Persistent Dropping: An Efficient Control of Traffic Aggregates

Hani Jamjoom Kang G. Shin
University of Michigan
{jamjoom,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 building a coherent model of 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.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—Flash Crowds; C.4 [**Performance of Systems**]: Modeling Techniques

General Terms

Management, Performance

Keywords

Queue Management, Flash Crowd Events, Modeling, Optimization

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. We show that persistent client behavior, ultimately rooted into TCP congestion control, can be responsible for increasing the aggregate traffic in an FCE by two folds. We also describe the internal dynamics of persistent clients with the goal of improving existing router and end-server controls to better handle client persistence.

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. Our approach differs from recent studies [2, 9, 25, 32], where aggregate traffic is treated as a black-box and is characterized with inter-arrival time, round-trip time, time-scale dynamics, etc. Instead, we look inside the box to characterize the behavior of the building blocks of the aggregates, namely, the individual clients. Similar to [1, 7, 17], we are interested in exploring the interactions between clients, the network, and the end-server. We observe the existence of a hierarchy of factors contributing to the behavior of clients in FCEs. This has led us to a new model — which we call *persistent clients* — that is different from traditional models in that clients do not simply go away when their requests are dropped. Instead, they keep trying until they succeed or eventually quit. Based on this new observation of client persistence, we build a coherent picture detailing the expected behavior of the entire aggregate. We found that this unfolds several unexpected behaviors of aggregate traffic during overload.

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 rate of (large) 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. We find that the combination of TCP's reaction to packet loss (first element) — namely, retransmitting after timing out — with the arrival of connection requests from new clients (second element) has an additive 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 [36].

Through the specialization of control, we are able to focus on new

SIGCOMM'03, August 25–29, 2003, Karlsruhe, Germany

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 to minimize the client-perceived latency as well as minimize 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 also 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.

The contributions of this paper are fourfold. First, we analyze the dynamics of the internal mechanisms of individual clients during an FCE. Second, we show that the arrival of new users is not the only cause of FCEs, but also show that the persistence of individual clients plays an important role, which is further exacerbated by the allowed parallelism of web browsers. Because requests are originating from legitimate clients, our measurements emphasize that SYN packets are the main contributing factor in FCEs; those already-established connections have limited contributions. This is different from DDoS attacks, where the adversary has the capability of spoofing any packet that is injected in the network. It is also different from the traditional view of network traffic where the majority of packets belong to established connections. Third, we examine the controllability of aggregate traffic and the shortcomings of existing control mechanisms in the context of SYN packets being the main cause of FCEs. Finally, we propose PD, an efficient mechanism for controlling traffic aggregates.

This paper is organized as follows. We first look at the anatomy of persistent clients in Section 2, where we focus on the behavior of real clients during FCEs, isolate the factors that impose real threats in an FCE, and combine these factors together to create a coherent model of persistent clients. We, then, propose a PD controller to deal with persistent clients in Section 3. In Section 4, we experimentally evaluate some performance issues. The paper ends with related work and concluding remarks in Sections 5 and 6, respectively.

2. ANATOMY OF PERSISTENT CLIENTS

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. We isolate five (non-user related) factors that can affect the persistence of a typical client's access to a web server (Figure 1). In the process of analyzing them, they are grouped in two separate categories: network-level and application-level factors. While we only focus on non-user related factors, our proposed control mechanisms ultimately reduce the client-perceived latency; this indirectly reduces the impact of user-related factors as, for example, users are less inclined to press the reload button on their web browsers.

2.1 Persistence in Network Protocols

In this subsection, we investigate how the combination of TCP congestion control and different 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. Consequently, our findings fit well with clients implementing the HTTP 1.0 specification. In Section 2.2, we extend our results to HTTP 1.1 clients, where several HTTP requests can be multiplexed into a single connection.

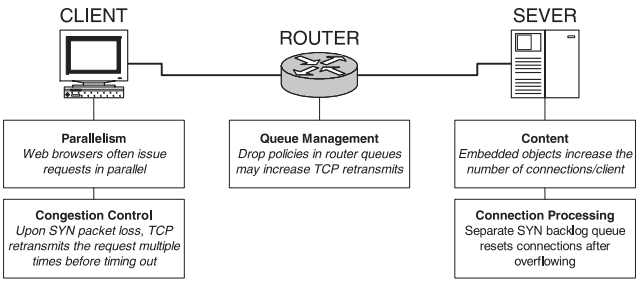

Figure 1: Non-user related factors affecting FCEs.

