrate its efficacy, up to 60% delay reduction for drop probabilities less than 0.5.

1. Introduction

Flash crowd events (FCEs) and distributed denial of service (DDoS) attacks have received considerable attention from the mass media and the research community. They are characterized by a large and sudden increase in demand for both the network and end-server resources. Similar to natural disasters, both phenomena are relatively infrequent but leave devastating damages behind. Their initial effect is a dramatic reduction in service quality to clients sharing the network and the server. Even worse, sustained overload can bring networks and especially end-servers to a complete halt. The cause of this overload need not be intentional nor need be originated by malicious clients or applications. FCEs, unlike DDoS attacks, are generally caused by a very large number of legitimate users all targeting the same network or server. Their sheer traffic volume exhausts any available network and server resources. In addition to high arrival rate, there is a second cause that is commonly overlooked, namely, the persistence of individual clients accessing the server that can be responsible for increasing the aggregate traffic in an FCE by two folds.

This paper focuses on the control of aggregate traffic destined for web servers, which are the targets of flash crowds. Unlike video or audio traffic, web servers are generally dominated by short-lived connections. Several research efforts have focused on the detection of, and/or protection from, FCEs and DDoS attacks. In particular, Aggregate-based Congestion Control (ACC) is introduced to deal with such attacks by limiting the (high) rate of aggregate traffic at the routers to reduce the impact of the added load on the underlying network and end-servers [21, 23]. We observed, however, that the reaction of the underlying traffic to a rate-limiting policy can, and often will, reduce the effectiveness of the applied control. This can be better explained by decoupling aggregate traffic into two elements. The first element describes how existing or on-going connections react to the applied controls; the second element describes how the arrival of new connections is affected by the applied control, which we call *the persistent behavior of client requests* or *client persistence* for short. We find that the combination of the arrival of connection requests from new clients with TCP's reaction to packet loss—namely, retransmitting after a timeout—has an additional effect that is not accounted for by current traffic controllers. To improve the controllability of FCEs, we advocate the classification of incoming connection requests (into new SYN packets and retransmitted SYN packets) and applying specialized controls to each traffic class—a similar concept to [34]. We are interested in controlling the interactions between clients, the network, and the end-server.

Through the specialization of control, we are able to focus on new connection requests, the main ingredient of an FCE. We are also able to take into account the persistence of clients accessing the server. We propose *persistent dropping (PD)*, an effective control mechanism, which we prove it minimizes the client-perceived latency and the effective aggregate traffic (includes new and retransmitted connection requests) while maintaining the same control targets as regular rate-control policies. PD randomly chooses a number of requests based on a target reduction in the effective aggregate traffic arrival rate and systematically drops them on every retransmission. PD is well suited for controlling aggregate traffic as it achieves three goals: (1) it enables routers and end-servers to quickly converge to their control targets, (2) it minimizes the portion of client delay that is attributed to aggregate control by Internet routers and end-servers while maintaining fairness to all packets, and (3) it is both easily implementable and computationally tractable. We emphasize that PD complements, but does not replace, existing control mechanisms that are optimized for controlling already-established TCP connections [6, 13]. We also emphasize that PD does not interfere with end-to-end admission-control policies as it represents an optimization of existing queue management techniques.

This paper is organized as follows. We first look at the

The work reported in this paper was supported in part by the NSF under Grant CCR-0216977. A subset of this paper was presented at ACM SIGCOMM 2003.

OS	Client Behavior			Server Behavior		
	Drops SYN *RTO (sec)*	Drops connection w/ sending reset	Drops connection w/o sending reset (implemented by Linux only)	Separate Backlog Queue	Supports SYN Cookies	Sends reset on listen queue overflow
FreeBSD	3, 9, 21, 45	Connects using 3-way handshake and sends a request packet. The server drops the connection and will either send the RST packet after the connection is dropped or after receiving the request packet. The client will then abort the connection	Connects using 3-way handshake and sends a request packet. The server drops the connection and will keep dropping subsequent packets from the client. The client will time out and retransmit the request packet. The timeout intervals are based on the computed RTT values during the connection establishment phase.	Yes* [ver < 4.5] Syncache [ver ≥ 4.5]	Yes	Yes
HP-UX 11	2, 3, 9, 21, 45			Yes	No	Yes
Linux 2.2/2.4	3, 9, 21, 45			Yes	Yes	Optional
Solaris 2.7	3.4, 10.1, 23.6, 50.6, 104.6, 164.6			Yes*	No	Yes
Win 9x, NT	3, 9, 21			No‡	No	Yes
Win 2000	3, 9			Yes‡	No	Yes

* The length of the SYN backlog queue is based on the length of the listen queue but multiplied by a fudge factor (often 2/3).
‡ Dynamic backlog was introduced in NT SP3 and windows 2000 to improve reaction against DoS attacks. Publicly available information does not specify if Windows uses a separate queue to hold incomplete connections. However, the length of incomplete connections can be separately specified in Windows NT SP3 and 2000, which, effectively, is the same as having a separate SYN backlog queue.

Table 1: Retransmission behavior for different OSs. The measurement assumes default OS configuration. Some parameters such as the timeout before the connection is aborted, can be dynamically configured.

persistent behavior in aggregate traffic in Section 2. We then propose a PD controller to deal with persistent clients in Section 3 and discuss implementation issues in Section 4. In Section 5, we experimentally evaluate some performance issues. The paper ends with related work and concluding remarks in Sections 6 and 7, respectively.

2. Persistence in Aggregate Traffic

As mentioned in Section 1, a more accurate representation of aggregate traffic accounts for those requests that belong to new clients entering the system and for those resulting from client persistence (i.e., requests that are retransmissions of previously-dropped ones). However, many factors contribute to the persistence of clients, whereby the client keeps trying to access the server (normally at a later time) even after server overload or network congestion is detected. Some factors of this persistence are embedded in the applications and protocols that clients use. These are not design flaws, but are often necessary to the proper operation of clients, e.g., TCP congestion control. Other factors are due to purely human habits. In this subsection, we investigate how the combination of TCP congestion control and rate-based queue-management techniques in routers and end-servers may raise the severity of FCEs. We use a simple model where a client issues a single Hyper-Text Transfer Protocol (HTTP) request using a separate TCP connection. This model allows us to study a single TCP connection in isolation and is representative of many browser implementations that issue HTTP requests in parallel to maximize their throughput [20].

Consider what could happen to our simple client's request during an FCE. Before examining the consequences of the request packets being dropped by routers or end-servers during the various stages of the request processing, we outline the stages that a successful request must go through before completion. The first stage of request processing is the three-way handshake. In this stage, the client sends a SYN packet to the server by performing an *active open*. The server then performs a *passive open* by queueing the SYN packet in a global backlog queue (with possibly per-application-port quotas) where proper structures (e.g., `skbuff` in Linux) are allocated and a SYN-ACK packet is sent back to the client. At this point, the connection at the server is said to be *half open*. In most operating systems (OSs), SYN packets are processed in the kernel independently from the corresponding application. Upon receiving the SYN-ACK packet from the server, the client sends an ACK packet followed immediately (in most implementations) by the request's meta-data. At the server, the client's ACK causes the half-open connection to be moved to the listen queue for processing by the application. Data packets are then exchanged between the server and the client to complete the request; the connection is optionally closed.

During overload, packets can get lost at different stages of request processing and at different points between the source and destination. Here we consider loss of packets on the path from the client to the server. An equivalent behavior occurs on the reverse direction; it is omitted for space consideration.

When a typical queues fills up (whether it is a router or a server SYN backlog queue) and packets are dropped, the request can be in the connection-establishment stage or the connection has already been established. In the first case, each time a SYN packet or its corresponding response is lost, an exponentially-increasing retransmission timeout (RTO) is used to detect the packet loss and the SYN packet is retransmitted.[1] The RTO values used by different client OSs are listed in Table 1. Established connections, in the latter case, detect and recover from packet loss in ways that are more complex. These have been investigated by several studies, both empirically and analytically, e.g., in [5, 27, 28].

To better understand the dynamics of FCEs, we extend some of the results in [5] that pertain to connection establishment. We follow the same modeling assumptions in [5] and build on

[1] Most TCP stack implementations follow Jacobson's algorithm [17, 30], where a SYN packet that is not acknowledged within an RTO period is retransmitted, but with the previous RTO period doubled. This is repeated until the connection is established or until the connection times out, at which point the connection is aborted.

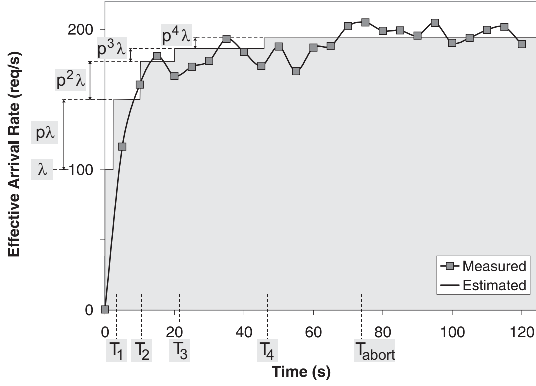

Figure 1: Increase in aggregate traffic under the persistent client model.

its estimate of the connection-establishment latency. We thus assume that end-points adhere to a TCP-Reno style congestion control mechanism [17].[2] However, to draw general conclusions for the entire aggregate, we must also characterize the arrivals of new connection requests: namely, their arrivals follow a Poisson process. Since the corresponding interarrival times are independently and identically distributed (i.i.d.), dropping one request packet does affect the arrival of future request packets. This matches well the observation that clients behave independently. It, however, does not consider the inter-dependency between requests from a single client. This is considered in Section 3. Furthermore, under a uniform drop policy, if we split incoming SYN requests into different streams, each representing SYN packets that have been retransmitted an equal number of times, then each stream can be approximated by a Poisson arrival process. This is a direct consequence of the fact that the RTO is measured from the client's transmission time of the SYN packet and the variance in the RTO value is very small, especially when compared to the RTO's time granularity [18].

