Re-synchronization and Controllability of Bursty Service Requests[*]

Hani Jamjoom
University of Michigan

Padmanabhan Pillai[†]
Intel Research Pittsburgh

Kang G. Shin
University of Michigan

{*jamjoom,pillai,kgshin*}*@eecs.umich.edu*

Abstract

There is an increasing prevalence of interactive Web sessions in the Internet. These are mostly short-lived TCP connections that are delay-sensitive and have transfer times dominated by TCP backoffs, if any, during connection establishment. Unfortunately, arrivals of such connections at a server tend to be bursty, and can trigger multiple retransmissions, resulting in long average client-perceived delays. Traditional traffic control mechanisms, such as token bucket filters, are designed to complement admission control mechanisms, by regulating throughput, bounding service times, and protecting systems from overload. However, they cannot control connection-establishment delays, and thus, do not provide effective control of client-perceived delays. We first present the surprising discovery of a re-synchronization property of retransmitted requests that exacerbates client-perceived delays when traditional control mechanisms are used. Then, we introduce a novel, multi-stage filtering scheme called *Abacus Filters* (AFs) that limits the client-perceived delay while maximizing server throughput even in the case of bursty connection arrivals. Analysis of delay-control properties of various filtering mechanisms is presented, along with a detailed performance evaluation. AFs are shown to exhibit tight delay control and better complement traditional admission control policies.

Keywords—**TCP Performance, TCP Measurements, SYN Packets, Admission Control, Re-synchronization**

1 Introduction

The cornerstone that protects the Internet from total collapse is congestion control in the Transmission Control Protocol (TCP). Designed as a distributed mechanism to protect the network from overload, it forces end-hosts to back off when packet losses are detected. Unfortunately, there is no concept of service quality in its design, and therefore, under certain conditions, it unnecessarily exacerbates client-perceived delay. This effect is particularly apparent for short-lived connections, which includes most Hyper Text Transfer Protocol (HTTP) traffic, when request packets are lost. Because TCP exponentially backs off before each retransmit, the connection establishment can quickly dominate the data transfer time in each connection. A request that is dropped twice, for instance, will typically exceed the 8-second delay that is commonly regarded as the response-time threshold after which an Internet server loses over 50% of its clients [42].

Especially during periods of heavy load, an Internet server or router must ensure bounded client-perceived latencies by deploying control mechanisms at various levels of the communication hierarchy. Admission control, whether it is based on resource reservation or measurements, has traditionally been used to provide the necessary coarse-grain delay controls by limiting the number of clients at the application or session level [2, 39]. However, since admitted streams themselves are often bursty [14, 37], these control mechanisms must still deal with occasional bursts of incoming requests. If a burst in the arrival rate is too high, or lasts too long, clients may experience excessive delays. In contrast, traffic control operates at the lower network layers and, when used in conjunction with admission control, defines the mechanism that enforces contracts between clients and the network. In this respect, traffic control is the first-line of defense and must be tailored to match the intrinsic behavior of the underlying traffic; for new connection requests, it must be sensitive to delays incurred due to request dropping.

Commonly-used traffic control or shaping techniques such as Token Bucket Filters (TBFs) [36] can reduce the burstiness of the filtered traffic and limit server load, but cannot provide the necessary control over the connection-establishment latencies, and thus, cannot bound overall client-perceived delay. Particularly, the dropping of "out-of-profile" packets alone cannot deal with delays due to TCP backoffs and retransmissions, and, in some cases, can introduce instabilities in the system. In contrast to this policing technique, smoothing involves buffering all incoming requests, and involves no explicit dropping. This is sufficient for small bursts, but with a large discrepancy between service rate and the burst arrival, a potential hazard of self-similar request arrivals [14], requests may be buffered for an extended period of time. With TCP timeouts and retransmissions, buffering an incoming request beyond some time has a similar effect on the client as dropping the request. In this paper, we investigate the effects of dropping SYN packets[1] on client-perceived delay and on system throughput. We also identify the shortcomings of *deterministic control-limit (DCL) mechanisms*,[2] which include traditional system queues and TBFs.

In order to develop control mechanisms that can provide the desired limits on client-perceived delays, we need to provide greater power to the server during periods of heavy load. In particular, we need to provide a mechanism by which the server can "reject" a client request. With traditional traffic control mechanisms, requests are simply dropped, giving clients a chance to reconnect again after a timeout period, at the expense of increased delay. However, dropping a request does not guarantee its admission at a later time, since the server could still be heavily-loaded. In fact, we will show that in some cases, DCL

[*]The work reported in this paper was supported in part by the NSF under Grant E1A9806280 and by HP under a generous grant.

[†]This work was done while at the University of Michigan

[1]Since sending a SYN packet is the first step of TCP connection request, controlling SYN packets is analogous to controlling connection requests.

[2]The term "Deterministic Control-Limit" in queueing theory applies to the class of control policies that drop incoming requests (with probability 1) after a certain threshold is reached. Otherwise, it accepts the incoming request (with probability 1). One can easily see why system queues and TBFs fall in this class.

		DROP BEHAVIOR			REJECT BEHAVIOR	
Client OS	% Clients	Drop SYN *RTO (sec)*	Drop After Connection Establishment	Drop After Sending Data	Receive ICMP *destination unreachable*	Receive RST packet
FreeBSD	0.01	3, 9, 21, 45	All clients connect using 3-way handshake, send a request packet, receive RST (since server dropped the connection), and abort the connection	All clients connect using 3-way handshake, send a request packet, receive ack, and wait	Client aborts the request	Client aborts the request
HP-UX 11	N/A	2, 3, 9, 21, 45				
Linux 2.2/2.4	0.3	3, 9, 21, 45				
Solaris 2.7	0.1	3.4, 10.1, 23.6, 50.6, 104.6, 164.6				
Win 9x, NT	81.5	3, 9, 21			Ignores ICMP packet and retransmits after timeout	Retransmits SYN packet immediately after receiving RST packet
Win 2000	12.4	3, 9				

Table 1: Controlling TCP connections for different client platforms. Client statistics in Column 2 were based on 600+ websites that Proteus (www.proteus.com) hosts which average about 3.4 million hits per week. Retransmission numbers in Column 3 are estimates based on a limited set of machines. We observed that initial RTO values are constant across multiple connections regardless of any estimate from previous connections to the same host.

implementations do not improve the number of served requests, and instead, only increase the average delay of all incoming requests. A client whose requests will never be served should, thus, be *explicitly* informed of this as soon as possible. The *reject* mechanism performs this action of informing a client that his/her request will not be served, and that s/he should not try to reconnect again. In a sense, the reject mechanism provides an extension to admission control that operates at fine granularity and is performed at the lowest networking levels. We elaborate on the actual technique in Section 2. A server will only resort to such extreme measures when it is certain that its overload will be sustained throughout the period of retransmission.

The goal of this paper is to limit the average total delay experienced/perceived by all clients. We use a simple economic formulation to describe our basic goal. First, assume that each request incurs a cost, $h \times T_k$, where h is some constant and T_k is the delay for request k. A request will also incur a fixed cost r if rejected. Our objective function is to minimize the total cost, which can minimize the average client-perceived delay, given an appropriate setting of h and r. We use this economic formulation to create an analytic model of the expected throughput and client-perceived delay of the system. We also describe in Section 4.2 how choice of h and r relates to the minimization of rejected packets and client-perceived delay.

Based on our model, we propose the *Abacus Filter* (AF), a hybrid token bucket implementation, tailored to the intrinsic behavior of SYN packets. AFs estimate the amount of available future service capacity and only admit a portion of any request burst that, with high probability, will eventually be serviced. This way, bursty traffic or a sudden jump in network load does not affect the overall delay of successful connections. AFs are particularly useful for controlling traffic aggregates that are destined to web servers — e.g., Flash Crowds, where a large number of legitimate users target the same network or server. As we show, without proper control, packet drops can cause future retransmissions, which can further cause another round of drops and retransmissions. This behavior can quickly increase the client-perceived delay. Our approach, however, maximizes the number of successfully-served requests, while keeping client-perceived delay in check. AFs complement *Persistent Dropping* [23] by specifying how much of the incoming SYN packets to drop and how much to reject. Furthermore, AFs give the flexibility to balance between minimizing the client-perceived delay versus maximizing the filter's throughput.

This paper is organized as follows. We first motivate connection-level controls and identify enforcement mechanisms in Section 2. We then characterize the behavior of new connection requests and discuss its impacts on DCL controls in Section 3. We formulate our analytic model and derive a predictive heuristic in Section 4. We present our new control mechanism in Section 5, and empirically evaluate some performance issues in Section 6. The paper ends with related work and concluding remarks in Sections 7 and 8, respectively.

2 Enforcing Traffic Control

Traffic control operates at the lowest network levels to enforce the proper behavior on the underlying traffic, and is used primarily to bound service times. Under moderate network-load conditions, servers can simply buffer the incoming requests to manage small bursts. In the presence of high and bursty traffic loads, a traffic shaping technique such as TBF may be used to regulate incoming load, but this is not sufficient to limit connection-establishment delays. To limit client-perceived delays, a more radical approach is required. As we shall show shortly, this will involve a combination of dropping and rejecting new connection requests. Two fundamental questions arise, however. First, why should we be only concerned with new connection requests? Second, how can we implement the rejection mechanism?

To answer the first question, we investigate dropping a single request at various points in its progress through a typical server. Briefly, for each request, a client must (1) establish a connection, (2) send meta-data describing the request, (3) receive the response from the server, and optionally (4) close the connection. This is a relatively accurate description for HTTP 1.0. For HTTP 1.1 using persistent connections, steps 2 and 3 may be repeated before closing the connection. Regardless of the underlying protocol, there are three points where a server can enforce its controls: (a) immediately after receiving the SYN request, (b) while establishing the connection, (c) while (or immediately after) receiving the request. Performing any control while sending the response is counterproductive for these short-lived TCP connections, as the request will have already received most of its service and consumed necessary server resources.

We instrumented the Linux kernel (version 2.2.17) to allow the dropping of requests at any time in the system. The introduced system call, `whack()`, simply wipes out the TCP connection state as if it were properly terminated by the OS. However, in the process of removing the connection, the OS does not inform the connected client of this action (if it did, this would basically revert to a proper shutdown of the connection). The `whack()` function only drops established connections. To drop connections during the establishment stage (steps a and b above), we modified the Linux firewall utility `ipchains` to drop a request at any point of the 3-way handshake. With slight modifications to the Apache 2.1.3 web server, we are able to observe the reaction of different clients to drop at steps a, b, and c. The clients cover a spectrum of popular web browsers running on various OS platforms.

The results of our experiments are detailed in Table 1. The drop behavior portion of the table highlights an important result: drop-

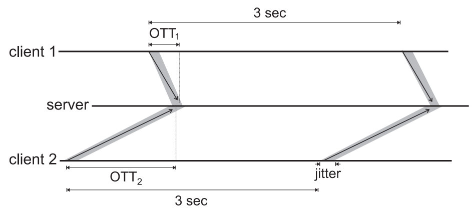

Figure 1: Synchronization of two SYN requests with different one-way transmission time (OTT) values and network jitter, but with the same retransmission timeout (RTO) values.

ping connections during steps b and c requires an equivalent amount of bandwidth and server resources since the connecting client is unaware of the drop and sends the request data regardless of the state of the connection. On the other hand, dropping SYN requests minimizes both bandwidth and server resources. However, as the table shows, clients often retransmit their requests over a very long period (by doubling the retransmission timeout (RTO) after each successive attempt). To avoid repeated attempts when a client's request will most likely time out, it is beneficial to equip servers with the ability to reject a request altogether and inform the client not to retransmit.