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 are lost due to any of the three types of queues filling up: router queues, server SYN backlog queues, and server (or application) listen queues. Packet drops by different queues may trigger different reactions from the client as part of recovery from packet loss. 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.

2.1.1 Packet Drops at Router Queues

When router queues fill up 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 em-

[1] Most TCP stack implementations follow Jacobson's algorithm [19, 33], 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.

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 SYN 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.

pirically and analytically — e.g., in [5, 30, 31].

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 its estimate of the connection-establishment latency. We, thus, assume that end-points adhere to a TCP-Reno style congestion control mechanism [19]. However, to draw general conclusions for the entire aggregate, we must also characterize the arrivals of new connection requests, namely, their interarrival times are *independently and identically distributed (i.i.d.)*, and are exponentially distributed. Assuming *i.i.d.* implies that 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. Based on these two assumptions and elementary queueing theory, one can also show that SYN-ACKs on the reverse direction have the same *i.i.d.* distribution. Furthermore, under a uniform drop policy (i.e., incoming requests are dropped with an equal probability), the retransmitted requests have *i.i.d.* and exponentially-distributed interarrival times.

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 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 p as the drop probability 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 *i.i.d.* stream of initial transmission attempts by λ_0 (i.e., SYN packets on their first transmission), the stream of first retransmissions by λ_1, up to λ_n. Then, the effective mean arrival rate Λ is:

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

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 at the router will, in theory, increase the amount of new connection requests by 75%.[2] 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.

The above illustrates that random dropping may not be well-suited for FCEs, and an alternative technique is needed. In Section 3, we present a better technique for controlling FCEs with minimal impact on connection-establishment latency.

2.1.2 Packet Drops at SYN Backlog Queues

When the backlog queue at the server fills up, the server can be configured to drop incoming SYN packets, which triggers a similar retransmission behavior as discussed above. The server can also be configured to 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 was 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.

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

Platform	Browser	Average parallel connections	Maximum parallel connections	Average interarrival time (ms)	Minimum interarrival time (ms)	Average connection duration (ms)	Average connections per page	Average reqs per connection	Distribution parameters for parallel connections **Weibull (α, β)**	Distribution parameters for interarrival times **Weibull (α, β)**
Windows	**IE 6**	5.9	15	161.2	0.2	359.8	8.1	2.7	2.41, 7.75	0.58,99.84
	Netscape 7	4.5	14	281.8	0.1	137.1	25.6	1.0	4.21, 5.75	0.91, 142.08
	Opera 6.05	10.3	32	366.3	0.3	328.5	16.6	1.3	1.13, 10.70	0.31, 38.84
Linux	**Mozilla 1.2**	5.9	16	200.2	0.1	272.4	10.5	2.1	1.20, 4.82	0.81, 133.67
	Netscape 4.7	7.8	14	274.4	0.1	395.4	18.8	1.2	1.176, 7.10	0.53, 76.62

Table 2: Parallelism of different browsers. Values obtained from accessing 250 different links from the top 25 websites. On average, there was 21.2 unique objects and 71.2 non-unique objects per page (excluding any Java or JavaScript).

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 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. As we show shortly, this is true even if SYN packets are not accepted when the listen queue of the application fills up.

2.1.3 Packet Drops at Application Listen Queues

Once application listen queues fill up, no connections can be established and the backlog queue drops incoming SYN requests.[3] However, some of the queued SYN packets can complete the three-way handshake while the listen queue is full. In this case, the OS typically drops the fully-established connections all together and optionally sends a reset packet back to the client.

This introduces an important consequence of having a separate SYN backlog queue that deserves careful examination. As explained earlier, the backlog queue is generally independent of applications' listen queues to increase the server's resilience to SYN flood attacks (Table 1). However, when the listen queue fills up, incoming SYN packets destined for the corresponding application are dropped even if there is room in the backlog queue. This is to avoid the situation where the three-way handshake is completed and the connection must be dropped due to the lack of room in the listen queue. However, connections can still get dropped when a burst of SYN packets arrive and the listen queue is almost full. All SYN packets in the burst are queued in the backlog queue, but only a small portion of the fully-established connections are moved to the listen queue, and the rest are dropped.

Once a connection is dropped, the server may or may not send a reset packet. As shown in Table 1, not sending a reset packet triggers further retransmissions by the client. In fact, since RTO is based on round-trip time (RTT) estimates, the client sends larger data packets more often than when the SYN packet is dropped and the connection is not allowed to complete. On the other hand, if a reset packet is sent back to the client when the connection is dropped, the server cycles between two phases. The first phase is when established connections are dropped and reset packets are sent back to the client. The second phase is when SYN packets are dropped and then retransmitted after exponentially-increasing timeouts.