For a new connection, consider the retransmission epochs of a dropped SYN packet as T_i, where $i = 0, \ldots, n$ represents the number of times the corresponding SYN packet has been dropped and n is the maximum number of connection attempts before aborting a connection. Let T_{abort} be the maximum time a connection waits before aborting. Note that $T_n \leq T_{abort}$. Since FCEs causes congestion on the path from the client to the server, we also consider the drop probability, p, in the forward direction. Extending the results to include drops in the reverse direction is trivial and omitted for space consideration. The expected connection-establishment latency EL_h can be expressed as:

$$EL_h = \sum_{j=0}^{n} [p^j(1-p)(T_j + RTT)] + p^{n+1}T_{abort}, \quad (1)$$

where p^{n+1} is the probability that the connection times out and RTT is the mean round-trip time. The first term, then, represents expected latency of successful connections, or $(1 - p^{n+1})E[L_h|x \text{ succeeds}]$, which was derived in [5]. By the independence assumption, it can be easily shown that EL_h is also the mean expected connection-establishment latency of all requests.

Under our network model, we also derive Λ, the effective or aggregate arrival rate of SYN packets. This aggregate is a collection of newly-transmitted requests and retransmission of the previously-dropped ones. It is divided into multiple streams, each representing the number of transmission attempts or *transmission class* of the corresponding connections. We denote the stream of initial transmission attempts by λ_0 (i.e., SYN packets on their first transmission), the stream of first retransmissions by λ_1, and so on, up to λ_n. Then, the effective mean arrival rate Λ is:

$$\begin{aligned} \Lambda &= \lambda_0 + \lambda_1 + \cdots + \lambda_n \\ &= \lambda_0 + p\lambda_0 + p^2\lambda_0 + \cdots + p^n\lambda_0 = \frac{1-p^{n+1}}{1-p}\lambda_0. \quad (2) \end{aligned}$$

Notice the simple relationship between the arrival rates of the different transmission classes. For example, the arrival rate of the first retransmission class, λ_1, is just the arrival rate of the initial transmissions, λ_0, times the drop probability p. Based on Eq. (2), a 50% drop will, in theory, increase the amount of SYN requests by 75%.[3] In fact, Eq. (2) shows that a typical rate controller only causes $p^{n+1}\lambda_0$ connections to time out; for the rest, increasing the number of retransmissions has a substantial impact on client-perceived delay as shown in Eq. (1). We argue that this probability, which we call the *effective timeout probability* (p^*), reflects the true impact of the control mechanism on the underlying traffic.[4]

We used real measurements to illustrate the effects of client persistence on the underlying traffic. A Linux-based machine ran Eve [19] to emulate clients arriving independently with exponentially-distributed inter-arrival time and with a mean arrival rate of 100 requests/sec. The server machine ran a home-grown server emulator, which is configured to emulate a server with a buffer capacity of 50 requests (i.e., listen buffer size = 50) and processes one request at a time with exponential service times with mean 1/200 sec. A third machine is configured as a router running NIST Net to emulate a WAN with a uniform one-way delay of 50 msec and drops incoming requests with a fixed probability (here, it was 0.5). The three machines are connected via FastEthernet link and are ensured to be bottleneck free. Finally, we used tcpdump to collect our measurements. Figure 1 plot the actual arrival rate of request from the client to the server as well as our estimated behavior. The figure illustrates some of the ideas behind our derivation, especially showing how traditional rate-based dropping can underestimate the resulting aggregate traffic.

[2] Other congestion control mechanisms can be assumed as long as they use Jacobson's algorithm to recover from SYN packet loss.

[3] When all packets are dropped, $\Lambda = (n+1)\lambda_0$.

[4] While Eq. 2 assumes a Poisson arrival process for new client arrivals, which is not affected by drops, in reality, a client may visit a server multiple times, which clearly can be affected by drops and timeouts. The degree that a client's future visits are affected by SYN drops will depend on several factors, which are described earlier. This paper only focuses on visits from new clients, which is a serious issue during an FCE.

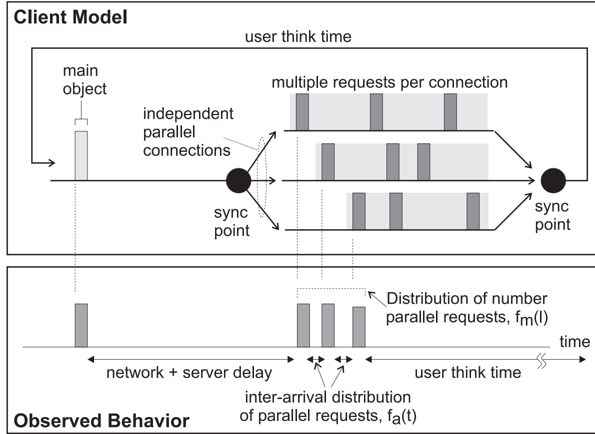

Figure 2: Persistent client model.

It is worth noting that unlike router queues, when the backlog queue at the server fills up, the server can be configured to drop the SYN packets and send SYN cookies to the client. A SYN cookie is simply a method for the server to avoid storing any state for half-open connections. In this case, a challenge is sent to the client and upon its return, the server can establish the connection as if the original SYN packet were queued properly in the backlog queue. The challenge is encoded in the TCP's sequence number and, thus, does not require any client modification. When SYN cookies are lost, the client times out and retransmits the request as described above. SYN caches are an alternative method to SYN cookies, which allow the server to store a large number of SYN packets by simply delaying the creation of connection data structures until the three-way handshake is completed [22]. Depending on the size of the cache and the arrival rate, SYN caches can fill up just like SYN backlog or router queues. Table 1 shows the OSs that support SYN cookies and SYN caches. Both SYN cookies and SYN caches are effective in handling a flood of SYN packets, the majority of which are spoofed (or fake). The mechanism relies on the fact that only a small portion of the SYN-ACKs will be replied back, after which the TCP connection is fully established. When requests originate from legitimate clients, both mechanisms can increase the additional work on the end-server as the resulting fully-established connections (from the clients' perspective) are dropped due to insufficient room in the application listen queue.

3. Controllability of Persistent Clients

Client persistence imposes an added challenge to the controllability of aggregate traffic. Because clients often send their requests in parallel to maximize their throughput, if that dropped request was part of a web page, the client may decide to reload the entire page causing multiple requests in future. It is an important problem in flash crowds. If a router or end-server is operating near or at full capacity, then any slight increase in load will trigger dropping of requests. These persistent requests, upon their retransmission, will set off further drops, creating a vicious cycle of drops causing future drops. A system that is trapped into such a cycle will require a long time to recover after the load subsides. As shown earlier, repeated dropping dramatically increases the client-perceived latency as it may require several timeouts before a client successfully establishes a connection.

A traffic controller that drops incoming requests must, therefore, deal with its retransmission in the future. To this end, we introduce *persistent dropping*, a novel drop strategy that chooses a small number of requests based on a target timeout probability and systematically drop them on every retransmission. We show that this drop policy minimizes the client's expected response time, the number of retransmissions, and the bandwidth requirement of the aggregate traffic. We also show that this technique does not affect the fairness of the control policy.

3.1 Modeling of Persistent Clients

Because our drop strategy is based on analytical optimization, the first step is to create accurate model of web clients. As with any analytical approach, the model has to be simple enough to allow for analytical tractability, yet accurate enough to reflect realism. To date, many of the analyzed models have been too simplistic as they assume independent client (or request) arrivals, with or without balking. One should not confuse modeling with characterization. The latter performs a similar analysis to what we did in the previous section to produce different characteristics (mostly using probabilistic distributions) of client behavior under certain load conditions. In fact, several studies have empirically studied the interaction between the client, network, and end-server to characterize the dynamics of underlying traffic [2, 10, 11]. Unfortunately, such studies often lack the clients' response to different control policies, which is the main ingredient for constructing effective controllers.

We are faced with the question of whether an effective traffic controller can be built without exact knowledge of client behavior. We argue that an optimal controller can be realized by approximating the internal structure of web clients. The model of persistent clients is presented in Figure 2; it captures the following four important elements.