Rejection of requests, at first glance, seems relatively easy to accomplish. RFC 1122 [9] defines that a host receiving an "`ICMP destination port unreachable`" message must abort the corresponding connection. This can therefore be used by servers to reject clients. Unfortunately, not all operating systems adhere to this specification. In particular, Microsoft OSs completely ignore these messages, as shown in Table 1, which details the reaction of various OSs to different control mechanisms. Furthermore, many servers are designed to limit the rate at which ICMP packets are sent out, following the specification in RFC 1812 [3]. So, the simple ICMP-based technique is not currently a promising reject mechanism.

Alternatively, the server can send a reset packet (which has the RST flag set in the TCP header) whenever it wants to reject a connection. This is similar to a mechanism used in some routers and firewalls to indicate queue overflow and avoid overload. A host receiving a reset packet must abort the corresponding connection as described in RFC 793 [34]. The use of reset in this context does violate its original intent, but does not conflict with TCP standards for correct client-application behavior, i.e., will not break existing client applications. Again, with the exception of Microsoft OSs, sending a reset achieves the desired effect. In this case, the Microsoft OSs did not wait for a timeout. Instead, they attempt to reconnect immediately when a reset is received. When the reconnect attempt also results in a reset, the client repeats the attempt, for a total of 4 tries (3 for Windows 2000) before giving up.[3] In this respect, a client is informed of the rejected request, but at the expense of several round-trip times as well as increased network load. Finally, we are unaware of any OS implementation that would limit the rate at which reset packets are sent.

The above two approaches show that there is no universal method for rejecting a client. One can argue, however, that higher-level mechanisms, such as sending back an HTTP 500 status code, can achieve the desired goal. Unfortunately, implementing such mechanisms at the application level incurs almost as much server overhead as accepting the request, while implementing this anywhere below the application level (e.g., in the kernel) suffers from two major drawbacks. First, it requires a separate TCP stack implementation to mimic the process of completing the three-way handshake, accepting the requests, and then sending the reply that rejects the requests. This not only violates protocol layering, but also consumes more resources as indicated in

[3]It has been argued that Microsoft's implementation is in response to faulty firewall implementations that send reset packets in response to connection-establishment requests.

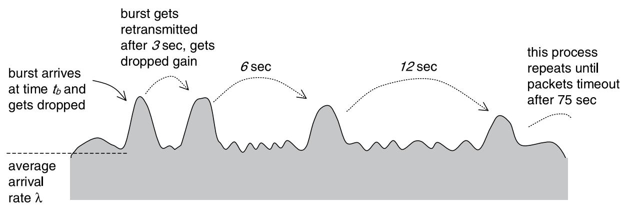
burst gets retransmitted after 3 sec, gets dropped gain
burst arrives at time t_b and gets dropped
6 sec
12 sec
this process repeats until packets timeout after 75 sec
average arrival rate λ

Figure 2: Typical dropping behavior of a burst of SYN requests

Table 1. Second, a different reject mechanism and the corresponding stack implementation would be required for each high-level protocol, which limits its applicability to future protocols. So, a low-level reject mechanism that can regulate requests prior to connection establishment is the best approach to providing a universal reject mechanism that will be handled by all client applications gracefully. As we will show that such a feature is crucial for bounding client-perceived delays, particularly with highly-loaded servers, this will hopefully motivate the means of providing a universal low-level reject mechanisms for TCP connection requests. For the remainder of this paper, we assume that such a reject mechanism is available. It is worth noting that we do not account for "human" factors in our model. Prematurely aborted connections and manual reconnect attempts by human clients require a separate study that can incorporate their unpredictable behavior.

3 Limitations of Deterministic Control-Limit Mechanisms

Every Internet server implicitly implements connection dropping through the nearly universal use of fixed-size protocol queues. This is, in a sense, a control mechanism that limits the load on Internet servers to some degree by dropping requests when queues are full. Rate-based mechanisms, such as TBFs, perform explicit dropping to achieve more configurable objectives: they regulate the arrival of requests such that it conforms to some average arrival rate and limit the maximum burst size admitted [24]. Unfortunately, uncontrolled drops directly impact both client- and server-perceived performance. An overloaded server that temporarily drops incoming requests will increase network load by causing an additional surge of retransmitted requests to appear at a later time. Furthermore, clients will experience exponentially longer delays when their connection requests are dropped.

The most critical and surprising consequence of controlling SYN requests is the re-synchronizing nature of the retransmissions of dropped requests. As we shall show, when a burst of requests is dropped, with high probability, this action will cause a similar burst of retransmitted requests at a later time. This re-synchronization is also independent of clients' link delays, and unless the burst is very small, link-delay jitter cannot significantly spread the retransmitted burst over time. In this section, we examine this issue and investigate its impact on a typical rate-control mechanism.

3.1 Bursty Traffic Behavior

To correctly model the effects of traffic control, we must identify the reactions of individual connections as well as flow aggregates to packet loss. Measuring the former is straightforward. One can independently identify the retransmission behavior of clients when requests are dropped (Table 1) and can further measure link-specific characteristics such as round-trip delay, jitter, throughput, and loss rate [40]. However, characterizing aggregate behavior remains a difficult problem. Although there are numerous Web traffic traces, they cannot be used to extract the precise aggregate reaction to packet loss under varying conditions for three reasons. First, HTTP access logs only provide information on accepted packets that are processed by the underlying server. Thus, the retransmission behavior of dropped bursts is not available in such logs. Second, in existing low-level packet traces, it is difficult to detect the dynamics of dropped bursts since packets can be dropped at any point from the client to the server. Third, traces are static and cannot be used to see the effects of control policies that will change the request arrival dynamics under various configuration scenarios. Measurements of such effects can only be performed under a controlled environment using live clients. Despite this lack of meaningful traffic traces, we are able to extrapolate from multiple individual client results an important characteristic of aggregate traffic, namely re-synchronization of SYN bursts.

Consider a scenario in which a burst of new connection requests arrive at a server and are then dropped. As shown in Table 1, most clients will time out after 3 seconds. This timeout is measured from the transmission of a packet, not the time it is dropped. Thus, assuming that network conditions remain similar, the retransmitted requests will arrive at the server roughly 3 seconds after the originals, independent of network latency from the client to the server. As a result, the burst of drops will cause a burst of retransmissions to arrive roughly simultaneously at the server. This is exemplified in Figure 1, where the arrival of the two retransmitted packets depends only on the timeout value, T_k, and network latency jitter for each request. Therefore, any dropped burst may be repeatedly dropped and retransmitted as a burst until the corresponding requests time out, as exemplified in Figure 2. This, however, only shows how synchronized retransmissions may arise and the general effect on the arrival rate at the server. Specifically, we identify three factors that have direct effect on the synchronization of retransmitted requests: (1) the distribution of RTO values, (2) the distribution of network jitter, and (3) the length of incoming bursts. If the network jitter or the RTOs are distributed over narrow intervals, we can expect retransmissions of a dropped burst to occur in highly coherent, synchronous bursts. Otherwise, the retransmissions will be less coherent, and spread out over a longer time interval than the original incoming burst.

First, we determine the distributions of RTOs. Let $f_{d_i}(x)$ be the probability density function of d_i, the RTO value after the i^{th} retransmission attempt, where $i = 0, \ldots, N-1$, and N is the maximum number of retransmissions. Recall that a connection request can be retransmitted several times before it is aborted, and that each attempt i has a different timeout d_i before another attempt is made. Here, d_0 corresponds to the initial RTO after the original request is sent. The overwhelming popularity of the Microsoft-based OSs ($\approx 94\%$ of all clients) implies that the vast majority of clients will enforce similar RTO values. Therefore, $f_{d_i}(x)$ is primarily governed by the accuracy of the timeout values on individual Windows machines. We can use empirical data to estimate $f_{d_i}(x)$. We collected RTO measurements from 22 different x86-based machines running the Windows 2000 operating system. These machines are carefully selected from a varying set of hardware vendors. Ten independent measurements are taken from each test machine by initiating a connection request to a laptop, which acts as a server and is connected via a crossover cable to minimize measurement error. The laptop (a Pentium III 733 MHz with 384 MB RAM running Linux kernel version 2.2.17) is configured to drop all connection requests and uses `tcpdump` to collect all measurements. We also perform a similar experiment, collecting data from 17 other machines running a mixture of Windows 98 and NT 4.0. The latter measurements show similar results which are omitted here for space considerations.

Figure 3(a) shows the resulting measurements for both the first and second RTOs. We present the histograms at two granularities: first, at a coarse 100 msec resolution to show the general trend, and the second at a finer 10 msec granularity to estimate the actual distribution. Generally, most clients have timeout values of approximately 3 and 6 seconds for the first two retransmissions. We observed only a pair of outlier machines, which had mean RTO values of 3.5 and 6.5 seconds. Both of these happened to have the same model of network interface adapters. Looking at a finer granularity, we find that the first RTO is almost uniformly distributed over a 100 msec range while the second RTO is very narrowly concentrated, mostly in a 10 msec range.

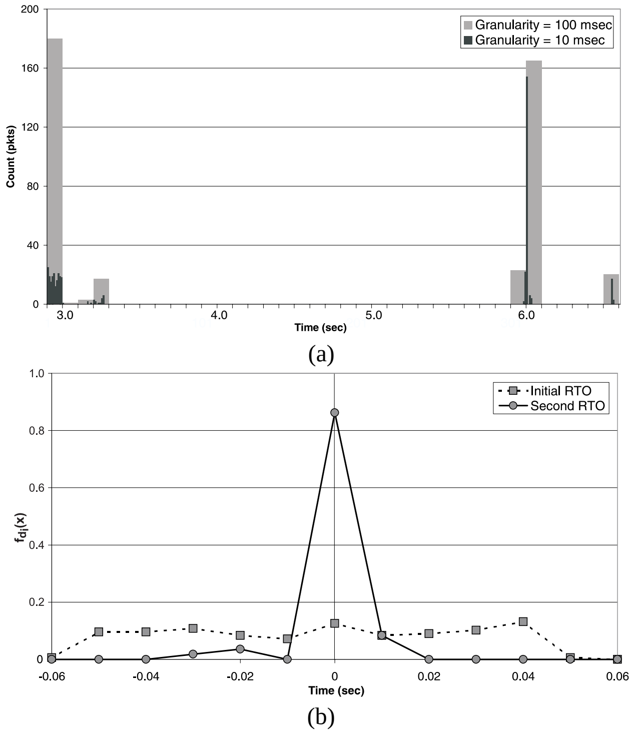

Figure 3: RTO distribution for Windows 2000 clients. Measurement were taken from 22 different machines with highly varying hardware. (a) The actual RTO for all clients. (b) Probability density function (pdf) around the mean for both initial and second RTO.