Depending on one's view, connection dropping is either an acceptable behavior that the server uses to shed load or an unacceptable behavior as it incorrectly drops clients' completed connections. If the latter view is taken, one can modify the TCP stack to completely eliminate this behavior. Our proposed solution allows the listen queue to temporarily grow beyond its specified limit to accept those fully-established connections that would otherwise be dropped. During that period, no new connection requests are allowed to be queued in the backlog queue. The listen queue is then allowed to decrease until it becomes smaller than the originally-specified limit; at that point, the backlog queue is allowed to accept new SYN packets. With this technique, we are able to completely eliminate "illegal" or undesirable dropping of fully-established connections. When developing our control mechanism, we assume that TCP stack implementation has been corrected, either using our approach or an alternative one, to improve the consistency of the results and to simplify our analysis.

2.1.4 Packet-level Analysis of FCEs

There still remains the question of whether connection requests (i.e., SYN packets) or actual data packets are the main cause of FCEs. Both of [21, 23] have characterized FCEs with a dramatic increase in the number of clients accessing the server without identifying the type of packets that cause the overload. Because of insufficient availability of public traces, we used real emulation to determine the cause of the overload. We used a collection of ten machines (500 MHz Pentium III, 512 MBytes RAM, and running Linux Kernel 2.4) all implementing Eve [20], a powerful home-grown client emulator that is able to emulate approximately 500 totally independent clients per machine. We configured our clients to bombard a single Apache 1.3 server (a 2.24 GHz Pentium 4 with 1 GByte RDRAM connected to the client machine through a FastEthernet switch) with a sustained request rate of up to 5000 requests/sec. We also instrumented the server kernel to report both backlog and listen queue behavior. This allowed us to observe — at the server — the cause of the overload and the reaction of the server. As expected, queues behaved in a manner that is consistent with what we described earlier and with [1, 27, 29]. Moreover, in our experiments we did not observe any dramatic surge in reverse traffic (i.e., traffic from the server to the client) especially when forward traffic is increased well beyond the server's capacity. In fact, the reverse traffic was always limited by the server's capacity, and the added load was mainly in the forward path and was dominated by SYN packets. This is due to the large numbers of new clients attempting to access the server without prior knowledge of the network or the server's condition. Even though they are sending a small number of packets (including retransmissions), their sheer volume overwhelmed the server. In Section 3, we show how to design a controller tailored to the arrival of new clients and retransmissions of their dropped requests.

2.2 Persistence in Client's Parallelism

Almost all the web content is organized such that a main object (or page) is first retrieved and followed by all of its embedded objects. Web browsers generally issue multiple requests to the embedded objects in parallel to maximize the throughput and minimize

[3] This behavior was verified by looking at the actual source code of the Linux and FreeBSD TCP stacks. For other OS implementations, we deduced this behavior using stress testing of tools.

the time of content retrieval. Even with the availability of persistent connections in HTTP 1.1, where browsers are encouraged to use a single connection to take advantage of TCP's larger window size,[4] parallelism is still used by most browsers. Traditionally, the added aggressiveness of parallelism was analyzed from the perspective of network bandwidth or server resource sharing [1]. In this subsection, we look at clients' parallelism as a factor contributing to the increase in the severity of FCEs since a single client can issue multiple *independent* connection requests. As shown in Figure 1, there exists an interplay between the client's browser implementation and the server's content organization (or encoding) that determines the degree of parallelism, and hence, the aggressiveness of the clients. On the server side, having many embedded documents forces clients to issue multiple HTTP requests (not necessarily in parallel). We have found from our experiments that there are an average of approximately 21 unique objects embedded in each page. Browsers on the client's side are free to request these objects using separate connections in parallel, series, or in a single connection using persistent connections in HTTP 1.1.

To investigate this issue more thoroughly, we analyzed the behavior of five popular browsers on two platforms, Linux (Kernel 2.4) and Microsoft Windows 2000. We focused on each browser's degree of parallelism, how the parallelism changes with different web content, and how each reacts to packet loss during the retrieval of the primary or the embedded web content. Ten random links from the top 25 websites[5] were used to test the five browsers. Each browser was configured to request the same 250 links. An intermediate machine running `tcpdump` was used to intercept and record all packets to and from the server. To improve the consistency of our results, we cleared the data cache after every visited page. Furthermore, each browser was configured to fetch the set of links 5 times over various times of the day. This way, we also minimize the effects of the time of day on our conclusions. We note, however, that there is a large set of configurations for each browser. Testing all of them would require long hours of manual labor. We, thus, used the default values as the basis for our conclusions.