- E1. Individual clients are independent of each other, and a client's requests are grouped into *visits*. Each visit represents a client accessing a web page and its entire content. Requests within a visit are correlated by the completion of the initial page that contains all the embedded links.
- E2. Once the main page is fetched, a batch of l parallel connections with probability distribution $f_m(l)$ are created to request the embedded objects with arrival distribution $f_a(t)$. We do not specify the exact distributions for $f_m(l)$ or $f_a(t)$, but in our subsequent derivations, they are assumed to be independent and have finite means. Moreover, the retransmissions of lost packets from parallel connections are independent as long as none of the connections is aborted.
- E3. The expected visit completion time, EV, is the sum of the time it takes to fetch the initial page and the longest finish

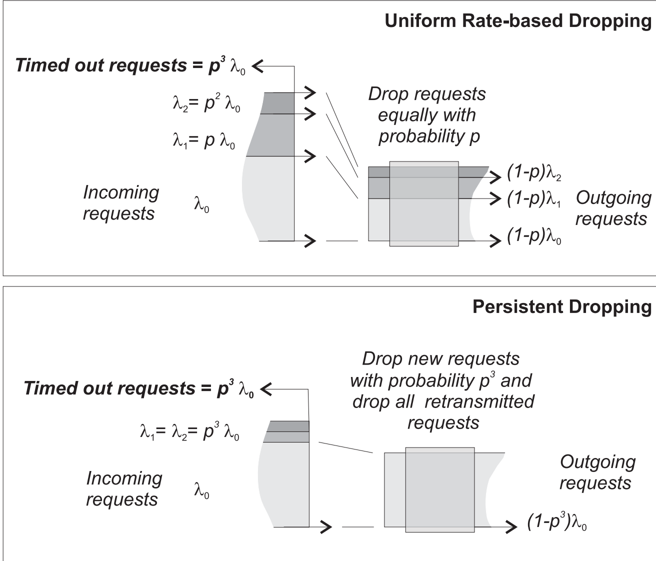

Figure 3: Illustrative comparison between rate-based dropping and PD. We view the outgoing link as a smaller pipe than the stream of incoming requests. We then show how the two strategies drop incoming requests to fit them into the pipe.

time of all parallel connections. Formally, consider p^* as the effective timeout probability, $q^* = 1 - p^*$ as the probability of success, m as the expected number of parallel connections, and γ as the mean interarrival times of the parallel connections. Also, consider ET_c as the expected latency for completing a single request (we consider better estimates for ET_c in Section 3.5). Then,

$$EV = p^* T_{abort} + q^* \left[ET_c + (1-(q^*)^m)(T_{abort} + \Psi) + (q^*)^m (ET_c + \Psi)\right]. \quad (3)$$

The second term in Eq. (3) estimates the expected delay when the first page is fetched successfully. The term $(1-(q^*)^m)$ represents the probability that at least one of the m connections times out and $\Psi = m\gamma$ is the approximate overhead of launching m parallel connections. Thus, the last product term in Eq. (3) is the expected delay for completing the parallel requests. It is derived by taking the expectation of their maximum completion time.

E4. A client may visit multiple pages within a web server before leaving the server. This is often referred to as a *user session*. The expected session time can be estimated in a manner similar to E3; it is omitted for space consideration.

In the absence of packet loss, our model is consistent with earlier ones (Observed Behavior in Figure 2) where it is assumed that a client sends a batch of closely-spaced connection requests (*active period*) followed by a relatively long period of user think time (*inactive period*) [2]. Our distributions have similar characteristics to those in [2, 9, 11] with very different distribution parameters. Our model, however, captures the effects of the applied control, which we use to construct our controller.

3.2 Persistent Dropping

Consider an Active Queue Management (AQM) technique that drops incoming SYN packets with probability p. Here, we do not consider how other packets are treated, and p is set in accordance with the underlying AQM technique. For instance, if packets are dropped in a router using RED, then p is based on dynamic measurements of queue length [13]. Alternatively, consider an unmodified SYN queue at the server. Here, the *drop-tail* queue at the server can be considered as a passive control mechanism where SYN requests will be dropped whenever the queue fills up. Averaging over any time window, the drop probability, p, can be computed by simply dividing the number of dropped SYN packets by the total number of arrivals over that time window. We, thus, view p as the percentage of packets that must be dropped regardless of how its value is chosen. Given a target drop probability p (or equivalently an effective timeout probability, p^*, as described earlier), our goal is to find the optimal drop policy that minimizes the effective arrival rate, Λ, and connection-establishment latency, EL_h. We base our development on the same network model introduced in Section 2, and still do not consider the parallelism of individual clients. This is addressed in Section 4.

The optimality is established by first looking at the retransmission epochs of individual connections. As shown in Section 2, given packet-loss probability p, estimates for connection-establishment latency and effective arrival rate can be derived. We use these relationships as bases to show that a drop policy that consistently drops retransmitted requests is able to minimize the additional latency that is caused by the applied control as well as minimize the aggregate traffic of all incoming requests.

Traditionally, a control policy that drops aggregate traffic with probability p does not take into account the transmission class of individual connections. Consider here a different mechanism that associates a drop probability p_i with each transmission class i. In order to assign these probabilities, we assume that incoming requests are classified into their corresponding transmission classes; we will show later how this can be achieved. Let us rewrite the aggregate arrival rate, Λ, in Eq. (2) using the per-class drop probabilities p_i's:

$$\Lambda = \lambda_0 + p_0\lambda_0 + p_0 p_1 \lambda_0 + \cdots + \left(\prod_{i=0}^{n-1} p_i\right)\lambda_0. \quad (4)$$

Notice here that the effective timeout probability is $p^* = \prod_{i=0}^{n} p_i$. For a traditional rate control policy, all requests are dropped with an equal probability (or $p_i = p$ for $\forall\, i$), which implies that $p^* = p^{n+1}$ (consistent with the results in Section 2).

To minimize the expected connection-establishment latency of clients, we start by writing the probability mass function of the connection-establishment latency using the per-class drop probabilities:

$$P\{L_h(x)=t\} = \begin{cases} (1-p_0) & \text{if } t = T_0 + RTT \\ \prod_{j=0}^{i-1} p_j(1-p_i) & \text{if } t = T_i + RTT, 1 \le i \le n \\ \prod_{j=0}^{h} p_j & \text{if } t = T_{abort} \\ 0 & \text{otherwise,} \end{cases} \quad (5)$$

where T_i is the time of the ith retransmission, $T_0 = 0$ is the time of the initial transmission, and T_{abort} is the time before the connection times out. Intuitively, Eq. (5) establishes the probability of connecting successfully, for example, after $RTT + T_2$ seconds, which represents the connection request being dropped during its initial transmission with probability p_0, then being dropped again on the second transmission with probability p_1, and finally succeeding on the third transmission attempt with probability $(1 - p_2)$. Notice that the minimum connection-establishment latency is $t = RTT$. Based on this, the expected connection-establishment delay, EL_h, can be computed as:

$$\begin{aligned} EL_h &= (1-p_0\cdots p_n)RTT + \\ & p_0(1-p_1)T_1 + p_0p_1(1-p_2)T_2 + \cdots + \\ & (p_0\cdots p_{n-1})(1-p_n)T_n + (p_0\cdots p_n)T_{abort}. \end{aligned} \quad (6)$$

The optimal drop strategy must, therefore, minimize EL_h with the constraint of having an effective timeout probability that is equal to the one obtained by traditional policies, i.e., $\prod_{i=0}^{n} p_i = p^*$. It suffices to show that if we set $p_0 = p^*$ and $p_i = 1$, for $i \neq 0$, then EL_h is minimized. This can be seen by observing that each term in Eq. (6) cancels out except for the last term. The minimum connection-establishment latency is then $EL_h^{g^*} = (1 - p^*)RTT + p^*T_{abort}$, where g^* denotes our optimal policy. Note that since $EL_h^{g^*}$ no longer has a delay component for successful connections, g^* breaks the dependency between the delay for successful connections and packet drop.

The above discussion implies that the optimal policy must decouple connection requests that belong to new connections (i.e., on their first attempt) from those that are not. Viewed another way, this is a form of low-level admission control where a new connection request can either be admitted into the system or denied access. But denying access at the connection-establishment level can be performed by either (1) sending back an explicit reject packet, such as an RST packet, notifying the sender to terminate the initiated connection, or (2) repeatedly dropping packets on every retransmission attempt. Unfortunately, the success of the first approach is predicated on the sender's cooperation. This issue is addressed in [18].

Based on the above discussion, we introduce *persistent dropping* (PD) as the optimal drop strategy that chooses $p^*\lambda_0$ new requests and systematically drop them on every retransmission. An example of PD is illustrated in Figure 3, where it shows how this new technique intelligently fills the outgoing link to minimize packet retransmissions. With persistent dropping, the resulting (or the aggregate) drop probability is substantially lower than traditional rate-based techniques. Specifically,

	Rate-Based Random Drop	Persistent Drop
Drop probability of new requests	p	p^{n+1}
Effective timeout probability, p^*	p^{n+1}	p^{n+1}
Expected connection-establishment latency, EL_h	$(1-p^{n+1})$ RTT + $\sum_{j=0}^{n}[p^j(1-p)T_j]+p^{n+1}T_{abort}$	$(1-p^{n+1})$ RTT + $p^{n+1}T_{abort}$
Expected number of retransmissions	$\frac{1-p^{n+1}}{1-p}$	$1+np^{n+1}$
Effective arrival rate, Λ	$\frac{1-p^{n+1}}{1-p}\lambda_0$	$(1+np^{n+1})\lambda_0$

Table 2: Comparison between PD and random dropping.

$$\begin{aligned} p^{g^*} &= \frac{p_0\lambda_0 + p_1\lambda_1 + \cdots + p_n\lambda_n}{\lambda_0 + \lambda_1 + \cdots + \lambda_n} \\ &= \frac{p_0\lambda_0 + p_0p_1\lambda_0 + \cdots + \left(\prod_{i=0}^{n} p_i\right)\lambda_0}{\lambda_0 + p_0\lambda_0 + \cdots + (p_0p_1\cdots p_{n-1})\lambda_0} \\ &= \frac{(n+1)p^*}{1+np^*}. \end{aligned} \quad (7)$$

Table 2 compares the performance improvement of PD over a traditional rate-control policy in terms of mean client-perceived latency, average number of retransmissions, and aggregate arrival rate for the same effective timeout probability. In Section 5, we also compare the variance in the latency, L_h, of the two schemes—they can be directly computed using Eq. (5).