To estimate $f_{d_i}(x)$, we shift each data point by the mean RTO value of the corresponding samples (excluding the outlier machines), and plot the resulting distributions. Figure 3(b) shows $f_{d_i}(x)$ for $i = 0, 1$. On visual inspection, f_{d_0} seem to follow a uniform distribution, whereas f_{d_1} seem to follow the normal distribution. We use the Kolmogorov-Smirnov (K-S) test [11] to verify this. The distribution of the initial RTO passes the K-S test, which indicates that $f_{d_0}(x)$ can indeed be represented by a uniform distribution. However, the second RTO does not pass the K-S test, due mainly to the higher concentration of the data around the mean. In fact, 75% of RTO values are within a 2 msec interval and 90% of the second RTOs are within a 5 msec interval, which is close to the expected error in our measurements. The measurements for the third RTO from our Windows 98 and NT experiments (not shown) also have similar characteristics. Based on this, we can describe $f_{d_i}(x)$ as follows:

$$f_{d_i}(x) = \begin{cases} 0.01 & \text{for } i = 0 \text{ and } |x-3| < 50 \text{ msec} \\ 0.5/\epsilon & \text{for } i = 1 \text{ and } |x-6| < \epsilon, \text{ as } \epsilon \to 0 \text{ msec} \\ 0.5/\epsilon & \text{for } i = 2 \text{ and } |x-12| < \epsilon, \text{ as } \epsilon \to 0 \text{ msec} \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

The RTO distribution alone is not sufficient to determine retransmission burst coherency. We must also look at the distribution of jitter in network delay. Characterizing network delay, in general, has been the focus of many studies [1, 30, 40]. There are, however, two relevant points that one can conclude from these studies. First, network delay varies across different links and over long time scales. Specifically, Mukherjee [30] described packet delay using a shifted gamma

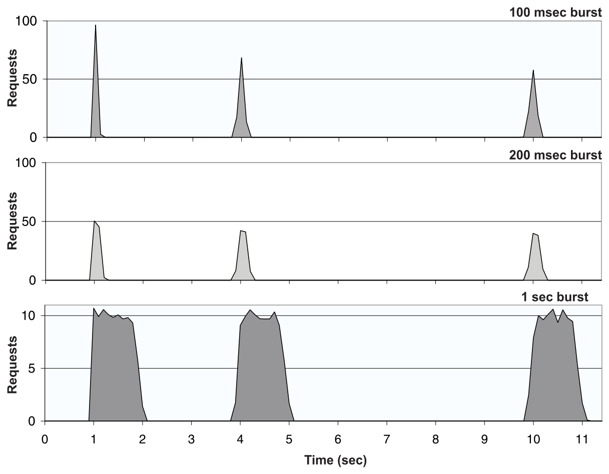

Figure 4: Re-synchronization of SYN bursts. A 100 requests burst arrives at a 100 msec, 200 msec, 1 sec intervals, respectively and are dropped.

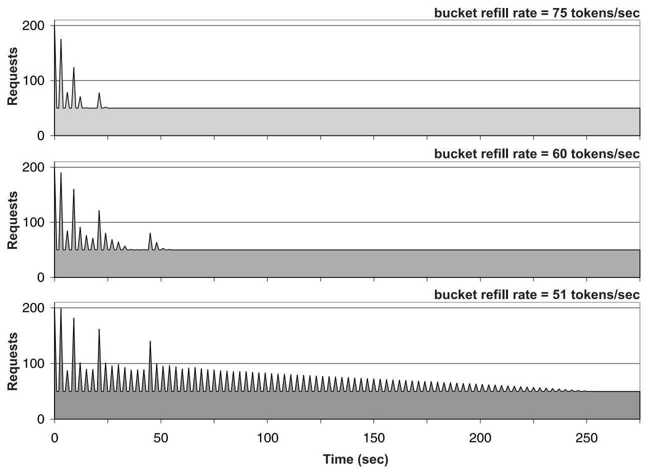

Figure 5: Burst convergence for TBFs with different acceptance rates r_t. A one second burst of 150 requests arrives on top of regular traffic arriving at a constant rate $\lambda = 50$ reqs/s.

distribution with varying parameters across different network paths. Second, over short time (less than 30 minutes) scales, link delay is relatively stable, except for the occasional spikes in delay. Paxson [40] found that standard predictors (such as weighted moving average) can give a good indication of future delays. Furthermore, Acharya [1] empirically verified that delay jitter is confined to a 10 msec interval for short time (less than 103 secs) scales, which is well below the RTO range. Since we are only interested in determining the degree to which a burst will remain concentrated as it is retransmitted, we can use this result to model jitter. Specifically, we can use a uniform 10 msec density function to describe network jitter, $f_J(x)$, over short time scales as follows:

$$f_J(x) = \begin{cases} 0.1 & \text{for } |x| < 5 \text{ msec} \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

Combining $f_{d_i}(x)$ with $f_J(x)$, we can obtain the following important results:

R1 A dropped burst $B(t)$, once retransmitted, will spread out at most over a 220 msec interval compared to the original burst. Subsequent retransmission attempts will spread the bursts by at most an additional 20 msec per retransmission. Therefore, short bursts, once dropped, will have retransmissions that spread out significantly, and will therefore be mitigated. Longer bursts, however, will not be significantly affected by these low spread-out times, so the retransmissions will be re-synchronized and arrive at the server as bursts of equal intensity.

R2 If a control mechanism looks at the aggregate traffic over a fairly long time interval (e.g., seconds), it will see retransmissions of dropped bursts arrive at predictable intervals as coherent bursts.

We use *eSim*, a simulation tool, to confirm this effect. Our simulator is based on the Linux implementation of the TCP stack and allows us greater flexibility in specifying server resources than would *ns*. A set of 100 clients are configured to send connection requests to a single server with enough network bandwidth to handle the maximum burst. Each client has an RTO distribution that follows the above $f_{d_i}(x)$ and a network latency jitter that is uniformly-distributed over a ± 5 msec range. We vary the time alignment of the initial client transmissions such that a burst of requests arrives at the server spread over 100 msec, 200 msec, and 1 sec time intervals. The server, in turn, drops all incoming connections. Figure 4 shows the arrival pattern at the server and the effects on the shape of retransmission bursts. Clearly, the narrow initial burst arrival is spread out significantly upon retransmission, and the burst intensity is reduced. As the burst duration increases, the effects of the variations in link delay and RTO become inconsequential, and the bursts occur at an equal intensity.

So far we have not discussed the burstiness of request arrivals, which remains an important issue. It is widely accepted to model new user (not request) arrivals using a Poisson process. However, since each new user generates request patterns that can be modeled by an ON/OFF process [12], the overall arrival of new request tend to be bursty [37]. When the persistent connection feature is used in HTTP 1.1, multiple requests can be sent over a single TCP connection, which — in theory — should eliminate bursty arrivals of SYN packets. However, as the authors of [41] have found, current browsers continue to issue multiple "persistent" connections in parallel to maximize their throughput. This implies that, once the main page is fetched by the browser, each client will still cause a burst of SYN packets, though this may not be as severe as with a browser using HTTP 1.0. Furthermore, Feldmann [14] concluded that connection arrivals are best modeled using a Weibull distribution, though the actual parameters vary across different traces. What is important, however, is the time scales over which the measured traffic is bursty. Feldmann found that burstiness is apparent across time scales of seconds to hours. However, sub-second bursts were difficult to observe, a fact also confirmed by Leland [26]. From these, we can see that the expected duration of bursts should be in the range of seconds or longer. Since this is much larger than the range of $f_J(x)$ and $f_{d_i}(x)$, we can expect the re-synchronization of the retransmissions of dropped bursts to occur, and that they will not be significantly reduced due to the distribution in delay jitter and RTO of the clients.

3.2 Limitations of Token Bucket Filters

A Token Bucket Filter (TBF) is a simple and well-known mechanism used for rate control [24]. In its general form, the TBF can be seen as a bucket of tokens, and is described by two parameters: a refill rate, r_t, and a bucket size, W. When a packet arrives at a particular TBF, a token is removed from the bucket and the packet is accepted into the system.[4] If the bucket is empty, then the packet is not accepted. The bucket is replenished with tokens at rate r_t, but is limited to a maximum of W tokens available at any time. This ensures an average acceptance rate of r_t while maintaining the ability to handle some

[4]In this paper, the "system" corresponds to the end-server OS; in general, it is not limited to this, since TBFs can be used in network switches and routers.

burstiness in the arrival stream. The behavior and simplicity of TBFs make them useful for regulating general traffic flows. As they can provide good approximations of application- or system-level queues (such as the TCP listen queue),[5] TBFs can be used to match the number of requests entering the system to the server capacity by setting W to the maximum queue length and r_t to the average processing rate of the system.

When controlling new connection requests, we can select the action the TBF will take when the bucket is empty. The TBF can either drop or reject the request. This plays an important role in determining the overall effectiveness of a TBF. Dropping SYN requests will delay the request to a later time. However, because the arrival of requests are inherently bursty (as discussed earlier), and because of the re-synchronization of retransmitted SYN requests described earlier, dropping requests may destabilize the system. This will also have direct impact on client-perceived delays since this additional instability causes increased retransmissions, timeouts, and connection-establishment delays.

Figure 5 shows an example of this, where retransmitted bursts trigger additional dropping of new incoming requests. In this experiment, we simulate the effects of TBFs on bursty request arrivals using *eSim*. The clients are assumed to follow the retransmission behavior and distributions discussed in the previous section. We use a TBF to regulate the admission of new requests, which arrive at a constant average rate, except for a single initial burst. This unrealistic load arrival is used to determine the burst response of the system as the system becomes critically-loaded. We will look at more realistic arrival distributions later. For now, we vary the token refill rate, setting it closer to the traffic arrival rate, and set the bucket size, W, such that it can be completely refilled in 3 seconds. The intuition behind this setting is that the TBF matches the processing capacity of the server and the admitted requests should not be queued for more than 3 seconds; otherwise, the client will assume packet loss and retransmit them anyway, wasting network bandwidth and server resources. We also vary the initial burst length and the average arrival rate.

Based on this simulation, we observe that the difference ($r_t - \lambda$) between the token refill rate and the average arrival rate greatly influences the stability of the system. As the system operates closer to full capacity (i.e., λ approaches r_t), it becomes critically-loaded, and any increase in the apparent arrival rate will trigger drops. The retransmissions of the original burst of requests that were dropped, due to their re-synchronization, will arrive at the server at roughly the same time, triggering further bursts. This will result in more synchronized drops of the retransmitted requests as well as many new requests. As a result of the cycle of retransmissions triggering new drops, which trigger further retransmissions, the instability may last for a very long time, continuing until all retransmissions time out or are slowly absorbed by the system. These behaviors are formally analyzed in Section 4.

Alternatively, we can try to avoid these stability issues altogether by rejecting requests when the token bucket is empty, instead of employing the drop-policy analyzed above. This will eliminate such instabilities, since the rejected clients do not retransmit their requests. Unfortunately, the added stability can be achieved at the cost of poor acceptance rates and low server utilization. This is most apparent with bursty arrivals, where the rejection policy is very sensitive to the choice of bucket size (Section 6). A small or moderate size bucket will provide the necessary traffic control and limit admitted bursts, but would reject a significant number of requests, resulting in poor server utilization. A very large bucket would permit more client connections, but translates into long queues in the system, effectively increasing service times and defeating the purpose of the traffic control.