Table 2 summarizes the parallelism of the different browsers. We derived these numbers by careful analysis of the generated logs. Our log analyzer kept track of connection start and end times by identifying the corresponding SYN, FIN, or reset packets. We were conservative in making measurements since we considered a connection terminated if it remains idle for 500 msec, which is roughly 5 times the average RTT value. Here we assumed that the implementer did not bother to close the connection before issuing a new one. The table shows (among other things) the statistical averages for the number of parallel connections and their interarrival times. The computed values are based on arrival epochs of new connections; they are not time averages. That is, each time a new connection is detected, a snapshot of the system is taken and based on their ensemble, the averages are computed. Table 2 also shows the mean number of HTTP requests per TCP connection, which is the number of per-page unique objects divided by the number of issued HTTP requests. We chose to use the number of unique objects as we assume browsers perform some intelligent caching. Our reported numbers are, thus, conservative since, in addition to using the number of unique objects, they also do not include any additional objects requested by any embedded Java applets or JavaScripts in the page.

Several conclusions can be drawn directly from the table.

[4]By using a single connection to request multiple objects, the TCP window, in theory, grows to match the throughput of using multiple short connections in parallel [26].

[5]Website rankings were based on a the December, 2002 statistics from Nielsen Netratings (www.nielsen-netratings.com).

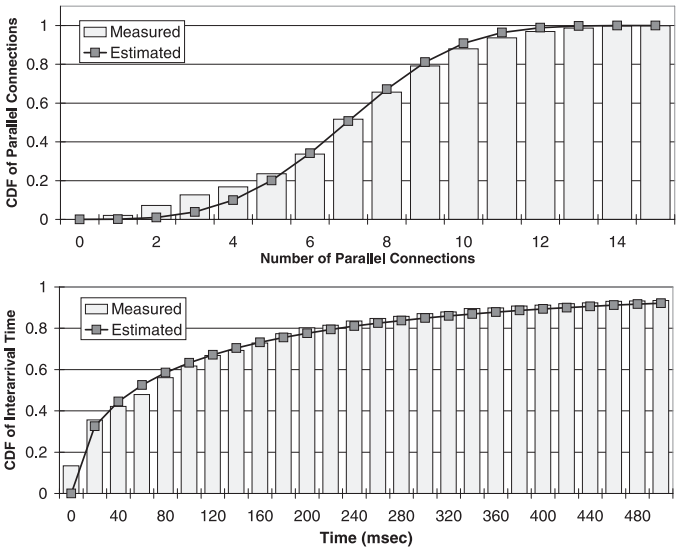

Figure 2: Internet Explorer 6.0 browser distributions: (top) distribution of parallel connections and (bottom) distribution of interarrival times. Both are approximated by the Weibull distribution with parameters in Table 2. In the bottom graph, the distribution converges to 1 after 4500 msec. We truncated the plot to magnify the important region.

C1. Even with the availability of persistent connections, browsers tend to use HTTP 1.0-style connections. This can be seen in values for the average requests per connection where a value that is close to 1 indicates a single connection is used to request a single page. This is also confirmed by comparing the average duration of a connection with the average interarrival times. Values that are close to each other imply that a browser is issuing a new connection as soon as an old one finishes.

C2. Browsers are configured with a specific maximum number of parallel connections. By pipelining their requests (C1), they try to maintain this maximum value when more objects need to be obtained.

C3. There is an initial burst of connections after obtaining the main page. The size of this burst is not the maximum allowed value described in C2; otherwise, the average interarrival time should be close to its minimum value. After the initial burst, browsers tend to space their parallel connections by an order of hundred milliseconds, roughly, the average time of completing a single request.

The last two points can be verified further by inspecting the distributions of the number of parallel connections and their interarrival times. These are shown in Figure 2 as Cumulative Distribution Functions (CDF). We only plotted the distributions for Internet Explorer (IE) 6.0 because of space limitation. A value on the ordinate (y-axis) of the top graph, for instance, should be read as the probability of having a maximum of x parallel connections, where x is drawn from the abscissa (x-axis). Both plots, then, confirm the behaviors in C2 and C3. We first note that because measurements of the number of parallel connections are taken at the arrival epochs of new connections, the computed average should approximately be a half of the allowed maximum when the maximum allowed number of connections is close to the number of embedded objects. For example, if there are 18 embedded objects and the maximum allowed number of parallel connections is 15, then we have the following set of measurements: $\{1, 2, 3, \ldots, 15, 15, 15, 15\}$, which yields an average value of $(1 + 2 + \cdots + 15 + 15 + 15 + 15)/18 = 9.17$. Here,