3.3 Fairness of Persistent Dropping

Albeit counterintuitive, the fact that PD chooses certain connections and consistently drop them on every retransmission does not imply that it is unfair. In this subsection, we show that PD has equal (or better) fairness when compared to traditional rate-based schemes.

Here, fairness is defined as giving each incoming request an equal probability of being accepted, given that p^* has not changed by the underlying controller. One, however, must be careful when specifying what constitutes an *incoming request*. Depending on whether incoming requests represent only new SYN packets or include all SYN packets (i.e., new and retransmitted ones), different types of fairness arise. To see this, consider the example of two clients. Client *A* has T_{abort} set small enough such that retransmissions are not allowed. Client *B* has T_{abort} set to allow for a single retransmission. Given equal treatment of incoming requests, client *B*'s request has a higher probability of getting accepted (i.e., $p + (1 - p)p$) than client *A*'s request (i.e., p). We, thus, distinguish between two types of fairness. The first is the *instantaneous fairness*, which measures the deviation in acceptance probability looking only at new SYN packets, ignoring any retransmissions. The second is *steady-state fairness*, which measures the deviation in acceptance probability accounting for all packets. Given these definitions of fairness, we can see that traditional rate-based schemes are fair from the perspective of instantaneous fairness, but un-

fair from the perspective of steady-state fairness (i.e., clients with different T_{abort} values have different acceptance probabilities). On the other hand, PD provides both steady-state and instantaneous fairness.

One can alternatively define fairness in terms of the resulting reduction in arrival rate. We have shown that a rate-control drop policy generally causes $p^*\lambda_0 = p^{n+1}\lambda_0$ connections to time out with an equal probability. This is identical (but less efficient due to the increased delay) to PD. Under this definition, both PD and rate-control policies yield the same fairness.

A question that arises is what happens if we design a scheme that gives retransmitted requests higher priority. That is, given two requests, one new and one retransmitted, the traffic controller would drop the new request first before dropping the retransmitted. Intuitively, this should appear preferable since those that have already been penalized in the past should get a better chance of being accepted in the future. However, one has to consider the goal of PD: controlling aggregate traffic when incoming requests are greater than available resources. Hence, this alternate scheme will not produce any favorable outcome. In fact, it follows directly from our derivation that this scheme will reduce the resulting number of accepted requests and will increase the average connection delay. When incoming requests are bursty, some tradeoffs can be made to maximize the acceptance rate while maintaining acceptable connection-establishment delays. We have explored this issue in [18], where we proposed a complementary mechanism to PD to deal with burstiness.

One possible argument against PD is that it may encourage users to press the reload button or possibly deploy a more aggressive TCP. However, this is the case for any rate-based scheme. Since those requests that will time out in a rate-based scheme will experience very long delays, they too will have a similar incentive to be more aggressive. As we show in Section 5, PD can be configured to further reduce the effects of aggressive users.

3.4 Applicability to Network of Queues

In most cases, requests must pass through multiple queues as they traverse different links on the network before reaching their destination. Fortunately, the above results also hold in this scenario, namely, when requests pass through a series of queues, each using a persistent drop policy g^*, the client's establishment latency and effective arrival rates are minimized. This is illustrated in Figure 4 where we assumed for simplicity that all queues have the same drop probability p. We see that for a rate-based drop strategy, the probability of a single request succeeding on a single attempt is $(1-p)^m$, where m is the number of queues that it must pass through. In contrast, the PD policy g^* has a probability $(1-p^{n+1})^m$. To put this in perspective, if $m = 5$, $p = 0.05$, and $n = 4$, then the probability of a request succeeding is 0.77 and 0.99 for the uniform rate-based and the PD policies, respectively. Using a similar development to our single queue analysis, we can prove that g^* is the optimal drop strategy even when each queue uses a different drop probability.

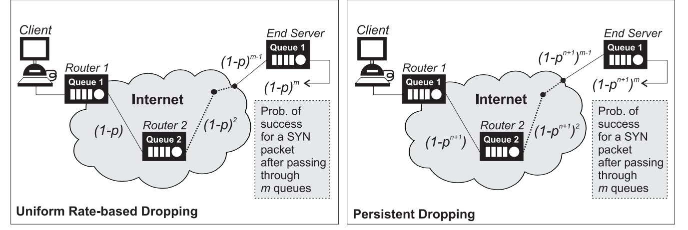

Figure 4: Probability of success in network of queues

3.5 Applicability to Persistent Clients

The development in the previous sections treated clients' connections as independent entities without considering the correlation between a group of connections originating from the same client (e.g., client visits or sessions as defined in Section 3.1). It is not difficult to verify PD's optimality in the case of correlated connections. Under the assumption that the controller does not distinguish between SYN packets that belong to an already-admitted visit and those that represent new visits, we provide here an intuitive sketch of the proof. Consider $EL_s = E[L_h|\text{ connection succeeds}]$ as the conditional expectation of the connection-establishment latency for successful connections. This is equivalent to Eq. (6), but excludes the last term and divides by $(1-p^*)$ to compensate for unaccounted timed-out connections

$$EL_s = RTT + \frac{p_0(1-p_1)T_1 + \cdots + (p_0 \cdots p_{n-1})(1-p_n)T_n}{1-p^*}. \quad (8)$$

For an established connection, let ET_s be the average time to send the request, process it by the server, and receive the reply. Estimates for ET_s are derived in [5, 27] as part of determining the expected latency of a TCP connection. We can now substitute the expression of EL_s into Eq. (3) using the relationship $ET_c = EL_s + ET_s$ to obtain EV as a function of per-class drop probabilities. Similar to the development in Section 3.1, when $p_0 = p^*$ and $p_i = 1$ for $i \neq 0$, EV is minimized.

4. Designing a Persistent Dropping Queue Manager

In the previous section, we have showed the optimality conditions of an admission-like control policy that can be implemented at the TCP-level protocol to minimize the portion of the client's delay that is due to queue management in routers or end-servers. Implementation of our optimal drop policy in routers and end-servers relies on the ability to (1) detect FCEs, (2) group requests originating from the same client visit, and (3) distinguish between new and retransmitted requests. If PD is to be deployed at servers or front-end switches, then SYN drops can be monitored as an effective way to detect FCEs. *Adaptive Packet Filters* can be used for deployment of PD [31]. Unfortunately, requirements (2) and (3) present design challenges, especially since we intend for our technique to operate at the packet-level. In fact, precise implementation requires violation of the protocol layering, similar to Layer-7 switches (e.g., Foundry, Alteon) to satisfy requirement (1) and need per-connection state information to satisfy requirement (2). How-

ever, one must not forget the original environment that this is intended for: large aggregate traffic causing an FCE. We are, thus, interested in constructing approximate implementations that are allowed to be less accurate than an exact implementation, but significantly improves on existing techniques.

The basic idea is to use an "appropriate" hash function to group requests from the same client and then, based on the mapping, decide to drop or allow packets to go through. The controller's logical operation is organized into two parts: classification and policy enforcement. The classification splits incoming requests into two streams, one representing new transmissions for new client visits and the other representing retransmitted requests. Policy enforcement then drops new connection requests with an equal probability, p^*, and drops retransmitted requests with probability 1.

The selection of a suitable hash function, $h(.)$, is not difficult. In fact, as we will show shortly, a simple XOR operation on the input parameters produces the desired uniform hashing [8]. On the other hand, we found that choice of the input parameters to the hash function is the most critical element in our design. Unfortunately, without the client-side's cooperation, packet-level information provides limited choices in achieving the desired classification. They are summarized as follows. We abbreviate IP source and destination addresses and TCP source and destination ports with src_addr, $dest_addr$, src_port, and $dest_port$, respectively.

H1. $h(src_addr, dest_addr)$: The src_addr allows per client classification and, with the combination of $dest_addr$, allows approximate user-session classification. Unfortunately, it is relatively coarse-grain classification since clients connecting through a proxy or a NAT (Network to Address Translation) service are treated as a single client. In case of high aggregate traffic, this seems to be an acceptable trade-off. It can be further improved by storing a separate list of high-priority IP addresses that contain preferred proxy servers (e.g., AOL, MSN). Packets originating from these addresses can then be excluded from dropping as long as the control target is met.

H2. $h(src_addr, dest_addr, src_port, dest_port)$: The combination of the four elements allows accurate connection-level classification even through proxies and NAT services. It, however, loses session semantics, which, as we show, still provides a considerable performance improvement over traditional mechanisms.

H3 $h(src_addr, dest_addr, IP\ options)$: One alternative solution is to require clients to encode their session information using IP options. This will produce the most accurate classification. It is, however, impractical as it requires client stack modification as well as high router overhead to process IP options. We will not investigate this alternative any further.