[5]In general, OSs queue SYN requests into a special queue of half-open connections. Once the three-way handshake is complete, they are moved into the listen queue. Therefore, controlling SYN packets also controls the listen queue, although indirectly.

The shortcomings of a TBF are due, in large, to its inability to balance between dropping and rejecting requests. Ideally, one would like to have a control mechanism that drops only those request packets that can eventually enter the system (when more tokens are available), and rejects the rest. We address this limitation by designing the multi-stage Abacus filter that accepts, drops, or rejects requests based on the expected future service capacity.

4 Optimal Control Policy

In order to create a control mechanism that can effectively limit the client-perceived delays caused by TCP backoffs and retransmissions, we need to have a good understanding of the client-server interaction behavior as control policies and arrival distributions vary. To this end, we first develop an analytic model to study the effects of request drops and rejects on aggregate clients' behavior. Based on this analytic model, we will later develop our improved traffic control mechanisms.

4.1 Network Model

In developing our analytic model, we need to specify carefully the parameters of the whole system. We can assume, without loss of generality, that senders adhere to a TCP-Reno style congestion control mechanism [20]. However, this alone does not suffice, as it identifies neither the arrival distribution of request aggregates nor the correlation between different requests in the presence of packet drops. The distribution of network traffic has been studied extensively [25, 26, 33] and is shown to exhibit a self-similar behavior. These results also extend to connection-level analysis due to the aggregation of independent ON/OFF traffic streams representing different user sessions [37]. Other studies [6, 12, 29] have measured the correlation between multiple users and across different requests or sessions originating from a single user. Unfortunately, self-similarity and user sessions do not follow traditional queuing models, complicating our analysis. In particular, accurately modeling a real system with many simultaneous clients, connecting to a server with a complex request arrival distribution and individual retransmission patterns will lead to mathematical intractability. As a result, we need to make several simplifying assumptions.

A1. The inter-arrival times of new connection requests (or SYN packets) are *independent and identically distributed (i.i.d.)*. Although connections may contain multiple requests, as with HTTP 1.1 persistent connections, for simplicity and to better match current browsers that issue parallel HTTP 1.1 requests [41], we assume each new connection will carry one request. Dropping one connection will not affect the arrival of new ones in the future in this model. Assuming *i.i.d.* connections fits well with traditional Poisson models. Although this does not allow for self-similar arrivals, it does make tractable the analysis of traffic control effects.

A2. All incoming connection requests follow the same retransmission behavior. This is justified by the fact that most current TCP implementations are based on TCP-Reno, which exponentially backs off after successive SYN timeouts. The behavior follows Jacobson's algorithm [20, 34], where a SYN packet that is not acknowledged within a RTO period is retransmitted, but with a doubling of the previous RTO period. This is repeated until the connection is established or until a connection timeout period, T_{conn}, is reached, at which point the connection is aborted. As mentioned earlier, we define d_i as the *length* of the timeout period after the i-th retransmission of the connection request, where the initial timeout period is represented by d_0.

A3. Each incoming request can be identified as a new request or the k^{th} retransmission (i.e., depending on the number of times it has been retransmitted, it is classified into one of N transmission classes, $\{Z_0, Z_1, \ldots, Z_N\}$, where N is the maximum number of times a request can be retransmitted). A delay is associated with each class, as described in A2. In practice, a state-based flow classification mechanism, similar to the one developed in [23], can be implemented. However, this assumption is necessary only for the development of our analytic model and is relaxed in our filter design.

A4. Only new requests, not retransmissions, may be rejected. If a request is dropped, but not rejected, it continues to be retransmitted until it is admitted by the system or times out as defined in A2. Given a mechanism to classify requests as in A3, implementing this is straightforward. However, as with A3, this assumption will also be relaxed later to simplify the implementation.

A5. A rejected packet will not be retransmitted again. As discussed in Section 2, this is not the case for certain implementations. We can account for the non-conforming implementations in our model by using a higher cost for rejection.

A6. We ignore link-delay effects in our analysis. We further assume that delay jitter and RTO variations are negligible. This essentially assumes perfect re-synchronization, but does not impact the correctness of our analysis, since jitter and initial RTOs are typically distributed over a very narrow range, as reasoned about in Section 3.1.

A7. The system can be observed at discrete time intervals of T_0. Thus, when referring to time t, we are implying the interval $[t, t + T_0)$. The assumption of perfect re-synchronization in A6 allows this, since all requests dropped within the current time interval will be retransmitted in a single future time interval, as long as we select T_0 such that d_0 is an integer multiple of T_0.

A8. We assume equal treatment in dropping requests. During each observation interval, $[t, t + T_0)$, all arriving requests have the same drop probability, regardless of the number of times they have been retransmitted (i.e., regardless of the transmission class defined in A3). Note that this is only for drops, not rejection (see A4).

The system presented here is used to develop an analytic model to describe the effects of traffic control on incoming connection requests. As we shall show in Section 6, despite the seemingly restrictive assumptions, analysis based on these assumptions can be used to develop control mechanisms that perform well under realistic load conditions, and provide effective control for real-world traffic.

4.2 Analytic Control Model

In this section, we analyze the effects of a general TBF control mechanism on the retransmission of clients' requests and the associated cost metric. Like all TBFs, we can describe this with two parameters, the maximum bucket size, W, and replenishment rate, r_t. The latter is also the average acceptance rate into the system. Unlike traditional TBFs, rather than simply dropping requests once all tokens are depleted, we will use a control policy, g, that decides whether to drop or reject an incoming request. To determine g, we first estimate the expected number of retransmitted requests for any arbitrary arrival rate $\lambda(t)$, assuming a control policy that only drops requests. Using this estimate, we can determine a threshold beyond which it is not beneficial to drop packets. Therefore, the control policy g can simply drop requests up to this maximum value, after which it rejects further requests.

The difficulty in the analysis of this system arises from the varying retransmission timeout periods for successive request drops. We can, however, form recursive equations that capture this behavior. Let $A(t)$ represent the number of packets arriving during interval $[t, t + T_0)$. This value includes both newly-arriving requests, $\lambda(t)$, as well as any retransmitted requests. More precisely,

$$A(t) = \lambda(t) + \sum_{i=0}^{N-1} X_i(t - d_i), \tag{3}$$

where $X_i(t)$ represents the number of request packets that are dropped over the time interval, $[t, t+T_0)$, and belong to the i-th transmission class. We can define $X_i(t)$ as:

$$X_i(t) = A_i(t)\left(1 - \frac{W(t)}{A(t)}\right)^+, \tag{4}$$

where operation $(y)^+$ is defined as $\max(0, y)$ to enforce a floor value of 0, $A_i(t)$ represents the number of requests belonging to the i-th transmission class that arrive during the interval $[t, t + T_0)$, and $W(t)$ is the number of tokens available at time t (the beginning of the observation period). The term $(1 - W(t)/A(t))^+$ defines the drop probability for any request during the time interval $[t, t + T_0)$, assuming an equal treatment of all incoming requests (Assumption A8). Since a dropped request belonging to the i-th class will be retransmitted after d_i, we can write $A_i(t)$ in terms of $X_i(t)$ as:

$$A_i(t) = \begin{cases} \lambda(t) & \text{for } i = 0 \\ X_{i-1}(t - d_{i-1}) & \text{for } 0 < i \leq N. \end{cases} \tag{5}$$

This forms a recursive equation in $X_i(t)$. Letting $P_d(t) = (1 - W(t)/A(t))^+$, we now have:

$$\begin{aligned} X_i(t) &= P_d(t) X_{i-1}(t - d_{i-1}) \\ &= P_d(t) P_d(t - d_{i-1}) X_{i-2}(t - d_{i-1} - d_{i-2}) \\ &= \left(\prod_{k=0}^{i} P_d(t - \sum_{l=1}^{k} d_{i-l})\right) \lambda(t - \sum_{l=1}^{i} d_{i-l}). \end{aligned} \tag{6}$$

Using the values for $A(t)$, we can determine the number of available tokens, $W(t)$, at any time t:

$$W(t) = \min\left((W(t - T_0) + T_0 r_t - A(t - T_0))^+,\ W\right), \tag{7}$$

where T_0 is the observation period defined earlier. The min is used to enforce a ceiling on the number of tokens in the bucket. It is important to note that the dependency of $W(t)$ on $A(t)$ does not form a cyclic definition, since $A(t)$ does not depend on $W(t)$, but only on $W(t - d_i)$, $0 \leq i < N$. Furthermore, $X_i(t)$ and $W(t)$, though tedious to compute, only depend on W, r_t and $\lambda(t_1)$, for $t_1 \leq t$. This implies that we only need to distinguish between new and retransmitted requests to compute these variables.

Based on the above analysis, we can now determine the expected cost of rejecting or admitting a given request. A simple economic formulation is used to balance between dropping versus rejecting client requests. We associate a cost h for each time unit the request is delayed. There is also a cost r for rejecting a client. A high rejection cost, for example, reflects that it is desirable to reject fewer clients, even at the expense of higher average client-perceived delay. We assume that a timed-out request will incur the same rejection cost r (since the end result is the same as if it were rejected) in addition to the cost of dropping the request until it times out.

It is important to note that our economic formulation assumes that all TCP connections are given an equal weight or priority. In reality, this may not be the case. Connections that belong to paying customers, for instance, may have a higher rejection cost than those that belong to non-paying ones. From that perspective, both delay and rejection costs may vary across different types of connections. The difficulty here is that a particular connection's real cost, a high-level application notion, is not directly available at the lower TCP-stack level, where traffic regulation (e.g., using TBF) is actually being performed. Therefore, we assume that if connections incur different costs, they should be classified first by an independent mechanism (possibly something similar to Layer-7 switching[6]) and that independent traffic regulators with different costs should be used for each class.

Let $c(t)$ be the cumulative cost that is incurred up to time t:

$$c(t) = c(t - T_0) + \left(\sum_{i=0}^{N} h d_i X_i(t) \right) + r X_N(t). \qquad (8)$$

If we define $d_N = T_{conn} - \sum_{i=0}^{N-1} d_i + r/h$, and solve this simple recursion, assuming that t is an integer multiple of T_0,

$$c(t) = \sum_{j=0}^{t/T_0} \sum_{i=0}^{N} h d_i X_i(jT_0). \qquad (9)$$

This cost function is monotonically non-decreasing. Thus, if we know the cost up to time t_1, we can compute the cost at time $t > t_1$ by using:

$$c(t) = c(t_1) + \sum_{j=1}^{\frac{t-t_1}{T_0}} \sum_{i=0}^{N} h d_i X_i(t_1 + jT_0), \qquad (10)$$

where the second term defines the holding cost from time t_1 to t.

So far, our analysis only deals with the cost of dropping requests due to the depletion of tokens. To account for the explicit rejection of new requests, let us define the control policy g as one that rejects a certain portion of incoming requests, and drops the rest. Here, $\lambda_a(t)$ represents the true request arrivals. Let $\lambda_r(t)$ be the number of rejected requests. Then, $\lambda(t) = \lambda_a(t) - \lambda_r(t)$ reflects the portion that is not rejected. In addition, we also need to account for the cost of rejecting a portion of the requests. The additional cost of rejection is thus $r\lambda_r(t)$. Formally, the cost $c^g(t)$ of control policy g can be captured by:

$$c^g(t) = c^g(t_1) + \left(\sum_{j=1}^{\frac{t-t_1}{T_0}} r\lambda_r(t_1 + jT_0) \right) + \sum_{j=1}^{\frac{t-t_1}{T_0}} \sum_{i=0}^{N} h d_i X_i(t_1 + jT_0), \qquad (11)$$

for $t \geq t_1$, where $X_i(t)$ is the same as defined in Eq. (6).