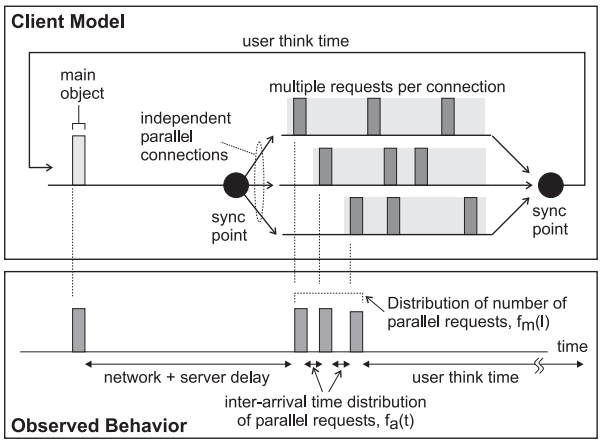

Figure 3: Persistent client model.

once we reached the maximum, the remaining three objects had to wait for three other connections to complete. This is the observed parallel behavior in Figure 2. The figure also verifies C3. It shows that 50% of the connections arrive less than 50 msec apart, but the bulk are spaced a few seconds apart.

Using standard distribution fitting techniques [16], we are able (in most cases) to approximate both parallel and interarrival distributions using the Weibull distributions with parameters detailed in Table 2. By observing the large variations in the distribution parameters, we find that no concrete conclusion can be drawn from our measurements regarding the exact distribution of browser behavior. Specifically, we observe that different times of the day produce large variations, which is probably due to changing server and network loads. As we show in Section 2.3, instead of using analysis of variance techniques to describe browser behavior under different load scenarios, we construct a simple model of their internal mechanisms. This model allows us to draw direct conclusions on the optimal way of controlling web clients.

Based on Table 2, we see the benefits of persistent connections in HTTP 1.1. This is shown in the large difference between the average number connections per page for IE 6.0 and Mozilla 1.2, and the other three browsers. There is, however, one caveat in using persistent connections: it may cause servers to run out of available file descriptors much faster than requiring connections to terminate after each request [21]. Once the application runs out of file descriptors, it no longer is able to accept additional connections, which causes the listen queue to fill up. To combat this problem, the server can dynamically reduce the keep-alive timeout in HTTP 1.1 as the number of descriptors approaches the maximum capacity.

Finally, we also tested the reaction of browsers to packet loss. We instrumented (using `iptables`) an intermediate machine to drop packets after the main object is fetched. Consequently, all browsers abort the client's request (after one of the parallel connections times out) and display an error dialog box.

2.3 Modeling of Persistent Clients

Creating accurate models of web clients is not an easy task. 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 3; 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 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 3.3). 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 3) 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 the ones in [2, 9, 11] with very different distribution parameters. Our model, however, captures the effects of the applied control, which we use to construct an optimal controller.

3. CONTROLLABILITY OF PERSISTENT CLIENTS

Client persistence imposes an added challenge to the controllability of aggregate traffic. 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. Repeated dropping also 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 new 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

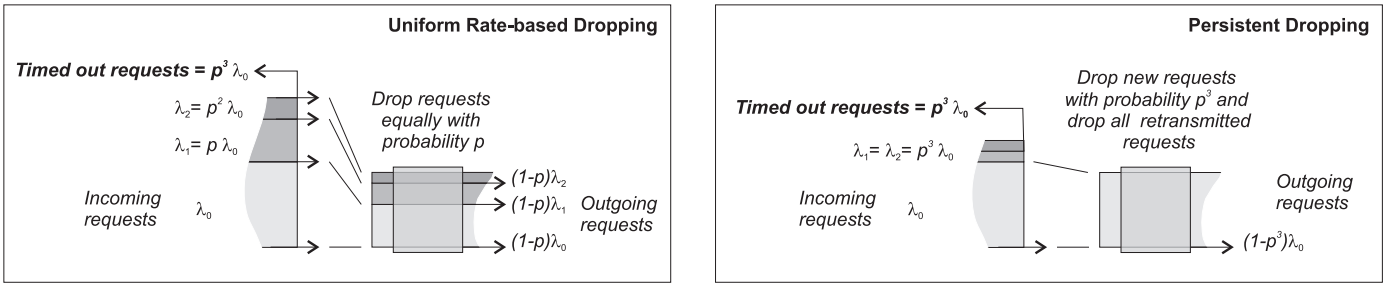

Figure 4: Illustrative comparison between rate-based dropping and PD. We view the outgoing link as a smaller pipe than the stream of incoming requests. We then show how the two strategies drop incoming requests to fit them into the pipe.