Since this classification must be performed at very high speeds, the hash function must be simple, yet still provide uniform hashing. We observe that the uniqueness of the source IP address, and when combined with the TCP port information, the probability of collision is minimized. We used a simple XOR operation to perform the required mapping:

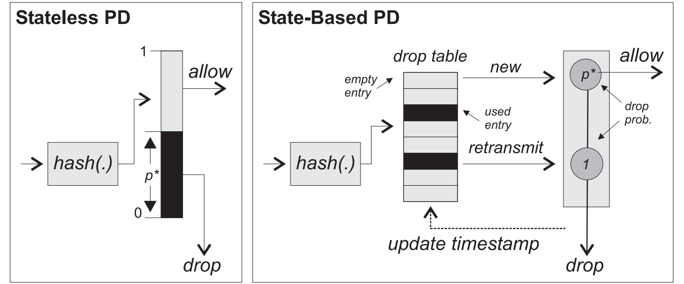

Figure 5: Stateless and state-based implementation of persistent drop controller.

$$h(x_1, x_2, \ldots, x_k) = x_1 \oplus x_2 \oplus \cdots \oplus x_k \times K(t) \bmod R, \quad (9)$$

where $K(t)$ is an appropriately-selected prime number that we use to randomize the hashing function (to be described shortly) and R is the range of the hash function. We performed a simple simulation, where IP addresses are randomly chosen and long runs of consecutive port numbers are used (since consecutive port numbers are commonly used by the underlying OS when multiple connections are issued). The distribution was almost uniform as we hoped and expected.[5]

We came up with two schemes to perform the desired classification: one is a stateless implementation and another stores a small per-connection state. We assume here that a preferred proxy list described in H1 is handled using a separate lookup operation.

4.1 Stateless Persistent Dropping (SLPD)

Upon arrival of a new connection, the hash in H1 or H2 is computed and normalized to a number within the range [0,1]. A threshold value, represented by the effective timeout probability, p^*, is used to drop those packets that have a hash value less than p^* and allow the rest to pass through (Figure 5). Depending on whether H1 or H2 is used, client- or connection-level persistent dropping can be achieved, respectively. The absence of state makes this scheme very simple to implement and fast to execute. However, this scheme can be unfair as it discriminates against a fixed set of clients. To mitigate this problem we use the term $K(t)$ in Eq. (9) to periodically change the function's mapping, hence its dependence on t [3]. The time interval between changes should be on the order of several minutes to minimize the error introduced by changing the set of dropped packets.

[5]Unfortunately, we are unable to use publicly available logs because they always anonymized, making logged IP will be uniformly distributed, which defeats the point of using an access log. We have used private logs from The University of Michigan's Electrical Engineering and Computer Science department. These logs were small and highly localized. Even then, the results seemed consistent.

4.2 State-based Persistent Dropping (SBPD)

Especially when connection-level control is desired (H2), storing a small (soft) state for each connection can further improve the accuracy of the classification. A hash table is used here to store the time at which a *new* request is dropped. Upon its retransmission, the controller is able to look up the request's initial drop time, and based on the age of the retransmission, determine the transmission class. Using hash tables is an efficient way to compactly organize the request's information such that its storage and retrieval are very efficient. The hash function described in H2 can be used to map the set of possible request headers into a much smaller number of table indices.

The basic operation of SBPD is split into two stages (Figure 5). The first stage consults the table to see if the request is a new or a retransmitted one. A table entry stores the time of the first drop time. Therefore, any incoming request that is mapped to a used entry is systematically marked as "retransmission" for a T_{abort}^{max} second window from the initial drop time. The window length is chosen based on the timeout value among most OS implementations. If the entry is empty or has an expired time-stamp, the request is marked as "new." The second stage of SBPD decides the control policy. Obviously, a request that is marked as "retransmission" is dropped. However, one that is marked as "new" is dropped with probability p^* and the hash table is appropriately updated.

Besides the hash function, there are two components that are important for an effective implementation of SBPD: the size of hash table, and the information stored in each entry. The size of hash table, M, determines the probability of collision between two requests. Recall that a collision occurs when two requests hash into the same entry. For a uniform hash function, the expected number of collisions at each table entry is K/M, where K is the number of possible request keys. To minimize the lookup and storage overheads, we do not store requests that are hashed into an occupied table entry (e.g., using chaining or open addressing). Consequently, M should be designed to reduce the probability of collision; it depends on the expected arrival rate, λ_0, and the connection timeout period, T_{abort}^{max}. The maximum (expected) values for these two parameters then dictate the worst-case scenario for which M should be provisioned. Assuming that our hash function is truly uniform, at most λ_0 requests may need to be dropped (and stored) per second.[6] But requests must be tracked for at least T_{abort}^{max}, so the table size is computed as:

$$M = \lambda_0 \times T_{abort}^{max} \times \sigma, \qquad (10)$$

where $\sigma \geq 1$ is an over-design factor that further reduces the probability of collisions; our experimental results have indicated that $\sigma = 1.2$ is adequate. For example, if we want to implement predictive drop that can accommodate the following specification: $\lambda_0 <$ 1,000 reqs/s, and $T_{conn} < 45$ s, then $K =$67,500 entries.

[6] In fact, even if we are designing for small $p*$ values, we cannot reduce the per-second size requirement to $p^* \lambda_0$. If we do, then the hash table will be fully occupied and the probability of collision will be close to 1.

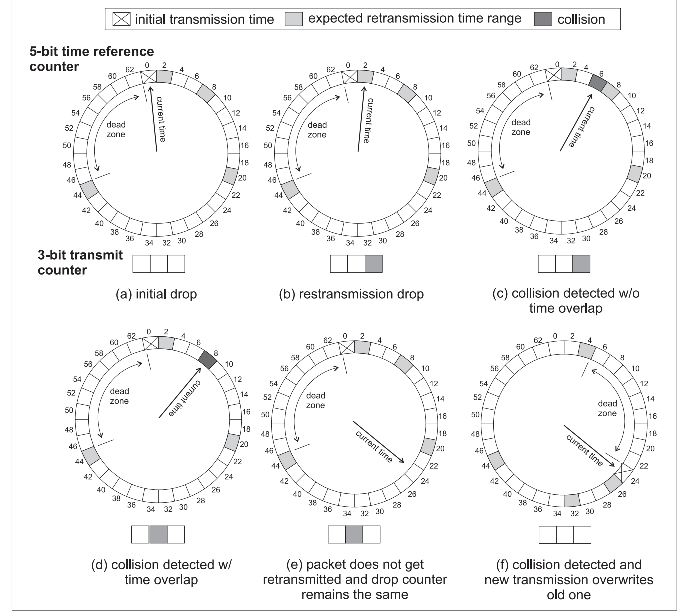

Figure 6: Example of request classification based on the time reference and transmit counters.

For individual table entries, we identify three criteria for encoding each entry in our hash table. First, the time or equivalently the age of a dropped request must be stored to identify its corresponding transmission class. Second, hashing collisions must be detected upon their occurrence, to maintain accurate classification of incoming requests. Third, the size of each entry must be limited (e.g., to 8 bits) to minimize memory requirements. To meet these requirements, we observe that almost all requests will time out within 5 retransmission attempts ($n = 4$), including the initial transmission (Table 1). This implies that the reference time counter should cover a range of 45 seconds to properly classify all transmissions in that range. Since retransmissions are on the order of multiple seconds, a two-second resolution is sufficient; it also accounts for the slight variations in transmission times.[7] With this in mind, only 5 bits are required to encode the reference time counter, which covers a range of 0 to $2 \times 2^5 - 1 = 63$ seconds. To detect collisions, we use three bits to account for the number of transmission attempts.

The basic process of classifying incoming requests are exemplified in Figure 6 and formally defined in the CLASSIFY function in Figure 7. Let $t = t_0$ be the arrival time of a *new* connection request that was dropped and hashed into a hash table entry. Assume that the hash entry was initially unoccupied. Logically, our time reference is a circular counter, and thus, can be represented by a time dial in Figure 6(a). All time values on the dial represent time ranges with respect to t_0, where the "X" in the figure marks arrival (and also drop) time of this connection request and the shaded boxes indicate the time periods for its expected retransmissions within a 2-second range

[7] As we will show, those TCP implementations that do not have the same timeout sequence, will not be discriminated against since they will be classified as "collision decidable."

```
COMPUTE HASH          \\ Assume the destination IP address and TCP
                      \\ port number are in IP_Addr and TCP_port.

    \\ Compute the lower and upper 16 bits of the IP address.
    IP_upper ← ( IP_address & xFFFF0000 ) >> 16
    IP_lower ← ( IP_address & 0x0000FFFF )

    hash ← TCP_port ⊕ IP_lower ⊕ IP_upper
    \\ where ( & is bitwise-and) , ( >> is shift-right), and ( ⊕ is bitwise-xor)

CLASSIFY    \\ Classify into four states: new transmission, retranmission
            \\ collision undetected, and collision detectable.

    index ← Compute Hash of incoming packet

    entry ← Hash_Table[index]
        \\ Retrieve entry corresponding to index from table.

    time_ref  ← lower 5-bits of entry
    xmit_cnt ← next 2-bits of entry
        \\ Extract time reference and xmit counters

    xmit_class ← min { x : T_{x-1} > current_time - time_ref }
        \\ Determine retransmission class of packet, -1 if none.
        \\ First retransmission has T_{-1} = 0

    if xmit_class = 0 or xmit_cnt < xmit_class – 1
        state ← new transmission
        Clear entry in hash table

    if xmit_cnt = xmit_class – 1 and T_x ≈ current_time - time_ref
        xmit_cnt ← xmit_cnt + 1
        state ← retransmission
        Write back entry into hash table

    if xmit_cnt = xmit_class and T_x ≈ current_time - time_ref
        state ← collision undetectable

    if T_x < current_time - time_ref
        state ← collision detectable
```

Figure 7: Hashing and classification algorithms. The algorithm presented ignores the counter wrap-around issues. So, $current_time - time_ref$ should always be assumed to be non-negative.