Eq. (11) can be illustrated by the following example. Consider a system that is under-loaded until time t_1, so $c(t_1) = 0$. Now, let a burst of 100 SYN packets arrive immediately after t_1; if all of the packets in the burst are rejected, then the cost at $t > t_1$ is just $100r$. Here the third term in Eq. (11) is 0 since no packets will be retransmitted. However, if only a portion is rejected, then the total cost is the cost of rejecting this portion (second term of Eq. (11)) plus the cost of the delay incurred by those packets that were dropped and then retransmitted at a later time (third term in Eq. (11)). Note that the second term also includes the effects of a retransmitted request inducing further drops by the system.

This analytic model is relatively general (within the limits of our assumptions), and can capture many different optimization scenarios. The parameters h and r may be changed to set the relative costs of rejecting a request.

[6]Layer-7 switching, e.g., Foundry's ServerIron, looks at a packet's payload to determine the corresponding request and can be used to enforce different priorities.

4.3 Predictive Control Policy

The above analytic model provides a mechanism for estimating the total cost incurred due to retransmissions, timeouts, and rejections of arriving requests. It works, however, only when we have full knowledge of new arrivals $\lambda_a(t)$. In implementing a control mechanism, we do not know this for future arrivals. Our objective is to find the control policy g that minimizes $c^g(t)$ as $t \to \infty$ and limits computation cost, without knowledge of the future $\lambda_a(t)$ values.

We look at a predictive heuristic that decides the control action by approximating the analytic cost function. Basically, when no tokens are available, the first requests, up to a threshold, λ_{thresh}, are dropped, and subsequent requests rejected. The predictive heuristic approximates the cost computation for future time t by looking only at the new requests that may arrive in the next time interval, and using this to select λ_{thresh}. Future retransmitted requests are still included in the heuristic. We can then simplify Eq. (11) as:

$$c^g(t) = c^g(t_1) + r\lambda_r(t_1 + T_0) + \sum_{j=1}^{\frac{t-t_1}{T_0}} \sum_{i=0}^{N} h d_i X_i(t_1 + jT_0), \qquad (12)$$

where t_1 is the current completed time interval, and $t > t_1$. Abbreviating the third term with $H(t_1, t)$, we have:

$$c^g(t) = c^g(t_1) + r\lambda_r(t_1 + T_0) + H(t_1, t), \qquad \text{for } t \geq t_1. \qquad (13)$$

Let us now consider the effect of changing λ_{thresh} on the cost computation. Take two control policies, g and g', such that $\lambda_r(t_1 + T_0) = \lambda'_r(t_1 + T_0) - 1$ (i.e, the policy g rejects one less request, so has a λ_{thresh} that is equal to that for g' plus 1). The cost function for switching to the policy g' after time t_1 is:

$$c^{g'}(t) = c^g(t_1) + r\lambda'_r(t_1 + T_0) + H'(t_1, t), \qquad \text{for } t \geq t_1. \qquad (14)$$

We can compute the cost difference, which reflects the change in cost when increasing λ_{thresh}:

$$c^g(t) - c^{g'}(t) = H(t_1, t) - (H'(t_1, t) + r), \qquad \text{for } t \geq t_1. \qquad (15)$$

Using the above formula, we can now find λ_{thresh}. Note that rejection of all requests will make the cost very high. As we reduce the number of rejected requests, the cost decreases. However, at some point, once we let too many requests in, and some begin to time out, the expected cost reaches its minimum and begins to increase. At this point, the cost difference in Eq. (15) will, for the first time, become positive. To find λ_{thresh} that minimizes cost, we can simply search for this occurrence:

$$\lambda_{thresh} = \min_{t \to \infty} \{\lambda(t_1 + T_0) : H(t_1, t) > H'(t_1, t) + r\}, \qquad (16)$$

where $\lambda(t) = \lambda_a(t) - \lambda_r(t)$. In practice, we only need to compute this over a window length $S_G = T_N$, where T_N is the time of the last retransmission before a request times out at T_{conn}. Hence, we only compute λ_{thresh} over time interval $[t_1, t_1 + T_N]$. If, on the other hand, $S_G < T_N$, then λ_{thresh} will be poorly computed, underestimating the cost of dropping packets and favoring this policy even if sufficient future capacity to handle retransmissions does not exist.

The predictive heuristic presented here uses an approximation of the analysis from Section 4.2, to determine the control policy that will minimize expected costs. This is really a greedy mechanism, that can, at best, provide a locally optimal cost under the assumption of no new arrivals in the future. Furthermore, as the cost incurred for a timed-out request is always greater than the cost for a new request that is

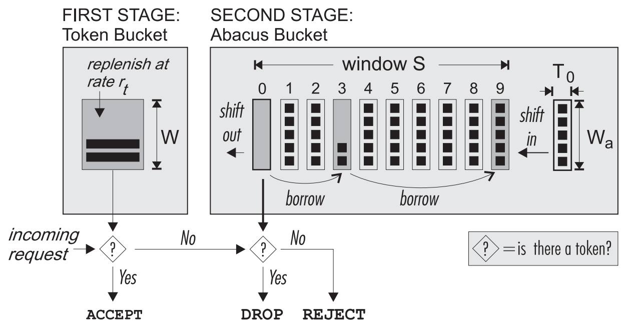

Figure 6: Abacus Filter architecture

rejected, this heuristic will switch to a reject policy at the threshold where an additional drop of a new request seems to cause some previous connection attempt to time out. In addition, this method is limited by the accuracy of its estimates of $\lambda(t)$ and $X_i(t)$. We will in Section 6 see how well this control mechanism performs using realistic traffic models that do not follow the assumptions we have outlined earlier.

5 Abacus Filters

A direct implementation of the predictive control mechanism developed above faces two critical challenges. First, the computation of λ_{thresh} at the beginning of each observation period, with its recursive variable dependencies, can inflict a very high computational overhead. This is especially counter-productive in servers that are already heavily-loaded. Second, we need an accurate measure of the rate of new requests in order to compute $X_i(t)$ values. As the computations are recursive, errors may accrue, degrading the accuracy of the estimates. In this section, we develop an approximate design that captures the benefits of our analytic model, but without its computational costs or the need for precise measurements. This new filter design, called *Abacus Filter (AF)*, is a two-stage hybrid token bucket implementation. The first stage limits the service load, as in a regular token bucket filter. The new second stage determines the appropriate control policy by estimating the number of packets that can be deferred for processing at a future time.

In order to develop a computationally-efficient approximation of our analytic model, we must break the recursive dependency between the state variables and past values. To this end, we disregard all historical information, but this introduces two limitations: (1) we cannot estimate the number of tokens, $W(t)$, that will be available at a future time, and (2) we cannot estimate the number of requests dropped in each transmission class, $X_i(t)$.

We can overcome the former by assuming the worst-case scenario that no tokens will be carried forward into future time periods. Therefore, the expected number of tokens, W_a, available during a future time interval to handle retransmission bursts is determined solely by the token refill rate, r_t, and the arrival rate of new requests, λ, over the time interval, T_0. Hence, $W_a = (r_t - \lambda)T_0$ represents the estimate of future excess capacity available to handle retransmission bursts for any future observation period, which is defined according to the guidelines in Section 4.1.

The second issue, classifying requests into their corresponding transmission classes, is not possible at runtime without significant overheads of tracking every request that arrives, even those dropped or rejected. Without the past drop information ($X_i(t)$ values) and the actual new request arrivals, we cannot compute the expected fraction of retransmissions that are accepted and the relative sizes of future retransmission bursts as in our analytical model. Instead, we simplify greatly by assuming that up to $(r_t - \lambda)T_0$ retransmitted packets are accepted during any observation period, corresponding to the estimated excess capacity based on the average new request arrival rate λ. Retransmissions arriving in excess of this will be dropped again. Assuming oldest packets are accepted first, we can predict the occurrence of retransmission bursts without $X_i(t)$ values or recursive computations.

Combining these ideas, we introduce the AF, a control mechanism that incorporates two stages: the first stage is a regular token bucket, while the second stage decides the traffic control policy. The novelty lies in this second stage, which consists of a series of buckets, each of which has size W_a (Figure 6). Each of these buckets represents the number of retransmissions that can be accepted over an interval of T_0, the *time granularity* of the filter, which corresponds to the observation period in A7. Together, these buckets represent the future retransmission burst handling capacity over a *time window*, S. Rather than replenishing the second-stage filter one token at a time, once every T_0 seconds the token buckets are shifted, dropping the first bucket and shifting-in a new bucket at the end. This models a discrete time progression over the time window S. The new bucket is filled with $W_a = (r_t - \lambda)T_0$ tokens, the estimate of excess resources available in the future based on an average new request arrival rate of λ.

When a burst of requests arrives, the first stage determines the packets that can be accepted immediately. The remainder are passed to the multi-bucket second stage for making a control policy decision. If tokens in these buckets are available, the packets are dropped and allowed to retransmit; otherwise, they are rejected. The AF looks for tokens in future buckets that represent the time the dropped packets will be retransmitted. An example is shown in Figure 6, where a burst arrives at an initialized AF. Consequently, the filter first looks d_0 or 3 seconds into the future, then d_1 or 6 seconds after that, and so on. Only if the entire series of buckets is exhausted, the particular request is rejected. This "borrowing" of tokens, somewhat akin to operating an abacus, performs the same function as the $A_i(t)$ values in our analytic model, which account for the future arrivals of request retransmissions. This way, future request arrivals contending for the same resources will be appropriately controlled.

The borrowing pattern depicted in Figure 6 assumes that all dropped requests belong to a single transmission class, i.e., new requests that are dropped. Processing capacity that will be consumed by the first retransmissions of these dropped requests, as well as all subsequent retransmissions for the portion of the retransmissions that is expected to be dropped again, is accounted for by the removal of tokens in the future buckets. At a future time, e.g., 3 seconds later, when the retransmissions arrive, a portion of these will be dropped and retransmitted again. However, the future capacity needed to handle the subsequent retransmissions has already been accounted for by the token borrowing during the initial drop of the requests. Hence, the portion of retransmissions not immediately accepted can be simply dropped without borrowing. The borrowing process is used only for new arrivals that are not accepted by the first stage token bucket.