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 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 routers using RED, then p is based on dynamic measurements of queue lengths [13]. We, thus, view p as the percentage of packets that must be dropped regardless of how it 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 will be addressed in Section 3.4.

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 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:

$$\Lambda = \lambda_0 + p_0\lambda_0 + p_0p_1\lambda_0 + \cdots + \left(\prod_{i=0}^{n-1} p_i\right)\lambda_0. \quad (4)$$

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 all i), implying that $p^* = p^{n+1}$ (consistent with the results in Section 2.1).

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

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

where T_i is the time of the i^{th} 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 after $RTT + T_2$ seconds, for example, is the probability of being dropped during the initial transmission with probability p_0, then being dropped again on the second transmission with probability p_1, and finally connecting 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:

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

The optimal drop strategy must, thus, 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 a RST packet, instructing 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.[6] It also requires the router or end-server to have enough network and processing capacity to respond to each rejected SYN packet.

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 4, showing how this new technique intelligently fills the outgoing link to minimize packet retransmissions. The fact that this dropping scheme chooses certain connections and consistently drops them on each retransmission does not imply that it is biased against these connections. We have shown that a rate-control drop policy generally causes $p^*\lambda_0 = p^{n+1}\lambda_0$ connections to time out, which is identical to (but less efficient than) PD. In this respect, both PD and rate-control policies have the same fairness. Table 3 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 4, we also compare the variance in the latency of the two schemes — they can be directly

[6] We have tested several modern OSs and found that there is no universal technique for rejecting a client. Microsoft Windows, in particular, ignores both RST and ICMP packets for this purpose.

	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_n	$(1-p^{n+1})\,\text{RTT} + \sum_{j=0}^{n}[p^j(1-p)T_j] + p^{n+1}T_{abort}$	$(1-p^{n+1})\,\text{RTT} + p^{n+1}T_{abort}$
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 3: Comparison between PD and random dropping.

computed using Eq. (5) and are omitted for space consideration.

3.2 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 new connection requests pass through a network of queues in series, each using a PD policy g^*, the client's connection-establishment latency and effective arrival rates are minimized. This is illustrated in Figure 5 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 policy, 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.

3.3 Applicability to Persistent Clients

The derivation in Section 3.1 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 2.3). 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 optimality 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

$$EL_s = RTT + \frac{p_0(1-p_1)T_1 + \cdots + (p_0\cdots p_{n-1})(1-p_n)T_n}{1-p^*}. \quad (7)$$

Assume that once a connection is established, the average time to send the request, have it processed by the server, and receive the reply is ET_s. Estimates for ET_s are derived in [5, 30] 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 2.3, when $p_0 = p^*$ and $p_i = 1$ for $i \neq 0$, EV is minimized.

3.4 Controller Architecture

Implementation of our optimal drop policy in routers and end-servers relies on the ability to (1) group requests originating from the same client visit and (2) distinguish between new and retransmitted requests. Unfortunately, both present a design challenge, 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). However, 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 improve on existing techniques.

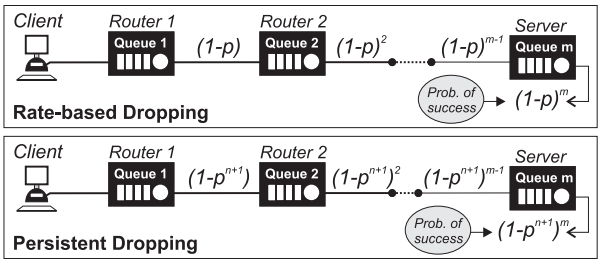

Figure 5: Probability of success in network of queues

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 client-side 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.

Since this classification must be performed at very high speeds, the hash function must be simple, yet still provides uniform hashing. We observe that the uniqueness of the source IP address, and when combined with the TCP port information, the probability of colli-

sion is minimized. We used a simple XOR operation to perform the required mapping:

$$h(x_1, x_2, \ldots, x_k) = x_1 \oplus x_2 \oplus \cdots \oplus x_k \times K(t) \bmod R, \qquad (8)$$

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.

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

3.4.1 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 6). Depending on whether H1 or H2 is used, client- or connection-level persistent dropping can be achieved. 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. (8) 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.

3.4.2 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. 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 operation of SBPD is split into two stages (Figure 6). 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 75 second window from the initial drop time. The window length was chosen based on T_{abort}^{max}, the maximum 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. Note that due to space limitation we have omitted several optimizations that reduce the number of lookup and store operations to the hash table. We also omitted a description of the periodic maintenance that is required to the table.