($[T_x - 1, T_x + 1]$). Because this is the first transmission, the transmission counter is cleared. Consider now $t = t_0 + 3$, the arrival time of the request's retransmission (Figure 6(b)). Since it will arrive during the first shaded box, it will be classified as a retransmission of the original request; the transmission counter is incremented accordingly. This will repeat until the connection times out.

Classification conflicts may arise when multiple requests are hashed to the same entry. Consider a second request arriving during a non-shaded period (Figure 6(c)). We call this a "decidable collision" since—with high probability—this request does not correspond to the original one. On the other hand, if a second request arrives during a shaded period (Figure 6(d)) and the transmission counter has already been incremented to reflect that a retransmission has been seen in this period, then this request can correspond to the actual retransmission or some new transmission; we, thus, classify it as an "undecidable collision."

When a collision is detected, a proper action must be taken to maintain proper (future) classification. As mentioned earlier, there are two types of collisions: decidable and undecidable. All undecidable collisions are classified as a retransmission and dropped, because an arriving request during a shaded-area will either correspond to the original request or to a new request, and such requests are indistinguishable from each other. Fortunately, new requests that are inappropriately dropped will be retransmitted during a non-shaded area, which are then classified as decidable collisions. When a decidable collision is detected, we interpret it as new transmission. Two possibilities exist in this case. First, upon arrival during a non-shaded area, if the transmission counter was not incremented during the most recent shaded area (Figure 6(e)) or if the counter equals 4 (i.e., its maximum value), then this means that the original request has either aborted the connection attempt or timed out. The entry can then be updated to reflect the new request (Figure 6(f)). Second, upon arrival during a non-shaded area, if the transmission counter was appropriately incremented during the most recent shaded area, then we are almost certain that the request is a new transmission, but cannot be stored in the entry. We, thus, let it pass through the filter. In general, because of the low collision probability, letting "decidable collisions" pass through will not affect the target drop probability or the corresponding delay.

When the hash table is used beyond its design range, the above classification technique can yield too many errors. To protect against such erroneous behavior, we use dynamic monitoring to detect and take corrective actions. Basically, the real drop probability is measured on-line by counting the total number of arrivals and dropped requests. If the measured drop probability is dramatically different from the aggregate drop probability in Eq. (7), then a uniform drop probability is used with $p = p^*$ for all incoming requests. This is a fall-back behavior, which is used only in extreme cases.

Finally, periodic maintenance of the hash table entries is required. This is equivalent to garbage-collection where old entries are cleared before the reference timer roles around. During this process, all hash entries are examined for expired values as follows. If the transmission counter does not correspond to the appropriate transmission class at the time of the maintenance (similar to the case in Figure 6(e)), then that entry is cleared.[8] Only once every 16 seconds all entries must be checked. This requirement can be verified by observing that at least once for every time the counters role over (64 seconds) we need to check during the "dead zone" (Figure 6) of every entry if it has expired. Note that if longer inter-maintenance periods are required, then more bits are needed to encode the reference timer to increase the length of the "dead zone".

4.3 Linux Implementation

We implemented working prototypes of SLPD and SBPD in Linux (Kernel 2.4) as filter extensions to `iptables`, Linux's firewalling architecture [24]. Using `iptables`, our implementation can be configured as part of the routing path, when our Linux box is configured as a router, or as a front-end, when

[8] We set all the bits in the transmission counter to indicate that the corresponding hash table entry is unused.

it is configured as a regular server.[9] In iptables, packets are filtered based on user-defined rules. Typically, a rule may include IP and TCP header information such as source or destination addresses/networks, ToS bits, SYN or RST flags, or TCP source or destination ports. A target function is also associated with each rule; it specifies what should be done to packets that match the corresponding rule. Typical targets include ACCEPT to accept the packet and DROP to drop the packet. Therefore, when an incoming packet matches a rule, the associated target function is invoked. For example, one can define a rule that matches all packets with the SYN flag set (indicating a new connection request) with a target of DROP. This would effectively block any connection attempt to the protected machine.

The architecture of iptables is designed to be easily extensible where the target function can be written as a separate kernel module and is free to implement any packet enforcement behavior. We defined two new targets in iptables called SLPD_Filt and SBPD_Filt that are kernel modules. These targets have a configurable effective timeout probability, p^*, and hash function, H1 or H2, that can be altered at runtime. Their implementation follows the exact description in this section. To activate either filter, we define a new rule that matches any packet with the SYN flag set and associate either module as its target. This way, new connection requests are dropped according to our optimal drop policy. As mentioned above, our implementation dynamically monitors the real drop probability. If the number does not match the expected value, incoming requests are dropped with probability p.

5. EVALUATION

To evaluate and demonstrate the efficacy of PD, we equipped a Linux server machine with working implementations of the SLPD and SBPD controllers (Section 4.3) as well as a rate-based drop (RBD) controller. The latter mimics traditional mechanisms where it uniformly drops all incoming requests with probability p and is used as the baseline for comparison [24]. Our main goal is to subject these controllers to realistic load conditions so that the results we obtain may be applicable to real-world deployment scenarios. We also want to avoid any unnecessary complexity without sacrificing accuracy. The three controllers are compared by studying their effects on the performance of clients during a synthesized FCE, which is emulated by generating high client arrival rates to a web server. In each scenario, we also compare the measured results with the predicted ones from our analytic models.

5.1 Experimental Setup

We employ a simple setup where the server machine (a 2.24 GHz Pentium 4 with 1 GBytes of RDRAM) runs Apache 1.3 to receive HTTP requests through a high-speed FastEthernet link. Clients on the other side are generated using Eve, a scalable highly-optimized client emulator. Eve follows the same design principles provided by SPECWeb99 [7] and Surge [2], widely-used tools to evaluate the performance of Web servers. The primary difference between Eve and the two load generators lies in the Eve's ability to sustain extremely high arrival rates regardless of the progress of on-going requests, which is critical to the accuracy of emulating an FCE. This is similar to httperf [26], but Eve has a more accurate client model. SPECWeb99, on the other hand, sends a fixed maximum number of requests; once the maximum is reached, a new request is sent only after the completion of a previous one.

Each of our emulated clients was based on the model described in Section 3.1, where the distributions for the number of parallel connections and their inter-arrival times were based on our estimates for IE 6.0 in Table [20]. Furthermore, we used IP aliasing to provide each client with a unique IP address, which is necessary for the H1 hashing metric. The arrival of clients (not their requests) followed a Poisson process with mean λ_0, a traditionally-accepted model. Furthermore, each client behaved independently from other clients and, on average, issued 6 (independent) parallel requests. Up to four (500 MHz Pentium III with 512 MBytes of SDRAM) machines were used to generate the desired client arrivals. Finally, an intermediate Linux machine was used as a router to implement one of the three controllers.

To eliminate external effects from our measurements, we observe that the client-perceived delay when connecting to a web server is the total wait time before a request completes and is the summation of three mostly independent components: connection-establishment latency (L_h), propagation delay, and service delay. As mentioned earlier, PD only affects the connection-establishment latency. Thus, by keeping the other two components constant, we are able to obtain an unbiased view of the performance of PD. We take two measures to minimize the variation in the other two components. First, we made sure that the client-to-server network path is bottleneck-free. Second, we over-provisioned the server to handle all incoming requests, and all requests issue the same document (e.g., index.html). Therefore, if a request passes through the controller, it successfully completes the HTTP request and has a similar service time to the other requests. Finally, because we need to conduct a large number of experiments to cover the wide range of variable parameters, we limit each run to 5 minutes. Each experiment was repeated until the 95% confidence interval was less than 5% (roughly $25 \sim 30$ times).

Our focus in this section is to evaluate the efficacy of PD at the request level and user-visit level based on the H1 and H2 metrics in Section 4, respectively, and to compare stateless and state-based implementations, SLPD and SBPD, respectively. Since PD is intended as a low-level control mechanism (and due to space considerations), we provide a limited discussion regarding higher-level semantics such as user-sessions. As previously noted, PD is not intended to replace high-level admission control mechanisms, but to improve the control of aggregate traffic in routers, especially during overload.

[9] To be more precise, iptables is built on top of netfilters, which allows packets to be intercepted at various points in the IP stack.