Although we cannot classify a request as new or retransmitted to select either simple drop or borrow and drop, given our assumption that $(r_t - \lambda)T_0$ retransmissions, oldest first, are accepted during an observation period, we can make do with just a count of the number of retransmissions that ought to be dropped in this period. A counter associated with each future bucket is incremented when chain borrowing occurs from the bucket. In other words, if we try to borrow a token from bucket i, indicating a retransimission will arrive i observation periods into the future, and the bucket is empty, requiring the borrowing of a token from a further bucket, this indicates that the retransmission is dropped again at the i-th observation period, so we increment counter B_i to keep count of the total number of retransmissions expected to be dropped. When deciding on the control policy, the second stage AF simply drops the first B_0 packets, the count for the current observation period, and then performs the borrowing

```
BORROW_TOKEN
  // Let A[i] and B[i] be the number of tokens and the number of
  // chain-borrowed tokens at i seconds in the future, respectively.
  // Let Z be a list of values of i; initially Z is empty.

  if B[0] $> 0$            // Drop first B[0] packets without borrowing
    B[0] $\leftarrow$ B[0] - 1
    Drop request and exit

  i $\leftarrow$ 3                // Starting index for borrowing
  k $\leftarrow$ 1
  Z $\leftarrow \emptyset$
Loop:
  if A[i] $> 0$            // Remove a token from the corresponding bucket
    A[i] $\leftarrow$ A[i] - 1
    for each j $\in$ Z       // Increment borrow counters for buckets
      B[j] $\leftarrow$ B[j] + 1 // that incur chain borrowing
    Drop request and exit
  else                     // Try to borrow from future bucket
    Z $\leftarrow$ Z $\cup$ {i}        // Track the bucket incurring chain borrowing
    i $\leftarrow$ i + 3*$2^k$       // Determine next bucket
    k $\leftarrow$ k+1

  if k $>$ N or i $>$ S     // Too many retransmissions, or beyond window
    Reject request and exit
  goto Loop
```

Figure 7: Borrowing algorithm of the Abacus Filter

scheme to decide on drop or reject for subsequent packets.

In addition to the token bucket parameters, W and r_t, two basic parameters determine the operation of the AF: *time granularity*, T_0, and *window size*, S. The value of T_0 has to be chosen such that it is much larger than the average jitter of the link delays experienced by arriving packets and also large enough to accommodate variations in d_0, as outlined in Section 3. The window size, S, limits the future borrowing scope of the AF. This value must be selected to minimize total cost in the system. Since for each second of delay for a request, cost is increased by h, while an immediate reject incurs cost r, intuitively, one should reject a request if its expected delay exceeds r/h, and drop it otherwise. This is enforced by setting window size $S = r/h$. Since r/h can be arbitrarily large, we show in Section 6 that setting $S = \max\{T_i : T_i \leq r/h\}$, where T_i is the time of the i-th retransmission ($T_i = \sum_{l=1}^{i} d_l$), does indeed minimize total cost.

The actual implementation of an AF is fairly straightforward. We extend the regular token bucket filter by adding the AF second stage to select the drop or reject policy. The AF's second stage is implemented as a circular array, where the elements represent the number of available tokens in each bucket. Each time a bucket is removed and a new one appended, an index indicating the element corresponding to the current bucket is increased. The array element associated with the removed bucket is reused for the newly-appended one, and is set to the value $W_a = (r_t - \lambda)T_0$. As the average arrival rate of new requests cannot be known *a proiri*, and can change over time, λ is actually an estimate based on the weighted-time average of the observed request arrivals over the most recent observation periods. To determine the number of requests that can be dropped, the filter looks at the appropriate array entries in the future according to the borrowing algorithm in Figure 7, using a simple modular arithmetic to handle wraparound of the index.

The AF that we have developed is computationally-efficient, and can be easily implemented as an alternative to token-bucket filters. We show in the following section that AFs can provide robust traffic control while bounding the client-perceived delay.

6 Evaluation

To evaluate and demonstrate the efficacy of the AF and optimization framework, we equipped our simulator, *eSim*, with working implementations of the following four filter designs.

TBF implements the traditional token bucket filter with a bucket size W and refill rate r_t. Once the bucket is depleted, all subsequent requests are dropped until tokens are replenished.

TBF w/ Reject (**Reject**, for short) is similar to TBF, except for the control action employed: when the bucket is empty, requests are rejected. This implements an aggressive control policy that enforces a strict ceiling on the number of accepted requests.

AF is a direct implementation of the Abacus Filter as proposed in Section 5. The AF is characterized by four parameters: bucket size, W, refill rate, r_t, window size, S, and time granularity, T_0. In all experiments, we set $T_0 = 1$ sec, sufficiently large to overshadow the effects of RTO and delay jitter, yet small enough to maintain a high level of accuracy in estimating server capacity.

Predictive Greedy (**Greedy**, for short) is based on the predictive control policy that was developed in Section 4.3. Our implementation here relaxes assumptions A3 and A4, so that packets are dropped/rejected indiscriminately across all service classes. This introduces inaccuracies in its predictions since our development in Section 4.3 relies on the fact that only new requests can be rejected, and retransmitted ones are always dropped if they cannot be accepted. As we shall show, this negatively affects the filter's overall performance.

In creating the simulated environment, our main goal is to subject these filters to realistic load conditions so that any results would be applicable to real-world deployment scenarios. We also want to avoid any unnecessary complexity without sacrificing accuracy. We, therefore, employ a simple setup where a server receives incoming requests through a high-speed link. Clients on the other side of the link also have enough resources to generate the desired request arrival distribution. Each incoming request to the server must first pass through one of the filters described above, which is located at the entry point of the server (the lowest level of its communication protocol stack). To eliminate external effects from our measurements, we make two assumptions about the system under test. First, we assume that the client-to-server network path is bottleneck-free, with negligible propagation delay (however, with a jitter distribution that follows the one in Section 3.1). Second, we assume that a request, once accepted, completes the connection handshake immediately, with no delay. The latter models a typical server, where SYN requests are handled in the operating system, and hence unaffected by processing delays and application queues. Therefore, the results presented in this section only reflect the effects of the underlying control mechanisms (i.e., one of the four filters) on the first half of the connection-establishment handshake. This way, one can apply the results in a wider range of operational scenarios.

A self-similar traffic model is used to generate the arrival distribution of new requests. Self-similarity is crucial in describing the bursty behavior of current network traffic. Specifically, we use the *Multifractal Wavelet Model (MWM)*, proposed by Riedi [35], as our underlying request generation model. While other approaches [14, 32] also produce self-similar traffic, the MWM-based approach is shown to have better flexibility and accuracy in modeling a wide range of real network traffic conditions. Traffic generation relies on four basic parameters: the mean, variance, and *Hurst* parameter of the arrival distribution, and the number of wavelet scales [35]. In our simulation, we use a mean of 100 reqs/sec and a Hurst parameter of 0.8, a

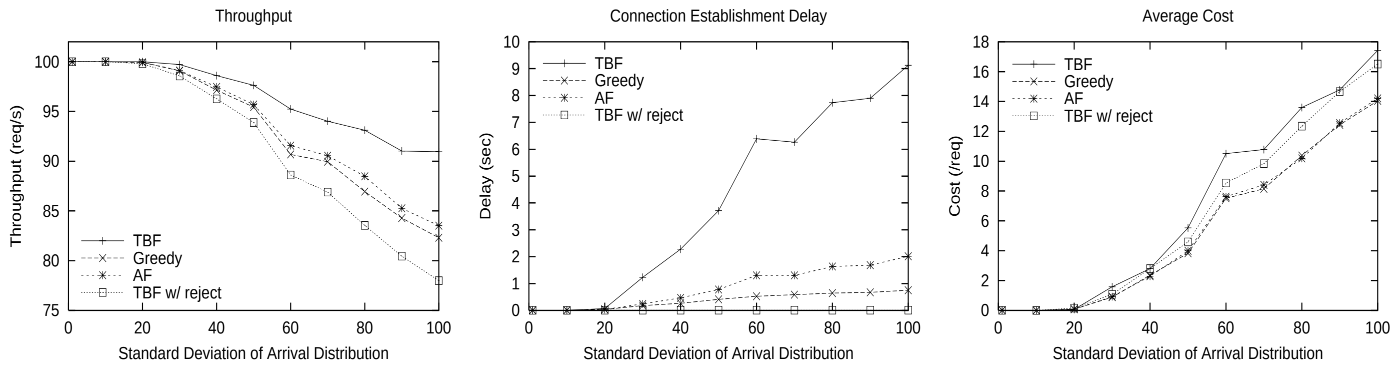

Figure 8: Effects of burstiness

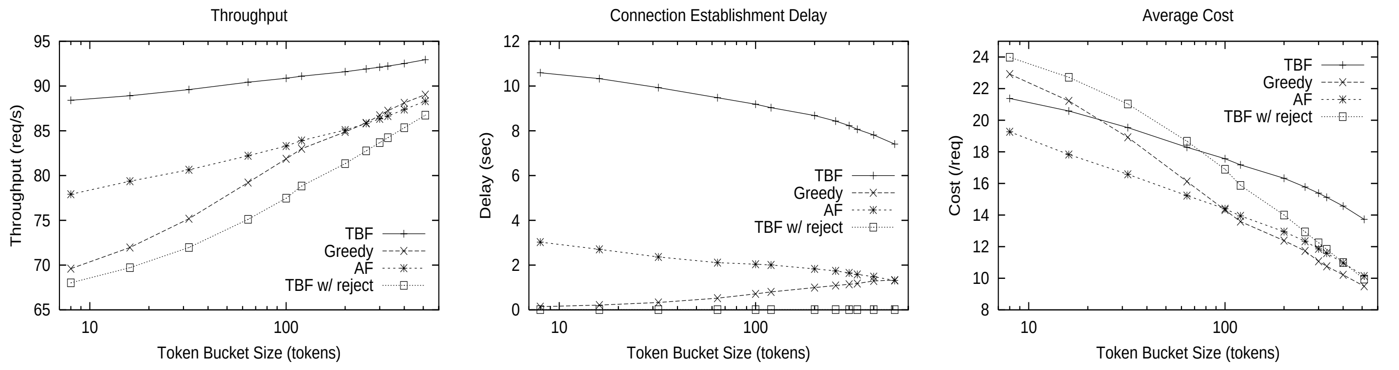

Figure 9: Effects of bucket size

commonly-measured value in real network traffic [26, 35]. The number of wavelet scales is set to 10 to generate 2^{10} data points, each of which represents the number of new client arrivals in a 1-sec interval. Finally, the variance is, in a sense, an indicator of the burstiness of the trace, so we use this parameter to vary the burstiness of the generated traffic.

Because we need to conduct a large number of experiments to cover the wide range of variable parameters, we evaluate the filters using an event-driven simulation, ensuring that long-duration simulations may be performed in reasonable time (simulating 54 days of non-stop testing in less than 6 hours).

As mentioned earlier, our simulator, *eSim*, is based on the Linux TCP stack implementation of SYN-request processing and connection-initiation processes. We have compared the simulation results against real-world measurements and validated the results obtained from the simulator to be within 5% of the real implementation.[7]

We want to characterize the behavior of the four filters when the underlying system is critically-loaded. By this we mean that the average arrival rate of new requests is close to the server processing rate, which is reflected in the token refill rate. Thus, we set $r_t = 120$ tokens/sec, reflecting a server that has 20% greater capacity than the mean request-arrival rate of 100 reqs/sec. Configuring the system to be under-utilized or over-utilized would yield predictable results: TBF would work well in under-utilized scenarios, while TBF w/ Reject would be best in over-utilized cases. In what follows, each experiment is set to simulate 600 seconds of incoming requests, and is repeated 30 times, amounting to roughly 1.8 million new arrivals for each plotted point.

[7]Our tests were performed with a Linux 2.2.17 kernel in an isolated testbed against simulated HTTP 1.0 clients. We used working prototypes of the four filters and compared measurements from 10 randomly selected configuration parameters (excluding network jitter) with ones obtained from our simulation.