Since the resolution of each time-stamp need only be on the order of seconds, 8 bits are sufficient to represent the time-stamp. The size of the table is then based on the worst-case scenario of the arrival rate, λ_0^{max}, and timeout value, T_{abort}^{max}:

$$M = \lambda_0^{max} \times T_{abort}^{max} \times \sigma, \qquad (9)$$

where $\sigma \geq 1$ is an over-design factor that further reduces the probability of hashing collisions; our experimental results have indicated that $\sigma = 1.2$ is adequate. When the hash table is used beyond its design range, the above classification technique can yield too many errors. To protect against such an 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 effective timeout probability, then a stateless classification can be used or even a uniform drop probability with $p = (p^*)^{\frac{1}{n+1}}$ for all incoming requests. This is a fallback behavior, which is used only in extreme cases.

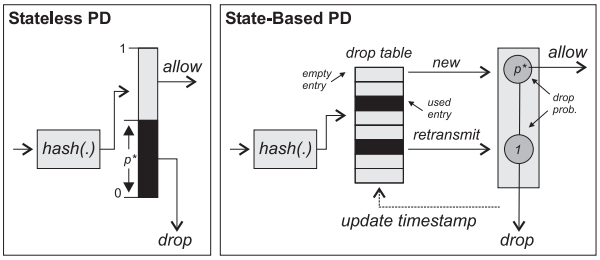

Figure 6: Stateless and state-based implementation of persistent drop controller.

3.5 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 it is configured as a regular server. 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 Section 3.4. 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 in Section 3.4, our implementation dynamically monitors the real drop probability. If the number does not match the expected value, incoming requests are dropped with probability p.

4. 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 3.5) 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.

4.1 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. Each of our emulated clients was based on the model

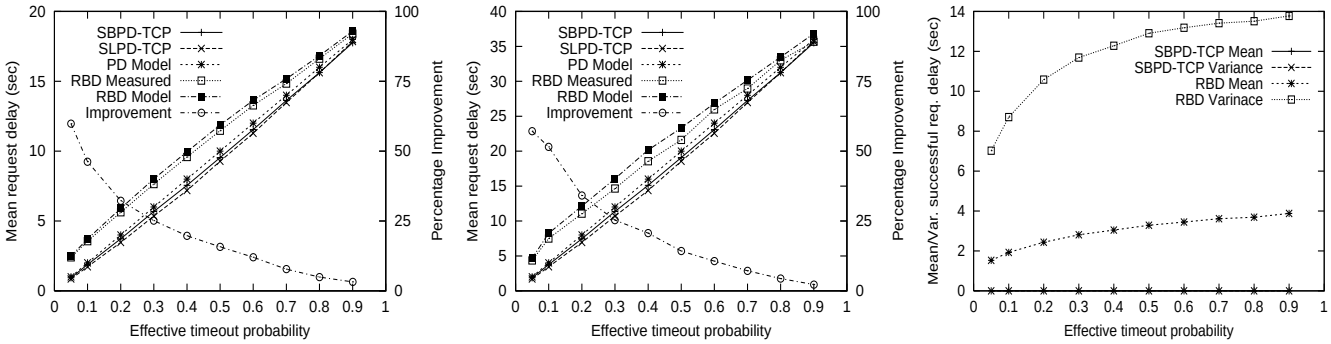

Figure 7: Request delay comparison, (left) Delay for λ_0 = 60 clients/sec and T_{abort} = 20 sec, (center) Delay for λ_0 = 80 clients/sec and T_{abort} = 40 sec, (right) Mean and variance for the delay of successful requests (same configuration as left).

described in Section 2.3, 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 2. 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 3.4, 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.

4.2 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 7(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). In Figure 7(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 allow a connection to pass through or (2) consistently drop it. We also observed that PD reduced the variability of the underlying aggregate traffic.