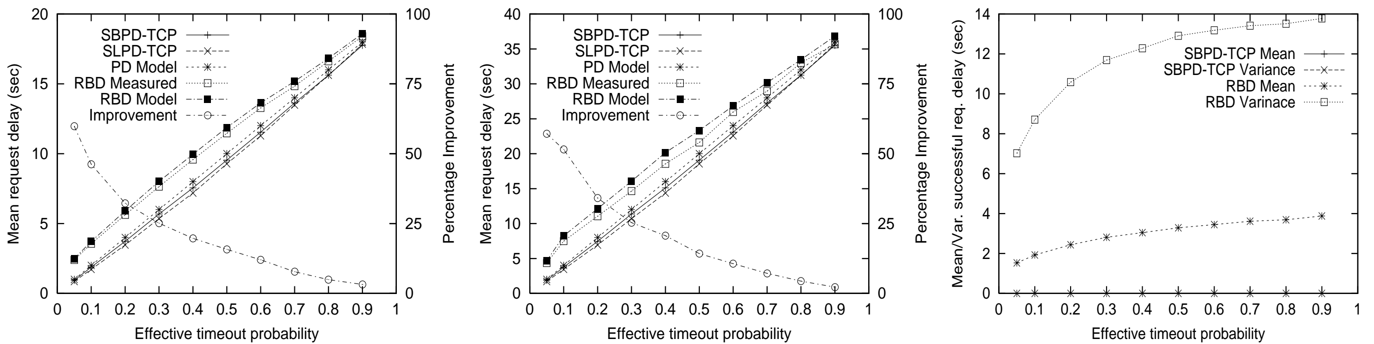

Figure 8: Request delay comparison, (left) Delay for λ_0 = 60 clients/sec and T_{abort} = 20 sec, (center) Delay for λ_0 = 80 clients/sec and T_{abort} = 40 sec, (right) Mean and variance for the delay of successful requests (same configuration as left).

5.2 Connection-Level Measurements

We now focus on characterizing client-perceived delay for rate-based and persistent dropping (both SLPD and SBPD). In our comparisons, we assume that both stateless and state-based PD controllers are using the connection-level hashing metric H2; they are denoted as SLPD-TCP and SBPD-TCP, respectively. In each experiment, we vary the effective timeout probability, p^*, and compare the three drop policies (SLPD, SBPD, and RBD) against each other and against their analytically-derived counterparts. Due to space limitation, we only present two configurations of source traffic. They are meant to confirm the efficacy of our new drop policy. We have performed an extensive evaluation while varying the various parameters over wide ranges. In all cases, our results were consistent with those presented here.

Two metrics are of particular interest to us: (1) the *mean request delay*, which is computed by averaging the elapsed time before a request is completed or timed out, (2) the *mean and variance of successful-request delay*, which is similar to the first metric but only looks at successful requests; it also looks at the variance of the delay. Figures 8(left) and (center) show the benefits of using PD. In the center plot, for example, clients experiencing an effective timeout probability of 0.1 had about a 50% reduction in their mean request delay (due to the reduction in the mean connection-establishment latency) when SLPD-TCP or SBPD-TCP, instead of RBD, is used. This is a dramatic reduction as it implies that a traffic controller that uses RBD to uniformly drop incoming requests with a probability of 0.56 achieves an effective timeout probability of 0.1 and produces 100% longer connection-establishment delays than the one that uses PD (SLPD-TCP or SBPD-TCP).[10] In Figure 8(right) we plotted the delay and variance for successful connections only. The figure shows the main benefit of PD, namely, decoupling the effects of the control policy on the delay of successful requests. The greatest impact can be seen on the variance of successful requests since PD produces one of two outcomes: (1) immediately allows a connection to pass through or (2) consistently drop it. We also observed that PD reduced the variability of the underlying aggregate traffic.

Figure 8 shows that SLPD-TCP achieves similar performance to SBPD-TCP. The real difference between the two schemes is fairness, which is not reflected in our performance metrics. In SLPD-TCP, packets are dropped based on their header information and the only randomness in the scheme is introduced by the prime multiplier, $K(t)$, in Eq. (9). On the other hand, SBPD-TCP has a built-in randomness in every packet it chooses to consistently drop. This, in our opinion, produces better fairness from the client's viewpoint.

We also verified the accuracy of our analytical models. We observe larger, but tolerable, errors in our estimates for smaller values of p^*. However, as p^* increases, T_{abort} dominates the computation of EL_h and thus, improves the accuracy of our prediction. Based on the presented results, our model still accurately predicts the expected delay even though incoming requests are highly dependent. This phenomenon seems counter-intuitive, but is explained by the strict enforcement of the effective timeout probability. Specifically, regardless of the instantaneous arrival rate, a fixed percentage of requests is dropped. Looking back at how the expected delay, EL_h, was derived (Section 2), one can observe that once the p_i's are held constant, the delay value becomes independent of the arrival rate. In fact, this type of policy enforcement is implemented by most Active Queue Management (AQM) techniques where a constant drop probability is enforced based on the average (not instantaneous) length of the underlying queues [13]. Furthermore, the effects of dependent traffic are apparent in other metrics, such as mean user-visit delay and probability of a successful visit (to be discussed shortly).

5.3 User-Visit Behavior

While the mean request delay provides a good indication of the performance of the underlying drop policy, it does not give a complete picture. Looking at the performance metrics that are associated with user-visits and the corresponding aggregate traffic better reflects what a typical client experiences in real systems. They also show the effects of dependent traffic more clearly than looking at individual requests by themselves. In the context of user-visits, we use three metrics to compare the per-

[10] One can argue that drop probabilities of larger than 10% can dramatically affect the throughput of established connections, thus questioning the usefulness of PD. However, bandwidth is commonly not the primary reason of the drops during an FCE, but rather the server's processing capacity. In such situations, the SYN packets are being dropped because the listen queue is full; established TCP connections are not affected. Simple filtering rules can distinguish between the two types of traffic.

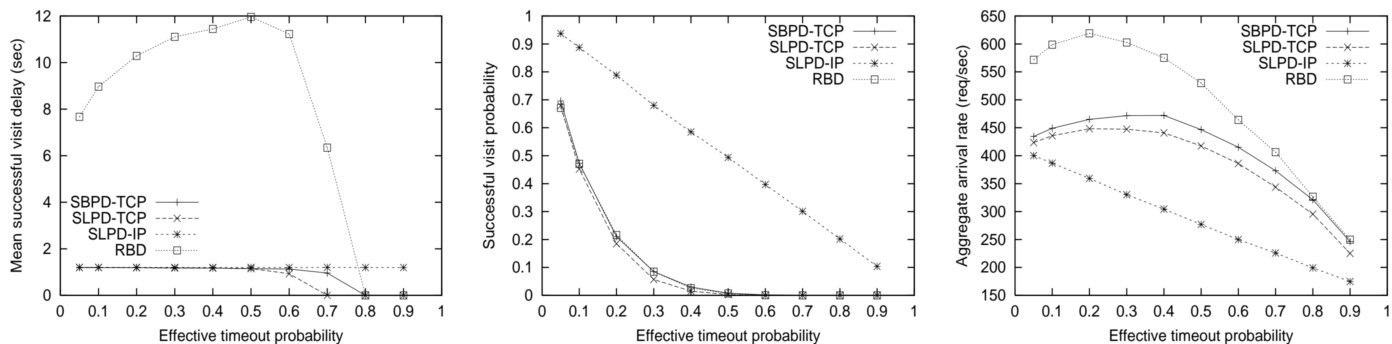

Figure 9: User-visit behavior. In all cases, λ_0 = 60 clients/sec and T_{abort} = 20 sec, (left) mean successful visit delay (a point with zero value implies that no visit was successful), (center) probability of successful visit, (right) effective arrival rate.

formance of the drop polices: (1) the *mean successful visit delay*, which measures the cumulative time for a successful visit as described in Eq. (3), excluding the aborted visits, (2) the *probability of a successful visit*, which reflects the sensitivity of dependent traffic to packet drops, and (3) the *effective arrival rate*, which looks at the change in arrival rate as the drop probability is varied.

Figures 9(left) and (center) plot the expected delay and success probability for the various drop policies. They also show the performance of a stateless PD that uses a client-level hashing metric (H1), referred to as SLPD-IP. Our analytical predictions for the expected user-visit delay were consistent with the measured values and omitted to reduce graph clutter. The figure clearly shows the advantage of PD, especially on the mean visit delay due to its additive nature (Eq. (3)). We note that while the delay seems to be decreasing as $p^* > 0.6$, it is only an artifact from having user-visits with fewer parallel connections that are actually succeeding. Eventually, all visits are aborted by the client and are represented by a zero-valued point in the figure.

Figure 9(center) shows how user visits are sensitive to connection-level and random dropping policies since a visit is successful only if none of its requests times out. This sensitivity is reduced when client-level dropping (SLPD-IP) is performed, which is apparent in the linear relationship between success probability and the effective timeout probability. In effect, SLPD-IP is performing a form of low-level admission control, which maximizes the performance of the controller. Unfortunately, SLPD-IP has the least fairness among our PD implementations as it targets entire clients. As mentioned earlier, unless care was taken to deal with NAT and proxy servers, SLPD-IP may unintentionally block a large number of clients.

Figure 9(right) shows how the aggregate traffic changes among the different policies. Two important points should be observed. First, because the source traffic model is highly dependent, the aggregate traffic, Λ, decreases as the effective timeout probability, p^*, is increased. Our analytical model assumed independent traffic sources and is, thus, not suited for predicting Λ in this case. Second, for any given p^*, we can see the dramatic improvement in using any of the PD policies compared to a rate-based drop policy. From that perspective, our estimate for Λ highlights the relative (not absolute) improvement in using PD over a rate-based drop policy.