We compare the performance of the four filters using three basic metrics. The first is *throughput*, expressed as a percentage of the connection requests that ultimately succeed, i.e., a token is available when the request or its retransmission arrives at the filter. The second is *connection-establishment delay*, which is computed by averaging the elapsed time before a client successfully connects, times out, or is rejected for all incoming requests. This does not include any propagation delay or internal system queueing time, as described earlier. The third is the *average cost*, which is computed much like the connection-establishment delay, but also includes a rejection cost for requests that are rejected and those who time out. We also measure the average number of retransmissions before a request is accepted, is rejected, or times out, as well as the percentage of requests that time out. These tie closely to our other metrics, as they directly affect client-perceived delay and the average cost.

6.1 Effects of Load Characteristics

We conduct two experiments to characterize the effects of changing network load. In the first experiment, we vary the standard deviation (effectively the variance) of the distribution of incoming requests while maintaining the mean arrival rate fixed at 100 reqs/s. Basically, as the standard deviation is increased, so is the burstiness of incoming requests. In this experiment, we set the size of the token bucket to match the refill rate (i.e., $W = 120$ tokens). We use a 1-second bucket size to better show the effects of burstiness on the tested filters. We set the AF and Greedy window sizes to 50 seconds (i.e., $S = S_G = 50$ seconds). Finally, we set the delay cost to $h = 1/(sec \cdot req)$, and the rejection cost to $r = 75/req$, equal to the timeout delay cost for all clients.

Figure 8 shows the throughput, connection-establishment delay, and average cost for all four filter designs. It shows that as the burstiness of incoming requests is increased, it negatively impacts the performance of all four filters. Furthermore, the TBF provides a higher throughput for bursty arrivals, but performs poorly in controlling client-perceived delay. The above figure also highlights the importance of using the AF or Greedy filter, namely, as the burstiness of traffic is varied, the expected delay remains in check. Of course, throughput is limited by the fixed system capacity, so when traffic becomes increasingly bursty, fewer requests will succeed (as more will be rejected), which will, in turn, decrease the overall percentage of accepted requests. The figure also shows that a reject policy (TBF w/ Reject) provides throughput only 5% and 8% less than the AF and Greedy filters, respectively, but with a cost about 15% greater.

There are two points to be made here. First, the throughput difference varies with the load condition and system utilization. In other experiments, where we allowed the mean to also increase, we find that as the traffic load approaches system capacity, the difference between these filter designs is maximized. Second, we use only a moderate reject cost parameter, but, as we will show shortly, the performance of the Reject filter quickly deteriorates as the cost of rejecting clients increases.

We also perform similar experiments to determine the effects of both RTO and delay jitters, where we vary the mean of the jitter distribution from 10 msec to 1 sec. The RTO and delay jitters have no noticeable impact on any of our performance metrics. The resulting graphs show essentially constant values across the entire range, and are, therefore, not presented here. This result is not surprising — even though jitter tends to spread out synchronized retransmissions, since the incoming traffic is so bursty across a wide range of time scales, the aggregate arrival remains very bursty. We expect that the jitter plays an important role when incoming requests include only occasional short bursts.

6.2 Effects of Configuration Parameters

The second step in our evaluation is to study the effects of the main configuration parameters on the performance of the filters: bucket size (W), rejection cost (r), and window size (S). We use a similar setup to that in Section 6.1, configure the token bucket size to $W = 120$ tokens, use AF and Greedy window sizes of 50 seconds, and set the costs of rejection and delay to $r = 75/req$ and $h = 1/(sec \cdot req)$, respectively. We ran a separate test for each parameter, varying it while fixing the others.

Figure 9 shows our three performance metrics as the bucket size is increased from 8 to 512 tokens. Note that W is doubled at each successive point, and the graph is plotted using a log scale. The figure shows that as the bucket size increases, so does the performance of the filter. Larger buckets allow the filters to accept larger bursts into the system, which causes increased performance by reducing the number of retransmissions, rejects, and timeouts. This would also result in longer system queues that may adversely affect overall service times, but, since we are only looking at the connection-establishment performance, such application queuing effects are not apparent in our metrics. It is interesting to see that the performance of the filters improves linearly with an exponential increase in bucket size, which implies that a linear increase in bucket size produces only a logarithmic improvement in overall performance of the filters. This is due to the self-similar nature of traffic across multiple time scales. The heavy-tailed distribution of burst sizes ensures that even as the bucket size is increased, there would still be a small percentage of bursts that are too large to be fully absorbed.

An increased bucket size also reduces the performance difference between AF and Greedy filter. This can be seen when the bucket size is larger than 100 tokens in Figure 9. This behavior is due to the fact that larger buckets decrease the prediction error that were introduced by relaxing assumptions A3 and A4. In particular, when the W is small, there are fewer tokens in the second stage buckets from which the borrowing algorithm can obtain a token, so any request passed to the second stage is likely rejected rather than dropped. These rejected requests may include those that are retransmitted and should only be dropped or accepted. As W increases, the rejections decrease, which also increases the accuracy of both mechanisms.

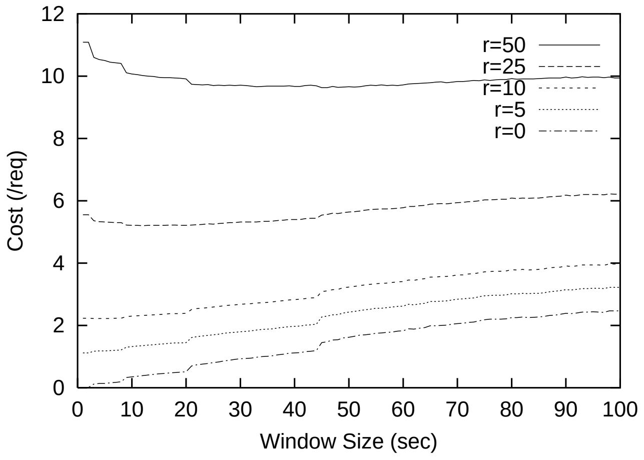

Figure 10: Relationship between Abacus window size and rejection cost

In Section 4.2, we showed the important role that the rejection cost plays in the derivation of the optimal control policy. A small rejection cost implies that it is more desirable to reject incoming requests than increasing the average connection delay. A large r indicates that it is worth sacrificing delay to avoid losing clients through rejects and timeouts. Recall that the cost of a timed-out connection request is the total delay cost plus the rejection cost, since it suffers the same effective result as a reject, but with a longer client notification time. To evaluate the performance impact of r, we first establish the relationship between the AF window sizes, S, and the costs h and r. Since it is the relative values of r and h that matter, i.e., ratio r/h, we fix $h = 1/(sec \cdot req)$ and vary both S and r. Figure 10 shows the performance, in terms of total cost, of the AF as S is increased. Each line in the figure represents different r values. Two observations are directly made from the figure:

O1: The total cost has noticeable step-like jumps around the boundaries of retransmission timeouts because the AF at each jump point has the ability of borrowing from an additional bucket.

O2: The cost is minimized when setting $S = \max\{T_i : T_i \leq r/h\}$, where T_i is the time of the i-th retransmission, i.e., $T_i = \sum_{l=1}^{i} d_l$. Therefore, even when r/h is very large, the maximum value of S is bounded to T_{conn}.

Using the above relationship to set the AF window size to minimize cost, we evaluate the impact of rejection cost on the performance of the underlying filters. Here we set S as described earlier and maintain $S_G = 50$ sec. Unlike the AF window size, the Greedy window size need not vary with r/h. As mentioned in Section 4.3, the value of S_G should be set to at least T_N. Figure 11 shows the performance impact of the rejection cost on the four filters. Overall, as the rejection cost is increased, the throughput of the TBF and Reject filters remain unchanged, but the differences between the average costs increase dramatically. Both the AF and Greedy filters, which explicitly account for costs, are significantly affected, and are biased towards rejecting requests when r is low. This changes as the cost of rejecting a clients becomes greater than the delay costs of retransmission. However, no matter how high the rejection cost is, they never switch to a drop-only policy. This is because the cost of a timed-out request is always greater than that of a rejected packet, so it is always preferrable

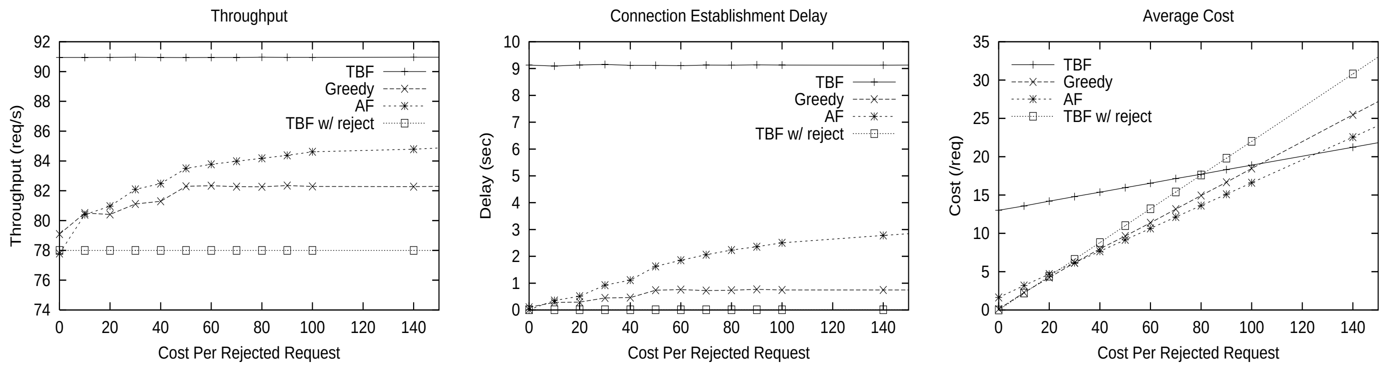

Figure 11: Effects of rejection cost

to limit the expected number of timed-out requests by rejecting some arrivals. This is directly reflected in the computation of λ_{thresh} for the Greedy filter and in the borrowing behavior of the AF. Over a wide range of rejection cost values (0 to 500), the average percentage of clients that time out is 5.5% for TBF, 0.01% for Greedy, 0.04% for AF, and 0% for Reject (as it does not allow retransmissions). Finally, when the rejection cost is very high, TBFs, the only filters that do not use reject, incur the lowest costs. This may indicate that both the Greedy and AF tend to predict more timeouts than really do occur, and reject more requests than they should. Of course, with varying traffic conditions, the actual crossover points in these graphs will change greatly, but for a large range of reasonable rejection-cost values, AFs seem to work very well.

Overall, our experiments demonstrate the effectiveness of the AF in controlling client-perceived delays. More remarkably, the simple design is able to find a balance between a lenient control policy (as with a TBF) and a strict one (as with a TBF w/ Reject) in response to changing request load conditions. It, therefore, shows the greatest advantage over the TBF under realistic, highly-bursty loads, while the latter performs well only under predictable loads. The Greedy filter, based on our analytic derivations, also achieves good results, but at a much higher computational cost. We note that this is just one of many possible implementations of our analytic results from Section 4, so with different assumptions and simplifications, it should be possible to derive filter designs that yield improved performance. In particular, improved estimation of arriving request distributions (e.g., using SYN caches [27]) and the request transmission classes is key to improving these control mechanisms. Specifically, this may reduce the apparent over-estimation in both the AF and the Greedy filter of future requests that would time out, which results in a larger number of requests rejected than is absolutely necessary to minimize costs.