Figure 7 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. (8). 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 analytic 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 3.1), 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

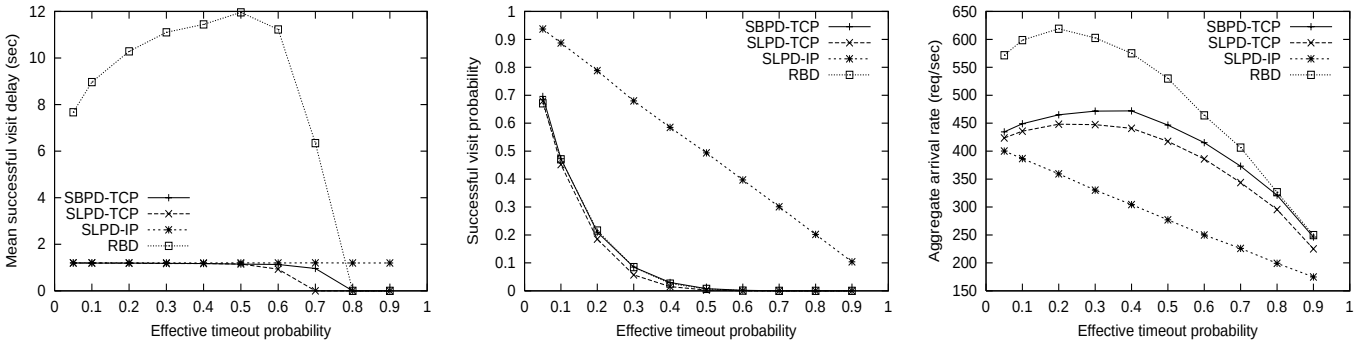

Figure 8: User-visit behavior. In all cases, λ_0 = 60 clients/sec and T_{abort} = 20 sec, (left) mean successful visit delay (a point with zero value implies that no visit was successful), (center) probability of successful visit, (right) effective arrival rate.

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).

4.3 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 performance 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 8(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 8(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 8(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.

4.4 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]. 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 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 no different from traditional control mechanisms.

5. RELATED WORK

Several recent studies have focused on characterizing aggregate traffic during FCEs [23, 21]. 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, 32, 35, 28], 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 [15, 30, 31, 34, 5, 18] presented analytical characterizations of the throughput of TCP's congestion control as a function of RTT and packet loss probability. We view our proposed client model 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 where we investigated both the influence of low-level network 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) [6, 13, 4], and Explicit Congestion Notification (ECN) [12] where we 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.

6. CONCLUSIONS

We characterized the dynamics of persistent clients in aggregate traffic. In particular, we showed that client's persistence, which is due mostly to TCP's congestion control, has a direct effect on the stability and effectiveness of traffic control mechanisms. Based on the analysis of real clients, we constructed an accurate model of client's persistence and used the model to derive the optimal drop strategy for controlling aggregate traffic. Furthermore, we proved that persistent dropping yields the lower bound that an AQM technique can achieve in reducing the effects of packet drop on client-perceived delay and on the effective arrival rate. 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 provides an optimal way to achieve quick convergence to the control targets with minimal intrusion on successful connections.

7. ACKNOWLEDGEMENTS

We gratefully acknowledge helpful discussions with, useful comments from, and the support of Padmanabhan Pillai, Haining Wang and Brian Nobel. The work reported in this paper was supported in part by the Saudi Ministry of Higher Education and NSF under Grant CCR-0216977.

8. REFERENCES

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

[2] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in *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] L. Cherkasova and P. Phaal, "Session Based Admission Control: a Mechanism for Improving Performance of Commercial Web Sites," in *Proceedings of Seventh IWQoS*. IEEE/IFIP event, May 1999.

[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," in *Self-Similar Network Traffic and Performance Evaluation (K. Park, W. Williner, eds.)*. Wiley-Interscience Publication, New York, 2000.

[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. Jacobsen, "Random Early Detection Gateways for Congestion Avoidance," *ACM/IEEE Trans. on Networking*, vol. 1, no. 4, pp. 397–417, 1993.

[14] S. Floyd and V. Jacobson, "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] G. J. Hahn and S. S. Shapiro, *Statistical Models in Engineering*. John Wiley and Sons, Inc., 1976.

[17] J. Heidemann, K. Obraczka, and J. Touch, "Modeling the Performance of HTTP Over Several Transport Protocols," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 616–630, November 1996.

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

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

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

[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," in *SIGCOMM Computer Comm. Review*, vol. 32, no. 3, July 2002.

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

[25] J. Mogul, "Observing TCP Dynamics in Real Networks," in *Proceedings of the ACM SIGCOMM '92*, 1992, pp. 305–317.

[26] J. C. Mogul, "The Case for Persistent-Connection HTTP," in *Proceedings of the ACM SIGCOMM '95*, 1995, pp. 299–313.

[27] J. C. Mogul and K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel," *Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, August 1997.

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

[29] D. P. Olshefski, J. Nieh, and D. Agrawal, "Inferring Client Response Time at the Web Server," in *Proceedings of the ACM SIGMETRICS '02 Conference*, Marina del Ray, CA, June 2002.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

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