5.4 Limitations of the Study

There are still three specific limitations to our study that are worth mentioning. First, we have not discussed how a traffic controller would adjust p^* based on the measured arrival rates or router queue lengths. We believe that PD can be easily integrated into existing AQM techniques, which already have built-in adaptation mechanisms [6, 13, 31]. Because PD reduces the variability of aggregate traffic, it will improve the stability and responsiveness of such mechanisms. Second, we have assumed that clients have unique IP addresses. This provided SLPD-IP with a clear advantage over the other schemes as it mimicked application-level admission control policies. For this reason, we believe that its performance numbers are overstated, but it still performs well when controlling large aggregate traffic as classification errors can be better tolerated. Finally, while our technique seems less effective in controlling or defending against DDoS attacks, it is indeed not more vulnerable than traditional rate-based techniques. The vulnerability of our scheme is only apparent in the choice of the hash function. This can be easily overcome by using more secure hash functions that an adversary cannot exploit. All that a DDoS attack can do is to increase the amount of traffic, which may force the controller to use a larger p^* value. This is not different from traditional control mechanisms.

6. RELATED WORK

Several recent studies have focused on characterizing aggregate traffic during FCEs [21, 23]. Looking at the broader scope, earlier studies can be categorized into empirical characterization or analytical modeling of TCP traffic. Measurement studies such as [1, 10, 11, 25, 29, 33], to name a few, have investigated the impact of TCP congestion control on the behavior of underlying traffic (e.g., throughput, variance, self-similarity). On the other side of the spectrum, the authors of [5, 15, 16, 27, 28, 32] presented analytical characterizations of the throughput of TCP's congestion control as a function of RTT and packet loss probability. Our proposed client model can be viewed as a direct extension to earlier ones, however, with the focus on the interaction between active traffic controls and the aggregate behavior of incoming requests. We have taken a bottom-up approach which investigates both the influence of low-level net-

work protocols as well as high-level application mechanism on the behavior (or persistence) of clients.

In general, our analysis is based on a different model of client behavior where we introduced the concept of persistent clients to capture the dynamics of client retransmissions. Our main objective is similar to queue-management solutions such as Class-Based Queueing (CBQ) [14], Active Queue Management (AQM) [4, 6, 13], and Explicit Congestion Notification (ECN) [12] all of which aim to improve the performance of the underlying network. Our work complements these solutions by specifying the exact mechanism for minimizing connection-establishment latency in the presence of active packet dropping by routers or end-servers.

7. CONCLUSIONS

We characterized the dynamics of persistent clients in aggregate traffic. In particular, we showed that client persistence, which is due mostly to TCP's congestion control, has a direct effect on the stability and effectiveness of traffic control mechanisms. To deal with client persistence, we introduced persistent dropping and showed that minimizes the average connection establishment delay as well as the effective arrival rate whenever the volume of incoming request exceeds the server's or router's capacity. We presented two working implementations of persistent dropping based on hash functions that can be deployed in routers or end-servers.

Persistent dropping can be considered as a low-level admission control policy. No application-level support is required for the correct operation of persistent dropping. In particular, when connection-level classification (H2) is performed, persistent dropping does not violate any end-to-end semantics and, at the same time, achieve the same control targets as the traditional rate-based control. Furthermore, the improvement in the connection-establishment latency does not interfere with higher-level admission control mechanisms. On the other hand, client-level classification (H1) does violate the end-to-end argument, and it is presented here to show the full potential of an intelligent dropping mechanism in routers. One can argue that connection-level controls should be avoided in routers and left to the end-servers. We addressed this exact issue by showing that in some high-congestion cases, such as FCEs, routers are forced to drop new connection requests. Our technique achieves quick convergence to the control targets with minimal intrusion on successful connections.

References

[1] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements," in *Proceedings of IEEE INFOCOM'98*, March 1998, pp. 252–262.

[2] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in *Proceedings of Performance'98/ACM Sigmetrics'98*, May 1998, pp. 151–160.

[3] N. L. Biggs, *Discrete Mathematics*. Oxford University Press, New York, 1989.

[4] B. Braden *et al.*, "Recommendations on Queue Management and Congestion Avoidance in the Internet," *RFC 2309*, 1998.

[5] N. Cardwell, S. Savage, and T. Anderson, "Modeling TCP Latency," in *Proc. of the IEEE INFOCOM 2000*, 2000, pp. 1742–1751.

[6] W. chang Feng, D. Kandlur, D. Saha, and K. G. Shin, "The BLUE Active Queue Management Algorithms," *IEEE/ACM Trans. on Networking*, vol. 10, no. 4, pp. 67–85, September 2002.

[7] S. D. Committee, "SPECweb," Tech. Rep., April 1999, http://www.specbench.org/osg/web/.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Press, 1990.

[9] A. Feldmann, *Characteristics of TCP Connection Arrivals*, ser. Self-Similar Network Traffic and Performance Evaluation. John Wiley and Sons, Inc., 2000, ch. 15, pp. 367–399.

[10] A. Feldmann, A. Gilbert, W. Willinger, and T. Kurtz, "The Changing Nature of Network Traffic: Scaling Phenomena," *Computer Communication Review*, vol. 28, no. 2, April 1998.

[11] A. Feldmann, A. C. Gilbert, P. Haung, and W. Willinger, "Dynamics of IP Traffic: A Study of the Role of Variability and Impact of Control," in *Proceedings of the ACM SIGCOMM '99*, 1999, pp. 301–313.

[12] S. Floyd, "TCP and Explicit Congestion Notification," *ACM Computer Communication Review*, vol. 24, no. 5, pp. 10–23, 1994.

[13] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *ACM/IEEE Trans. on Networking*, vol. 1, no. 4, pp. 397–417, 1993.

[14] ——, "Link-sharing and Resource Management Models for Packet Networks," *IEEE/ACM Transactions on Networking*, vol. 3, no. 4, pp. 365–386, August 1995.

[15] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-Based Congestion Control for Unicast Applications," in *Proceedings of the ACM SIGCOMM '00*. ACM, August 2000.

[16] C. Hollot, V. Misra, D. Towsley, and W. Gong, "A Control Theoretic Analysis of RED," in *Proceedings of the IEEE INFOCOM 2001*, 2001.

[17] V. Jacobson, "Congestion Avoidance and Control," in *Proceedings of the ACM SIGCOMM '88*, August 1988.

[18] H. Jamjoom, P. Pillai, and K. G. Shin, "Re-synchronization and Controllability of Bursty Service Requests," *To appear in IEEE/ACM Transactions on Networking*, August 2004.

[19] H. Jamjoom and K. G. Shin, "Eve: A Scalable Network Client Emulator," University of Michigan Technical Report, Tech. Rep. CSE-TR-478-03, 2003.

[20] ——, "Persistent dropping: An efficient control of traffic aggregates," in *Proceedings of the ACM SIGCOMM '03*, Karlsruhe, Germany, August 2003, pp. 287–298.

[21] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites," in *Proceedings of the 11th International World Wide Web Conference*, Honolulu, Hawaii, May 2002.

[22] J. Lemon, "Resisting SYN Flood DoS Attacks with a SYN cache," in *BSDCon 2002*, Feb 2002.

[23] R. Manajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling High Bandwidth Aggregates in the Network," *SIGCOMM Computer Comm. Review*, vol. 32, no. 3, July 2002.

[24] F. Marie, "Netfilter Extensions HOWTO," http://www.netfilter.org.

[25] R. Morris and D. Lin, "Variance of Aggregated Web Traffic," in *Proceedings of the IEEE INFOCOM 2000*, vol. 1, 2000, pp. 360–366.

[26] D. Mosberger and T. Jin, "httperf — A Tool for Measuring Web Server Performance," HP Research Labs, Tech. Rep., http://www.hpl.hp.com/personal/David_Mosberger/httperf.html.

[27] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," in *Proceedings of the ACM SIGCOMM '98*, 1998, pp. 303–314.

[28] J. Pahdye and S. Floyd, "On Inferring TCP behavior," in *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*. ACM Press, 2001.

[29] V. Paxon, "End-to-end Internet Packet Dynamics," in *Proceedings of the ACM SIGCOMM '97*, 1997, pp. 139–152.

[30] J. Postel, "RFC793: Transmission Control Protocol," *Infomation Science Institute*, September 1981.

[31] J. Reumann, H. Jamjoom, and K. Shin, "Adaptive Packet Filters," in *Proceeedings of the IEEE GLOBCOM'01*, November 2001.

[32] S. Sahu, P. Nain, C. Diot, V. Firoiu, and D. F. Towsley, "On Achievable Service Differentiation with Token Bucket Marking for TCP," in *Measurement and Modeling of Computer Systems*, 2000, pp. 23–33.

[33] S. Sarvotham, R. Riedi, and R. Baraniuk, "Connection-level Analysis and Modeling of Network Traffic," in *Proceedings of the ACM SIGCOMM Internet Measurment Workshop*, November 2001.

[34] H. Zhang and D. Ferrari, "Rate-Controlled Static Priority Queueing," in *Proc. of the IEEE INFOCOM 1993*, San Francisco, 1993, pp. 227–236.