7 Related Work

Bounding client-perceived delay can be addressed by various server layers (application, middleware, OS) and network components (routers, front-end switches). On the server side, much work has focused on providing resource reservation and protection [2, 4, 38]. Unfortunately, existing OS techniques, such as Resource Containers [5] and IBM's Workload Manger [2], operate at a higher level of the communication stack, and do not consider the fine-grain interaction between their control and incoming traffic. Some network-level control mechanisms, such as Lazy Receiver Processing [4] and QGuard [22], do regulate server load, but do not relate control to client-perceived delay. Our work in this paper complements general OS controls, defining a novel method for maximizing the number of accepted requests while bounding the average client-perceived delay. Because the AF is devised as a network filter and is computationally efficient, it can easily be implemented as a plug-in OS module or deployed on network devices (e.g., load-balancers).

In [10, 16, 31, 36], analytical characterizations of the throughput of TCP's congestion control as a function of round-trip time (RTT) and drop probability are presented. While we also follow an analytical approach, our model focuses on the interaction between network-level controls and the aggregate behavior of incoming service requests, not just a single TCP connection. Looking at the aggregate is necessary to characterize the change in drop probability as a function of control policy and incoming traffic. Our work can be viewed as an extension to earlier models [17, 19] that assume predetermined or measured drop probability. It also relates to measurement-based admission control schemes [21] that maximize the number of admitted connections without violating specific performance constraints (e.g., response time). Finally, our work follows the same theme in [7, 13], where prediction is used to profile future demands of real-time tasks. This prediction is used to schedule or reject tasks such that the overall probability of tasks meeting their deadlines is maximized.

One cause of increased client-perceived delay is the queuing of new connection requests. Queue management solutions such as Class-Based Queues (CBQ) [18], Active Queue Management (AQM) [8, 17], and Explicit Congestion Notification (ECN) [15] can provide fine-grain controls. Some studies [28] have also focused on finding buffer bounds for self-similar incoming traffic. In all of the studies, the assumed arrival distribution includes neither the retransmissions nor the re-synchronization behavior of dropped connection requests. Our work can be considered as a specialized queue management solution for new connections. In some respect, the AF uses TCP retransmission timeouts as implicit queuing to effectively reduce internal queuing requirements by only admitting those requests that can be handled within a timeout period. Thus, it reduces buffer requirements and permits rapid adaptation to changing load conditions.

8 Conclusions

Traffic control is used in servers to limit load and bound service latencies in the presence of bursty request arrivals. However, these controls may adversely affect client-perceived delays by increasing the average connection-establishment latencies. We have presented an analysis of traffic shaping and the impact this has on the delays perceived by clients. Based on this analysis, we have proposed a predictive control mechanism that estimates future delay costs due to current control actions. As a practical approximation of this model, we introduce *Abacus Filters* (AFs), which is a novel mechanism of regulating incoming requests to limit client-perceived delays. Experimentally, we have shown that as compared to the *de facto* traffic shaping standard, token bucket filters, AFs provide much stricter control over delays, while avoiding any significant decreases in throughput. As the request traffic burstiness increases, the AF limits degradation of client-perceived delay, matching the drop rate to estimated future capacity, while TBF performance degrades since many requests eventually time out. For delay-sensitive, short-lived TCP connections, including the vast majority of HTTP traffic, where transfer times are often dominated by connection-establishment delays, AFs can provide traffic regulation to heavily-loaded servers, while providing good performance to clients. Furthermore, AFs can be implemented very efficiently, and, as they do not need to be on the end host, may be migrated to front-end switches or firewall devices.

References

[1] ACHARYA, A., AND SALTZ, J. A Study of Internet Round-trip Delays. Tech. Rep. CS-TR-3736, University of Meryland Technical Report, 1996.

[2] AMAN, J., EILERT, C. K., EMMES, D., YOCOM, P., AND DILLENBERGER, D. Adaptive Algorithms for Managing Distributed Data Processing Workload. *IBM Systems Journal 36*, 2 (1997), 242–283.

[3] BAKER, F. RFC1812: Requirements for IP Version 4 Routers. *IETF* (June 1995).

[4] BANGA, G., AND DRUSCHEL, P. Lazy Receiver Processing LRP: A Network Subsystem Architecture for Server Systems. In *Second Symposium on Operating Systems Design and Implementation* (October 1996).

[5] BANGA, G., DRUSCHEL, P., AND MOGUL, J. Resource Containers: A New Facility for Resource Management in Server Systems. In *Third Symposium on Operating Systems Design and Implemenation* (February 1999), pp. 45–58.

[6] BARFORD, P., AND CROVELLA, M. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *In Proceedings of Performance'98/ACM Sigmetrics'98* (May 1998), pp. 151–160.

[7] BESTAVROS, A. Load Profiling in Distributed Real-Time Systems. In *The 17th International Conference on Distributed Computer Systems* (May 1997).

[8] BRADEN, B., ET AL. Recommendations on Queue Management and Congestion Avoidance in the Internet. *RFC 2309* (1998).

[9] BRADEN, R. RFC1122: Requirements for Internet Hosts - Communication Layers. *IETF* (October 1989).

[10] CARDWELL, N., SAVAGE, S., AND ANDERSON, T. Modeling TCP Latency. In *Proc. of the IEEE INFOCOM 2000* (2000), pp. 1742–1751.

[11] CHAKRAVARTI, I. M., LAHA, R. G., AND ROY, J. *Handbook of Methods of Applied Statistics*, vol. 1. John Wiley and Sons, Inc., 1967.

[12] CHERKASOVA, L., AND PHAAL, P. Session Based Admission Control: a Mechanism for Improving Performance of Commercial Web Sites. In *Proceedings of Seventh IwQoS* (May 1999), IEEE/IFIP event.

[13] DINDA, P. A., KALLIVOKAS, L. F., LOWEKAMP, B., AND O'HALLARON, D. R. The Case for Prediction-Based Best-Effort Real-Time Systems. In *IPPS/SPDP Workshops* (1999), pp. 309–318.

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

[15] FLOYD, S. TCP and Explicit Congestion Notification. *ACM Computer Communication Review 24*, 5 (1994), 10–23.

[16] FLOYD, S., HANDLEY, M., PADHYE, J., AND WIDMER, J. Equation-Based Congestion Control for Unicast Applications. In *Proceedings of the ACM SIGCOMM '00* (August 2000), ACM.

[17] FLOYD, S., AND JACOBSON, V. Random Early Detection Gateways for Congestion Avoidance. *ACM/IEEE Trans. on Networking 1*, 4 (1993), 397–417.

[18] FLOYD, S., AND JACOBSON, V. Link-sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking 3*, 4 (August 1995), 365–386.

[19] HOLLOT, C., MISRA, V., TOWSLEY, D., AND GONG, W. A Control Theoretic Analysis of RED. In *Proceedings of the IEEE INFOCOM 2001* (2001).

[20] JACOBSON, V. Congestion Avoidance and Control. In *Proceedings of the ACM SIGCOMM '88* (August 1988).

[21] JAMIN, S., DANZIG, P., SHENKER, S., AND ZHANG, L. A Measurement-based Admission Control Algorithm for Integrated Services Packet Networks. *IEEE/ACM Transactions on Networking 5*, 1 (Feb 1997), 56–70.

[22] JAMJOOM, H., REUMANN, J., AND SHIN, K. G. QGuard: Protecting Internet Servers from Overload. Tech. Rep. CSE-TR-427-00, University of Michigan Technical Report, 2000.

[23] JAMJOOM, H., AND SHIN, K. G. Persistent dropping: An efficient control of traffic aggregates. In *Proceedings of the ACM SIGCOMM '03* (Karlsruhe, Germany, August 2003), pp. 287–298.

[24] KESHAV, S. *An Engineering Approach to Computer Networking*. Addison-Wesley Publishing Company, 1997.

[25] KHAUNTE, S. U., AND LIMB, J. O. Statistical Characterization of a World Wide Web Browsing Session. Tech. rep., Georgia Institute of Technology, 1997.

[26] LELAND, W. E., TAQQU, M. S., WILLINGER, W., AND WILSON, D. V. On the Self-Similar Nature of Ethernet Traffic (extended version). In *IEEE/ACM Transactions on Networking* (1994), pp. 2:1–15.

[27] LEMON, J. Resisting SYN Flood DoS Attacks with a SYN cache. In *BSDCon 2002* (Feb 2002).

[28] LIKHANOV, N. *Bounds on the Buffer Occupancy Probability with Self-similar Input Traffic*. Self-similar Network Traffic and Performance Evaluation. John Wiley and Sons, Inc., 2000, ch. 8, pp. 193–215.

[29] MORRIS, R., AND LIN, D. Variance of Aggregated Web Traffic. In *Proceedings of the IEEE INFOCOM 2000* (2000), vol. 1, pp. 360–366.

[30] MUKHERJEE, A. On the Dynamics of Significance of Low Frequency Components of Internet Load. In *Internetworking: Research and Experience* (December 1994), vol. 5, pp. 163–205.

[31] PADHYE, J., FIROIU, V., TOWSLEY, D., AND KUROSE, J. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proceedings of the ACM SIGCOMM '98* (1998), pp. 303–314.

[32] PAXSON, V. Fast, Approximate Synthesis of Fractional Gaussian Noise for Generating Self-Similar Network Traffic. *Computer Communication Review 27*, 5 (October 1997), 5–18.

[33] PAXSON, V., AND FLOYD, S. Wide Area Traffic: the Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking 3*, 3 (1995), 226–244.

[34] POSTEL, J. RFC793: Transmission Control Protocol. *Infomation Science Institute* (September 1981).

[35] RIEDI, R. H., CROUSE, M. S., RIBEIRO, V. J., AND BARANIUK, R. G. A Multifractal Wavelet Model with Application to Network Traffic. *IEEE Transactions on Information Theory 45*, 4 (1999), 992–1018.

[36] SAHU, S., NAIN, P., DIOT, C., FIROIU, V., AND TOWSLEY, D. F. On Achievable Service Differentiation with Token Bucket Marking for TCP. In *Measurement and Modeling of Computer Systems* (2000), pp. 23–33.

[37] SARVOTHAM, S., RIEDI, R., AND BARANIUK, R. Connection-level Analysis and Modeling of Network Traffic. In *Proceedings of the ACM SIGCOMM Internet Measurment Workshop* (November 2001).

[38] SPATSCHECK, O., AND PETERSON, L. L. Defending Against Denial of Service Attacks in Scout. In *Third Symposium on Operating Systems Design and Implementation* (February 1999), pp. 59–72.

[39] ZHANG, L., DEERING, S., AND ESTRIN, D. RSVP: A New Resource ReSerVation Protocol. *IEEE network 7*, 5 (September 1993).

[40] ZHANG, Y., DUFFIELD, N., PAXSON, V., AND SHENKER, S. On the Constancy of Internet Path Properties. In *Proceedings of the ACM SIGCOMM Internet Measurment Workshop* (November 2001).

[41] ZHE WANG AND PEI CAO. Persistent Connection Behavior of Popular Browsers. ttp://www.cs.wisc.edu/ cao/papers/persistent-connection.html.

[42] ZONA RESEARCH INC. The Need for Speed. July 1999.